Companion Course Notes for
Volume 6 of the Software Foundations Series
https://softwarefoundations.cis.upenn.edu/slf-current/

Foundations of Separation Logic for Sequential Programs

Arthur Charguéraud

Document compiled on November 28, 2023

Abstract

Separation Logic brought a major breakthrough in the area of program verification. The all-in-Coq course entitled *Separation Logic Foundations* is published as Volume 6 of the *Software Foundations* series. The present document corresponds to the companion course notes for that volume. It covers the key definitions, formatted in traditional LaTeX style rather than in Coq. It also includes an historical survey of Separation Logic for sequential programs.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In the 90's, the use of formal methods was mainly motivated by safety-critical applications, where a *bug* in the code could mean that people get hurt. Of course, such applications remain relevant. Yet, two game-changing evolutions related to software have significantly broadened the scope of application of formal methods: massive-scale deployment, and digital security concerns.

Regarding deployment, consider that a given piece of code may be executed by a couple *billion* users. The cost of one bug, multiplied by the number of users, adds up to such a large amount that it motivates corporations to invest considerable efforts in eliminating bugs. The Big Tech companies, which do provide software to billions of users, are among those that go beyond traditional testing, and leverage formal methods for critical products. For example, AWS (Amazon's cloud) exploits TLA+ [Yu et al., 1999] to apply model checking to detect flaws in the design of their fault-tolerant, distributed systems [Newcombe et al., 2015]. Meta exploits the Infer tool [Calcagno and Distefano, 2011] for static analysis of its Android and iOS apps, in particular. The Infer tool is now also being used by Spotify, Uber, Mozilla, Microsoft, AWS, and many others. Besides, Meta has invested efforts in verifying parts of a microkernel for embedded devices [Carbonneaux et al., 2022]. Likewise, Google recently announced KataOS, an operating system for embedded devices that run machine-learning applications [Google, 2022], implemented on top of the seL4 mechanically-verified microkernel [Klein et al., 2010]. We can reasonably expect the deployment of software at a very large scale, and thus the interest in bug-free programs, to keep growing.

The second critical aspect is security. Software is widespread in every aspect of society, from corporations and factories to daily consumer products such as phones, TVs, cars, etc. A software bug may induce a source of vulnerability, that an attacker may exploit to crash a system, or (usually worse) to steal data, or (much worse) to take remote control of a physical system, such as a car, a factory, or a city-management system. Attackers may also exploit a flaw to set up a backdoor for a future attack. Attackers may be motivated by extorting ransoms, by stealing valuable information or technology, or by taking an advantage in the cyberwarfare. Attacks are carried out not only by individuals and small teams of hackers, but also by official and undercover government agencies. The cumulative cost of cyberattacks is very hard to evaluate, with estimates ranging from hundreds to thousands of billion US dollars per year. Because cyberattacks can be performed remotely from anywhere on earth, possibly by leaving very few tracks behind, with limited investment and possibly huge returns, we can expect such attacks to continue at a sustained rate. The use of formal methods alone certainly does not make software systems invulnerable, but it can help reduce the attack surface.

The large number of users concerned, combined with the desperate need for increased soft-

ware security, will, one can speculate, motivate unprecedented growth in the use of formal methods. One key question is whether existing verification tools can be improved to decrease the cost of verifying large and complex systems. Another question is how many years it will take to train the workforce necessary for specifying and verifying a significant fraction of the highly sensitive software components in use.

1.2 Separation Logic

Separation Logic brought a major breakthrough in the area of program verification [O'Hearn, 2019]. Since its introduction, it has made its way into a number of practical tools that are used on a daily basis for verifying programs ranging from pieces of operating systems kernels [Xu et al., 2016; Carbonneaux et al., 2022] and file systems [Chen et al., 2015] to data structures [Pottier, 2017] and state-of-the-art algorithms [Guéneau et al., 2019; Haslbeck and Lammich, 2021]. These programs are written in various programming languages, including machine code [Myreen and Gordon, 2007], assembly [Ni and Shao, 2006; Chlipala, 2013], C-language [Appel and Blazy, 2007], OCaml [Charguéraud, 2011], SML [Kumar et al., 2014], 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].

The first paragraph from Reynold's paper [2002] summarizes the situation prior to Separation Logic in the following words.

> *Approaches to reasoning about [the use of shared mutable data structures] have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size.*

Today, the core definitions of Separation Logic may appear as *the obvious thing to write*, or even as *the only thing that would make sense to write*. Perhaps the best way to truly value the contribution of Separation Logic is to realize that, following the introduction of the first program logics in the late sixties [Floyd, 1967; Hoare, 1969; Dijkstra, 1975], people have tried for 30 years to verify programs *without* Separation Logic.

1.3 Foundational Verification

The term *formal methods* covers a broad range of tools, with different purposes, e.g., to check functional properties, to check convergence properties, to verify cryptographic protocols, to verify hardware circuits, to analyze resource consumption, to verify time-channel attacks, etc. *Deductive program verification* aims at formally verifying that all possible behaviors of a given program satisfy formally defined properties. These properties constitute the *specification* of the program.

The *verification* process is said to be *machine-checked* if a program called a *theorem prover* is used to validate every step of the reasoning involved in the process. A theorem prover may consist either of an *automated* theorem prover, or of an *interactive* proof assistant (e.g., *Coq*).

The verification process is said to be *foundational* if the reasoning on the program of the behavior is established, via machine-checked proofs, with respect to a formalization of the *operational semantics* of the source programming language. Foundational verification, when combined with

the use of a machine-checked compiler, yields very high confidence on the fact that the machine code produced does indeed satisfy the desired formal specification.

1.4 Contents of the Book

This book presents the key ideas involved in the construction of **an interactive framework based on Separation Logic, allowing to establish functional correctness and termination, in a foundational way.**

This book focuses on reasoning about *sequential* programs. Reasoning about *concurrent* programs, with multiple threads interacting with each other, is left as matter for another book.

The present book accompanies an all-in-Coq course, entitled *Foundations of Separation Logic*, and released as Volume 6 of the *Software Foundation* series, edited by Benjamin Pierce. This course includes more than 130 exercises of various difficulties. Its prerequisites are the contents of Volume 1 (Logical Foundations) and Volume 2 (Programming Language Foundations) of the series.

Chapter 2

Features of Separation Logic

This chapter gives an overview of the features that are specific to Separation Logic: (1) the *separating conjunction* and the *frame rule*, which enable *local reasoning* and *small-footprint specifications*; (2) the treatment of aliasing; (3) the specification of recursive pointer-based data structures such as mutable linked lists; and (4) 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. This fragment must include 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 then 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 traditionally called *heaps*, and predicates 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 manuscript, 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 Section 6.1.) 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 may be stated in the form shown below:

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

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 work 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 conjunction. The most important operators are summarized below—they appear in examples throughout the rest of this section, and are formally defined further on (Section 4.2).

- $p \hookrightarrow v$, 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 that we will consider 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 $\{H\}\ (\texttt{incr}\ p)\ \{Q\}$, as shown below.

Example 2.2.1 (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 that stores an integer value, 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 `incr`p, namely the unit value, needs not be assigned a name.

Throughout the rest of the manuscript, the outermost universal quantification (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 2.3.1 (Applying the frame rule to 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 2.2.1 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:

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

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 `incr_two` 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 2.4.1 (Potentially aliased arguments) *The function:*

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

admits the following two specifications.

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

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 2.5.1 (Specification of primitive operations on references)

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

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 Section 6.1). 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, namely 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 $(p \hookrightarrow v) \star (q \hookrightarrow v')$ that appears below.

Example 2.5.2 (Application of the frame rule to the specification of allocation)

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

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.

Chapter 3

Representation Predicates

3.1 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 3.1.1 (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:*

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

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

A mutable linked list is described by a heap predicate of the form 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 Mlist $L'\,q$ describes the tail of the list. The case disjunction is expressed using Coq's pattern-matching construct.

Definition 3.1.2 (Representation of a mutable list)

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

Example 3.1.1 (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.*

$$\begin{array}{rcl} \mathit{Mlist}\,(8 :: 5 :: 6 :: \mathit{nil})\,p_0 & \equiv & \exists p_1.\ (p_0.\mathit{head} \hookrightarrow 8) \star (p_0.\mathit{tail} \hookrightarrow p_1) \\ & & \star\ \exists p_2.\ (p_1.\mathit{head} \hookrightarrow 5) \star (p_1.\mathit{tail} \hookrightarrow p_2) \\ & & \star\ \exists p_3.\ (p_2.\mathit{head} \hookrightarrow 6) \star (p_2.\mathit{tail} \hookrightarrow p_3) \\ & & \star\ [p_3 = \mathit{null}] \end{array}$$

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 3.1.2 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 3.1.3 (Alternative definition for Mlist)

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

Note that this alternative definition is not recognized as structurally-recursive by Coq. Its statement may be formulated as an equality, and proved correct with respect to Definition 3.1.2.

3.2 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 3.2.1 (Specification of in-place append for mutable lists)

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

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 Mlist $L_2\, p_2$ from the postcondition.

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

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

We follow the convention of placing pure hypotheses as premises outside of triples, as in general it 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 Mlist $L\, p$, and the postcondition describes the output heap as Mlist $L\, p \star$ Mlist $L\, p'$, where p' denotes the address of the new list.

Example 3.2.2 (Specification of a copy function for mutable lists)

$$\{\mathit{MList}\, L\, p\}\ (\mathit{mcopy}\, p)\ \{\lambda r.\ \exists p'.\ [r = p'] \star (\mathit{MList}\, L\, p) \star (\mathit{MList}\, L\, p')\}$$

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 may be found in Section 9.5. The key steps of that proof are summarized next. Details may be found in [Charguéraud, 2020, Appendix E].

Proof *The specification of* `mcopy` *is proved by induction on the length of the list L. If the list L is empty, the result p' is the null pointer, and $\mathit{Mlist}\,\mathit{nil}\,p'$ is equivalent to the empty heap predicate. When the list is nonempty, $\mathit{Mlist}\,L\,p$ unfolds as $(p.\mathit{head} \hookrightarrow x) \star (p.\mathit{tail} \hookrightarrow q) \star (\mathit{Mlist}\,L'\,q)$. 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 $\mathit{Mlist}\,L'\,q$. Over the scope of that call, the frame rule is used to put aside the head cell, described by $(p.\mathit{head} \hookrightarrow x) \star (p.\mathit{tail} \hookrightarrow q)$. 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:*

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

which may be folded to $(\mathit{Mlist}\,L\,p) \star (\mathit{Mlist}\,L\,p')$, 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.

Verification of termination via proofs by induction. The previous example shows the proof of a recursive function. A key aspect of this proof is that the specification is proved by induction, using Coq's support for well-founded induction. More precisely, we aim to establish a specification for a mutable linked list whose logical model is the Coq list L. By induction principle, we may assume this specification to hold for any mutable linked list whose logical model is a sublist of L. More generally, the CFML framework manipulates total-correctness triples. Hence, when one establishes a triple for a term, one establishes in particular a proof of termination for that term.

One may wonder what happens if trying to establish a triple for a term that diverges. Consider for example the definition `let rec f x = f x`. The term `f 0` diverges. To establish a triple for the term `f 0`, one would need to establish a triple for its body, which is also `f 0`. Such a hypothesis may only come from an induction principle, yet there exist no measure or well-founded relation for which the argument `0` could be viewed as *smaller* than itself. Thus, a user would get stuck trying to establish a triple for `f 0`. More generally, by virtue of the soundness of the framework, no total-correctness triple can be established for a term that diverges.

3.3 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 `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 3.3.1 (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 given away. 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 3.3.1 (Deallocation of a list cell) *The function* `mfree_cell` *deallocates a list cell.*

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

Example 3.3.2 (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 begin
    let q = p.tail in
    mfree_cell p;
    mfree_list q
  end
```

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

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

Remark (Languages with implicit garbage collection) *For languages equipped with a garbage-collector, Separation Logic can be adapted to allow freely discarding heap predicates (see Chapter 11).*

Chapter 4

Heap Predicates

4.1 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 Chapter 6.

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 fmap $A\,B$ denote the type of finite maps from a type A to an (inhabited) type B.

Definition 4.1.1 (Representation of heaps) *The type state 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 is unspecified when applied to non-disjoint arguments; in other words, it may return arbitrary results for arguments with overlapping domains.

4.2 Core Heap Predicates

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

Definition 4.2.1 (Heap predicates) *A heap predicate is a predicate of type: state → Prop.*

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

Definition 4.2.2 (Core heap predicates)

Operator	Notation	Definition
empty predicate	$[\,]$	$\lambda h.\ h = \varnothing$
pure fact	$[P]$	$\lambda h.\ h = \varnothing \wedge P$
singleton	$p \mapsto v$	$\lambda h.\ h = (p \to v) \wedge p \neq \mathit{null}$
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 (state $\to$ 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 (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 $[\mathsf{True}]$.*

4.3 Other Heap Predicates

Traditional presentations of Separation Logic include four additional operators, $\bot$, $\top$, $\vee$, and $\wedge$. These four operators may be encoded in terms of the ones from Definition 4.2.2, 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.

Operator	Notation	Definition	Encoding
bottom	$\bot$	$\lambda h.\ \mathsf{False}$	$[\mathsf{False}]$
top	$\top$	$\lambda h.\ \mathsf{True}$	$\exists(H : \mathsf{state} \to \mathsf{Prop}).\ H$
disjunction	$H_1 \vee H_2$	$\lambda h.\ (H_1\ h \vee H_2\ h)$	$\exists(b : \mathsf{bool}).\ \mathsf{If}\ b\ \mathsf{then}\ H_1\ \mathsf{else}\ H_2$
non-separating conjunction	$H_1 \wedge H_2$	$\lambda h.\ (H_1\ h \wedge H_2\ h)$	$\forall(b : \mathsf{bool}).\ \mathsf{If}\ b\ \mathsf{then}\ H_1\ \mathsf{else}\ H_2$

Definition 4.3.1 (Representation predicate for lists defined with disjunction) *The representation predicate for lists introduced in Definition 3.1.1 can be reformulated using the disjunction operator instead of relying on pattern-matching. The corresponding definition, which may be useful if the host logic does not feature a pattern-matching construct, is as follows.*

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

Chapter 5

Entailment

5.1 Definition and Properties of Entailment

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

Definition 5.1.1 (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 5.1.1 (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.[1] The antisymmetry property plays a critical role for stating the key properties of Separation Logic operators in the form of equalities, as detailed next.

The useful properties of entailment involving pure facts and quantifiers appear in Figure 5.1. The application of a number of reasoning rules for entailment can be automated by means of a tactic. One such tactic, called `xsimpl`, is illustrated in Section 9.5, and is specified in [Charguéraud, 2020, Appendix G]. 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.

[1] 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 practice, we take FUNCTIONAL-EXTENSIONALITY and PROPOSITIONAL-EXTENSIONALITY as axioms in Coq, then derive PREDICATE-EXTENSIONALITY from these two.

PURE-L
$$\frac{P \Rightarrow (H \vdash H')}{([P] \star H) \vdash H'}$$

EXISTS-L
$$\frac{\forall x.\ (H \vdash H')}{(\exists x.\, H) \vdash H'}$$

FORALL-L
$$\frac{([a/x]\, H) \vdash H'}{(\forall x.\, H) \vdash H'}$$

EXISTS-MONOTONE
$$\frac{\forall x.\ (H \vdash H')}{(\exists x.\, H) \vdash (\exists x.\, H')}$$

PURE-R
$$\frac{(H \vdash H') \qquad P}{H \vdash (H' \star [P])}$$

EXISTS-R
$$\frac{H \vdash ([a/x]\, H')}{H \vdash (\exists x.\, H')}$$

FORALL-R
$$\frac{\forall x.\ (H \vdash H')}{H \vdash (\forall x.\, H')}$$

FORALL-MONOTONE
$$\frac{\forall x.\ (H \vdash H')}{(\forall x.\, H) \vdash (\forall x.\, H')}$$

Figure 5.1: Useful properties for pure facts and quantifiers, with respect to entailment.

5.2 Properties of Separating Conjunction

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 extraction rule STAR-EXISTS is stated using an equality because the entailment relation holds in both directions. On the contrary, the extraction rule STAR-FORALL is stated using a simple entailment relation because the reciprocal entailment does not hold—for a counterexample, consider the case where the type of x is inhabited. The last rule, STAR-MONOTONE-R, describes a monotonicity property; it is explained afterwards.

Lemma 5.2.1 (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:
$$\frac{H_1 \vdash H_1'}{H_1 \star H_2 \vdash H_1' \star H_2}$$

The monotonicity rule STAR-MONOTONE-R 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 (Symmetric version of the monotonicity rule) *The monotonicity rule may be equivalently presented in its symmetric form, stated below.*

$$\frac{H_1 \vdash H_1' \qquad H_2 \vdash H_2'}{H_1 \star H_2 \vdash H_1' \star H_2'}\ \textit{STAR-MONOTONE}$$

5.3 Entailment for Extracting Pure Facts

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 5.3.1 (Properties of the singleton heap predicate)

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

5.4 Entailment between Postconditions

In the imperative λ-calculus considered in this manuscript and formalized further on (Section 6.1), a term evaluates to a value. A postcondition thus describes both an output value and an output state.

Definition 5.4.1 (Type of postconditions) *A postcondition has type: val $\to$ state $\to$ Prop.*

Thereafter, we let Q range over postconditions. To obtain concise statements of the reasoning rules, it is convenient to extend separating conjunction and entailment to operate on postconditions. To that end, we generalize $H \star H'$ and $H \vdash H'$ by introducing the predicate $Q \mathbin{\dot\star} H'$ and the judgment $Q \mathrel{\dot\vdash} Q'$, written with a dot to suggest *pointwise extension*. These two predicates are formalized next.

Definition 5.4.2 (Separating conjunction between a postcondition and a heap predicate)

$$Q \mathbin{\dot\star} H \;\equiv\; \lambda v.\ (Q\, v \star H)$$

This operator appears for example in the statement of the frame rule (recall Section 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 5.4.3 (Entailment between postconditions)

$$Q \mathrel{\dot\vdash} Q' \;\equiv\; \forall v.\ (Q\, v \vdash Q'\, v)$$

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.

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

Chapter 6

Definition of Triples

The definition of triples depends on the details of the programming language. Section 6.1 presents the syntax of embedded programs. Section 6.2 presents the standard big-step semantics. As we argue, the standard big-step semantics is not the most well-suited for capturing safety and termination of nondeterministic programs. Section 6.3 presents a variant of the big-step semantics called the omni-big-step semantics, and Section 6.4 presents its key properties. Section 6.5 presents the definition of triples in terms of the omni-big-step semantics. Section 6.6 presents an alternative definition of triples directly in terms of the big-step semantics, but only applicable to languages that are deterministic (up to the choice of memory allocations).

6.1 Syntax of Embedded Programs

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 term-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 addition, division, and random number generation. The nondeterministic operation rand n, where n is a positive integer, evaluates to any integer in the range $[0, n)$.

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 6.1.1 (Syntax of the language)

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

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. Our Coq formalization actually includes these two constructs explicitly

in the grammar to avoid unnecessary complications associated with the elimination of dummy variables.

Restriction to A-normal form. Although our syntax technically allows for arbitrary terms, for simplicity we assume in this chapter terms to be written in "administrative normal form" (*A-normal form*). In A-normal form, "let $x = t_1$ in t_2" is the sole sequencing construct: no sequencing is implicit in any other construct. For instance, the conditional construct "if t_0 then t_1 else t_2", where t_0 is not a value, must be encoded as "let $x = t_0$ in if x then t_1 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 A-normalization as a preliminary step. Nevertheless, in Section 14.2, we present the *bind* rule, which allows to reason about a subterm in an evaluation context, and thereby handle programs that are not in A-normal form.

Details on the syntax of function definitions. In the grammar of terms, $\mu f.\lambda x.t$ denotes a function definition, where the body t may refer to free variables. In the grammar of values, $\hat{\mu} f.\lambda x.t$ denotes a closure, that is, a closed recursive function, without any free variable. The distinction between functions and closed functions usually does not appear in research papers. It appears, however, naturally in mechanized formalization. We define the type `val` for *closed values* in mutual recursion with the type `trm` for *terms*.

```
Inductive val : Type :=
  | val_int : int → val
  | val_fix : var → var → trm → val
  ...
with trm : Type :=
  | trm_val : val → trm
  | trm_var : var → trm
  | trm_fix : var → var → trm → trm
  ...
```

The fundamental benefit of considering a grammar for *closed* values is that the substitution operation needs not traverse values.

