A Modern Eye on Separation Logic for Sequential Programs

Arthur Charguéraud

UNIVERSITÉ DE STRASBOURG

École Doctorale:
Mathématiques, sciences de l'information et de l'ingénieur (ED 269)

Habilitation Manuscript

A Modern Eye on Separation Logic for Sequential Programs

Arthur Charguéraud

Febuary 27th, 2023

Jury

Sandrine	Blazy	Examiner	Professeur des universités	Université Rennes 1
Dominique	Devriese	Reviewer	Professor	KU Leuven
Derek	Dreyer	Examiner	Professor	Max Planck Institute (MPI-SWS)
Robert	Harper	Reviewer	Professor	Carnegie Mellon University
Philippe	Helluy	Examiner	Professeur des universités	Université de Strasbourg
Nicolas	Magaud	Examiner	Maître de conférences HDR	Université de Strasbourg
Xavier	Rival	Reviewer	Directeur de recherche	Inria

Abstract

Separation Logic brought a major breakthrough in the area of program verification. Since its introduction, Separation Logic has made its way into a number of practical tools that are used on a daily basis for verifying programs, ranging from operating systems kernels and file systems to data structures and graph algorithms. These programs are written in a wide variety of programming languages at different abstraction levels, ranging from machine code and assembly, to C, Java, OCaml, and Rust, just to name a few. Numerous extensions to Separation Logic have been proposed over the past two decades. In this habilitation manuscript, I present an overview of my own contributions—and that of my co-authors—over the period from 2009 to 2022.

The manuscript is organized in three main parts. The first part describes a foundational set up of Separation Logic, with the logic being proved sound with respect to a semantics mechanized in an interactive proof assistant. The presentation targets an imperative λ-calculus, sufficiently minimalistic to allow for an easy-to-teach presentation of the theory, yet sufficiently rich to support the verification of realistic programs. The second part presents the technique of characteristic formulae, which enables smooth proofs of practical programs in a proof assistant. Compared with the characteristic formulae introduced in my PhD thesis, I here give a simplified presentation based on weakest preconditions and, most importantly, I show how to justify characteristic formulae in a foundational manner. The third part of this manuscript describes extensions to Separation Logic for resource analysis: time credits for establishing amortized execution bounds, big-O notation to support asymptotic reasoning, and space credits to establish space bounds in the presence of a garbage collector.

The manuscript ends with two closing chapters. One provides a survey of publications on Separation Logic for sequential programs. The other covers research perspectives.

Table of Contents

Chapter 1

Introduction

1.1 Context

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. Let me cite just a few examples. 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, I 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.

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 analyse resource consumption, to verify time-channel attacks, etc. My research is concerned with *deductive program verification*, which 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. My work has focused on providing interactive proofs for establishing, in a foundational manner, functional correctness and termination, as well as bounds on the asymptotic execution time and on the space usage.

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., 2019a; 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.

I started working on Separation Logic during my PhD, in 2009. Since then, my work has focused on the development of practical techniques for reasoning about sequential programs, with applications in particular to data structures and algorithms. My contributions to Separation Logic is 3-fold.

1. I have contributed to extensions of Separation Logic, and to simplifications of its formalization. The major contributions are the following. I have developed *characteristic formulae* as a way to smoothly integrate Separation Logic in interactive proof assistants. By being grounded on a non-deterministic big-step-style semantics, this approach inherently supports reasoning about termination. I have introduced a feature for accommodating both *linear* and *affine* predicates in a lightweight manner. I have proposed *higher-order representation predicates* for specifying polymorphic containers that may store other mutable objects, including other containers. Besides, together with François Pottier and our students Armaël Guéneau and Alexandre Moine, we have developed extensions of Separation Logic for reasoning about resource consumption, establishing asymptotic time bounds, with support for the big-O notation, and establishing bounds on the space usage in the presence of a garbage collector.
2. I am the lead developer of an interactive program verification tool, called CFML. This tool has been used to implement and validate the aforementioned contributions, with the exception of space bounds for which the Iris framework [Jung et al., 2018b] has been used. This tool consists, on the one hand, of a *characteristic formulae generator*, which parses OCaml syntax and produces logical formulae; and, on the other hand, of a Coq library that provides definitions, lemmas, and tactics for carrying out interactive program verification proofs. CFML has been used to verify several thousand lines of OCaml in total.
3. I have written 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 only assumes as prerequisite the contents of Volume 1 (Logical Foundations) and Volume 2 (Programming Language Foundations). The current version contains 13 chapters, 140 exercises, and covers most of the material from Chapters 2, 3, 4 and 5 of the present manuscript, as well as one chapter covering the basics of representation predicates. I have recently started the writing of a second book, focused on the practice of specifying and verifying data structures and algorithms.