```
Fixpoint subst (y:var) (w:val) (t:trm) : trm :=
  match t with
  | trm_val v ⇒ trm_val v (* no traversal of v *)
  | trm_var x ⇒ if var_eq x y then trm_val w else t
  | trm_fix f x t1 ⇒ trm_fix f x (if var_eq y f || var_eq y x
                                  then t1 else subst y w t1)
  ...
```

If values were allowed to contain free variables, we would indeed save the need to distinguish between $\mu f.\lambda x.t$ and $\hat{\mu} f.\lambda x.t$ (a.k.a. `trm_fix` and `val_fix`). However, we would need to carry around invariants of the form "the function f is closed". To see why, assume that f is a function defined as $\mu f.\lambda x.t$ and for which a specification triple has already been established, and consider the example program let $a = 3$ in $f\,a$. To reason about this program, one needs to reason about the term $([3/a]\,f)\,([3/a]\,a)$. One naturally expects this term to simplify to $f\,3$. Yet, the equality $[3/a]\,f = f$ only holds under the knowledge that f is a closed value.

Checking that f is closed may be easily verified by a syntactic operation, but only if the definition of f is available—this is not the case in the presence of abstraction barriers. For example, if f is a function coming from a module taken as argument of a functor, then the definition of f is not available. In such case, the interface that provides f must be accompanied by a lemma

BIG-VAL
$$\frac{}{v/s \Downarrow v/s}$$

BIG-FIX
$$\frac{}{(\mu f.\lambda x.t)/s \Downarrow (\hat{\mu} f.\lambda x.t)/s}$$

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

BIG-LET
$$\frac{t_1/s \Downarrow v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow v/s''}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \Downarrow v/s''}$$

BIG-IF
$$\frac{\mathsf{If}\ b\ \mathsf{then}\ (t_1/s \Downarrow v'/s')\ \mathsf{else}\ (t_2/s \Downarrow v'/s')}{(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow v'/s'}$$

BIG-REF
$$\frac{p \notin \mathsf{dom}\, s}{(\mathsf{ref}\, v)/s \Downarrow p/(s[p := v])}$$

BIG-FREE
$$\frac{p \in \mathsf{dom}\, s}{(\mathsf{free}\, p)/s \Downarrow tt/(s \setminus p)}$$

BIG-GET
$$\frac{p \in \mathsf{dom}\, s}{(\mathsf{get}\, p)/s \Downarrow (s[p])/s}$$

BIG-SET
$$\frac{p \in \mathsf{dom}\, s}{(\mathsf{set}\, p\, v)/s \Downarrow tt/(s[p := v])}$$

BIG-ADD
$$\frac{}{((+)\, n_1\, n_2)/s \Downarrow (n_1 + n_2)/s}$$

BIG-DIV
$$\frac{n_2 \neq 0}{((\div)\, n_1\, n_2)/s \Downarrow (n_1 \div n_2)/s}$$

BIG-RAND
$$\frac{0 \leq m < n}{(\mathsf{rand}\, n)/s \Downarrow m/s}$$

Figure 6.1: Evaluation rules in big-step style

asserting that f is a closed value. Stating and exploiting such lemmas would induce a significant overhead in practice. We avoid the issue altogether by considering a grammar for closed values, associating to the type `val` the property that its inhabitants are closed values.

A second motivation for closed values is performance of proof-checking. If we do not distinguish between $\mu f.\lambda x.t$ and $\hat{\mu} f.\lambda x.t$, then the syntactic check performed to establish that a function definition is a closed value would need to traversing not only the body of that function, but also the body of all the functions that it refers to. Likewise, the substitution function would need to traverse in depth through all values, and would need to recurse through functions that appear inside those values, and through values and functions that appear inside those functions.

6.2 Standard Big-Step Semantics

Thereafter, we use the meta-variable s to denote a variable of type state 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 may correspond to only a piece of the memory state.

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 6.2.1 (Big-step semantics of the language) *The evaluation rules appear in Figure 6.1.*

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: $\mathsf{dom}\, s$ 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.

$$\textsc{omni-big-val}\quad \frac{(v,s) \in Q}{v/s \Downarrow Q} \qquad \textsc{omni-big-let}\quad \frac{t_1/s \Downarrow Q_1 \qquad (\forall (v',s') \in Q_1.\ ([v'/x]\,t_2)/s' \Downarrow Q)}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \Downarrow Q}$$

$$\textsc{omni-big-app}\quad \frac{v_1 = \mu f.\lambda x.t_1 \qquad ([v_1/f]\,[v_2/x]\,t_1)/s \Downarrow Q}{(v_1\, v_2)/s \Downarrow Q} \qquad \textsc{omni-big-if}\quad \frac{\mathsf{If}\ b\ \mathsf{then}\ (t_1/s \Downarrow Q)\ \mathsf{else}\ (t_2/s \Downarrow Q)}{(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q} \qquad \textsc{omni-big-add}\quad \frac{(n_1 + n_2,\ s) \in Q}{((+)\, n_1\, n_2)/s \Downarrow Q}$$

$$\textsc{omni-big-div}\quad \frac{n_2 \neq 0 \qquad (n_1 \div n_2,\ s) \in Q}{((\div)\, n_1\, n_2)/s \Downarrow Q} \qquad \textsc{omni-big-rand}\quad \frac{n > 0 \qquad (\forall m.\ 0 \le m < n \ \Rightarrow\ (m,s) \in Q)}{(\mathsf{rand}\, n)/s \Downarrow Q}$$

$$\textsc{omni-big-ref}\quad \frac{\forall p \notin \mathsf{dom}\, s.\ (p,\ s[p := v]) \in Q}{(\mathsf{ref}\, v)/s \Downarrow Q} \qquad \textsc{omni-big-free}\quad \frac{p \in \mathsf{dom}\, s \qquad (tt,\ s \setminus p) \in Q}{(\mathsf{free}\, p)/s \Downarrow Q}$$

$$\textsc{omni-big-get}\quad \frac{p \in \mathsf{dom}\, s \qquad (s[p],\ s) \in Q}{(\mathsf{get}\, p)/s \Downarrow Q} \qquad \textsc{omni-big-set}\quad \frac{p \in \mathsf{dom}\, s \qquad (tt,\ s[p := v]) \in Q}{(\mathsf{set}\, p\, v)/s \Downarrow Q}$$

$$\textsc{omni-big-bind}\quad \frac{\neg\, \mathsf{value}\, t \qquad t/s \Downarrow Q_1 \qquad (\forall v s'.\ Q_1\, v\, s' \ \Rightarrow\ E[v]\,/\,s' \Downarrow Q)}{E[t]\,/\,s \Downarrow Q}$$

Figure 6.2: Evaluation rules in omni-big-step style

Observe that the rules of Figure 6.1 define a semantics that is *deterministic up to the choice of memory locations.*

6.3 Definition of Omni-Big-Step Semantics

The omni-big-step semantics judgment is written $t/s \Downarrow Q$. It asserts that all possible evaluations starting from the configuration t/s reach final configurations that belong to the set Q. Whereas the standard big-step judgment $t/s \Downarrow v/s'$ describes the behavior of one possible execution of t/s, the omni-big-step judgment describes the behavior of all possible executions of t/s. Omni-big-step semantics appear to have first appear in Schäfer et al. [2016]. The use of such semantics in the context of Separation Logic appears in Charguéraud et al. [2022].

Definition 6.3.1 (Omni-big-step semantics of the language) *The evaluation rules in Figure 6.2 define the inductive judgment $t/s \Downarrow Q$.*

The omni-big-step rules capture the same semantics as the standard big-step semantics from Figure 6.1. [Charguéraud et al., 2022, §2.2].

The set Q that appears in $t/s \Downarrow Q$ corresponds to an overapproximation of the set of final configurations: it may contain configurations that are not actually reachable by executing t/s. (A discussion of why to consider an overapproximation of the set of results as opposed to an *exact* set of results may be found in [Charguéraud et al., 2022, §2.3].)

The set Q contains pairs made of values and states. Such a set can be described equivalently by a predicate of type "val $\to$ state $\to$ Prop" or by a predicate of type "(val $\times$ state) $\to$ Prop". In the beginning of this chapter, to present definitions in the most idiomatic style, we use set-theoretic notation such as $(v, s) \in Q$ for stating semantics and typing rules. We then use the logic-oriented notation $Q\, v\, s$ for discussing applications of this judgment to program logics.

We next describe the key evaluation rules of Figure 6.2.

The rule for values, OMNI-BIG-VAL, asserts that a final configuration v/s satisfies the postcondition Q if this configuration belongs to the set Q.

The let-binding rule, OMNI-BIG-LET, ensures that all possible evaluations of an expression let $x = t_1$ in t_2 in state s terminate and satisfy the postcondition Q. First, we need all possible evaluations of t_1 to terminate. Let Q_1 denote (an overapproximation of) the set of results that t_1 may reach, as captured by the first premise $t_1/s \Downarrow Q_1$. One can think of Q_1 as the type of t_1, in a very precise type system where any set of values can be treated as a type. The second premise asserts that, for any configuration v'/s' in that set Q_1, we need all possible evaluations of the term $[v'/x]\, t_2$ in state s' to satisfy the postcondition Q. The rule OMNI-BIG-BIND generalizes the let-rule to arbitrary evaluation contexts.

The evaluation rule OMNI-BIG-APP explains how to evaluate a beta-redex by evaluating the result of the relevant substitution. Omnisemantics can be understood as an inductively defined weakest-precondition semantics (or more generally, predicate-transformer semantics) that does not involve invariants for recursion (or loops), but instead uses unrolling rules like in traditional small-step and big-step semantics.

The rule for conditional, OMNI-BIG-IF, is stated using a Coq if-statement, written using a capitalized "If" keyword. Alternatively, it could be stated using the rule OMNI-BIG-IF' shown below, or using the pair of rules OMNI-BIG-IF-TRUE and OMNI-BIG-IF-FALSE.

OMNI-BIG-IF'
$$\frac{(\mathsf{If}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q}{(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q}$$

OMNI-BIG-IF-TRUE
$$\frac{t_1/s \Downarrow Q}{(\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q}$$

OMNI-BIG-IF-FALSE
$$\frac{t_2/s \Downarrow Q}{(\mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q}$$

The evaluation rule OMNI-BIG-ADD for an addition operation is almost like that of a value: it asserts that the evaluation of $(+)\, n_1\, n_2$ in state s satisfies the postcondition Q if the pair $((n_1 + n_2), s)$ belongs to the set Q. The rule OMNI-BIG-DIV is similar, only with an additional premise to disallow division by zero.

The nondeterministic rule OMNI-BIG-RAND is more interesting. The term $\mathsf{rand}\, n$ evaluates safely only if $n > 0$. Under this assumption, its result, named m in the rule, may be any integer in the range $[0, n)$. Thus, to guarantee that every possible evaluation of $\mathsf{rand}\, n$ in a state s produces a result satisfying the postcondition Q, it must be the case that every pair of the form (m, s) with $m \in [0, n)$ belongs to the set Q.

The evaluation rule OMNI-BIG-REF, which describes allocation at a nondeterministically chosen, fresh memory address, follows a similar pattern. For every possible new address p, the pair made of p and the extended state $s[p := v]$ needs to belong to the set Q.

The remaining rules, OMNI-BIG-FREE, OMNI-BIG-GET and OMNI-BIG-SET, are deterministic and follow the same pattern as OMNI-BIG-ADD, only with a side condition $p \in \mathsf{dom}\, s$ to ensure that the address being manipulated does belong to the domain of the current state.

On the one hand, omni-big-step semantics can be viewed as *operational semantics*, because they are not far from traditional operational semantics or executable interpreters. On the other hand, omni-big-step can be viewed as *axiomatic semantics*, because they are not far form reasoning rules. In particular, they directly give a practical, usable definition of a weakest-precondition judgment, which can be used for verifying concrete programs. The fact that they are both closely

related to operational semantics and to axiomatic semantics is precisely the strength of omni-big-step semantics.

6.4 Properties of Omni-Big-Step Semantics

In this section, we discuss key properties of the omni-big-step judgment $t/s \Downarrow Q$ that are involved in the soundness proof of the rules of Separation Logic. For other properties and other applications of omni-semantics, we refer to Charguéraud et al. [2022]. Recall that the metavariable Q denotes an of the set of possible final configurations.

The judgment $t/s \Downarrow Q$ is preserved when the postcondition Q is replaced with a larger set. In other words, the postcondition can always be weakened, like in Hoare logic.

Lemma 6.4.1 (Consequence property for big-step omnisemantics)

$$\frac{t/s \Downarrow Q \qquad Q \subseteq Q'}{t/s \Downarrow Q'} \text{ OMNI-BIG-CONSEQUENCE}$$

A fundamental result is that the omni-big-step judgment inherently satisfies the frame property. The corresponding lemma captures the preservation of the omni-big-step judgment $t/s_1 \Downarrow Q$ when the input state s_1 is extended with a disjoint piece of state s_2.

Lemma 6.4.2 (Frame property for big-step omnisemantics)

$$\frac{t/s_1 \Downarrow Q \qquad s_1 \perp s_2}{t/(s_1 \uplus s_2) \Downarrow (Q \star (\lambda s'.\ s' = s_2))} \text{ OMNI-BIG-FRAME}$$

Proof *The proof is carried out by induction on the omnisemantics judgment. We next show the two most interesting cases of the proof: the treatment of an allocation (4 lines of Coq script) and that of a let-binding (3 lines of Coq script). In each case, we assume* $s_1 \perp s_2$.

Case 1: t *is* $\mathrm{ref}\, v$. *The assumption is* $(\mathrm{ref}\, v)/s_1 \Downarrow Q$. *It is derived by the rule* OMNI-BIG-REF, *whose premise is* $\forall p \notin \mathrm{dom}\, s_1.\ Q\, p\, (s_1[p := v])$. *We need to prove* $(\mathrm{ref}\, v)/(s_1 \uplus s_2) \Downarrow (Q \star (\lambda s'.\ s' = s_2))$. *By* OMNI-BIG-REF, *we need to justify:* $\forall p \notin \mathrm{dom}\, (s_1 \uplus s_2).\ (Q \star (\lambda s'.\ s' = s_2))\, p\, ((s_1 \uplus s_2)[p := v])$. *Consider a location* p *not in* $\mathrm{dom}\, s_1$ *nor in* $\mathrm{dom}\, s_2$. *The predicate* $(Q \star (\lambda s'.\ s' = s_2))\, p$ *is equivalent to* $(Q\, p) \star (\lambda s'.\ s' = s_2)$. *The state update* $(s_1 \uplus s_2)[p := v]$ *is equivalent to* $(s_1[p := v]) \uplus s_2$. *Thus, there remains to prove:* $((Q\, p) \star (\lambda s'.\ s' = s_2))\ ((s_1[p := v]) \uplus s_2)$. *By definition of separating conjunction and exploiting* $(s_1[p := v]) \perp s_2$, *it suffices to show* $Q\, p\, (s_1[p := v])$. *This fact follows from* $\forall p \notin \mathrm{dom}\, s_1.\ Q\, p\, (s_1[p := v])$.

Case 2: t *is "let* $x = t_1$ *in* t_2*". The assumption is* $t/s_1 \Downarrow Q$. *It is derived by the rule* OMNI-BIG-LET, *whose premises are* $t_1/s_1 \Downarrow Q_1$ *and* $\forall v' s'.\ Q_1\, v'\, s' \Rightarrow ([v'/x]\, t_2)/s' \Downarrow Q$. *We need to prove* $(\mathrm{let}\, x = t_1\, \mathrm{in}\, t_2)/(s_1 \uplus s_2) \Downarrow (Q \star (\lambda s'.\ s' = s_2))$. *To that end, we invoke* OMNI-BIG-LET. *For its first premise, we prove* $t_1/(s_1 \uplus s_2) \Downarrow (Q_1 \star (\lambda s'.\ s' = s_2))$ *by exploiting the induction hypothesis applied to* $t_1/s_1 \Downarrow Q_1$. *For the second premise, we have to prove* $\forall v' s''.\ (Q_1 \star (\lambda s'.\ s' = s_2))\, v'\, s'' \Rightarrow ([v'/x]\, t_2)/s'' \Downarrow (Q \star (\lambda s'.\ s' = s_2))$. *Consider a particular* v' *and* s''. *The assumption* $(Q_1 \star (\lambda s'.\ s' = s_2))\, v'\, s''$ *is equivalent to* $((Q_1\, v') \star (\lambda s'.\ s' = s_2))\, s''$. *By definition of separating conjunction, we deduce that* s'' *decomposes as* $s_1' \uplus s_2$, *with* $s_1' \perp s_2$ *and* $Q_1\, v'\, s_1'$, *for some* s_1'. *There remains to prove* $([v'/x]\, t_2)/(s_1' \uplus s_2) \Downarrow (Q \star (\lambda s'.\ s' = s_2))$. *We first exploit* $\forall v' s'.\ Q_1\, v'\, s' \Rightarrow ([v'/x]\, t_2)/s' \Downarrow Q$, *on* $Q_1\, v'\, s_1'$ *to obtain* $([v'/x]\, t_2)/s_1' \Downarrow Q$. *We then conclude by applying the induction hypothesis to the latter judgment.* □

6.5 Separation Logic Triples

Consider a possibly nondeterministic semantics. A total-correctness Hoare triple $\{H\}\ t\ \{Q\}$ asserts that, for any input state s satisfying the precondition H, every possible execution of t/s terminates and satisfies the postcondition Q. This property can be captured using the *inductive* omni-big-step judgment as follows.

Definition 6.5.1 (Separation Logic triples in terms of the omni-big-step judgment)

$$\{H\}\,t\,\{Q\} \quad \equiv \quad \forall s.\ H\,s \ \Rightarrow\ (t/s \Downarrow Q)$$

Note that, reciprocally, an omni-big-step judgment may be interpreted as a particular Hoare triple, featuring a singleton precondition to constrain the input state:

$$(t/s \Downarrow Q) \quad \Longleftrightarrow \quad \{(\lambda s'.\ s' = s)\}\,t\,\{Q\}.$$

6.6 Alternative Definition of Triples for Deterministic Languages

An alternative route to defining Separation Logic triples is to use the technique of the *baked-in frame rule* [Birkedal et al., 2005], for defining Separation Logic triples *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. This definition captures termination: it defines a *total correctness* triple.

Definition 6.6.1 (Total correctness Hoare triple)

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

Such Hoare triples do not yet give Separation Logic reasoning, because they lack support for the frame rule (presented in Section 2.1). Let us see why.

Counterexample (The Hoare triples defined above do not satisfy the frame rule)
The BIG-REF *evaluation rule associated with the definition of the big-step judgment asserts that a term of the form "ref v" may evaluate to any fresh memory location. Thus, we can prove that, starting from an empty heap, the program ref 5 returns a* specific *memory location, say the address number 2. We are therefore able to establish the triple:* $^{\mathit{Hoare}}\{[\,]\}\,(\mathit{ref}\,5)\,\{\lambda p.\ [p = 2] \star (2 \hookrightarrow 5)\}$*, where p denotes the address of the fresh location, specified to be equal to 2.*

To see why the judgment does not satisfy the frame rule, let us attempt to extend the pre- and the postcondition of this triple with the heap predicate $2 \hookrightarrow 6$, which denotes a reference at location 2 storing the value 6. If the frame rule were to hold on Hoare triples, we would be able to derive: $^{\mathit{Hoare}}\{2 \hookrightarrow 6\}\,(\mathit{ref}\,5)\ \{\lambda p.\ [p = 2] \star (2 \hookrightarrow 5) \star (2 \hookrightarrow 6)\}$*. This triple does not hold because, even though the precondition is satisfiable, and even though the program ref 5 evaluates safely, the separating conjunction $(2 \hookrightarrow 5) \star (2 \hookrightarrow 6)$ that appears in the postcondition is equivalent to $[\mathit{False}]$ by the rule* SINGLE-CONFLICT *(Chapter 5). Hence, we derive a contradiction.* □

Whereas a Hoare triple describes the evaluation of a term with respect to the whole memory state, a Separation Logic triple describes 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 6.6.2 (Total correctness Separation Logic triple)

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

To fully grasp the meaning of a Separation Logic triple, it helps to consider an alternative definition expressed directly with respect to the evaluation judgment. This alternative, provably-equivalent definition is shown below. It reads as follows: if the input state 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 6.6.3 (Alternative definition of total correctness Separation Logic triples)

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

The two definitions of triples shown above are appropriate for semantics that are deterministic, or deterministic *up to the choice of memory locations*.

Chapter 7

Reasoning Rules

The reasoning rules of Separation Logic fall in three categories. Section 7.1 presents the structural rules, which do not depend on the details of the language. Section 7.2 presents the reasoning rules for terms: there is one such rule for each term construct of the language. Section 7.3 presents the specification of the primitive operations: there is one such rule for each primitive operation. Section 7.4 details a few proofs.

7.1 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.

Lemma 7.1.1 (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 6.6.2.*

CONSEQUENCE

$$\frac{H \vdash H' \quad \{H'\}\ t\ \{Q'\} \quad Q' \vdash Q}{\{H\}\ t\ \{Q\}}$$

FRAME

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

PROP

$$\frac{P \Rightarrow \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}$$

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 7.1.2 (Combined consequence-frame rule)

$$\frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \star H_2 \vdash Q}{\{H\}\ t\ \{Q\}}\ \textit{CONSEQUENCE-FRAME}$$

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 $Q_1 \star H_2$. The "framed" heap predicate H_2 can be computed as the difference between H and H_1. In practice, though, rather than trying to instantiate H_2 in the consequence-frame rule, it may be more effective to exploit the *ramified frame rule* presented further on (Section 10.4).

7.2 Rules for Terms

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

Lemma 7.2.1 (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 6.6.2.*

$$\frac{H \vdash (Q\,v)}{\{H\}\ v\ \{Q\}}\ \textit{VAL} \qquad \frac{H \vdash (Q\,(\hat{\mu}f.\lambda x.t))}{\{H\}\ (\mu f.\lambda x.t)\ \{Q\}}\ \textit{FIX} \qquad \frac{v_1 = \hat{\mu}f.\lambda x.t \quad \{H\}\ ([v_2/x]\,[v_1/f]\,t)\ \{Q\}}{\{H\}\ (v_1\,v_2)\ \{Q\}}\ \textit{APP}$$

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

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

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 7.2.2 (Small-footprint reasoning rules for values)

$$\frac{}{\{\,[\,]\,\}\ v\ \{\lambda r.\,[r = v]\}}\ \textit{VAL'} \qquad \frac{}{\{\,[\,]\,\}\ (\mu f.\lambda x.t)\ \{\lambda r.\,[r = (\hat{\mu}f.\lambda x.t)]\}}\ \textit{FIX'}$$

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 "$\textsf{let}\,x\ =\ t_1\,\textsf{in}\,t_2$". 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 "let $x = t_1$ 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.

7.3 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 (Section 2.5 and Section 3.3). The arithmetic operations admit specifications that involve only empty heaps.

Lemma 7.3.1 (Specification for primitive operations)

$$
\begin{array}{lllll}
\textit{REF:} & & \{[\,]\} & (\texttt{ref}\,v) & \{\lambda r.\ \exists p.\ [r = p] \star (p \hookrightarrow v)\} \\
\textit{GET:} & & \{p \hookrightarrow v\} & (\texttt{get}\,p) & \{\lambda r.\ [r = v] \star (p \hookrightarrow v)\} \\
\textit{SET:} & & \{p \hookrightarrow v\} & (\texttt{set}\,p\,v') & \{\lambda \_.\ (p \hookrightarrow v')\} \\
\textit{FREE:} & & \{p \hookrightarrow v\} & (\texttt{free}\,p) & \{\lambda \_.\ [\,]\} \\
\textit{ADD:} & & \{[\,]\} & ((+)\,n_1\,n_2) & \{\lambda r.\ [r = n_1 + n_2]\} \\
\textit{DIV:} & n_2 \neq 0 \ \Rightarrow & \{[\,]\} & ((\div)\,n_1\,n_2) & \{\lambda r.\ [r = n_1 \div n_2]\}
\end{array}
$$

7.4 Proofs of Reasoning Rules

Consider, e.g., a let-binding. Compare the OMNI-BIG-LET rule with the corresponding Separation Logic rule.

OMNI-BIG-LET

$$\frac{t_1/s \Downarrow Q_1 \qquad \big(\forall v' s'.\ Q_1\,v'\,s' \ \Rightarrow\ ([v'/x]\,t_2)/s' \Downarrow Q\big)}{(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/s \Downarrow Q}$$

LET

$$\frac{\{H\}\ t_1\ \{Q_1\} \qquad \big(\forall v'.\ \{Q_1\,v'\}\ ([v'/x]\,t_2)\ \{Q\}\big)}{\{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}}$$

The only difference between the two rules is that the first one considers one specific state s, whereas the second rule considers a set of possible states satisfying the precondition H. To prove the LET rule, we first unfold the definition of $\{H\}\ t\ \{Q\}$ as $\forall s.\ H\,s \ \Rightarrow\ (t/s \Downarrow Q)$. We then consider a particular state s, and apply the rule OMNI-BIG-LET for that state. The two premises of OMNI-BIG-LET are justified by applying each of the two premises of the LET rule. The corresponding Coq proof script witnesses the simplicity of this proof: "`intros. eapply mbig_let; eauto.`"

Chapter 8

Weakest-Precondition Style

8.1 Semantic Weakest Precondition

The notion of weakest precondition has been used pervasively in the development of automated tools based on Hoare logic. Work on the Iris framework [Jung et al., 2015] has shown that this notion also helps to streamline the set-up of interactive tools based on Separation Logic.

The *semantic weakest precondition* of a term t with respect to a postcondition Q denotes a heap predicate, written $\mathrm{wp}\, t\, Q$, 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 5.1.1). 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 8.1.1 (Semantic weakest precondition) *The predicate wp is equivalently characterized by:*

1. $\mathsf{wp}\, t\, Q \ \equiv\ \min_{(\vdash)} \{\ H \mid \{H\}\ t\ \{Q\}\ \}$
2. $\big(\{\mathsf{wp}\, t\, Q\}\ t\ \{Q\}\big) \ \wedge\ \big(\forall H.\ \{H\}\ t\ \{Q\} \ \Rightarrow\ H \vdash \mathsf{wp}\, t\, Q\big)$
3. $\mathsf{wp}\, t\, Q \ \equiv\ \lambda h.\ \big(\{\lambda h'.\, h' = h\}\ t\ \{Q\}\big)$
4. $\mathsf{wp}\, t\, Q \ \equiv\ \exists H.\ H \star \big[\{H\}\ t\ \{Q\}\big]$
5. $H \vdash \mathsf{wp}\, t\, Q \ \Leftrightarrow\ \{H\}\ t\ \{Q\}$
6. $\mathsf{wp}\, t\, Q \ \equiv\ \lambda h.\ (t/s \Downarrow Q)$

The first 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 second characterization consists of a reformulation of the first characterization in terms of basic logic operators.

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 $H \vdash \mathrm{wp}\, t\, Q$.

The sixth characterization defines $\mathsf{wp}\, t\, Q$ directly in terms of the omni-big-step judgment, bypassing the notion of triple.

The developer of a practical tool based on Separation Logic may choose to take either triples or weakest-preconditions as a primitive notion; the other notion may then be derived in terms of that primitive notion. One may define wp using characterization (6), then derive the notion of triple using equivalence (5) in the right-to-left direction. Reciprocally, assuming a definition of triples, one may derive wp. The choice of the encoding depends on the strength of the host logic with respect to existential quantification. Definition (3) makes weaker assumptions, but require reasoning at the level of heaps. Definition (4) leverages the ability to existentially quantify over heap predicates, allowing for higher-level reasoning.

8.2 WP-Style Structural Rules

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

Lemma 8.2.1 (Structural rules in weakest-precondition style)

$$\frac{Q \mathrel{\dot{\vdash}} Q'}{\mathsf{wp}\, t\, Q \vdash \mathsf{wp}\, t\, Q'}\ \textit{WP-CONSEQUENCE} \qquad \frac{}{(\mathsf{wp}\, t\, Q) \mathbin{\dot{\star}} H \vdash \mathsf{wp}\, t\, (Q \mathbin{\dot{\star}} H)}\ \textit{WP-FRAME}$$

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 two structural rules may be combined into a single rule, called WP-RAMIFIED-FRAME. This rule alone suffices to capture all the structural properties of Separation Logic.

Lemma 8.2.2 (Ramified frame rule in weakest-precondition style)

$$\frac{}{(\mathsf{wp}\, t\, Q) \mathbin{\dot{\star}} (Q \mathbin{\dot{-\!\!\star}} Q') \vdash (\mathsf{wp}\, t\, Q')}\ \textit{WP-RAMIFIED-FRAME}$$

8.3 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.

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

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 8.3.1 (Reasoning rules for terms in weakest-precondition style)

$$\frac{}{Q\, v \vdash \mathsf{wp}\, v\, Q}\ \textit{WP-VAL} \qquad \frac{}{Q\, (\hat{\mu} f.\lambda x.t) \vdash \mathsf{wp}\, (\mu f.\lambda x.t)\, Q}\ \textit{WP-FIX} \qquad \frac{v_1 = \hat{\mu} f.\lambda x.t}{\mathsf{wp}\, ([v_2/x]\, [v_1/f]\, t)\, Q \vdash \mathsf{wp}\, (v_1\, v_2)\, Q}\ \textit{WP-APP}$$

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

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

8.4 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 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 fresh address p is quantified universally in the left-hand side of the entailment.

Lemma 8.4.1 (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 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 8.4.2 (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.*

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

Stating specifications in weakest-precondition style is not at all mandatory for working with reasoning rules in weakest-precondition style. Indeed, as we do in CFML, it is possible to continue stating specifications using conventional triples, which are more intuitive to read. In that setting (presented in Section 9.6), we exploit the following rule for reasoning about every function call.

Lemma 8.4.3 (Variant of the ramified frame rule for proof obligations in wp style)

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

Chapter 9

Characteristic Formulae

This chapter describes the technique of *characteristic formulae* for smoothly integrating Separation Logic in an interactive proof assistant. Section 9.1 explains the concept of characteristic formula, and in particular how it relates to the concept of weakest precondition. Section 9.2 and Section 9.3 show how to define the characteristic formulae generator as a function that effectively computes within Coq. Section 9.4 presents the soundness proof. Section 9.5 and Section 9.6 describe the set-up used for carrying interactive program verification in practice using characteristic formulae, through the use of tactics that lead to concise proof scripts.

9.1 Principle of Characteristic Formulae

Recall from the previous chapter that the predicate $\mathsf{wp}\, t\, Q$ describes the *weakest precondition* of a term t with respect to a postcondition Q. This predicate satisfies the equivalence $\big(H \vdash \mathsf{wp}\, t\, Q\big) \Leftrightarrow \{H\}\ t\ \{Q\}$. It comes with a number of reasoning rules, such as WP-LET, which is expressed as the entailment: $\mathsf{wp}\, t_1\, (\lambda v.\ \mathsf{wp}\, ([v/x]\, t_2)\, Q) \vdash \mathsf{wp}\, (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\, Q$. The predicate wp can be defined in numerous ways, but ultimately all definitions refer to the inductively defined semantics of the programming language.

This chapter presents a function that *effectively computes* the weakest precondition of a term. This function, called cf, is defined by recursion over the *syntax* of the source term. In the particular case where cf reaches a function application, the formula that it produces simply refers to the weakest precondition (wp) associated with that application. Compared with a *weakest-precondition calculus*, as found typically in Hoare-logic based tools that rely on automated solvers for discharging proof obligations, the main difference is that cf operates on a raw source term, without requiring any specification or invariant to accompany the term. One may thus view a computation of the characteristic formula as a *most general weakest-precondition calculus*.

The central theorem of this chapter establishes $\mathsf{cf}\, t\, Q \vdash \mathsf{wp}\, t\, Q$. Exploiting this entailment allows to establish the specification of a function by following the structure of the logical formula produced by cf. In particular, to establish that a function satisfies a given specification, one can apply the rule CF-TRIPLE-FIX shown below. This rule reveals the characteristic formula associated with the body the function instantiated on the argument provided to the function. This rule is a corollary of the reasoning rule APP and of the entailment $\mathsf{cf}\, t\, Q \vdash \mathsf{wp}\, t\, Q$.

$$\frac{v_1 = \hat{\mu} f.\lambda x.t \qquad f \neq x \qquad H \vdash \mathsf{cf}\, ([v_2/x]\, [v_1/f]\, t)\, Q}{\{H\}\ (v_1\, v_2)\ \{Q\}}\ \textsc{cf-triple-fix}$$

Compared with carrying out proofs using wp directly, the added value of characteristic formulae is three-fold.

First, the characteristic formula function cf produces a logical formula that no longer refer to the deeply-embedded syntax of the term t. All variables appear as logical variables, that is, Coq variables. Furthermore, there is no need to simplify substitutions expressed on the deep embedding, such as $[v/x]\,t_2$ in the rule WP-LET.

Second, the characteristic formula in some sense *pre-applies* all the reasoning rules of the program logic. For example, when processing a let-binding, there is no need to apply the lemma WP-LET, because this lemma is somehow already exploited as part of the statement of the characteristic formula. The only bookkeeping work that remains to make progress through a let-binding is to instantiate an existential quantifier and split a conjunction. These two benefits should appear more clearly when we present examples further on.

Third, characteristic formulae enable the introduction of the *lifting* technique described in the next chapter. This technique allows describing program values directly using Coq values. In particular, constructors of OCaml algebraic data types may be represented using corresponding Coq inductive constructors. Among other benefits, the lifting techniques leads to considerable simplifications in the logical formulae produced for reasoning about pattern matching.

9.2 Building a Characteristic Formulae Generator, Step by Step

We next describe a 6-step process that leads to the definition of the function cf. For simplicity, we assume the term t to be in A-normal form, meaning that arguments of functions and conditionals are expected to be either variables or values. The generalization beyond A-normal form makes the definitions slightly more technical, so we do not present it here. Such a generalization may be found in the implementation of CFML.

Step 0: Weakest-precondition style reasoning rules. We start from the wp-style reasoning rules (Section 8.3), which we reproduce below for convenience. In the rule WP-LET that handles a term of the form $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$, the variable named X has type val and corresponds to the value produced by t_1. The substitution $[X/x]\,t_2$ replaces the program variable x (represented as a string) with an abstract value represented by the Coq variable X.

WP-VAL:	$Q\,v \;\vdash\; \mathsf{wp}\,v\,Q$	
WP-FIX:	$Q\,(\hat{\mu} f.\lambda x.t) \;\vdash\; \mathsf{wp}\,(\mu f.\lambda x.t)\,Q$	
WP-APP:	$\mathsf{wp}\,([v_2/x]\,[v_1/f]\,t)\,Q \;\vdash\; \mathsf{wp}\,(v_1\,v_2)\,Q$	where $v_1 = \hat{\mu} f.\lambda x.t$
WP-LET:	$\mathsf{wp}\,t_1\,(\lambda X.\ \mathsf{wp}\,([X/x]\,t_2)\,Q) \;\vdash\; \mathsf{wp}\,(\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2)\,Q$	
WP-IF:	$\mathsf{If}\, b\, \mathsf{then}\, (\mathsf{wp}\,t_1\,Q)\, \mathsf{else}\, (\mathsf{wp}\,t_2\,Q) \;\vdash\; \mathsf{wp}\,(\mathsf{if}\, b\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\,Q$	
WP-FRAME:	$(\mathsf{wp}\,t\,Q) \star (Q \mathrel{-\!\!\star} Q') \;\vdash\; (\mathsf{wp}\,t\,Q')$	

Step 1: Recursion over the syntax. We consider a first version of $\mathsf{cf}\,t\,Q$, defined by recursion over its argument t. For all term constructs except applications, we mimic the weakest-precondition rule. For a function application, we simply refer to the wp judgment for that application. Indeed, we do not have at hand the specification of the function being called. In the case of a conditional, we need to existentially quantify over the boolean value b that corresponds to the argument of the conditional, because in the original program that argument could be a variable

and not a boolean value. Support for the frame rule will be added later on, at step 5.

$$\begin{array}{lcl}
\mathsf{cf}\, v\, Q & \equiv & Q\, v \\
\mathsf{cf}\, (\mu f.\lambda x.t)\, Q & \equiv & Q\, (\hat{\mu} f.\lambda x.t) \\
\mathsf{cf}\, (t_1\, t_2)\, Q & \equiv & \mathsf{wp}\, (t_1\, t_2)\, Q \\
\mathsf{cf}\, (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\, Q & \equiv & \mathsf{cf}\, t_1\, \big(\lambda X.\ \mathsf{cf}\, ([X/x]\, t_2)\, Q\big) \\
\mathsf{cf}\, (\mathsf{if}\, t_0\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\, Q & \equiv & \exists (b : \mathsf{bool}).\ [t_0 = b] \star \mathsf{If}\, b\, \mathsf{then}\, (\mathsf{cf}\, t_1\, Q)\, \mathsf{else}\, (\mathsf{cf}\, t_2\, Q) \\
\mathsf{cf}\, x\, Q & \equiv & \bot
\end{array}$$

On the last line above, $\mathsf{cf}\, x\, Q$ is defined as the always-false assertion. Indeed, variables should all have been removed via the substitutions performed when traversing let-bindings. If the computation of cf reaches a free variable, it means that this variable was a dangling (unbound) free variable of the original input program. A dangling free variable is a stuck term in the semantics, hence its weakest precondition is the false predicate.

Step 2: Refinement for local functions. Let us refine the definition of characteristic formulae for local functions. Consider a local function definition of the form $\mu f.\lambda x.t$. The formula $Q\, (\hat{\mu} f.\lambda x.t)$ is sound and complete: it enables the user to state and prove property about that function, by exploiting its syntactic definition. Yet, when working with characteristic formulae, we would like to never manipulate program syntax. Instead, we would like to obtain a logical formula that enables the user to reason about the extensional behavior of the function. Such a behavior can be achieved by leveraging the characteristic formula recursively computed for the body of that function. The relevant definition is shown below and explained next.

$$\mathsf{cf}\, (\mu f.\lambda x.t)\, Q \quad \equiv \quad \forall (F : \mathsf{val}).\ [\forall X Q'.\ \mathsf{cf}\, ([X/x]\, [F/f]\, t)\, Q' \vdash \mathsf{wp}\, (F\, X)\, Q'] \mathrel{-\!\!\star} Q\, F$$

The universally quantified variable F denotes the value $\hat{\mu} f.\lambda x.t$ that corresponds to the function closure. Yet this information is not revealed. What is provided is an assumption that may be exploited to establish properties about calls of the form $F\, X$. This assumption asserts that, for any argument X, to establish that the application $F\, X$ admits a particular behavior described by a postcondition Q', one has to show that the term $[X/x]\, [F/f]\, t$ admits the same behavior.

An equivalent formulation of $\mathsf{cf}\, (\mu f.\lambda x.t)\, Q$, slightly more convenient when specifying functions using triples, is shown below. It specifies the application $F\, X$ using a triple, and involves a heap predicate H to denote the precondition of that application.

$$\begin{array}{l}
\mathsf{cf}\, (\mu f.\lambda x.t)\, Q \ \equiv \\
\quad \mathsf{framed}\, (\lambda Q.\, \forall F.\ [\forall X H Q'.\ H \vdash \mathsf{cf}_{(f,F)::(x,X)::E}\, t\, Q' \ \Rightarrow\ \{H\}\ (F\, X)\ \{Q'\}] \mathrel{-\!\!\star} Q\, F)
\end{array}$$

Step 3: Obtaining a structural recursion. The recursive function cf defined at steps 1 and 2 is not structurally recursive. Indeed, in the processing of $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$, the second recursive call is not performed on t_2 but on $[X/x]\, t_2$. The function cf does terminate on all input, because the substitution involved replaces variables not with arbitrary terms but with values. These values are handled at the base case ($\mathsf{cf}\, v\, Q \equiv Q\, v$), where no recursive call is involved. To simplify the Coq formalization of the function cf, we are going to recast the function in a way that makes it structurally recursive.

We introduce an environment, written E, to keep track of the *delayed* substitutions. This environment plays the same role as a typing environment in a type-checker, except that it binds a program variable not to its type but to its corresponding Coq variable. Concretely, we define a function of the form $\mathsf{cf}_E\, t\, Q$. For simplicity, we represent E as an association list binding values

to variables. We note, however, that an appropriate tree data structure (e.g., a Patricia tree) could improve performance.

The definition of the structurally recursive function cf is shown below. The context E gets extended in the let-binding case. When the function reaches a free variable x, it performs a lookup for this variable in the environment E, using the operation written $E[x]$. Besides, observe that the definition of $\mathsf{cf}_E\, v\, Q$ does not involve any substitution, because values in our language are always closed value.

$$
\begin{array}{lcl}
\mathsf{cf}_E\, x\, Q & \equiv & \mathsf{If}\ (x \in \mathsf{dom}\ E)\ \mathsf{then}\ Q\,(E[x])\ \mathsf{else}\ \bot \\
\mathsf{cf}_E\, v\, Q & \equiv & Q\, v \\
\mathsf{cf}_E\, (\mu f.\lambda x.t)\, Q & \equiv & \forall F.\ [\forall X Q'.\ \mathsf{cf}_{(f,F)::(x,X)::E}\, t\, Q' \vdash \mathsf{wp}\,(F\ X)\, Q'] \mathrel{-\!\!\star} Q\, F \\
\mathsf{cf}_E\, (t_1\, t_2)\, Q & \equiv & \mathsf{wp}\,(\mathsf{subst}\ E\ (t_1\, t_2))\, Q \\
\mathsf{cf}_E\, (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)\, Q & \equiv & \mathsf{cf}_E\, t_1\, \big(\lambda X.\ \mathsf{cf}_{(x,X)::E}\, t_2\, Q\big) \\
\mathsf{cf}_E\, (\mathsf{if}\ t_0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\, Q & \equiv & \exists (b : \mathsf{bool}).\ [t_0 = b] \star \mathsf{If}\ b\ \mathsf{then}\ (\mathsf{cf}_E\, t_1 Q)\ \mathsf{else}\ (\mathsf{cf}_E\, t_2 Q)
\end{array}
$$

To invoke the characteristic formulae generator on a closed program, we let $\mathsf{cf}\, t\, Q \equiv \mathsf{cf}_{\mathsf{nil}}\, t\, Q$.

Step 4: Reformulation as a function that does not depend on the postcondition. For reasons that will only appear clear in the following steps, we next swap the place where Q is taken as an argument with the place where the pattern matching on t occurs. In other words, we define the recursive function $\mathsf{cf}_E\, t$, whose output is a function that expects a postcondition Q as argument. The function $\mathsf{cf}_E\, t$, reformulated below, admits the type: $(\mathsf{val} \to \mathsf{Hprop}) \to \mathsf{Hprop}$.

$$
\begin{array}{lcl}
\mathsf{cf}_E\, x & \equiv & \lambda Q.\, \mathsf{If}\ (x \in \mathsf{dom}\ E)\ \mathsf{then}\ Q\,(E[x])\ \mathsf{else}\ \bot \\
\mathsf{cf}_E\, v & \equiv & \lambda Q.\, Q\, v \\
\mathsf{cf}_E\, (\mu f.\lambda x.t) & \equiv & \lambda Q.\, \forall F.\ [\forall X Q'.\ \mathsf{cf}_{(f,F)::(x,X)::E}\, t\, Q' \vdash \mathsf{wp}\,(F\ X)\, Q'] \mathrel{-\!\!\star} Q\, F \\
\mathsf{cf}_E\, (t_1\, t_2) & \equiv & \lambda Q.\, \mathsf{wp}\,(\mathsf{subst}\ E\ (t_1\, t_2))\, Q \\
\mathsf{cf}_E\, (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) & \equiv & \lambda Q.\, \mathsf{cf}_E\, t_1 \big(\lambda X.\ \mathsf{cf}_{(x,X)::E}\, t_2\, Q\big) \\
\mathsf{cf}_E\, (\mathsf{if}\ t_0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & \equiv & \lambda Q.\, \exists (b : \mathsf{bool}).\ [t_0 = b] \star \mathsf{If}\ b\ \mathsf{then}\ (\mathsf{cf}_E\, t_1 Q)\ \mathsf{else}\ (\mathsf{cf}_E\, t_2 Q)
\end{array}
$$

Step 5: Adding support for the frame rule. The frame rule $(\mathsf{wp}\, t\, Q) \star (Q \mathrel{-\!\!\star} Q') \vdash (\mathsf{wp}\, t\, Q')$ is not syntax directed. Thus, we do not know, a priori, where in a proof the user may wish to exploit this rule. Our approach to handling structural rules is to introduce a *predicate transformer*, written framed $\mathcal{F}$, at every node of the characteristic formula. For example:

$$
\mathsf{cf}_E\, (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) \quad \equiv \quad \mathsf{framed}\ (\lambda Q.\, \mathsf{cf}_E\, t_1 \big(\lambda X.\ \mathsf{cf}_{(x,X)::E}\, t_2\, Q\big))
$$

We will come back to the definition and properties of "framed" in Section 9.3. Suffices to know at this point that: (1) if needed, this predicate can be exploited to mimic the frame rule; (2) if not needed, this predicate can be discarded, before pursuing through the remaining of the formula at hand.

The idea of applying a predicate transformer at every node of the characteristic formula originates from [Charguéraud, 2010]. Yet, the characteristic formulae presented here operate on weakest-precondition style predicates, thus the predicate framed used here admits a totally different shape than in the prior work on characteristic formulae.

Step 6: Introduction of auxiliary definitions. We introduce one auxiliary definitions per term construct. Their purpose is to improve the readability of the output of calls to cf by means of a set of custom notation, and to ease the statement of the lemmas that contribute to the soundness

proof. In the definitions shown below, the meta-variable $\mathcal{F}$ denotes a formula of type (val $\to$ Hprop) $\to$ Hprop, and $\mathcal{G}$ denotes a formula that depends on one or several arguments of type val.

Definition 9.2.1 (Auxiliary definitions for the characteristic formulae)

$$
\begin{array}{lcl}
\mathit{cf\_fail} & \equiv & \mathit{framed}\,(\lambda Q.\ \bot) \\
\mathit{cf\_val}\,v & \equiv & \mathit{framed}\,(\lambda Q.\ Q\,v) \\
\mathit{cf\_var}\,E\,x & \equiv & \mathit{framed}\,(\lambda Q.\ \mathit{If}\,(x \in \mathit{dom}\,E)\ \mathit{then}\ \mathit{cf\_val}\,(E[x])\ \mathit{else}\ \mathit{cf\_fail}) \\
\mathit{cf\_app}\,t & \equiv & \mathit{framed}\,(\lambda Q.\ \mathit{wp}\,t\,Q) \\
\mathit{cf\_fix}\,\mathcal{G} & \equiv & \mathit{framed}\,(\lambda Q.\ \forall F.\ [\forall X Q'.\ \mathcal{G}\,F\,X\,Q' \vdash \mathit{wp}\,(F\,X)\,Q'] \mathrel{-\!\!\star} Q\,F) \\
\mathit{cf\_let}\,\mathcal{F}_1\,\mathcal{G}_2 & \equiv & \mathit{framed}\,(\lambda Q.\ \mathcal{F}_1\,(\lambda X.\ \mathcal{G}_2\,v\,Q)) \\
\mathit{cf\_if}\,t_0\,\mathcal{F}_1\,\mathcal{F}_2 & \equiv & \mathit{framed}\,(\lambda Q.\ \exists(b : \mathit{bool}).\ [t_0 = b] \star \mathit{If}\,b\ \mathit{then}\ \mathcal{F}_1\,Q\ \mathit{else}\ \mathcal{F}_2\,Q)
\end{array}
$$

For example, we can now define: "$\mathrm{cf}_E\,(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)$" as "$\mathrm{cf\_let}\,(\mathrm{cf}_E\,t_1)\,\big(\lambda X.\ \mathrm{cf}_{(x,X)::E}\,t_2\big)$". Furthermore, we introduce the Coq syntax "`Let x := F1 in F2`" for the predicate "$\mathrm{cf\_let}\,\mathcal{F}_1\,\mathcal{G}_2$". As a result, the characteristic formula of a term of the form "$\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$" is displayed to the user in the form "`Let X := F1 in F2`". In other words, as we will illustrate further on (Section 9.5), the display of characteristic formulae gives the user the illusion of reading source code, even though in fact what is being manipulated is not a piece of program syntax but instead a Coq logical formula.

The definition of the characteristic formulae generator in terms of the auxiliary definitions is as follows.

Definition 9.2.2 (Characteristic formulae generator) $\mathit{cf}\,t\,Q \equiv \mathit{cf}_{\mathit{nil}}\,t\,Q$ *with*

$$
\begin{array}{lcl}
\mathit{cf}_E\,x & \equiv & \mathit{cf\_var}\,E\,x \\
\mathit{cf}_E\,v & \equiv & \mathit{cf\_val}\,v \\
\mathit{cf}_E\,(\mu f.\lambda x.t) & \equiv & \mathit{cf\_fix}\,(\lambda F X.\ \mathit{cf}_{(f,F)::(x,X)::E}\,t) \\
\mathit{cf}_E\,(t_1\,t_2) & \equiv & \mathit{cf\_app}\ (\mathit{subst}\,E\,(t_1\,t_2)) \\
\mathit{cf}_E\,(\mathit{let}\,x = t_1\ \mathit{in}\,t_2) & \equiv & \mathit{cf\_let}\,(\mathit{cf}_E\,t_1)\,(\lambda X.\ \mathit{cf}_{(x,X)::E}\,t_2) \\
\mathit{cf}_E\,(\mathit{if}\,t_0\ \mathit{then}\,t_1\ \mathit{else}\,t_2) & \equiv & \mathit{cf\_if}\,(\mathit{subst}\,E\,t_0)\,(\mathit{cf}_E\,t_1)\,(\mathit{cf}_E\,t_2)
\end{array}
$$

9.3 Properties and Definition of the "framed" Predicate

As said earlier, the purpose of the predicate framed is to provide the user with the possibility to access the expressiveness of the frame rule while carrying out proofs via characteristic formulae. Proof obligations take the form $H \vdash \mathsf{framed}\,\mathcal{F}\,Q$, where $\mathcal{F}$ is an application of an auxiliary definition such as cf_let, or an application of wp. On such proof obligations, we wish to exploit the following reasoning rules, which mimic the application of consequence and frame on triples, and to be able to eliminate the transformer "framed" when it is not needed.

CF-FRAMED-CONSEQ
$$\frac{H \vdash \mathsf{framed}\,\mathcal{F}\,Q \qquad Q \vdash Q'}{H \vdash \mathsf{framed}\,\mathcal{F}\,Q'}$$

CF-FRAMED-FRAME
$$\frac{H \vdash \mathsf{framed}\,\mathcal{F}\,Q}{H \star H' \vdash \mathsf{framed}\,\mathcal{F}\,(Q \mathbin{\ast\!\!-} H')}$$

CF-FRAMED-ERASE
$$\frac{H \vdash \mathcal{F}\,Q}{H \vdash \mathsf{framed}\,\mathcal{F}\,Q}$$

There remains to exhibit a definition of "framed" that satisfies the above rules. Recall that the frame and consequence rules are subsumed by the rule WP-FRAME. This rule asserts that the assertion $\mathsf{wp}\,t\,Q$ is entailed by the assertion $\exists Q'.\,(\mathsf{wp}\,t\,Q') \star (Q' \mathrel{-\!\!\star} Q)$. We can mimic this definition by *defining* the assertion "framed $\mathcal{F}\,Q$" as "$\exists Q'.\,(\mathcal{F}\,Q') \star (Q' \mathrel{-\!\!\star} Q)$".

Furthermore, to account for the fact that we aim for a program logic in which certain heap predicates can be considered *affine* as opposed to *linear*, we include an affine-top predicate in the definition of "framed", in a way reminiscent of the rule WP-RAMIFIED-FRAME-ATOP (Section 11.2).

Definition 9.3.1 (Predicate "framed")

$$\mathit{framed}\,\mathcal{F} \quad \equiv \quad \lambda Q.\ \ \exists Q'.\ (\mathcal{F}\,Q') \star (Q' \mathrel{-\!\!\star} (Q \star \top))$$

The key properties of this predicate appear below. The three first properties justify the three reasoning rules stated above. The next two properties are useful in the soundness proof: FRAMED-MONO asserts that framed is covariant in the formula it applies to; FRAMED-WP asserts that framed does not add to the expressiveness of a weakest-precondition formula $\mathsf{wp}\,t$, because such a formula already supports consequence-frame reasoning. The last property FRAMED-IDEM, is a sanity check. It shows that two nested applications of the framed predicate are redundant; it is reminiscent of the fact that two applications of the frame rule (or of the consequence rule) can always be merged into a single application of that rule.

Definition 9.3.2 (Properties of the predicate "framed")

$$\begin{array}{ll}
\textsc{framed-conseq:} & Q \vdash Q' \ \Rightarrow\ \mathit{framed}\,\mathcal{F}\,Q \vdash \mathit{framed}\,\mathcal{F}\,Q' \\
\textsc{framed-frame:} & (\mathit{framed}\,\mathcal{F}\,Q) \star H \ \vdash\ \mathit{framed}\,\mathcal{F}(Q \star H) \\
\textsc{framed-erase:} & \mathcal{F}\,Q \ \vdash\ \mathit{framed}\,\mathcal{F}\,Q \\
\\
\textsc{framed-mono:} & (\forall Q.\ \mathcal{F}\,Q \ \vdash\ \mathcal{F}'\,Q) \ \Rightarrow\ \mathit{framed}\,\mathcal{F}\,Q \vdash \mathit{framed}\,\mathcal{F}'\,Q \\
\textsc{framed-wp:} & \mathit{framed}\,(\mathit{wp}\,t) \ =\ \mathit{wp}\,t \\
\textsc{framed-idem:} & \mathit{framed}\,(\mathit{framed}\,\mathcal{F}) \ =\ \mathit{framed}\,\mathcal{F}
\end{array}$$

9.4 Soundness of Characteristic Formulae

As announced earlier, our aim is to prove $\mathsf{cf}\,t\,Q \ \vdash\ \mathsf{wp}\,t\,Q$. Recall that $\mathsf{cf}\,t\,Q \ \equiv\ \mathsf{cf}_{\mathsf{nil}}\,t\,Q$ where the recursive function has the form $\mathsf{cf}_E\,t$, for an environment E. A key insight is that the formula computed by $\mathsf{cf}_E\,t\,Q$ is equivalent to the one computed by $\mathsf{cf}\,(\mathsf{subst}\,E\,t)\,Q$, where $\mathsf{subst}\,E\,t$ corresponds to the term t in which all bindings from E have been substituted. We define the iterated substitution operation subst as a recursive function over the term t, for efficiency reasons. Alternatively, this operation can be defined by recursion over the environment E as follows.

$$\begin{array}{lcl}
\mathsf{subst}\,\mathsf{nil}\,t & \equiv & t \\
\mathsf{subst}\,((x,v) :: E)\,t & \equiv & \mathsf{subst}\,E\,([v/x]\,t)
\end{array}$$

In fact, our soundness proof exploits the fact that these two definitions of subst are equivalent.

Our soundness proof establishes the following result:

$$\mathsf{cf}_E\,t\,Q \vdash \mathsf{wp}\,(\mathsf{subst}\,E\,t)\,Q.$$

The proof is by structural induction on the term t. To ease the statement of the lemmas involved in the soundness proof, we introduce an auxiliary judgment to reformulate the proof obligations. This judgment, written "$\mathsf{sound}\,t\,\mathcal{F}$", asserts that $\mathcal{F}$ is a logical formula stronger than the weakest-precondition of t.

Definition 9.4.1 (Auxiliary soundness judgment)

$$\mathit{sound}\,t\,\mathcal{F} \quad \equiv \quad \forall Q.\ \mathcal{F}\,Q \ \vdash\ \mathit{wp}\,t\,Q$$

Using this judgment, the proposition $\mathsf{cf}_E\,t\,Q \vdash \mathsf{wp}\,(\mathsf{subst}\,E\,t)\,Q$ reformulates as shown below.

Lemma 9.4.1 (Statement of the induction principle for the soundness proof) *We prove:*

$$\forall t E.\ \ \mathit{sound}\,(\mathit{subst}\,E\,t)\,(\mathit{cf}_E\,t)$$

The proof is by induction on t. We next list the key lemmas involved in the proof.

SOUND-WP :	$\mathsf{sound}\,t\,(\mathsf{wp}\,t)$
SOUND-FRAMED :	$\mathsf{sound}\,t\,\mathcal{F} \Rightarrow \mathsf{sound}\,t\,(\mathsf{framed}\,\mathcal{F})$
SOUND-FAIL :	$\mathsf{sound}\,t\,\mathsf{cf\_fail}$
SOUND-VAL :	$\mathsf{sound}\,v\,(\mathsf{cf\_val}\,v)$
SOUND-APP :	$\mathsf{sound}\,t\,(\mathsf{cf\_app}\,t)$
SOUND-IF :	$\mathsf{sound}\,t_1\,\mathcal{F}_1 \wedge \mathsf{sound}\,t_2\,\mathcal{F}_2$ $\Rightarrow \mathsf{sound}\,(\mathsf{if}\,t_0\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2)\,(\mathsf{cf\_if}\,t_0\,\mathcal{F}_1\,\mathcal{F}_2)$
SOUND-LET :	$\mathsf{sound}\,t_1\,\mathcal{F}_1 \wedge \big(\forall X.\ \mathsf{sound}\,([X/x]\,t_2)\,(\mathcal{G}_2\,X)\big)$ $\Rightarrow \mathsf{sound}\,(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\,(\mathsf{cf\_let}\,\mathcal{F}_1\,\mathcal{G}_2)$
SOUND-FIX :	$\big(\forall F X.\ \mathsf{sound}\,([X/x]\,[F/f]\,t)\,(\mathcal{G}\,F\,X)\big)$ $\Rightarrow \mathsf{sound}\,(\mu f.\lambda x.t)\,(\mathsf{cf\_fix}\,\mathcal{G})$

Each of these lemmas admits a short proof. The lemma SOUND-FIX requires 4 lines of Coq script, the lemmas SOUND-LET and SOUND-IF each require 2 lines of Coq script, and all others require a single line of proof. For example, the proof of SOUND-LET is as follows.