In summary, my work on Separation Logic consists of (1) extending or simplifying the theory of Separation Logic, and developing new specification patterns, (2) putting all the new ideas to practice for verifying actual implementations, and (3) writing pedagogical material to share knowledge on Separation Logic.

1.3 Contents

Foundations. Chapter 2 gives a streamlined presentation of the core ideas of Separation Logic. The presentation is carried out using a sequential, deterministic, imperative λ-calculus. The absence of mutable variables considerably simplifies the formalization of the reasoning rule of the logic. Starting from a big-step operational semantics, I derive reasoning rules in the form of lemmas, expressed either in the forms of triples, or in weakest-precondition style. This chapter does not contain novel ideas, but gathers in one place a state-of-the-art presentation of Separation Logic

that, I believe, will prove useful for teaching purpose. This Chapter also serves as a basis for the rest of the manuscript.

Language extensions. Chapter 3 presents techniques involved for reasoning in Separation Logic about a richer programming language, in which realistic programs can be written. It covers a number of extensions, such as n-ary functions, loops, arrays, records, dynamic checks, and handling of programs that are not in A-normal form. Besides, I explain how to treat programming languages equipped with a garbage collector. To that end, I present a *partially-affine* program logic, which allows to freely discard certain classes—and only certain classes—of heap predicates.

Omnisemantics. The foundational definition of Separation Logic triples presented in Chapter 2 applies only to deterministic semantics or, technically, to semantics that are deterministic up to the choice of fresh memory locations. In Chapter 4, I explain how to define foundational triples for the more general case of nondeterministic semantics. The construction is based on the inductively defined *omni-big-step* judgment, written $t/s \Downarrow Q$. This judgment asserts that every possible evaluation starting from the configuration t/s reaches a final configuration that belongs to the set Q. This set Q is isomorphic to a postcondition. As I show, the omni-big-step judgment directly yields a definition of a weakest-precondition operator that inherently satisfies the frame rule.

Characteristic formulae. Chapter 5 describes the technique of *characteristic formulae*, whose purpose is to smoothly integrate Separation Logic in an interactive proof assistant. A characteristic formulae generator is a function that, given a program without any specification or invariant, computes its weakest precondition by recursion over its syntax. One may view the computation of a characteristic formula as a *most-general weakest-precondition calculus*. Compared with characteristic formulae introduced in my PhD thesis, there are two major novelties. First, I generate characteristic formulae using a function that computes inside Coq, as opposed to using an external tool. This function is proved correct once and for all, thus every formula produced is inherently sound with respect to the term it describes. Second, I switched from characteristic formulae that operate on triples, with a precondition and a postcondition, to formulae that operate only on postconditions, in weakest-precondition style. This technical change significantly simplifies the presentation.

The lifting technique. Chapter 6 presents a technique, called *lifting*, for specifying program values using logical values, i.e. Coq values. With this technique, the user never sees deeply embedded syntax for terms and values. The user can work with typed Coq values, and with representation predicates over such values. Moreover, numerous proof steps can be eliminated when working with lifted characteristic formulae. Like in my PhD work, to exploit the lifting technique, I rely on an external characteristic formulae generator. In that prior work, characteristic formulae were generated as axioms. The key challenge that I have solved in the recent years is to find a way to justify the correctness of lifted characteristic formulae in a foundational way with respect to the operational semantics of the programming language.

Resource analysis. The most common aim of program verification is to establish the *safety* and *functional correctness* of a program, that is, to prove that this program does not crash and computes a correct result. Beyond safety and functional correctness, it may be desirable to establish bounds on *resource consumption*, that is, to prove that the resource requirements of a program do not exceed a certain predictable bound. Indeed, a program that requires an unexpectedly large amount of *time* may be unresponsive. A program that requires an unexpectedly large amount of *stack space*

may crash with a stack overflow. A program that requires an unexpectedly large amount of *heap space* may exhaust the available memory and make the system unstable. Chapter 7 is concerned with extensions of Separation Logic for establishing bounds on resource consumption.