```
Lemma sound_let : ∀F1 G2 x t1 t2,
  sound t1 F1 →
  (∀ v, sound (subst1 x v t2) (G2 v)) →
  sound (trm_let x t1 t2) (cf_let F1 G2).
Proof using.
  introv S1 S2. intros Q. unfolds cf_let. applys himpl_trans wp_let.
  applys himpl_trans S1. applys wp_conseq. intros v. applys S2.
Qed.
```

With these lemmas at hand, the Coq script for the soundness proof is no more than a dozen lines long.

```
Lemma sound_cf : ∀E t,
  sound (isubst E t) (cf E t).
Proof using.
  intros. gen E. induction t; intros; simpl;
   applys sound_framed.
  { applys sound_val. }
  { rename v into x. unfold cf_var. case_eq (lookup x E).
    { intros v EQ. applys sound_val. }
    { intros N. applys sound_fail. }}
  { introv IHt1. applys sound_fix.
    intros F X. rewrite ← isubst_rem_2. applys IHt1. }
  { applys wp_sound. }
  { applys sound_let.
    { applys IHt1. }
    { intros X. rewrite ← isubst_rem. applys IHt2. }}
  { applys sound_if. { applys IHt2. }{ applys IHt3. }}
Qed.
```

The only tedious parts of the proof are the lemmas `isubst_rem` and `isubst_rem_2`, which explain how substitutions commute. For example, `isubst_rem` establishes the equality:

$$\mathsf{subst}\,((x,v) :: E)\,t \;=\; [v/x]\,(\mathsf{subst}\,(E \setminus \{x\})\,t)$$

where $E \setminus \{x\}$ denotes a copy of E with bindings on x removed.

Equipped with these results, we derive our final theorem justifying the soundness of characteristic formulae by instantiating Lemma 9.4.1 on the empty environment.

Theorem 9.4.1 (Soundness of characteristic formulae)

$$\mathit{cf}\,t\,Q \vdash \mathit{wp}\,t\,Q$$

In practice, this soundness theorem is exploited by means of the rule CF-TRIPLE-FIX, which allows establishing a specification triple for a function by processing the characteristic formula of its body (recall Section 9.1).

9.5 Interactive Proofs using Characteristic Formulae

In this section, we describe the process of reasoning about untyped code by exploiting characteristic formulae that are computed inside Coq. The examples from this section can be played interactively by opening the first two Coq files, `Basic.v` and `Repr.v`, from [Charguéraud, 2021].

Consider as an example program the function `incr`, which increments the contents of a mutable cell that stores an integer. In OCaml syntax, this function could be defined (in A-normal form) as shown below.

```
let incr =
  fun p ->
    let n = !p in
    let m = n + 1 in
    p := m
```

Thanks to the use of a Coq *custom syntax*, enclosed in specific delimiters written <{ .. }>, we can parse source code using a readable syntax, not too far from that of OCaml. The only caveat is that we need to prefix all variables with a quote symbol, to distinguish between program variables and Coq constants. The definition shown below defines a Coq constant named `incr`. This constant has type val, the type of closed values in our deep embedding.

```
Definition incr : val :=
  <{ fun 'p ⇒
      let 'n = ! 'p in
      let 'm = 'n + 1 in
      'p := 'm }>.
```

We next state a specification for that function. This specification takes the form `triple t H Q`, where `t` corresponds to an application of the function `incr` to an argument, written <{ `incr p` }> in our custom syntax. The precondition is $p \hookrightarrow n$, and the postcondition is $p \hookrightarrow (n+1)$. The "`fun _ ⇒ ...`" that appears at the head of the postcondition denotes the fact that the function returns a unit value that does not need to be named. The argument p and the auxiliary variable n are quantified outside the triple. The variable n has type int, which is an alias for $\mathbb{Z}$. Our semantics indeed assumes an arbitrary-precision arithmetic, as implemented, e.g., in the CakeML verified compiler [Kumar et al., 2014].

```
Lemma triple_incr : ∀(p:loc) (n:int),
  triple <{ incr p }>
    (p ↪ n)
    (fun _ ⇒ (p ↪ (n+1))).
```

We next discuss the proof establishing that the code of `incr` satisfies its specification. First, we explain how proof obligations are displayed. Second, we show a naive, explicit proof script. Third, we show how the use of specialized tactics called *x-tactics* or *CFML-style tactics* can shorten proof scripts.

Interactive feedback. After introducing the variables p and n in the context, the proof begins with an application of the rule CF-TRIPLE-FIX. Throughout the proof, the proof obligations involving characteristic formulae take the form $H \vdash \mathcal{F}\, Q$, where $\mathcal{F}$ is a formula associated with a subterm of the program. We display such proof obligations in Coq using a custom notation of the form PRE H CODE F POST Q. In the CODE section, the characteristic formula is displayed using our custom notation for formulae introduced earlier at step 6. Up to alpha-renaming of bound variables, the initial proof obligation reads as follows. Observe how one can somewhat recognize the body of the function `incr`.

```
PRE (p ↪ n)
CODE <[ Let n := App val_get p in
        Let m := App val_add n 1 in
        App val_set p ) ]>
POST (fun _ ⇒ (p ↪ (n+1))).
```

Proofs without x-tactics. Carrying a proof from first principles is quite verbose. We show below a corresponding proof script. The name `triple_get` refers to the lemma that corresponds to the specification of the "get" operation on references. Likewise, `triple_add` and `triple_set` refer to specification lemmas. The tactic `xsimpl` simplifies an entailment; it is detailed further on. The tactic `xpull` simplifies the left-hand side of an entailment; it is a restricted version of `xsimpl`. The Coq tactic `intros ? →` introduces two quantifiers of the form $\forall(x : A)(H : x = e)$...., then immediately substitutes x away, replacing its occurrences with the expression e.