Asymptotic notation. Chapter 8 presents the challenges and key ideas associated with formalization of the asymptotic *big-O* notation, pervasively used in algorithms textbooks. Reasoning and working with asymptotic complexity bounds is not as simple as one might hope, especially when several variables are involved in the bounds—i.e., in the *multivariate* case. As I illustrate through several examples, typical paper proofs using the big-O notation rely on informal reasoning principles, which can easily be abused to prove a contradiction. I explain how to formally state specifications featuring asymptotic bounds, and how to establish such bounds in practice.

Garbage-collected space. The approach for resource analysis presented in Chapter 7 applies to different kinds of resources. The requirement is that it must be evident at which program points these resources are introduced and consumed. This requirement is *not* met by heap space in the presence of garbage collection. Indeed, when using a garbage collector, the programmer does not include explicit deallocation instructions in the code. Thus, reasoning about garbage-collected heap space poses unique challenges. Chapter 9 presents the key ideas associated with the setup of a Separation Logic with *space credits* that allows establishing space bounds for programs equipped with a garbage collector. Doing so involves, in particular, reasoning about *roots* and about *unreachability*.

Conclusion. This manuscript ends with two closing chapters. Chapter 10 gives an historical account of the contributions to Separation Logic for sequential programs. Chapter 11 discusses the research perspectives that I am particularly interested in investigating in the next decade.

Chapter 2

Foundations of Separation Logic

I begin with an overview of the key features of Separation Logic (Section 2.1). I then present the operators of the logic (Section 2.2), the syntax and semantics of the language used for the formal presentation (Section 2.3), and the statement of the reasoning rules (Section 2.4). I complete this chapter with the description of two key Separation Logic ingredients: the magic wand operator (Section 2.5), and the formulation of reasoning rules in weakest-precondition style (Section 2.6).

The contents of this chapter is an excerpt from my ICFP'20 paper [Charguéraud, 2020]. In this chapter and the next one, unless stated otherwise, all the material presented is standard knowledge of the Separation Logic literature. I discuss the origins of every ingredient in Chapter 10. My contribution here has been to assemble a streamlined presentation of Separation Logic.

2.1 Overview of the Features of Separation Logic

This first section 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.

2.1.1 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 2.3.1.) 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:

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

2.1.2 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 2.2.2).

- $p \hookrightarrow v$, 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 2.1.1 (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 $\texttt{incr}\, p$, namely the unit value, needs not be assigned a name.

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

2.1.3 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 2.1.2 (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 2.1.1 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:

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

2.1.4 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 2.1.3 (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. $\{p \hookrightarrow n\}\ (\texttt{incr\_two}\ p\ p)\ \{\lambda\_.\ (p \hookrightarrow n+2)\}$

2.1.5 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 2.1.4 (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 `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 2.3.1). The operation `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 `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 `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 2.1.5 (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.

2.1.6 Representation of Mutable Lists

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

Definition 2.1.1 (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 "$\mathit{head} \equiv 0$" and "$\mathit{tail} \equiv 1$" 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 2.1.2 (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 2.1.6 (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 2.1.2 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 2.1.3 (Alternative definition for Mlist)

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

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 2.1.2.

2.1.7 Operations on Mutable Lists

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

Example 2.1.7 (Specification of in-place append for mutable lists)

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

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 $\mathsf{Mlist}\, L_2\, p_2$ 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:*

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

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 $\mathsf{Mlist}\, L\, p$, and the postcondition describes the output heap as $\mathsf{Mlist}\, L\, p \star \mathsf{Mlist}\, L\, p'$, where p' denotes the address of the new list.

Example 2.1.8 (Specification of a copy function for mutable lists)

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

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

Proof *The specification of $\mathtt{mcopy}$ 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 $\mathsf{Mlist}\, \mathit{nil}\, p'$ is equivalent to the empty heap predicate. When the list is nonempty, $\mathsf{Mlist}\, L\, p$ 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 $\mathtt{mcopy}$ on the tail of the list, with the precondition $\mathsf{Mlist}\, L'\, q$. Over the scope of that call, the frame rule is used to put aside the head cell, described by $(p.\mathit{head} \hookrightarrow x) \star (p.\mathit{tail} \hookrightarrow q)$. 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.[1] 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.