```
Proof using.
  intros.
  applys cf_triple_fix.{ reflexivity.} simpl.
  applys cf_let.
  applys cf_app.{ apply triple_get.}{ xsimpl.}
  xpull; intros ? →.
  applys cf_let.
  applys cf_app.{ apply triple_add.}{ xsimpl.}
  xpull; intros ? →.
  applys cf_app.{ apply triple_set.}{ xsimpl.}
  xsimpl.
Qed.
```

Simplification of entailments. Each call to the tactics `xpull` and `xsimpl` may apply dozens of lemmas for exploiting the associativity and commutativity of the separating conjunction, as well as extraction rules (STAR-EXISTS and EXISTS-L and PURE-L, Chapter 5). The tactic `xsimpl` is

a procedure able to simplify and/or prove nontrivial entailments, including ones involving certain cancellations of magic wand operators. Here are two example entailments that `xsimpl` can discharge.

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

The behavior of `xsimpl` is described in details in [Charguéraud, 2020, Appendix K]. This tactic is currently implemented using Ltac, the tactic programming language of Coq. Yet, due to limitation of Ltac, this implementation is quite slow, and is the major performance bottleneck.

Proofs using x-tactics. Let us revisit the proof of the specification lemma for the increment function using specialized tactics for manipulating characteristic formulae. The corresponding script, shown below, consists of a series of *x-tactics*. Each tactic applies one or several rules (i.e., lemmas) specifically tailored for processing characteristic formulae—details are given in Section 9.6.

```
Proof.
  xwp. xapp. xapp. xapp. xsimpl.
Qed.
```

A more complex example. To give an idea of what a typical proof script looks like, let us consider a more complex example. The example consists of the copy function for C-style, null-terminated linked list, where `mnil` and `mcons` are smart constructors for the empty list and for a list cell, respectively.

```
let rec mcopy p =
  let b = (p == null) in
  if b then
    mnil ()
  else
    let x = p.head in
    let q = p.tail in
    let q2 = mcopy q in
    mcons x q2
```

The specification of this function has been presented in Section 3.2.2. We reproduce it here using Coq syntax.

```
Lemma triple_mcopy : ∀(p:loc) (L:list val),
  triple (mcopy p)
    (MList L p)
    (fun (r:val) ⇒ ∃(p':loc), \[r = p'] ⋆ (MList L p) ⋆ (MList L p')).
```

The proof script is shown below. The first line sets up a well-founded induction on the list. The remaining lines follow the structure of the program: `xwp` enters the proof; `xapp` is used to handle each function call; `xif` handles the conditional and leaves one subgoal for each branch. The tactic `xchange` exploits the consequence rule to fold or unfold the representation predicate for mutable lists (Mlist, see Definition 3.1.3). The star symbol that appears after tactic names denotes a call to Coq's automation tactic `eauto`.

```
Proof using.
  intros. gen p. induction_wf IH: list_sub L.
  xwp. xapp. xchange MList_if. xif; intros C; case_if; xpull.
```

```
  { intros → . xapp. xsimpl*. subst. xchange* ← MList_nil. }
  { intros x q L' → . xapp. xapp. xapp. intros q'.
    xapp. intros p'. xchange ← MList_cons. xsimpl*. }
Qed.
```

Summary. The above script is representative of many CFML-style proofs. It consists of:

1. the set-up of a proof by induction, in the case of a recursive function;
2. x-tactics for handling a term construct and thereby make progress through the code;
3. interleaved between the former, x-tactics that apply the structural reasoning rules, which are not syntax-directed;
4. also interleaved between x-tactics for term constructs, calls to conventional Coq tactics for performing rewriting operations, or performing case analyses (i.e., *inversions*);
5. calls of `xsimpl` and `xpull` for simplifying entailments;
6. conventional Coq tactics for discharging pure obligations in the leaves of the proof tree.

The use of x-tactics for processing characteristic formulae and for simplifying entailments allows achieving fairly concise proof scripts. Besides, the x-tactics that follow the term constructs help organize the proof script in a way that matches the structure of the code. If either the code or the specification is modified, the user greatly benefits from these structuring tokens for figuring out where and how to fix the proof script. We next give a brief overview of how x-tactics are defined.

9.6 Implementation of CFML-Style Tactics

We next describe the construction of a few key CFML tactics. We do not aim here for exhaustiveness. Details may be found in the Coq file `WPgen.v`.

Processing of specification triples. As mentioned earlier, the user begins a proof with the tactic `xwp`. First, the tactic introduces the variables universally quantified in the specification. Second, it applies the rule CF-TRIPLE-FIX, reproduced below. Third, it launches the evaluation in Coq of the application of `cf` to the body of the function.

$$\frac{v_1 = \hat{\mu} f.\lambda x.t \qquad f \neq x \qquad H \vdash \mathsf{cf}\,([v_2/x]\,[v_1/f]\,t)\,Q}{\{H\}\ (v_1\,v_2)\ \{Q\}}\ \textsc{cf-triple-fix}$$

Application of the frame rule. The tactic `xframe` enables the user to invoke the frame rule. This tactic leverages the rule `cf-frame` shown below. The first premise asserts that the characteristic formula $\mathcal{F}$ at hand contains a leading "framed" transformer. This premise is always verified by construction of characteristic formulae.

$$\frac{\mathcal{F} = \mathsf{framed}\,\mathcal{F}' \qquad H \vdash H_1 \star H_2 \qquad H_1 \vdash \mathcal{F}\,Q_1 \qquad Q_1 \mathbin{-\!\!\star} H_2 \vdash Q}{H \vdash \mathcal{F}\,Q}\ \textsc{cf-framed}$$

In practice, the tactic comes in two flavors: one tactic for specifying which heap predicates that should be *kept* (i.e., providing H_1), and one for specifying which heap predicates should be *excluded* (i.e., providing H_2). In both cases, the heap predicate that corresponds to the complement is computed by invoking `xsimpl` on the second premise.

Processing of values. The tactic `xval` invokes the lemma `cf-val`, which reformulates the definition of cf_val v.

$$\frac{H \vdash Q\,v}{H \vdash \mathsf{cf\_val}\,v\,Q}\ \textsc{cf-val}$$

Processing of let-bindings. Likewise, the tactic `xlet` invokes the lemma CF-LET, which reformulates the definition of cf_let $\mathcal{F}_1\,\mathcal{G}_2$.

$$\frac{H \vdash \mathcal{F}_1\,(\lambda X.\ \mathcal{G}_2\,X\,Q)}{H \vdash \mathsf{cf\_let}\,\mathcal{F}_1\,\mathcal{G}_2\,Q}\ \textsc{cf-let}$$

Processing of applications. The tactic `xapp` is the most interesting. First, `xapp` invokes `xlet` if it faces a let-binding. Then, `xapp` expects to face a proof obligation of the form $H \vdash \mathsf{wp}\,t\,Q$, where t corresponds to a function application. It applies the rule RAMIFIED-FRAME-FOR-WP, introduced in Section 8.4 and reproduced below.

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

CFML provides a mechanism for registering a specification lemma for every top-level function. Using this "database", we are able to automatically look up and instantiate the relevant lemma. Using the instantiated specification lemma, we discharge the first premise of the rule RAMIFIED-FRAME-FOR-WP. We also offer means of providing explicit arguments for instantiating the specification lemma in nontrivial cases.

The second premise is handled by `xsimpl`, which, in particular, computes on-the-fly the *frame* that applies to the function call. More precisely, by cancelling the current heap predicate H against the precondition H_1, `xsimpl` simplifies the remaining proof obligation to $Q_1 \star H_2 \vdash Q$, where H_2 denotes the framed predicate.

For function calls inside a let-binding, the postcondition Q is of the form $\lambda X.\ \mathcal{G}_2\,X\,Q'$. In such case, the proof obligation simplifies further to: $\forall X.\ (Q_1\,X) \star H_2 \vdash \mathcal{G}_2\,X\,Q'$. Here, X denotes the value produced by the function call, $Q_1\,X$ characterizes the heap produced by that function call, H_2 denotes the heap predicate framed during the call, and $\mathcal{G}_2\,X\,Q'$ denotes the characteristic formula of the continuation, which may refer to the value X.

In many cases, the postcondition Q_1 includes an equality on X, so the proof obligation takes the form: $\forall X.\ [X = V] \star (Q_1'\,X) \star H_2 \vdash \mathcal{G}_2\,X\,Q'$. Such proof obligations are further simplified by `xapp` into the form: $(Q_1'\,V) \star H_2 \vdash \mathcal{G}_2\,V\,Q'$. Doing so saves the user the need to perform the substitution for X by hand. A tactic `xapp_nosubst` can be used to avoid this automated simplification, in the rare cases where it is preferable to preserve an explicit equality on X.

Summary. Our methodology in developing CFML-tactics is to capture as much as possible of the reasoning in the statement of lemmas, in order to limit the required amount of tactic programming to the minimum. In each tactic, in particular in `xapp`, we integrate a number of "processing by default" to obtain the behavior that is best-suited for the majority of the cases encountered in practice. We provide variants of the tactics to handle the rarer cases appropriately. Overall, we are able to implement a set of robust, well-specified tactics for processing characteristic formulae through concise proof scripts. In practice, when verifying the implementation of an algorithm using CFML, x-tactics account for a tiny fraction of our proof scripts: most of the reasoning is concerned with explaining why the algorithm at hand is correct, tackling its inherent complexity rather than its implementation details.

Chapter 10

The Magic Wand Operator

10.1 Definition of the Magic Wand

The magic wand, also known as *separating implication*, 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 10.1.1 (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.*

$$\frac{(H_1 \star H_0) \vdash H_2}{H_0 \vdash (H_1 \mathrel{-\!\!\star} H_2)}\ \textit{WAND-INTRO} \qquad \frac{}{H_1 \star (H_1 \mathrel{-\!\!\star} H_2) \vdash H_2}\ \textit{WAND-CANCEL}$$

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.

Each of these four characterizations of the magic wand operator have appeared in various papers on Separation Logic, yet Charguéraud [2020] appears to provide the first mechanized proof of their equivalence.

10.2 Properties of the Magic Wand

In practice, the properties stated below are useful for working with the magic wand and for implementing a tactic that simplifies the proof obligations that arise from the *ramified frame rule* (Section 10.4).

Lemma 10.2.1 (Useful properties of the magic wand)

$$\frac{H_1' \vdash H_1 \qquad H_2 \vdash H_2'}{(H_1 \mathrel{-\!\!\star} H_2) \vdash (H_1' \mathrel{-\!\!\star} H_2')}\ \textit{WAND-MONOTONE} \qquad \frac{}{[\,] \vdash (H \mathrel{-\!\!\star} H)}\ \textit{WAND-SELF} \qquad \frac{P}{([P] \mathrel{-\!\!\star} H) = H}\ \textit{WAND-PURE-L}$$

$$\frac{}{((H_1 \star H_2) \mathrel{-\!\!\star} H_3) = (H_1 \mathrel{-\!\!\star} (H_2 \mathrel{-\!\!\star} H_3))}\ \textit{WAND-CURRY} \qquad \frac{}{((H_1 \mathrel{-\!\!\star} H_2) \star H_3) \vdash (H_1 \mathrel{-\!\!\star} (H_2 \star H_3))}\ \textit{WAND-STAR}$$

Lemma 10.2.2 (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 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.

10.3 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 (Section 10.4). 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 10.3.1 (Magic wand for postconditions) *The operator* $(\mathrel{-\!\!\!\star})$ *is equivalently defined by:*

1. $Q_1 \mathrel{-\!\!\!\star} Q_2 \;\equiv\; \forall v.\ \big((Q_1\, v) \mathrel{-\!\!\star} (Q_2\, v)\big)$
2. $Q_1 \mathrel{-\!\!\!\star} Q_2 \;\equiv\; \lambda h.\ \big(\forall v h'.\ h \perp h' \wedge Q_1\, v\, h' \Rightarrow Q_2\, v\, (h \uplus h')\big)$
3. $Q_1 \mathrel{-\!\!\!\star} Q_2 \;\equiv\; \exists H_0.\ H_0 \star \big[(Q_1 \mathbin{\dot\star} H_0) \mathrel{\dot\vdash} Q_2\big]$
4. $H_0 \vdash (Q_1 \mathrel{-\!\!\!\star} Q_2) \;\Leftrightarrow\; (Q_1 \mathbin{\dot\star} H_0) \mathrel{\dot\vdash} Q_2$
5. $Q_1 \mathrel{-\!\!\!\star} Q_2$ *satisfies the following introduction and elimination rules.*

$$\frac{(Q_1 \mathbin{\dot\star} H_0) \mathrel{\dot\vdash} Q_2}{H_0 \vdash (Q_1 \mathrel{-\!\!\!\star} Q_2)}\ \textit{QWAND-INTRO} \qquad \frac{}{Q_1 \mathbin{\dot\star} (Q_1 \mathrel{-\!\!\!\star} Q_2) \mathrel{\dot\vdash} Q_2}\ \textit{QWAND-CANCEL}$$

Lemma 10.3.1 (Useful properties of the magic wand for postconditions)

$$\frac{Q_1' \mathrel{\dot\vdash} Q_1 \qquad Q_2 \mathrel{\dot\vdash} Q_2'}{(Q_1 \mathrel{-\!\!\!\star} Q_2) \vdash (Q_1' \mathrel{-\!\!\!\star} Q_2')}\ \textit{QWAND-MONOTONE} \qquad \frac{}{[\,] \vdash (Q \mathrel{-\!\!\!\star} Q)}\ \textit{QWAND-SELF}$$

$$\frac{}{((Q_1 \mathrel{-\!\!\!\star} Q_2) \mathbin{\dot\star} H) \vdash (Q_1 \mathrel{-\!\!\!\star} (Q_2 \mathbin{\dot\star} H))}\ \textit{QWAND-STAR} \qquad \frac{}{\big(Q_1 \mathrel{-\!\!\!\star} Q_2\big) \vdash \big((Q_1\, v) \mathrel{-\!\!\!\star} (Q_2\, v)\big)}\ \textit{QWAND-SPECIALIZE}$$

10.4 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 $Q_1 \mathbin{⋆} H_2 \vdash Q$ from the CONSEQUENCE-FRAME rule is equivalent to $H_2 \vdash (Q_1 \mathbin{-\!\!⋆} Q)$, by the 4th characterization of Definition 10.3.1. Thus, in the other premise $H \vdash H_1 \mathbin{⋆} H_2$, the heap predicate H_2 may be replaced with $Q_1 \mathbin{-\!\!⋆} Q$. The RAMIFIED-FRAME rule appears below.

Lemma 10.4.1 (Ramified frame rule) *RAMIFIED-FRAME reformulates CONSEQUENCE-FRAME.*

CONSEQUENCE-FRAME

$$\frac{H \vdash H_1 \mathbin{⋆} H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \mathbin{⋆} H_2 \vdash Q}{\{H\}\ t\ \{Q\}}$$

RAMIFIED-FRAME

$$\frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \mathbin{⋆} (Q_1 \mathbin{-\!\!⋆} Q)}{\{H\}\ t\ \{Q\}}$$

Chapter 11

Partially-Affine Separation Logic

11.1 Linear and Affine Heap Predicates

The Separation Logic presented in the previous chapter is well-suited for a language with explicit deallocation. It is, however, 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.

Technically, a Separation Logic is said to be *linear* if only pure heap predicates, of the form $[P]$, can be discarded. The seminal papers on Separation Logic describe linear logics [O'Hearn et al., 2001; Reynolds, 2002]. On the contrary, a Separation Logic is said to be *affine* if any heap predicates may be freely discarded at any time. The purpose of this section is to set up a *partially-affine* Separation Logic, in which both *linear* and *affine* heap predicates may coexist. The contents of this chapter follows the presentation of Charguéraud [2020, §8].

11.2 Customizable Characterization of Affine Heap Predicates

The *discard rules* of our program logic are expressed using a predicate written affine H, to assert 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 pieces of heap may be discarded. The predicate haffine h is a parameter of the program logic: by suitably instantiating this predicate, the user can choose which predicates should be treated as affine, as opposed to linear.

When defining the predicate haffine h, the user only has to satisfy two basic well-formedness constraints, expressed below.

Definition 11.2.1 (Axiomatization of affine heaps) *The predicate haffine h must satisfy two rules:*

$$\frac{}{\mathit{haffine}\,\varnothing}\ \textit{STAFFINE-EMPTY} \qquad \frac{\mathit{haffine}\,h_1 \qquad \mathit{haffine}\,h_2 \qquad h_1 \perp h_2}{\mathit{haffine}\,(h_1 \uplus h_2)}\ \textit{STAFFINE-UNION}$$

The predicate affine H captures the idea that a heap predicate H can be discarded. By definition, affine H holds if the heap predicate H is restricted to affine heaps.

Definition 11.2.2 (Definition of affine heap predicates)

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

The rules presented next establish that the composition of affine heap predicates yield affine heap predicates. In other words, the predicate affine is stable by composition. For example, a heap predicate $H_1 \star H_2$ is affine provided that H_1 and H_2 are both affine. A heap predicate $\exists x.\, H$ is affine provided that H is affine for any variable x. Likewise, a heap predicate $\forall x.\, H$ is affine provided that H is affine for any variable x, with a technical restriction asserting that the type of x must be inhabited (because, otherwise, the hypothesis would be vacuous). The last rule, AFFINE-STAR-PURE, asserts that to prove $[P] \star H$ affine, it suffices to prove H affine under the hypothesis that the proposition P holds.

Lemma 11.2.1 (Sufficient conditions for affinity of a heap predicate)

$$\frac{}{\mathit{affine}\,[\,]}\ \textit{AFFINE-EMPTY} \qquad \frac{}{\mathit{affine}\,[P]}\ \textit{AFFINE-PURE} \qquad \frac{\mathit{affine}\,H_1 \qquad \mathit{affine}\,H_2}{\mathit{affine}\,(H_1 \star H_2)}\ \textit{AFFINE-STAR}$$

$$\frac{\forall x.\ \mathit{affine}\,H}{\mathit{affine}\,(\exists x.\, H)}\ \textit{AFFINE-EXISTS} \qquad \frac{\forall x.\ \mathit{affine}\,H \qquad \text{the type of } x \text{ is inhabited}}{\mathit{affine}\,(\forall x.\, H)}\ \textit{AFFINE-FORALL} \qquad \frac{P \Rightarrow \mathit{affine}\,H}{\mathit{affine}\,([P] \star H)}\ \textit{AFFINE-STAR-PURE}$$

In practice, the application of these rules is automated using a tactic. The process of justifying that a heap predicate is affine is in most cases totally transparent for the user.

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 "$\lambda h.\,$True") holds of *any heap*, the affine top predicate holds only of *any affine heap*.

Definition 11.2.3 (Affine top) *The predicate $\mathbb{T}$ can be equivalently defined in two ways.*

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

There are three important properties of the affine top predicate. 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 11.2.2 (Properties of affine top)

$$\frac{\mathit{affine}\,H}{H \vdash \mathbb{T}}\ \textit{ATOP-R} \qquad \frac{}{\mathit{affine}\,\mathbb{T}}\ \textit{AFFINE-ATOP} \qquad \frac{}{(\mathbb{T} \star \mathbb{T}) = \mathbb{T}}\ \textit{STAR-ATOP-ATOP}$$

All the aforementioned definitions and lemmas hold for any predicate haffine satisfying the axiomatization from Definition 11.2.1.

Two extreme instantiations of haffine are particularly interesting. The first instantiation treats all heaps as discardable, leading to a *fully-affine* logic. The second instantiation treats none heaps as discardable, leading to a *fully-linear* logic, equivalent to the logic from the previous chapter.

Example 11.2.1 (Fully-affine Separation Logic) *The definition "haffine $h \equiv$ True" satisfies the requirements of Definition 11.2.1, and leads to a Separation Logic where all heap predicates may be freely discarded. In that setting, (affine H) $\Leftrightarrow$ True, and $\mathbb{T} = \top = (\lambda h.\,\mathit{True}) = (\exists H.\, H)$.*

Example 11.2.2 (Fully-linear Separation Logic) *The definition "haffine $h \equiv (h = \varnothing)$" satisfies the requirements of Definition 11.2.1, and leads to a Separation Logic where only pure heap predicates may be freely discarded. In that setting, (affine H) $\Leftrightarrow$ $(H \vdash [\,])$, and $\mathbb{T} = [\,] = (\lambda h.\, h = \varnothing)$.*

11.3 Triples for a Partially-Affine Separation Logic

To accommodate reasoning rules that enable freely discarding affine heap predicates, it suffices to refine the definition of a Separation Logic triple (Definition 6.6.2) by integrating the affine top predicate $\top\!\!\top$ into the postcondition of the underlying Hoare triple, as formalized next.

Definition 11.3.1 (Refined definition of triples for Separation Logic)

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall H'.\ ^{HOARE}\{H \star H'\}\ t\ \{Q \star H' \star \top\!\!\top\}$$

Note that, with the fully-linear instantiation described in Example 11.2.2, the predicate $\top\!\!\top$ is equivalent to the empty heap predicate, therefore Definition 11.3.1 is strictly more general than Definition 6.6.2.

Lemma 11.3.1 (Reasoning rules for refined Separation Logic triples) *All the previously mentioned reasoning rules, in particular the structural rules (Lemma 7.1.1) and the reasoning rules for terms (Lemma 7.2.1), remain correct with respect to the refined definition of triples (Definition 11.3.1).*

The *discard rules*, which enable discarding affine heap predicates, may be stated in a number of ways. The three variants that are most useful in practice are shown below. These three variants have equivalent expressive power with respect to discarding heap predicates.

The rule DISCARD-PRE allows discarding a user-specified predicate H' from the precondition, provided that H' is affine. Without this rule, the user would have to carry this heap predicate H' through the proof until it appears in a postcondition.

The rule ATOP-POST allows extending the postcondition with $\top\!\!\top$, allowing a subsequent proof step to yield an entailment relation of the form $Q_1 \vdash (Q \star \top\!\!\top)$, allowing to discard unwanted pieces from Q_1. This rule is useful in "manual" proofs, i.e., proofs carried out with limited tactic support.

The rule RAMIFIED-FRAME-ATOP extends the ramified frame rule so that its entailment integrates the predicate $\top\!\!\top$, allowing to discard unwanted pieces from either H or Q_1. This rule is a key building block for a practical tool that implements a partially-affine Separation Logic.

Lemma 11.3.2 (Discard rules for triples)

$$\frac{\{H\}\ t\ \{Q\} \qquad \mathit{affine}\ H'}{\{H \star H'\}\ t\ \{Q\}}\ \textit{DISCARD-PRE} \qquad \frac{\{H\}\ t\ \{Q \star \top\!\!\top\}}{\{H\}\ t\ \{Q\}}\ \textit{ATOP-POST}$$

$$\frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \star (Q_1 \mathrel{-\!\!\star} (Q \star \top\!\!\top))}{\{H\}\ t\ \{Q\}}\ \textit{RAMIFIED-FRAME-ATOP}$$

These three rules can be equivalently formulated in weakest-precondition style, shown below. Let us point out the strength of the third rule, namely WP-RAMIFIED-FRAME-ATOP. This rule subsumes all the other structural rules of our Separation Logic: CONSEQUENCE, FRAME, PROP, EXISTS, DISCARD-PRE, and ATOP-POST (stated in Lemma 7.1.1 and Lemma 11.3.2).

Lemma 11.3.3 (Discard rules in weakest-precondition style)

$$\frac{\mathit{affine}\ H}{(\mathit{wp}\, t\, Q) \star H \ \vdash\ (\mathit{wp}\, t\, Q)}\ \textit{WP-DISCARD-PRE} \qquad \frac{}{\mathit{wp}\, t\, (Q \star \top\!\!\top) \ \vdash\ \mathit{wp}\, t\, Q}\ \textit{WP-ATOP-POST}$$

$$\frac{}{(\mathit{wp}\, t\, Q) \ \star\ (Q \mathrel{-\!\!\star} (Q' \star \top\!\!\top)) \ \vdash\ (\mathit{wp}\, t\, Q')}\ \textit{WP-RAMIFIED-FRAME-ATOP}$$

Chapter 12

Arrays

This chapter present specifications for operations on ML-style arrays and C-style arrays. The representation predicate Array $p\,L$ to assert that at location p is allocated an array whose elements are described by the list L. In particular, the length of the array is equal to the length of the list L, written $|L|$.

12.1 Representation of ML-style Arrays

Let us start by explaining how the predicate Array $p\,L$ may be defined in a foundational manner with respect to the memory model. In ML, an array is laid out in memory as a header cell followed by a sequence of cells.

- The header block is described by a heap predicate written ArrayHeader $p\,n$, where p denotes the address of the array and n its length.
- An individual array cell is described by a heap predicate written Cell $p\,i\,v$ where p denotes the base of the address, i an index in the array, and v the value stored in the cell at that index.
- The heap predicate ArraySeg $p\,j\,L$ describes an array segment from the array p, starting at index j, and covering a range of cells whose elements are described by the list L.
- The high-level heap predicate Array $p\,L$ describes both the array header and the *full* segment, which covers elements from index 0 to index $|L| - 1$, inclusive.

In the formal definitions shown below, $L[i]$ stands for "List.nth $i\,L$" and $L[i := v]$ stands for "List.update $i\,v\,L$". Besides, the construction $\star_{i \in [0,|L|)}\ H_i$ denotes an iterated separating conjunction, indexed by valid indices in the list L.

Definition 12.1.1 (Definitions of representation predicates for arrays)
The first two definitions are implementation-dependent and should not be revealed to the end user.

$$
\begin{array}{lcll}
\mathit{ArrayHeader}\,p\,n & \equiv & p \hookrightarrow n & \text{(not revealed)} \\
\mathit{Cell}\,p\,i\,v & \equiv & (p + 1 + i) \hookrightarrow v & \text{(not revealed)} \\
\mathit{ArraySeg}\,p\,j\,L & \equiv & \star_{i \in [0,|L|)}\ \mathit{Cell}\,p\,(j + i)\,(L[i]) & \\
\mathit{Array}\,p\,L & \equiv & \mathit{ArrayHeader}\,p\,|L| \star \mathit{ArraySeg}\,p\,0\,L &
\end{array}
$$

12.2 Operations on Arrays

We can assign two sets of specifications to array operations: large-footprint specifications expressed in terms of the high-level predicate $\mathsf{Array}\,p\,L$, or small-footprint specifications expressed in terms of the finer-grained predicates $\mathsf{Cell}\,p\,i\,v$ and $\mathsf{ArrayHeader}\,p\,n$. Let us first show the large-footprint specifications, which are generally more convenient to work with.

Lemma 12.2.1 (Large-footprint specifications for array operations)

$$\begin{array}{rcl} n \geq 0 & \Rightarrow & \{[]\}\ (\mathit{Array.make}\,n\,v)\ \{\lambda p.\ \mathsf{Array}\,p\,(\mathit{List.make}\,n\,v)\} \\ & & \{\mathsf{Array}\,p\,L\}\ (\mathit{Array.length}\,p)\ \{\lambda r.\ [r = |L|] \star \mathsf{Array}\,p\,L\} \\ 0 \leq i < |L| & \Rightarrow & \{\mathsf{Array}\,p\,L\}\ (\mathit{Array.get}\,p\,i)\ \{\lambda r.\ [r = L[i]] \star \mathsf{Array}\,p\,L\} \\ 0 \leq i < |L| & \Rightarrow & \{\mathsf{Array}\,p\,L\}\ (\mathit{Array.set}\,p\,i\,v)\ \{\lambda \_.\ \mathsf{Array}\,p\,(L[i := v])\} \end{array}$$

Small-footprint specifications reveal useful for reasoning about algorithms that operate on a clearly delimited subset of the array cells. For example, a recursive call to quicksort operates on a specific array segment. By using smaller-footprint specifications, one benefits from the frame rule, which inherently captures the fact that all the array cells that are not mentioned in the precondition remain unmodified. The small-footprint specifications may be expressed either at the level of individual cells or at the level of array segments. We first show the specifications at the level of cells.

Lemma 12.2.2 (Small-footprint specifications for array operations, for individual cells)

$$\begin{array}{rcl} \{\mathsf{Cell}\,p\,i\,v\}\ (\mathit{Array.get}\,p\,i)\ \{\lambda r.\ [r = v] \star \mathsf{Cell}\,p\,i\,v\} \\ \{\mathsf{Cell}\,p\,i\,v'\}\ (\mathit{Array.set}\,p\,i\,v)\ \{\lambda \_.\ \mathsf{Cell}\,p\,i\,v\} \\ \{\mathsf{ArrayHeader}\,p\,n\}\ (\mathit{Array.length}\,p)\ \{\lambda r.\ [r = n]\} \end{array}$$

In the last specification stated above, observe that reading the length of the array requires only access to the header, described by $\mathsf{ArrayHeader}\,p\,n$. Remark: in Cosmo [Mével et al., 2020], a concurrent Separation Logic for multicore OCaml, $\mathsf{ArrayHeader}\,p\,n$ appears to the user as a duplicatable heap predicate, more convenient to manipulate.

12.3 Borrowing and Splitting

The *borrowing* lemma presented next may reveal useful for exploiting small-footprint specifications without revealing the iterated star that enumerates all the array cell. This lemma allows isolating the i-th cell out of an array, to perform read and write operations on that cell in isolation from the rest of the array. Subsequently, the cell with its updated contents, named v below, may be merged back into the array representation. This logical operation involves cancelling a magic wand.

Lemma 12.3.1 (Borrowing of a cell) *Assume* $0 \leq i < |L|$.

$$\big(\mathsf{Array}\,p\,L\big) \ \vdash\ \big(\mathsf{Cell}\,p\,i\,(L[i])\big) \star \big(\forall v.\ \mathsf{Cell}\,p\,i\,v \mathrel{-\!\!\star} \mathsf{Array}\,p\,(\mathit{List.update}\,i\,v\,L)\big)$$

We next present specifications based on array segments. There, i denotes the absolute index, j denotes the start of the segment, and d denotes the index of the targeted cell relative to the start of the segment—thus $i = j + d$.

Lemma 12.3.2 (Small-footprint specifications for array operations, for segments)

$$d = i - j \ \wedge\ 0 \le d < |L| \Rightarrow \{\mathit{ArraySeg}\, p\, j\, L\}\ \ (\mathit{Array.get}\, p\, i)\ \ \{\lambda r.\ [r = L[d]] \star \mathit{ArraySeg}\, p\, j\, L\}$$
$$d = i - j \ \wedge\ 0 \le d < |L| \Rightarrow \{\mathit{ArraySeg}\, p\, j\, L\}\ (\mathit{Array.set}\, p\, i\, v)\ \{\lambda\_.\ \mathit{ArraySeg}\, p\, j\, L[d := v]\}$$

For functions that process an array by making recursive calls to increasingly-smaller segments of the array, the following *range splitting* lemma allows splitting the segment at hand. Typically, one would provide one of the two segments to a recursive call (e.g., in quicksort), while the other fragment may be framed over the scope of that call.

Lemma 12.3.3 (Splitting of array segments)

$$\mathit{ArraySeg}\, p\, j\, (L_1 +\!\!+ L_2) \ =\ \mathit{ArraySeg}\, p\, j\, L_1 \star \mathit{ArraySeg}\, p\, (j + |L_1|)\, L_2$$

12.4 Arrays in a C-like Language

We complete the discussion of arrays with a specification of arrays in a C-like language featuring pointer arithmetic. In C, there are no header blocks stored at the front of every array. However, there is a notion of *malloc-ed block* that needs to be tracked by the program logic. Indeed, the deallocation operation (free) may only be called on pointers obtained as a result of an allocation operation (malloc). In the formal definitions shown below, the predicate MallocBlock $p\, n$ captures the fact that a memory block of size n was allocated at location p.

Definition 12.4.1 (Alternative definitions for arrays in a C-like language)

$$\begin{array}{lcl}
\mathit{MallocBlock}\, p\, n & \equiv & \ldots \quad \text{(depends on the memory allocator)} \\
\mathit{Cell}\, p\, i\, v & \equiv & (p + i) \hookrightarrow v \qquad \text{(transparently)} \\
\mathit{Array}\, p\, L & \equiv & \mathit{ArraySeg}\, p\, 0\, L \star \mathit{MallocBlock}\, p\, |L| \\
\mathit{ArraySeg}\, p\, j\, L & \equiv & \star_{i \in [0, |L|)}\ \mathit{Cell}\, p\, (j + i)\, (L[i])
\end{array}$$

The segment representation predicate in a C-like language features additional equalities thanks to the exposure of pointer arithmetic by the language.

Lemma 12.4.1 (Properties of the array representation predicate in a C-like language)

$$\begin{array}{lcl}
\mathit{ArraySeg}\, p\, j\, (L_1 +\!\!+ L_2) & = & \mathit{ArraySeg}\, p\, j\, L_1 \star \mathit{ArraySeg}\, p\, (j + |L_1|)\, L_2 \\
\mathit{ArraySeg}\, p\, j\, L & = & \mathit{ArraySeg}\, (p + j)\, 0\, L \\
\mathit{ArraySeg}\, p\, 0\, (L_1 +\!\!+ L_2) & = & \mathit{ArraySeg}\, p\, 0\, L_1 \star \mathit{ArraySeg}\, (p + |L_1|)\, 0\, L_2
\end{array}$$

The alloc operation allocates an array of a given size. Its cells are initialized with a special value called uninit. This value cannot be read by the get operation. The specification of get is thus updated with an additional precondition of the form $v \neq \mathsf{uninit}$. The free operation, when applied to the address of a block, requires as precondition all the cells that were allocated as part of that block.

Lemma 12.4.2 (Specification of operations on arrays in a C-like language)

Specification of alloc:	$n \ge 0 \ \Rightarrow\ \{[]\}\ (\mathit{alloc}\, n)\ \{\lambda p.\ \mathit{Array}\, p\, (\mathit{List.make}\, n\ \mathit{uninit})\}$
Specification of get:	$v \neq \mathit{uninit} \ \Rightarrow\ \{p \hookrightarrow v\}\ (\mathit{get}\, p)\ \{\lambda x.\ [x = v] \star p \hookrightarrow v\}$
Specification of set:	$\{p \hookrightarrow v\}\ (\mathit{set}\, p\, v')\ \{\lambda\_.\ (p \hookrightarrow v')\}$ (where v could be uninit)
Specification of free:	$\{\mathit{Array}\, p\, L\}\ (\mathit{free}\, p)\ \{\lambda\_.\ []\}$

The view of arrays and array segments as iterated separating conjunctions of cells comes from the original papers on Separation Logic [O'Hearn et al., 2001; Reynolds, 2002]. A foundational formalization of small-footprint specifications for ML-style arrays is implemented in CFML since 2021.

Chapter 13

Records