2.1.8 Reasoning about Deallocation

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

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

Definition 2.1.4 (Specification of the free operation)

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

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 2.1.9 (Deallocation of a list cell) *The function `mfree_cell` deallocates a list cell.*

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

Example 2.1.10 (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.*

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

[1] Details of the proof of the list copy function may be found in Appendix E of my ICFP'20 paper. The Coq formalization appears in lemma `triple_mcopy` from: https://softwarefoundations.cis.upenn.edu/slf-current/Repr.html

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 Section 3.1).*

2.2 Heap Predicates and Entailment

2.2.1 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 Section 2.3.

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 $\alpha\,\beta$ denote the type of finite maps from a type α to an (inhabited) type β.

Definition 2.2.1 (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.

2.2.2 Heap Predicates

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

Definition 2.2.2 (Heap predicates) *A heap predicate is a predicate of type: state → Prop.*

The *core heap predicate operators*, informally introduced in Section 2.1.2, are realized as predicates over heaps, as shown below and explained next.

Definition 2.2.3 (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 → 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 $[\mathit{True}]$.*

Remark (Other heap predicate operators) *Traditional presentations of Separation Logic include four additional operators, $\bot$, $\top$, $\curlyvee$, and $\curlywedge$. These four operators may be encoded in terms of the ones from Definition 2.2.3, with the help of Coq's conditional construct. The table below presents the relevant encodings, in addition to providing direct definitions of these operators as predicates over heaps.*

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

Definition 2.2.4 (Representation predicate for lists defined with disjunction) *The representation predicate for lists introduced in Definition 2.1.6 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.*

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

2.2.3 Entailment

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

Definition 2.2.5 (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 2.2.1 (Entailment defines an order on the set of heap predicates)

$$\textit{HIMPL-REFL}\ \frac{}{H \vdash H} \qquad \textit{HIMPL-TRANS}\ \frac{H_1 \vdash H_2 \qquad H_2 \vdash H_3}{H_1 \vdash H_3} \qquad \textit{HIMPL-ANTISYM}\ \frac{H_1 \vdash H_2 \qquad H_2 \vdash H_1}{H_1 = H_2}$$

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

[2] 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

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 2.2.2 (Fundamental properties of the star)

$$\begin{array}{lrcll}
\textit{STAR-ASSOC:} & (H_1 \star H_2) \star H_3 & = & H_1 \star (H_2 \star H_3) & \\
\textit{STAR-COMM:} & H_1 \star H_2 & = & H_2 \star H_1 & \\
\textit{STAR-NEUTRAL-R:} & H \star [\,] & = & H & \\
\textit{STAR-EXISTS:} & (\exists x.\, H_1) \star H_2 & = & \exists x.\, (H_1 \star H_2) & (\text{if } x \notin H_2) \\
\textit{STAR-FORALL:} & (\forall x.\, H_1) \star H_2 & \vdash & \forall x.\, (H_1 \star H_2) & (\text{if } x \notin H_2)
\end{array}$$

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

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.*

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

The useful properties of entailment involving pure facts and quantifiers appear in Figure 2.1. 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 5.5, 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.

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 2.2.3 (Properties of the singleton heap predicate)

$$\begin{array}{lrcl}
\textit{SINGLE-NOT-NULL:} & (p \hookrightarrow v) & \vdash & (p \hookrightarrow v) \star [p \neq \mathit{null}] \\
\textit{SINGLE-CONFLICT:} & (p \hookrightarrow v_1) \star (p \hookrightarrow v_2) & \vdash & [\mathit{False}]
\end{array}$$

for propositions. These standard axioms are formally stated as follows.

$$\begin{array}{lllllll}
\text{PREDICATE-EXTENSIONALITY:} & \forall A. & \forall (P\ P' : A \to \mathsf{Prop}). & (P\ x \Leftrightarrow P'\ x) & \Rightarrow & (P = P') \\
\text{FUNCTIONAL-EXTENSIONALITY:} & \forall A\ B. & \forall (f\ f' : A \to B). & (f\ x = f'\ x) & \Rightarrow & (f = f') \\
\text{PROPOSITIONAL-EXTENSIONALITY:} & & \forall (P\ P' : \mathsf{Prop}). & (P \Leftrightarrow P') & \Rightarrow & (P = P')
\end{array}$$

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

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

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

Figure 2.1: Useful properties for pure facts and quantifiers, with respect to entailment.