This chapter presents specifications for operations on ML-style records. Compared the treatment of arrays, there are two important differences. The first difference is that the cells are indexed by fields names, instead of being indexed by an integer value. As a result, the model of a record is not a list of values, but a list or set of pairs each made of a field identifier and a value. It is usually desirable for representation predicates to be insensitive to the order of fields, following common practice in ML languages. The second difference is that, because ML does not expose a function to read the number of fields of a record, there is no need to keep track in the program logic of representation predicates for the record headers. For the remaining aspects, our formalization of records shares a lot of similarities with that of arrays.

13.1 Representation of Records

We introduce the representation predicate $\mathsf{Record}\,p\,K$ to assert that at location p is allocated a record whose fields are described by the list K, which consists of a list of pairs each made of a field identifier and a value. An individual record field is described by a heap predicate written $\mathsf{Field}\,p\,k\,v$ where p denotes the base of the address, k denotes a field identifier, and v the value stored in the cell at that index.

To realize record predicates in a foundational manner, we have to consider a particular memory layout for records. To that end, field identifiers are viewed as natural numbers representing the offset of the corresponding field. However, in the high-level presentation exposed to the user, fields are referred to by name—the offset need not be exposed. Thereafter, we use the term *field name* to refer to field identifiers, reflecting the choice of presenting reasoning rules using the concepts of that appear in source code.

The foundational definitions appear next. The predicate $\mathsf{Field}\,p\,k\,v$ is meant to be presented as an abstract predicate to the end user. The predicate $\mathsf{Record}\,p\,K$ is defined by iterating over the fields. The definition inherently ensures that $\mathsf{Record}\,p\,K$ is equal to $\mathsf{Record}\,p\,K'$ for any K' being a permutation of the list K. We deliberately choose to represent K as a Coq list rather than a set to ensure that fields remain ordered in the same way as the user writes them in formal specifications.

Definition 13.1.1 (Definitions of representation predicates for records)

$$\begin{array}{lcll} \mathit{Field}\,p\,k\,v & \equiv & (p+1+k) \hookrightarrow v & \text{(not revealed)} \\ \mathit{Record}\,p\,K & \equiv & \star_{(k,v)\in K}\ \mathit{Field}\,p\,k\,v & \end{array}$$

Observe that the definition of $\mathsf{Record}\,p\,K$ allows one to convert between the *record view*, which

describes multiple fields at once, and the *field view*, which enables manipulating individual fields. There is also the possibility to consider a *subset view* of a record, by means of the following split rule, which allows isolating any subset of the record fields. Considering a subset may be helpful to reason about a program component that only interacts with a subset of the fields associated with a record data type.

Lemma 13.1.1 (Decomposition rules for the record representation predicate)

$$\begin{array}{lcl} \mathit{Record}\, p\, (K_1 +\!\!+ K_2) & = & \mathit{Record}\, p\, K_1 \star \mathit{Record}\, p\, K_2 \\ \mathit{Record}\, p\, ((k, v) :: \mathit{nil}) & = & \mathit{Field}\, p\, k\, v \\ \mathit{Record}\, p\, \mathit{nil} & = & [\,] \end{array}$$

13.2 Operations on Records

The specification of operations on records share a lot of similarities with the operations on arrays. We next present small-footprint and large-footprint specifications.

Lemma 13.2.1 (Small-footprint specifications for record operations, for individual fields)

$$\begin{array}{c} \{\mathit{Field}\, p\, k\, v\} \quad (\texttt{p.k}) \quad \{\lambda r.\ [r = v] \star \mathit{Field}\, p\, k\, v\} \\ \{\mathit{Field}\, p\, k\, v'\}\ (\texttt{p.k <- v})\ \{\lambda \_.\ \mathit{Field}\, p\, k\, v\} \end{array}$$

Lemma 13.2.2 (Large-footprint specifications for record operations)

$$\begin{array}{lcccl} & & \{[\,]\}\ (\texttt{\{k1:=v1;k2:=v2\}}) & & \{\lambda p.\ \mathit{Record}\, p\, ((k1, v1) :: (k2, v2) :: \mathit{nil})\} \\ k \in \mathit{dom}\, K & \Rightarrow & \{\mathit{Record}\, p\, K\} & (\texttt{p.k}) & \{\lambda r.\ [r = K[k]] \star \mathit{Record}\, p\, K\} \\ k \in \mathit{dom}\, K & \Rightarrow & \{\mathit{Record}\, p\, K\} & (\texttt{p.k <- v}) & \{\lambda \_.\ \mathit{Record}\, p\, (K[k := v])\} \end{array}$$

13.3 Record-With Construct

OCaml's *record-with* operation applies to an existing record and creates a fresh copy of that record, with a subset of the fields being updated. Consider the specification shown below.

$$\begin{array}{lcl} k1, k2 \in \mathsf{dom}\, K & \Rightarrow & \{\mathsf{Record}\, p\, K\} \\ & & (\{\texttt{p with k1:=v1;k2:=v2}\}) \\ & & \{\lambda p'.\ \mathsf{Record}\, p'\, (K[k1 := v1][k2 := v2])\} \end{array}$$

This specification captures the semantics of the *record-with* operation, but only under the assumption that the program logic at hand is affine. Indeed, the predicate $\mathsf{Record}\, p\, K$ might cover only a subset of the fields. For every field that is not described by K, the corresponding field in the fresh record is implicitly discarded when exploiting the above specification. It is thus the responsibility of the user to gather the heap predicates associated with all the fields before reasoning about a *record-with* operation. (In a linear program logic, one would need to involve the record header predicate to constrain the length of the list K, and thereby enforce that K covers all the fields of the input record.)

CFML features a mechanism to smoothen the reasoning about read and write operations on record fields. Given an operation on a field k of a record at address p, this mechanism searches the precondition at hand for a predicate of the form $\mathsf{Field}\, p\, k\, v$, or of the form $\mathsf{Record}\, p\, K$ with $k \in K$. It then exploits the appropriate specification. In the case of a large-footprint specification, the record update of the form $K[k := v]$ is computed. In terms of syntax, CFML provides

the syntax p ~> ‘ {k1 := v1 ; k2 := v2} for Record p $((k1, v1) :: (k2, v2) :: \mathsf{nil})$. Overall, from the perspective of the end user, a heap predicate for a record closely resembles the source code for the corresponding record definition, and operations on records are processed automatically by the framework.

Chapter 14

Additional Language Extensions

This chapter presents reasoning rules for dynamic checks (Section 14.1), for terms that are not in A-normal form (Section 14.2), for while-loops and for-loops, and for n-ary functions (Section 14.4).

14.1 Treatment of Dynamic Checks (Assertions)

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 14.1.1 (Evaluation rules and reasoning rule for assertions)

$$\frac{t/s \Downarrow \mathit{true}/s'}{(\mathit{assert}\ t)/s \Downarrow tt/s'}\ \textit{BIG-ASSERT-ENABLED} \qquad \frac{}{(\mathit{assert}\ t)/s \Downarrow tt/s}\ \textit{BIG-ASSERT-DISABLED} \qquad \frac{\{H\}\ t\ \{\lambda r.\ [r = \mathit{true}] \star H\}}{\{H\}\ (\mathit{assert}\ t)\ \{\lambda\_.\ H\}}\ \textit{ASSERT}$$

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

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 in the rule ASSERT.

14.2 Beyond A-normal Form: The Bind Rule

In this section, we explain how to reason about programs that are not in A-normal form. We follow the approach of the *bind rule*, popularized by Iris [Jung et al., 2018] in the context of program logics. The bind rule follows the pattern of the let-binding rule but allows for evaluation of a subterm t that appears in an *evaluation context* E.

For the syntax introduced in Section 6.1 and used so far, we can define evaluation contexts by the following grammar, where $\square$ denotes the *hole*, i.e., the empty context. We fix here a left-to-right evaluation order; other orders could be considered.

$$E \quad := \quad \square \quad | \quad \mathsf{let}\, x = E\, \mathsf{in}\, t \quad | \quad (E\, t) \quad | \quad (v\, E) \quad | \quad \mathsf{if}\, E\, \mathsf{then}\, t\, \mathsf{else}\, t$$

We write $E[t]$ for the context E whose hole is filled with the term t. We write $\mathsf{value}\, t$ for the predicate that asserts that t is a value. The bind rule describes how to evaluate or reason about subterms that appear in evaluation contexts and that are not already values. The big-step bind rule takes the following form.

$$\frac{\neg\, \mathsf{value}\, t \qquad t/s \Downarrow v/s' \qquad E[v]\,/\,s' \Downarrow v'/s''}{E[t]\,/\,s \Downarrow v'/s''}\ \textsc{big-bind}$$

Note that the premise $\neg\, \mathsf{value}\, t$ is technically optional. Indeed, if t is a value, then the last premise is simply equivalent to the conclusion of the rule. (Nevertheless, we like to include this premise because it is necessary when considering a coinductive interpretation of the evaluation rules, see [Charguéraud et al., 2022, §3.4].)

The corresponding reasoning rules, expressed using either triples or weakest preconditions, appear next. Observe that these two rules need not include a premise of the form $\neg\, \mathsf{value}\, t$. Indeed, the rules remain valid even in the case where t is already a value. (Here, including the premise would add a serious burden onto the end user, who will have to perform additional case analyses.)

Lemma 14.2.1 (Bind rules)

BIND
$$\frac{\{H\}\ t\ \{Q_1\} \qquad (\forall v.\ \ \{Q_1\, v\}\ E[v]\ \{Q\})}{\{H\}\ E[t]\ \{Q\}}$$

WP-BIND
$$\frac{}{\mathsf{wp}\, t\, (\lambda v.\ \mathsf{wp}\, (E[v])\, Q) \ \vdash\ \mathsf{wp}\, (E[t])\, Q}$$

14.3 Inductive Reasoning for Loops

Pointer-manipulating programs are typically written using loops. Although loops can be simulated using recursive functions, proofs are simpler in the presence of a while-loop construct. We write it "$\mathsf{while}\, t_1\, \mathsf{do}\, t_2$".

A loop "$\mathsf{while}\, t_1\, \mathsf{do}\, t_2$" 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 BIG-WHILE and WHILE shown below capture this one-step unfolding principle.

Lemma 14.3.1 (Evaluation rule and reasoning rules for while loops)

BIG-WHILE
$$\frac{(\mathit{if}\, t_1\ \mathit{then}\ (t_2\,;\ \mathit{while}\, t_1\ \mathit{do}\, t_2)\ \mathit{else}\ tt)/s \Downarrow v/s'}{(\mathit{while}\, t_1\ \mathit{do}\, t_2)/s \Downarrow v/s'}$$

WHILE
$$\frac{\{H\}\ (\mathit{if}\, t_1\ \mathit{then}\ (t_2\,;\ \mathit{while}\, t_1\ \mathit{do}\, t_2)\ \mathit{else}\ tt)\ \{Q\}}{\{H\}\ (\mathit{while}\, t_1\ \mathit{do}\, t_2)\ \{Q\}}$$

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 14.3.1 (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 loop is specified by the triple:

$$\begin{array}{l}\{\mathit{Mlist}\, L\, p \star r \hookrightarrow p \star s \hookrightarrow 0\} \\ (\texttt{while ... done}) \\ \{\lambda\_.\, \mathit{Mlist}\, L\, p \star r \hookrightarrow \mathit{null} \star s \hookrightarrow |L|\}\end{array}$$

and its proof is conducted by induction on the following statement.

$$\begin{array}{l}\forall L n p.\ \{\mathit{Mlist}\, L\, p \star r \hookrightarrow p \star s \hookrightarrow n\} \\ \quad (\texttt{while ... done}) \\ \quad \{\lambda\_.\, \mathit{Mlist}\, L\, p \star r \hookrightarrow \mathit{null} \star s \hookrightarrow n + |L|\}\end{array}$$

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* $(p.\mathit{head} \hookrightarrow x) \star (p.\mathit{tail} \hookrightarrow q)$*, and to apply the induction hypothesis to the tail of the list described by* $\mathit{Mlist}\, L'\, q$*, 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.

The statement of a reasoning rule for loops that allows to frame over the remaining iterations had been devised independently by Tuerk [2010] and Charguéraud [2010].

14.4 Treatment of Functions of Several Arguments

Functions of several arguments may be represented as curried functions, as tupled functions, or as native n-ary functions (like, e.g., in the C language). Regardless of the representation of functions, the rules for reasoning about a proper function call—i.e., with the expected number of arguments—are stated essentially in the same way. For example, one may state the following rule for reasoning about the call to a function of two arguments. The predicate noduplicates involved here captures the fact that the name of the function and of its arguments do not clash.

Lemma 14.4.1 (Reasoning rule for functions of arity 2)

APP2

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

More interestingly, we can state an arity-generic version of the rule, that works for any arity. It applies to a function f expecting a list $\overline{x}$ of arguments of the form "$x_1 :: \ldots :: x_n ::$ nil". The term $v_0\, \overline{v}$ denotes the application of a value v_0 to a list of arguments $\overline{v}$ of the form "$v_1 :: \ldots :: v_n ::$ nil". The corresponding rule, which is used in the current version of CFML, is stated as follows.

Lemma 14.4.2 (Reasoning rule for n-ary functions)

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

Remark: in CFML, we set up a Coq coercion such that an application written in curried style "$v_0\, v_1 \ldots v_n$" is elaborated to the n-ary application "$v_0\, (v_1 :: \ldots :: v_n :: \mathsf{nil})$".

To reason about n-ary applications that are not in A-normal forms, the bind rule can be used, with a suitably adapted grammar of contexts.

Chapter 15

A Survey of Separation Logic for Sequential Programs

This chapter focuses on research on Separation Logic, excluding work on concurrent Separation Logic. This survey corresponds, up to minor updates, to the version published in [Charguéraud, 2020]. For a broader survey of Separation Logic, we refer to O'Hearn's CACM paper [2019]. In particular, the appendix to his survey covers practical automated and semi-automated tools based on Separation Logic, such as Infer [Calcagno et al., 2015], VeriFast [Philippaerts et al., 2014], or Viper [Müller et al., 2016].

Section 15.1 starts by listing the ingredients that were already present in the seminal papers on Separation Logic [O'Hearn et al., 2001; Reynolds, 2002]. Section 15.2 attempts to trace the origin of every other ingredient. Section 15.3 gives a tour of the work that involves mechanized presentations of Separation Logic. Finally, Section 15.4 provides a review of course notes on Separation Logic.

15.1 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 appeared in the first descriptions of Separation Logic [O'Hearn et al., 2001; Reynolds, 2002]:

- 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 rule;
- a variant of the rule EXISTS, named EXISTS2 in the discussion further below;
- the fundamental properties of the star operator described in Lemma 5.2.1;
- the small footprint specifications for primitive state-manipulating operations,
- the definition of Mlist, stated by pattern-matching over the list structure like in Definition 3.1.2;
- the characterization of the magic wand operator via characterizations (1), (3) and (4) from Definition 10.1.1, but not characterization (2), which involves quantification over heap predicates;

- the example of a copy function for binary trees;
- the encoding of records and arrays using pointer arithmetics.

My presentation of structural reasoning rules (Lemma 7.1.1) features two extraction rules named PROP and EXISTS. These rules did not appear in that form in the original papers on Separation Logic. Instead, these papers included the following two rules.

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

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 5.1). Compared with EXISTS2, the statement of EXISTS is more concise and better-suited for practical purpose. The rule PROP for extracting pure facts may be seen as a particular instance of the rule EXISTS for extracting existential quantifiers. Indeed, as pointed out in Remark 4.2.1, the heap predicate $[P]$ is equal to $\exists(p : P).\ [\,]$. The rule FORALL does not need to be included in the core set of rules. Indeed, it is derivable, via the rule of CONSEQUENCE, from the rule FORALL-L, which enables instantiating universal quantifiers in entailments (Figure 5.1).

15.2 Additional Features of Separation Logic

The original presentation of Separation Logic consists of a first-order logic for a first-order language. Follow-up work aimed for higher-order logics and languages.

Biering et al. [2005, 2007] tackled the generalization to *higher-order quantification*—the possibility to quantify over propositions and heap predicates—using *BI-hyperdoctrines*. Krishnaswami et al. [2007] formalized the subject-observer pattern with a strong form of information hiding between the subject and the client. This work illustrated how higher-order Separation Logic supports data abstraction.

Birkedal et al. [2005, 2006] tackled the generalization of Separation Logic to *higher-order languages*, where functions may take functions as arguments. To avoid complications with mutable variables, the authors considered a version of Algol with immutable variables and first-order heaps—heap cells can only store integer values. Specifications are presented using dependent types: a triple $\{H\}\ t\ \{Q\}$ is expressed by the fact that the term t admits the type "$\{H\} \cdot \{Q\}$". One key idea from this work is to bake-in the frame rule into the interpretation of triples, that is, to quantify over a heap predicate describing the rest of the state, as in Definition 6.6.2. The technique of the baked-in frame rule later proved successful in mechanized proofs. For example, it appears in the HOL4 formalization by Myreen and Gordon [2007] (see §3.2, as well as §2.4 from Myreen's PhD thesis [2008]) and in the Coq formalization by [Appel and Blazy, 2007] (see Definition 9).

Reus and Schwinghammer [2006] presented a generalization of Separation Logic to *higher-order stores*, where heap cells may store functions whose execution may act over the heap. The former work targets a language that features storable, parameter-less procedures. Its model, developed on paper, was then simplified by Birkedal et al. [2008] using the technique of the baked-in frame rule.

Another approach to tackling the circularity issues associated with higher-order quantification and higher-order stores consists of using the *step indexing* technique [Appel and McAllester, 2001; Ahmed, 2004; Appel et al., 2007]. In that approach, a heap predicate depends not only on a heap but also on a natural number, which denotes the number of execution steps for which the predicate is guaranteed to hold. This approach was later exploited in VST, which provided the first higher-order *concurrent* Separation Logic [Hobor et al., 2008], and in Iris [Jung et al., 2017].

Ni and Shao [2006] presented the XCAP framework, formalized in Coq. It targets an assembly-level language with embedded code pointers, thereby supporting both higher-order functions and higher-order stores. XCAP features *impredicative polymorphism*, allowing heap predicates to quantify over heap predicates. This work addresses the same problem as the aforementioned work through a more syntactic approach.

When reasoning about first-class functions, the notion of *nested triple* naturally appears: triples may occur inside the pre- or post-condition of other triples. Nested triples were described in work by Schwinghammer et al. [2009] for functions stored in the heap, and in work by Svendsen et al. [2010] for higher-order functions (more precisely, for delegate functions). Nested triples are naturally supported by shallow embeddings of Separation Logic in higher-order logic proof assistants. This possibility is mentioned explicitly by Wang et al. [2011], but was already implicitly available in earlier formalizations, e.g. [Appel and Blazy, 2007].

Krishnaswami et al. [2010] introduced the idea of a ramified frame rule. The general statement of the ramified rule stated as in Lemma 10.4.1 appeared in [Hobor and Villard, 2013]. Users of the tools VST [Cao et al., 2018b] and Iris [Jung et al., 2017] have advertised for the interest of this rule.

The magic wand between postconditions, written $Q_1 \mathrel{-\!\!\star} Q_2$, as opposed to the use of an explicit quantification $\forall v.\, Q_1\, v \mathrel{-\!\!\star} Q_2\, v$, appears to have first been employed by Bengtson et al. [2012]. This operator is described in the book by Appel et al. [2014]. The five equivalent characterizations of this operator give in Definition 10.3.1 appear to be a (minor) contribution of Charguéraud [2020].

Regarding while loops, the possibility to frame over the remaining iterations (Section 14.3) is inherently available when a loop is encoded as a recursive function, or when a loop is presented in CPS-style—typical with assembly-level code [Ni and Shao, 2006; Chlipala, 2011]. The statement of a reasoning rule directly applicable to a non-encoded loop construct, and allowing to frame over the remaining iterations, has appeared independently in work by Charguéraud [2010] and Tuerk [2010].

A number of interesting extensions of Separation Logic for deterministic sequential programs are beyond the scope of the present survey. Let us cite a few.

- *Fractional permissions* have been introduced by Boyland [2003] in the context of a type system with linear capabilities. Soon afterwards, the idea was identified as essential for specifying concurrent threads in Separation Logic [Bornat et al., 2005]. It appears that fractions may also be useful for reasoning about sequential programs. For example, we use them pervasively for keeping track of pointers when reasoning about space usage in the presence of a garbage collector [Moine et al., 2023].
- The *higher-order frame* [Birkedal et al., 2005, 2006] and the *higher-order anti-frame* [Pottier, 2008; Schwinghammer et al., 2010] allow reasoning about hidden state in sequential programs.
- The notion of *Separation Algebra* [Calcagno et al., 2007; Dockins et al., 2009; Gotsman et al., 2011; Klein et al., 2012] is useful for developing a Separation Logic framework independently of the details of the programming language.
- Costanzo and Shao [2012] present a refined definition of *local reasoning* to ensure that, whenever a program runs safely on some state, adding more state would have no effect on the program's behavior; their definition is useful in particular for nondeterministic programs and programs executed in a finite memory.
- *Fictional Separation Logic* [Jensen and Birkedal, 2012] generalizes the interpretation of separating conjunction beyond physical separation, and explains how to combine several separation algebras.

- *Temporary read-only permissions* [Charguéraud and Pottier, 2017] provide a simpler alternative to fractional permission for manipulating duplicatable read-only resources in a sequential program.
- *Time credits* allow for amortized cost analysis [Charguéraud and Pottier, 2015; Charguéraud and Pottier, 2019]. *Time receipts* provide the dual notion: they may be used to establish lower bounds on the execution time. Mével et al. [2019] and Pottier et al. [2024] formalize time credits and time receipts in Iris. Spies et al. [2021] introduce *transfinite time credits* in Iris for reasoning about the termination of programs whose execution time cannot be bound upfront.

15.3 Mechanized Presentations of Separation Logic

Gordon [1989] presents the first mechanization of Hoare logic in higher-order logic, using the HOL tool. Gordon's pioneering work was followed by numerous formalizations of Hoare logic, targeting various programming languages. Mechanizations of Separation Logic appeared later. Here again, we restrict our discussion to the verification of sequential programs.

Yu et al. [2003, 2004] present the CAP framework, implemented in Coq. It supports reasoning about low-level code using Separation Logic-style rules, and is applied to the verification of a dynamic storage allocation library. Ni and Shao [2006] present the XCAP framework, already mentioned in the previous section, to reason about embedded code pointers. XCAP was also applied to reasoning about x86 context management code [Ni et al., 2007]. Feng et al. [2006] present the SCAP framework, for reasoning about stack-based control abstractions, including exceptions and setjmp/longjmp operations. SCAP is also applied to the verification of Baker's incremental copying garbage collector [McCreight et al., 2007]. Feng et al. [2007] present the OCAP framework that generalizes XCAP for supporting interoperability of different verification systems, including SCAP. Cai et al. [2007] present the GCAP framework for reasoning about self-modifying code, and apply Separation Logic to support local reasoning on both program code and regular data structures. Feng et al. [2008] presents the first verified implementation of a preemptive thread runtime that exploits hardware interrupts; this runtime is linked to verified context switch primitives, using the OCAP and the SCAP frameworks. Wang et al. [2011] present ISCAP, a step-indexed, direct-style operational semantics with support for first-class pointers.

Weber [2004] formalizes in Isabelle/HOL a first-order Separation Logic for a simple while language. This work includes a soundness proof for the frame rule, and the verification of the classic in-place list reversal example.

Preoteasa [2006] formalize in PVS a first-order Separation Logic, with the additional feature that it supports recursive procedures. This work includes the verification of a collection of recursive procedures for computing the parse tree associated with an arithmetic expression.

Marti et al. [2006] formalize in Coq a Separation Logic library, and used it for the verification of the heap manager of an operating system.

Tuch et al. [2007] present a shallow embedding of Separation Logic in Isabelle/HOL, for a subset of the C language, with support for interpreting values at the byte level when required. Their framework is applied to the verification of the memory allocator of a microkernel. Its logic was later extended to support predicates for mapping virtual to physical addresses, and thereby reason about the effects of virtual memory [Kolanski and Klein, 2009]. Klein et al. [2012] present a re-usable library for Separation Algebras, including support for automation.

Appel and Blazy [2007] formalize in Coq a Separation logic for Cminor. This work led to the VST tool, which supports the verification of concurrent C code [Appel, 2011; Appel et al., 2014;

Cao et al., 2018a]. VST leverages step-indexed definitions and features a *later modality* [Hobor et al., 2008; Dockins et al., 2008; Hobor et al., 2010].

Myreen and Gordon [2007] formalize Separation Logic in HOL4. This work eventually lead to the CakeML compiler, described further on.

Varming and Birkedal [2008] demonstrate the possibility to formalize *higher-order* Separation Logic as a shallow embedding in Isabelle/HOLCF.

Nanevski et al. [2008b] and Chlipala et al. [2009] present the Ynot tool, which consists of an axiomatic embedding in Coq of Hoare Type Theory (HTT) [Nanevski et al., 2006, 2008a]. HTT is a presentation of higher-order Separation Logic with higher-order stores in the form of a type system for a dependently typed functional language. In Ynot, like in HTT, a Coq term t admits the Coq type "ST H Q" to express the specification $\{H\}\ t\ \{Q\}$. In Ynot, programs are shallowly embedded in Coq: they are expressed using Coq primitive constructs and axiomatized monadic constructs for effects. The frame rule takes the form of an identity coercion of type $\mathsf{ST}\,H\,Q \to \mathsf{ST}\,(H \star H')\,(\lambda v.\ Q\,v \star H')$. For specifications involving auxiliary variables, Ynot supports *ghost* arguments, which appear like normal function arguments except that they are erased at runtime.

Charguéraud [2011] presents the CFML tool, which supports 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 was extended to support asymptotic cost analysis [Charguéraud and Pottier, 2015; Charguéraud and Pottier, 2019]. CFML initially hard-wired fully-affine triples, featuring unrestricted discard rules, and later integrated the customizable predicate haffine (Chapter 11) [Guéneau et al., 2019].

Tuerk [2011] presents in HOL4 the *Holfoot* tool, formalizing in particular the rules of *Abstract Separation Logic* [Calcagno et al., 2007].

Chlipala [2011, 2013] presents in Coq the Bedrock framework, for the verification of programs written at the assembly level. Bedrock has been, for example, put to practice to verify a cooperative threading library and an implementation of a domain-specific language for XML processing. These software components were interfaced with hardware components of mobile robots [Chlipala, 2015].

Bengtson et al. [2011] present a shallow embedding of higher-order Separation Logic in Coq, demonstrating the use of nested triples for reasoning about object-oriented code. Following up on that work, Bengtson et al. [2012] developed in Coq the *Charge!* tool, which handles a subset of Java.

Jensen et al. [2013] give a modern presentation of a Separation Logic for low-level code, exploiting in particular the (higher-order) frame connective [Birkedal et al., 2005; Birkedal and Yang, 2007; Krishnaswami, 2012]. Building on that work, Kennedy et al. [2013] show how to write assembly syntax and generate x86 machine code inside Coq.