2.2.4 Generalization to Postconditions

In the imperative λ-calculus considered in this manuscript and formalized further on (Section 2.3), a term evaluates to a value. A postcondition thus describes both an output value and an output state.

Definition 2.2.6 (Type of postconditions) *A postcondition has type: val → state → 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 2.2.7 (Separating conjunction between a postcondition and a heap predicate)

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

This operator appears for example in the statement of the frame rule (recall Section 2.1.1).

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 2.2.8 (Entailment between postconditions)

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

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

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

2.3 Language Syntax and Semantics

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

2.3.1 Syntax

We consider an imperative call-by-value λ-calculus. The syntactic categories are primitive functions π, values v, and terms t. The grammar of values is intended to denote *closed* values, that is, values without occurrences of free variables. This design choice leads to a simple 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 the addition and division operations.

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

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

Definition 2.3.1 (Syntax of the language)

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

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, "$\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$" is the sole sequencing construct: no sequencing is implicit in any other construct. For instance, the conditional construct "$\mathsf{if}\ t_0\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$", where t_0 is not a value, must be encoded as "$\mathsf{let}\, x = t_0\, \mathsf{in}\, \mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$". This presentation is intended to simplify the statement of the evaluation rules and reasoning rules. Note that many practical program verification tools perform code A-normalization as a preliminary step. Nevertheless, in Section 3.2, 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 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.

2.3.2 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 2.3.2 (Semantics of the language) *The evaluation rules appear in Figure 2.2.*

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

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

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

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

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

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

Figure 2.2: Evaluation rules, in big-step style

Observe that the rules of Figure 2.2 define a semantics that is *deterministic up to the choice of memory locations*. In Chapter 4, we extend the semantics with a nondeterministic construct.

2.4 Triples and Reasoning Rules

2.4.1 Separation Logic Triples

Separation Logic is a refinement of Hoare logic. Interestingly, Separation Logic triples can be defined *in terms of* Hoare triples. A Hoare triple, written $^{\text{HOARE}}\{H\}\, t\, \{Q\}$, asserts that in any state s satisfying the precondition H, the evaluation of the term t terminates and produces output value v and output state s', as described by the evaluation judgment $t/s \Downarrow v/s'$. Moreover, the output value and output state satisfy the postcondition Q, in the sense that $Q\, v\, s'$ holds. This definition captures termination: it defines a *total correctness* triple.

Definition 2.4.1 (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 2.1.1). 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* [*False*] *by the rule* SINGLE-CONFLICT *(Section 2.2.3). 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 2.4.2 (Total correctness Separation Logic triple)

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

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 2.4.3 (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*. In Chapter 4, we present an alternative definition well-suited for the more general case of nondeterministic semantics.

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

2.4.2 Structural Rules

The structural rules of Separation Logic include the consequence rule and the frame rule, which were already discussed, and two rules for extracting pure facts and existential quantifiers out of preconditions. (The role of these extraction rules is illustrated in the example proof presented [Charguéraud, 2020, Appendix D].)

Lemma 2.4.1 (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 2.4.2.*

$$\frac{H \vdash H' \quad \{H'\}\ t\ \{Q'\} \quad Q' \vdash\!\!\!-\ Q}{\{H\}\ t\ \{Q\}}\ \textit{CONSEQUENCE} \qquad \frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H'\}}\ \textit{FRAME}$$

$$\frac{P \Rightarrow \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}}\ \textit{PROP} \qquad \frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{Q\}}\ \textit{EXISTS}$$

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 $\{H \star H'\}\ t\ \{Q \mathbin{\dot\star} H'\}$. 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 2.4.2 (Combined consequence-frame rule)

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

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 $Q_1 \mathbin{\dot\star} H_2$. 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 2.5.3).

2.4.3 Rules for Terms

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

Lemma 2.4.3 (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 2.4.2.*

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

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

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

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 2.4.4 (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 "let $x = t_1$ 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.