The CakeML verified compiler [Kumar et al., 2014], implemented in HOL, takes SML-like programs as input and produces machine code as output. It exploits Separation Logic to prove the garbage collector [Sandberg Ericsson et al., 2019]. It also exploits Separation Logic to set up a CFML-style characteristic formulae generator, extended with support for catchable exceptions and I/O [Guéneau et al., 2017]. The characteristic formulae are used to verify the standard library for CakeML.

The Iris framework [Jung et al., 2015, 2016; Krebbers et al., 2017; Jung et al., 2017, 2018], implemented in Coq, supports higher-order concurrent Separation Logic. Like VST, Iris features a later modality and step-indexed definitions. Iris exploits weakest-precondition style reasoning rules (Chapter 8) and function specifications are stated as in Lemma 8.4.2, although using syntac-

tic sugar to make specifications resemble conventional triples. Iris is defined as a fully-affine logic, with an affine entailment. Tassarotti et al. [2017] present an extension of Iris featuring linear heap predicates, and an *affine modality* written $\mathcal{A}(H)$. An alternative approach is proposed by Bizjak et al. [2019], who present the encoding on top of Iris of two logics that enable tracking of linear resources, transferable among dynamically allocated threads. The first one, called Iron, leverages fractional permissions to encode *trackable resources*, and allow, e.g., reasoning about deallocation of shared resources. The second one, called Iron++, hides away the use of fractions, and offers the user with the illusion of a *linear* Separation Logic with support for *trackable invariants*. Spies et al. [2021] extend Iris with *transfinite time credits* for, in particular, reasoning about termination.

The Mosel framework [Krebbers et al., 2018] generalizes Iris' tooling to a large class of separation logics, targeting both affine and linear separation logics, and combinations thereof.

Bannister et al. [2018] discuss techniques for forward and backward reasoning in Separation Logic. Their work, presented in Isabelle/HOL, introduces the *separating coimplication* operator to improve automation. Separating coimplication is the dual of separating conjunction, just like *septraction* [Vafeiadis and Parkinson, 2007] is the dual of separating implication. Separating coimplication forms a Galois connection with septraction, just like separating conjunction forms a Galois connection with separating implication.

Lammich [2019b] present a refinement framework that leverages Separation Logic to refine from Isabelle/HOL definitions to verified code in LLVM intermediate representation. It is applied to the production of a number of algorithms, including an efficient KMP string search implementation [Lammich, 2019a].

15.4 Course Notes 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 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]. More recently, Appel and Cao [2020] published a volume part of the Software Foundations series, entitled *Verifiable C*. This volume is a tutorial for VST [Cao et al., 2018a], a tool that supports reasoning about actual C code. As of 2022, the tutorial covers the verification of data structures, including linked lists, stacks, hashtables, as well as string-manipulating functions. 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.

The Iris tutorial by Birkedal and Bizjak [2018] presents the core ideas of Iris' concurrent Separation Logic [Krebbers et al., 2017; Jung et al., 2018]. 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 only the case of an affine logic is covered. Dietrich [2021] wrote, as part of her Bachelor's thesis, *A beginner's guide to Iris, Coq and Separation Logic*. It provides a gentle

introduction on how to use the framework in practice, illustrated with a few case studies.

Chlipala's course notes [Chlipala, 2018a] feature a chapter on Separation Logic, accompanied with a corresponding Coq formalization meant to be followed by students [Chlipala, 2018b]. The material includes a proof of soundness, as well as the verification of a few example programs. Chlipala's chapter focuses on the core of Separation Logic—it does not cover any of the enhancements listed in the introduction. 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 cmd α. Altogether, this design allows for a concise formalization of the source language, yet, we believe, at the price 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]. 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. Chlipala's approach appears 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, a total correctness proof carried out with respect to a big-step semantics yields a stronger result, via a simpler proof.

In 2020, Arthur Charguéraud released the first version of an all-in-Coq course entitled *Foundations of Separation Logic* [Charguéraud, 2021]. It is distributed as Volume 6 of the *Software Foundations Series*, edited by Benjamin C. Pierce. A second version was released in 2023, together with a first version of the the present companion document. An earlier version of this material appeared in Charguéraud's habilitation manuscript [2023].

Bibliography

Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004. URL http://www.cs.indiana.edu/~amal/ahmedsthesis.pdf.

Andrew W. Appel. 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, page 1–17, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642197178. URL https://doi.org/10.1007/978-3-642-28891-3_2.

Andrew W Appel. *Program logics for certified compilers*. Cambridge University Press, 2014. URL https://doi.org/10.1017/CBO9781107256552. With Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy.

Andrew W Appel and Sandrine Blazy. Separation logic for small-step Cminor. In Klaus Schneider and Jens Brandt, editors, *International Conference on Theorem Proving in Higher Order Logics*, pages 5–21, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74591-4. URL https://doi.org/10.1007/978-3-540-74591-4_3.

Andrew W. Appel and Qinxiang Cao. *Verifiable C*, volume 5beta of *Software Foundations*. Electronic textbook, 2020. URL http://softwarefoundations.cis.upenn.edu. Version 0.9.5.

Andrew W. Appel and David McAllester. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001. ISSN 0164-0925. doi: 10.1145/504709.504712. URL https://doi.org/10.1145/504709.504712.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 109–122, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1595935754. doi: 10.1145/1190216.1190235. URL https://doi.org/10.1145/1190216.1190235.

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, USA, 2014. ISBN 110704801X. URL https://doi.org/10.1017/CBO9781107256552.

Callum Bannister, Peter Höfner, and Gerwin Klein. Backwards and Forwards with Separation Logic. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 68–87, Cham, 2018. Springer International Publishing. ISBN 978-3-319-94821-8. URL https://doi.org/10.1007/978-3-319-94821-8_5.

Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying Object-Oriented Programs with Higher-Order Separation Logic in Coq. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 22–38, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22863-6. URL https://doi.org/10.1007/978-3-642-22863-6_5.

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 315–331, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32347-8. URL https://doi.org/10.1007/978-3-642-32347-8_21.

Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI Hyperdoctrines and Higher-Order Separation Logic. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, page 233–247, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540254358. doi: 10.1007/978-3-540-31987-0_17. URL https://doi.org/10.1007/978-3-540-31987-0_17.

Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-Hyperdoctrines, Higher-Order Separation Logic,

and Abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24–es, August 2007. ISSN 0164-0925. doi: 10.1145/1275497.1275499. URL https://doi.org/10.1145/1275497.1275499.

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

Lars Birkedal and Hongseok Yang. Relational Parametricity and Separation Logic. In Helmut Seidl, editor, *Foundations of Software Science and Computational Structures*, pages 93–107, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71389-0. URL https://doi.org/10.1007/978-3-540-71389-0_8.

Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 260–269. IEEE, 2005. URL https://doi.org/10.1109/LICS.2005.47.

Lars Birkedal, Noah Torp-smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for algol-like languages. volume 2. Logical Methods in Computer Science e.V., Nov 2006. doi: 10.2168/lmcs-2(5:1)2006. URL http://dx.doi.org/10.2168/LMCS-2(5:1)2006.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. A Simple Model of Separation Logic for Higher-Order Store. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming (ICALP)*, pages 348–360, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70583-3. URL https://doi.org/10.1007/978-3-540-70583-3_29.

Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290378. URL https://doi.org/10.1145/3290378.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*, pages 259–270, January 2005. URL http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/permissions_paper.pdf.

John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, June 2003. URL http://www.cs.uwm.edu/~boyland/papers/permissions.pdf.

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

Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified Self-Modifying Code. *SIGPLAN Not.*, 42(6): 66–77, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250743. URL https://doi.org/10.1145/1273442.1250743.

Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM)*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465. Springer, April 2011. URL http://www.eecs.qmul.ac.uk/~ddino/papers/nasa-infer.pdf.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local Action and Abstract Separation Logic. In *Logic in Computer Science (LICS)*, pages 366–378, July 2007. URL http://www.doc.ic.ac.uk/~ccris/ftp/asl-short.pdf.

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing. ISBN 978-3-319-17524-9. URL https://doi.org/10.1007/978-3-319-17524-9_1.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1-4):367–422, 2018a. URL https://doi.org/10.1007/s10817-018-9457-5.

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

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. Applying Formal Verification to Microkernel IPC at Meta. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 116–129, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391825. doi: 10.1145/3497775.3503681. URL https://doi.org/10.1145/3497775.3503681.

Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi: 10.1145/2034773.2034828. URL https://doi.org/10.1145/2034773.2034828.

Arthur Charguéraud. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408998. URL https://doi.org/10.1145/3408998.

Arthur Charguéraud. *A Modern Eye on Separation Logic for Sequential Programs*. Habilitation à diriger des recherches, Université de Strasbourg, February 2023. URL https://inria.hal.science/tel-04076725.

Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning (JAR)*, 62 (3):331–365, March 2019. ISSN 0168-7433. doi: 10.1007/s10817-017-9431-7. URL https://doi.org/10.1007/s10817-017-9431-7.

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. Omnisemantics: Smooth Handling of Nondeterminism. To appear in ACM Transactions on Programming Languages and Systems (TOPLAS), September 2022. URL https://hal.inria.fr/hal-03255472.

Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, December 2010. URL http://www.charguer aud.org/research/2010/thesis/thesis_final.pdf.

Arthur Charguéraud. *Separation Logic Foundations*, volume 6 of *Software Foundations*. 2021. http://softwarefoundations.cis.upenn.edu.

Arthur Charguéraud and François Pottier. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 137–153. Springer, August 2015. URL http://cambium.inria.fr/~fpottier/publis/charguer aud-pottier-uf.pdf.

Arthur Charguéraud and François Pottier. Temporary Read-Only Permissions for Separation Logic. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 260–286. Springer, April 2017. URL http://cambium.inria.fr/~fpottier/publis/charguer aud-pottier-slro.pdf.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815402. URL https://doi.org/10.1145/2815400.2815402.

Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, June 2011. URL http://adam.chlipala.net/papers/BedrockPLDI11/BedrockPLDI11.pdf.

Adam Chlipala. 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*, volume 48, page 391–402, New York, NY, USA, September 2013. Association for Computing Machinery. doi: 10.1145/2544174.2500592. URL https://doi.org/10.1145/2544174.2500592.

Adam Chlipala. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *Principles of Programming Languages (POPL)*, pages 609–622, January 2015. URL http://adam.chlipala.net/papers/BedrockPOPL15/BedrockPOPL15.pdf.

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

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

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-Order Imperative Programs. In *ACM International Conference on Functional Programming (ICFP)*, ICFP '09, page 79–90, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583327. doi: 10.1145/1596550.1596565. URL https://doi.org/10.1145/1596550.1596565.

David Costanzo and Zhong Shao. A Case for Behavior-Preserving Actions in Separation Logic. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 332–349, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35182-2. doi: 10.1007/978-3-642-35182-2_24.

Elizabeth Dietrich. A beginner guide to Iris, Coq and separation logic. *CoRR*, abs/2105.12077, 2021. URL https://arxiv.org/abs/2105.12077.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. URL http://doi.acm.org/10.1145/360933.360975.

Robert Dockins, Andrew W. Appel, and Aquinas Hobor. Multimodal Separation Logic for Reasoning About Operational Semantics. *Electronic Notes in Theoretical Computer Science*, 218:5 – 20, 2008. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2008.10.002. URL http://www.sciencedirect.com/science/article/pii/S1571066108003964. Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 161–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10672-9. URL https://doi.org/10.1007/978-3-642-10672-9_13.

Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. *SIGPLAN Not.*, 41(6):401–414, June 2006. ISSN 0362-1340. doi: 10.1145/1133255.1134028. URL https://doi.org/10.1145/1133255.1134028.

Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An Open Framework for Foundational Proof-Carrying Code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 67–78, New York, NY, USA, January 2007. ACM Press. URL https://doi.org/10.1145/1190315.1190325.

Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 170–182, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375603. URL https://doi.org/10.1145/1375581.1375603.

R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967. URL https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf.

Google. Announcing KataOS and Sparrow, oct 2022. URL https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html.

Michael J. C. Gordon. *Mechanizing Programming Logics in Higher Order Logic*, page 387–439. Springer-Verlag, Berlin, Heidelberg, 1989. ISBN 0387969888. URL https://doi.org/10.1007/978-1-4612-3658-0_10.

Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electronic Notes in Theoretical Computer Science*, 276:171–190, 2011. URL http://www0.cs.ucl.ac.uk/staff/b.cook/pdfs/precision_and_the_conjunction_rule_in_concurrent_seperation_logic.pdf.

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, pages 584–610, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54434-1. URL https://doi.org/10.1007/978-3-662-54434-1_22.

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *Interactive Theorem Proving (ITP)*, volume 141 of *Leibniz International Proceedings in Informatics*, pages 18:1–18:20, September 2019. URL http://cambium.inria.fr/~fpottier/publis/gueneau-jourdan-chargueraud-pottier-2019.pdf.

Maximilian P. L. Haslbeck and Peter Lammich. For a Few Dollars More - Verified Fine-Grained Algorithm Analysis Down to LLVM. In *European Symposium on Programming (ESOP)*, volume 12648 of *Lecture Notes in Computer Science*, pages 292–319. Springer, March 2021. URL https://www21.in.tum.de/~haslbema/documents/Haslbeck_Lammich_LLVM_with_Time.pdf.

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

Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 523–536, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318327. doi: 10.1145/2429069.2429131. URL https://doi.org/10.1145/2429069.2429131.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, pages 353–367, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78739-6. URL https://doi.org/10.1007/978-3-540-78739-6_27.

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A Theory of Indirection via Approximation. *SIGPLAN Not.*, 45(1):171–184, January 2010. ISSN 0362-1340. doi: 10.1145/1707801.1706322. URL https://doi.org/10.1145/1707801.1706322.

Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages (POPL)*, pages 14–26, January 2001. URL http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/bi-assertion-lan.pdf.

Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-Level Separation Logic for Low-Level Code. *SIGPLAN Not.*, 48(1):301–314, January 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429105. URL https://doi.org/10.1145/2480359.2429105.

Jonas Braband Jensen and Lars Birkedal. Fictional Separation Logic. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 377–396, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28869-2. URL https://doi.org/10.1007/978-3-642-28869-2_19.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-Order Ghost State. *SIGPLAN Not.*, 51(9):256–269, September 2016. ISSN 0362-1340. doi: 10.1145/3022670.2951943. URL https://doi.org/10.1145/3022670.2951943.

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

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018. URL https://doi.org/10.1017/S0956796818000151.

Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. Coq: The World's Best Macro Assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, page 13–24, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321549. doi: 10.1145/2505879.2505897. URL https://doi.org/10.1145/2505879.2505897.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6): 107–115, 2010. URL http://ertos.nicta.com.au/publications/papers/Klein_EHACDEEKNSTW_10.pdf.

Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised Separation Algebra. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 332–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32347-8. URL https://doi.org/10.1007/978-3-642-32347-8_22.

Rafal Kolanski and Gerwin Klein. Types, Maps and Separation Logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 276–292, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9. URL https://doi.org/10.1007/978-3-642-03359-9_20.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*, page 696–723, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 9783662544334. doi: 10.1007/978-3-662-54434-1_26. URL https://doi.org/10.1007/978-3-662-54434-1_26.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi: 10.1145/3236772. URL https://doi.org/10.1145/3236772.

Neel R. Krishnaswami, Lars Birkedal, and Jonathan Aldrich. Verifying Event-Driven Programs Using Ramified Frame Properties. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, page 63–76, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588919. doi: 10.1145/1708016.1708025. URL https://doi.org/10.1145/1708016.1708025.

Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2012. URL http://www.cs.cmu.edu/~neelk/thesis.pdf.

Neelakantan R. Krishnaswami, Jonathan Aldrich, and Lars Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *In Proceedings of Formal Techniques for Java-like Programs (FTfJP)*, 2007.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–191. ACM Press, January 2014. doi: 10.1145/2535838.2535841. URL https://cakeml.org/popl14.pdf.

Peter Lammich. Refinement to Imperative HOL. *Journal of Automated Reasoning*, 62(4):481–503, April 2019a. URL https://www21.in.tum.de/~lammich/pub/jar_ref_imp_hol.pdf.

Peter Lammich. Refinement to Imperative HOL. *Journal of Automated Reasoning (JAR)*, 62(4):481–503, April 2019b. ISSN 0168-7433. doi: 10.1007/s10817-017-9437-1. URL https://doi.org/10.1007/s10817-017-9437-1.

Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. 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, page 400–419, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540474609. doi: 10.1007/11901433_22. URL https://doi.org/10.1007/11901433_22.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A General Framework for Certifying Garbage Collectors and Their Mutators. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 468–479, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250788. URL https://doi.org/10.1145/1250734.1250788.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020. doi: 10.1145/3408978. URL https://doi.org/10.1145/3408978.

Alexandre Moine, Arthur Charguéraud, and François Pottier. A High-Level Separation Logic for Heap Space under Garbage Collection. To appear at POPL'23, January 2023. URL http://www.charguer aud.org/research/2022/space_with_gc/space_with_gc.pdf.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49122-5. doi: 10.1007/978-3-662-49122-5_2.

Magnus O Myreen. *Formal verification of machine-code programs*. PhD thesis, December 2008.

Magnus O. Myreen and Michael J. C. Gordon. 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, page 568–582, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540712084. URL https://doi.org/10.1007/978-3-540-71209-1_44.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 1–27. Springer, April 2019. URL http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-time-in-iris-2019.pdf.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. *SIGPLAN Not.*, 41(9):62–73, September 2006. ISSN 0362-1340. doi: 10.1145/1160074.1159812. URL https://doi.org/10.1145/1160074.1159812.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare Type Theory, Polymorphism and Separation. *J. Funct. Program.*, 18(5–6):865–911, September 2008a. ISSN 0956-7968. doi: 10.1017/S0956796808006953. URL https://doi.org/10.1017/S0956796808006953.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent Types for Imperative Programs. *SIGPLAN Not.*, 43(9):229–240, September 2008b. ISSN 0362-1340. doi: 10.1145/1411203.1411237. URL https://doi.org/10.1145/1411203.1411237.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

Zhaozhong Ni and Zhong Shao. Certified Assembly Programming with Embedded Code Pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 320–333, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi: 10.1145/1111037.1111066. URL https://doi.org/10.1145/1111037.1111066.

Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to Certify Realistic Systems Code: Machine Context Management. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 189–206, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74591-4. URL https://doi.org/10.1007/978-3-540-74591-4_15.

O'Hearn, Reynolds, and Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL: 15th Workshop on Computer Science Logic*. LNCS, Springer-Verlag, 2001. URL https://doi.org/10.1007/3-540-44802-0_1.

Peter W. O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019. doi: https://doi.org/10.1145/3211968. URL https://dl.acm.org/doi/10.1145/3211968. The appendix is linked as supplementary material from the ACM digital library.

Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic*, 5 (2):215–244, 1999. ISSN 10798986. URL https://doi.org/10.2307/421090.

Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software Verification with VeriFast: Industrial Case Studies. *Sci. Comput. Program.*, 82:77–97, March 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.01.006. URL https://doi.org/10.1016/j.scico.2013.01.006.

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. Thunks and Debits in Separation Logic with Time Credits. In *POPL 2024 - 51st ACM SIGPLAN Symposium on Principles of Programming Languages*, Londres, United Kingdom, January 2024. SIGPLAN, ACM. URL https://hal.science/hal-04238691.

François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *IEEE Symposium on Logic In Computer Science (LICS)*, pages 331–340, Pittsburgh, Pennsylvania, June 2008. URL https://doi.org/10.1109/LICS.2008.16.

François Pottier. Verifying a hash table and its iterators in higher-order separation logic. In *Certified Programs and Proofs (CPP)*, pages 3–16, January 2017. URL http://cambium.inria.fr/~fpottier/publis/fpottier-hashtable.pdf.

Viorel Preoteasa. Mechanical Verification of Recursive Procedures Manipulating Pointers Using Separation Logic. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 508–523, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37216-5. URL https://doi.org/10.1007/11813040_34.

Bernhard Reus and Jan Schwinghammer. Separation Logic for Higher-Order Store. In Zoltán Ésik, editor, *Computer Science Logic*, pages 575–590, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-45459-5. URL https://doi.org/10.1007/11874683_38.

John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

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

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning (JAR)*, 63, 2019. doi: 10.1007/s10817-018-9487-z. URL https://link.springer.com/content/pdf/10.1007%2Fs10817-018-9487-z.pdf.

Steven Schäfer, Sigurd Schneider, and Gert Smolka. Axiomatic Semantics for Compiler Verification. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 188–196, St. Petersburg FL USA, January 2016. ACM. ISBN 978-1-4503-4127-1. doi: 10.1145/2854065.2854083. https://dl.acm.org/doi/10.1145/2854065.2854083.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic*,

pages 440–454, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04027-6. URL https://doi.org/10.1007/978-3-642-04027-6_32.

Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. A Semantic Foundation for Hidden State. In Luke Ong, editor, *Foundations of Software Science and Computational Structures*, pages 2–17, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12032-9. URL https://doi.org/10.1007/978-3-642-12032-9_2.

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 80–95, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454031. URL https://doi.org/10.1145/3453483.3454031.

Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying Generics and Delegates. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 175–199, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14107-2. URL https://doi.org/10.1007/978-3-642-14107-2_9.

Joseph Tassarotti, Ralf Jung, and Robert Harper. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 909–936. Springer, April 2017. URL https://iris-project.org/pdfs/2017-esop-refinement-final.pdf.

Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, Bytes, and Separation Logic. *SIGPLAN Not.*, 42 (1):97–108, January 2007. ISSN 0362-1340. doi: 10.1145/1190215.1190234. URL https://doi.org/10.1145/1190215.1190234.

Thomas Tuerk. Local reasoning about while-loops. Unpublished, August 2010. URL http://www.cl.cam.ac.uk/~tt291/talks/vstte10.pdf.

Thomas Tuerk. A separation logic framework for HOL. Technical Report UCAM-CL-TR-799, University of Cambridge, Computer Laboratory, June 2011. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-799.pdf.

Viktor Vafeiadis and Matthew Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74407-8. URL https://doi.org/10.1007/978-3-540-74407-8_18.

Carsten Varming and Lars Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science*, 218:371 – 389, 2008. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2008.10.022. URL http://www.sciencedirect.com/science/article/pii/S1571066108004167. Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

Wei Wang, Zhong Shao, Xinyu Jiang, and Yu Guo. A Simple Model for Certifying Assembly Programs with First-Class Function Pointers. In Zhenhua Duan and C.-H. Luke Ong, editors, *5th IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2011, Xi'an, China, 29-31 August 2011*, pages 125–132. IEEE Computer Society, 2011. doi: 10.1109/TASE.2011.16. URL https://doi.org/10.1109/TASE.2011.16.

Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, pages 250–264, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30124-0. URL https://doi.org/10.1007/978-3-540-30124-0_21.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive OS kernels. In Swarat Chaudhuri and Azadeh Farzan, editors, *International Conference on Computer Aided Verification*, pages 59–79, Cham, 2016. Springer, Springer International Publishing. URL https://doi.org/10.1007/978-3-319-41540-6_4.

Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. In Pierpaolo Degano, editor, *Programming Languages and Systems*, pages 363–379, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36575-4. URL https://doi.org/10.1007/3-540-36575-3_25.

Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building Certified Libraries for PCC: Dynamic Storage Allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004. URL https://doi.org/10.1007/3-540-36575-3_25.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48153-9.