2.4.4 Specification of Primitive Operations

The third and last category of reasoning rules corresponds to the specification of the primitive operations of the language. The operations on references have already been discussed (Section 2.1.5 and Section 2.1.8). The arithmetic operations admit specifications that involve only empty heaps.

Lemma 2.4.5 (Specification for primitive operations)

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

This completes the presentation of the reasoning rules of Separation Logic. Technically, these 18 reasoning rules suffice to verify imperative programs, although additional infrastructure helps obtain more concise proof scripts. The Coq formalization of the material from Section 2.2, Section 2.3, and Section 2.4 amount to 564 non-blank lines of Coq script. It includes 23 definitions, 59 lemmas, 24 lines of tactic definitions, and 117 lines of proofs. The corresponding formalization may be found in the file called LibSepMinimal.v distributed with the Separation Logic Foundation course [Charguéraud, 2021].

2.5 The Magic Wand Operator

2.5.1 Definition and Properties 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 2.5.1 (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.

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 2.5.3).

Lemma 2.5.1 (Useful properties of the magic wand)

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

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

Lemma 2.5.2 (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.

2.5.2 Magic Wand for Postconditions

Just as useful as the magic wand is its generalization to postconditions, which is involved for example in the statement of the ramified frame rule (Section 2.5.3). This operator, written $Q_1 \mathrel{-\!\!\star} Q_2$, takes as argument two postconditions Q_1 and Q_2 and produces a heap predicate.

Definition 2.5.2 (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. $Q_1 \mathrel{-\!\!\star} Q_2 \;\equiv\; \exists H_0.\ H_0 \star \big[(Q_1 \mathbin{\dot\star} H_0) \vdash Q_2\big]$
4. $H_0 \vdash (Q_1 \mathrel{-\!\!\star} Q_2) \;\Leftrightarrow\; (Q_1 \mathbin{\dot\star} H_0) \vdash Q_2$
5. $Q_1 \mathrel{-\!\!\star} Q_2$ *satisfies the following introduction and elimination rules.*

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

Lemma 2.5.3 (Useful properties of the magic wand for postconditions)

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

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

2.5.3 Ramified Frame Rule

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

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

The benefit of the ramified frame rule is that it eliminates the problem altogether. The key idea is to observe that the premise $Q_1 \mathbin{\dot\star} H_2 \vdash Q$ from the CONSEQUENCE-FRAME rule is equivalent to $H_2 \vdash (Q_1 \mathrel{-\!\!\star} Q)$, by the 4th characterization of Definition 2.5.2. Thus, in the other premise $H \vdash H_1 \star H_2$, the heap predicate H_2 may be replaced with $Q_1 \mathrel{-\!\!\star} Q$. The RAMIFIED-FRAME rule appears below.

Lemma 2.5.4 (Ramified frame rule) *RAMIFIED-FRAME reformulates CONSEQUENCE-FRAME.*

$$\textit{CONSEQUENCE-FRAME}\ \frac{H \vdash H_1 \star H_2 \qquad \{H_1\}\ t\ \{Q_1\} \qquad Q_1 \mathbin{\dot\star} H_2 \vdash Q}{\{H\}\ t\ \{Q\}} \qquad \textit{RAMIFIED-FRAME}\ \frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \star (Q_1 \mathrel{-\!\!\star} Q)}{\{H\}\ t\ \{Q\}}$$

2.6 Weakest-Precondition Style

2.6.1 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 $\mathsf{wp}\, t\, Q$, 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 2.2.1). 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 2.6.1 (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. $H \vdash \mathsf{wp}\, t\, Q \ \Leftrightarrow\ \{H\}\ t\ \{Q\}$

The first characterization asserts that $\mathsf{wp}\, t\, Q$ 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 $\mathsf{wp}\, t\, Q$.

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

The third characterization defines $\mathsf{wp}\, t\, Q$ as a predicate over a heap h, asserting that $\mathsf{wp}\, t\, Q$ 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 $\mathsf{wp}\, t\, Q$ 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 $H \vdash \mathsf{wp}\, t\, Q$.

Here again, Charguéraud [2020] appears to provide the first mechanized proof of equivalence relating all these well-known characterizations of the weakest-precondition operator. Yet another possible definition of wp will be presented in Chapter 4, with a judgment defined as a generalized form of inductive big-step semantics.

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. Concretely, the notion of triple may be defined in terms of wp using characterization (5), in the right-to-left direction. Reciprocally, the definition of wp may be defined in terms of triples, with a choice for the encoding that depends on the strength of the host logic with respect to existential quantification: Definition (3) makes weaker assumptions, whereas Definition (4) leverages the ability to existentially quantify over heap predicates. We have found that using Definition (4), which is expressed at the level of heap predicates, helps to simplify and to automate proofs.

2.6.2 WP-Style Structural Rules

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

Lemma 2.6.1 (Structural rules in weakest-precondition style)

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

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 2.6.2 (Ramified frame rule in weakest-precondition style)

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

2.6.3 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 2.6.3 (Reasoning rules for terms in weakest-precondition style)

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

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

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

2.6.4 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 2.6.4 (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 2.6.5 (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 5.6), we exploit the following rule for reasoning about every function call.

Lemma 2.6.6 (Variant of the ramified frame rule for proof obligations in wp style)

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

This concludes the first chapter of our presentation of a *foundational Separation Logic*. The next two chapters are concerned with language extensions, and with the generalization to nondeterministic languages.

Chapter 3

Language Extensions

This chapter presents techniques involved for reasoning in Separation Logic about a richer programming language, in which realistic programs can be written. I cover several extensions of the source programming language. First, I discuss the treatment of programming languages equipped with a garbage collector (Section 3.1). Such languages require an *affine* program logic, that is, a logic featuring the ability to freely discard certain classes of heap predicates. Second, I explain how to reason about programs that are not in A-normal form using the bind rule (Section 3.2). Third, I present reasoning rules for n-ary functions (Section 3.3), dynamic checks (Section 3.4), and loops (Section 3.5). Last, I specify operations on arrays (Section 3.6 and Section 3.7) and records (Section 3.8). Data constructors and pattern matching are postponed to Chapter 6.

The contents of this chapter contains excerpts from my ICFP'20 paper and its appendix [Charguéraud, 2020]. I have revised and augmented the sections on arrays and records to match the semantics of an ML-style language. I have also added a section explaining the bind rule, popularized by Iris [Jung et al., 2018b] for reasoning about terms that are not in A-normal form. Reasoning about floating-point programs by leveraging the Gappa tool [Boldo et al., 2009; Boldo and Melquiond, 2017] is left to future work.

3.1 Partially-Affine Separation Logic

3.1.1 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.

Early presentations of Separation Logic focused on C-style language, and did not discuss garbage collection. My PhD work on CFML provides an *affine* program logic, in which any heap predicate may be discarded at any time [Charguéraud, 2010; Charguéraud, 2011]. When Armaël Guéneau proposed *possibly negative time credits* [Guéneau et al., 2019a; Guéneau, 2019], CFML's original approach revealed too restrictive. Indeed, it would be unsound to allow discarding negative amounts of time credits. Guéneau adapted CFML to hardwire the fact that heap predicates may only be discarded if they account for a nonnegative amount of time credits.

Subsequently, I generalized CFML to allow distinguishing between *affine heap predicates*, which may be freely discarded, and *linear heap predicates*, which must remain accounted for. Besides time credits, linearity is useful to ensure, e.g., that in a terminating program every file handle opened eventually gets closed, or to ensure that every lock acquired eventually gets released. To that end, the reasoning rules should not allow discarding the heap predicates that represent linear resources.

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. Its contents was published in [Charguéraud, 2020, §8].

The formalization of a Separation Logic with support for both *linear* and *affine* predicates has been previously investigated in the context of Iris [Jung et al., 2015, 2018b]. On the one hand, Iris supports invariants and modalities that make the logic richer and more complex than the one we considered. On the other hand, Iris only supports affine predicates. Let me first explain why, before discussing extensions of Iris. All the heap predicates defined and used in Iris are *upwards-closed*. A predicate H is upwards-closed if, when it holds for one heap, it also holds for any larger heap: $H\,h \;\wedge\; h \perp h' \;\Rightarrow\; H(h \uplus h')$. In particular, the empty heap predicate is defined in Iris in such a way that it holds of any heap. As a result, the entailment $H \vdash [\,]$ holds for any heap predicate H. By exploiting this entailment in the rule of consequence, one may discard any heap predicate.

Tassarotti et al. [2017a] present an extension of Iris that supports both affine and linear assertions. Their model imposes a strong separation between affine resources and linear resources, visible at the level of the heaps that appear in the model. This separation might be too constraining for resources that could be both affine or linear, such as the *time credits* that we discuss in Section 7.3. In contrast, rather than imposing a *physical* separation between linear and affine heaps, our presentation relies on the use of a customizable predicate for characterizing the heaps that should be treated as affine.

Another approach to handling linear predicates in Iris is proposed by Bizjak et al. [2019]. The authors first describe Iron, in which linear resources (a.k.a. *trackable resources*) are encoded in the affine logic of Iris using fractional permissions to enforce that no linear heap predicate can be discarded. They then present Iron++, which is a layer of abstraction on top of Iron that hides away the use of fractions. In short, all heap predicates become implicitly parameterized by a fraction. It is not at all clear that time credits could be defined in Iron++ in a way that validates all the desired reasoning rules. Indeed, the fact that certain time credits may be discarded means that some fractions of "credits ownership" may be definitely lost.

In the rest of this section, I present my construction for a partially-affine Separation Logic. I leave to future work the investigation of whether this approach could be adapted to Iris, that is, to understand how it could combine with invariants and modalities.

3.1.2 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 3.1.1 (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 3.1.2 (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 3.1.1 (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 "$\lambda h.\,\mathsf{True}$") holds of *any heap*, the affine top predicate holds only of *any affine heap*.

Definition 3.1.3 (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 3.1.2 (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 3.1.1.

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 3.1.1 (Fully-affine Separation Logic) *The definition "haffine $h \equiv$ True" satisfies the requirements of Definition 3.1.1, 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 3.1.2 (Fully-linear Separation Logic) *The definition "haffine $h \equiv (h = \varnothing)$" satisfies the requirements of Definition 3.1.1, and leads to a Separation Logic where only pure heap predicates may be freely discarded. In that setting, (affine H) $\Leftrightarrow$ $(H \vdash [\,])$, and $\mathbb{T} = [\,] = (\lambda h.\ h = \varnothing)$.*

3.1.3 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 2.4.2) by integrating the affine top predicate $\mathbb{T}$ into the postcondition of the underlying Hoare triple, as formalized next.

Definition 3.1.4 (Refined definition of triples for Separation Logic)

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

Note that, with the fully-linear instantiation described in Example 3.1.2, the predicate $\mathbb{T}$ is equivalent to the empty heap predicate, therefore Definition 3.1.4 is strictly more general than Definition 2.4.2.

Lemma 3.1.3 (Reasoning rules for refined Separation Logic triples) *All the previously mentioned reasoning rules, in particular the structural rules (Lemma 2.4.1) and the reasoning rules for terms (Lemma 2.4.3), remain correct with respect to the refined definition of triples (Definition 3.1.4).*

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 $\mathbb{T}$, allowing a subsequent proof step to yield an entailment relation of the form $Q_1 \vdash (Q \mathbin{\dot\star} \mathbb{T})$, 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 $\mathbb{T}$, 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 3.1.4 (Discard rules for triples)

DISCARD-PRE
$$\frac{\{H\}\ t\ \{Q\} \qquad \mathit{affine}\ H'}{\{H \star H'\}\ t\ \{Q\}}$$

ATOP-POST
$$\frac{\{H\}\ t\ \{Q \mathbin{\dot\star} \mathbb{T}\}}{\{H\}\ t\ \{Q\}}$$

RAMIFIED-FRAME-ATOP
$$\frac{\{H_1\}\ t\ \{Q_1\} \qquad H \vdash H_1 \star (Q_1 \mathbin{\dot{-\!\star}} (Q \mathbin{\dot\star} \mathbb{T}))}{\{H\}\ t\ \{Q\}}$$

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 2.4.1 and Lemma 3.1.4).

Lemma 3.1.5 (Discard rules in weakest-precondition style)

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

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

3.2 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., 2018b] 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 4.1 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 3.2.1 (Bind rules)

$$\textit{BIND}\ \frac{\{H\}\ t\ \{Q_1\} \qquad (\forall v.\ \ \{Q_1\, v\}\ E[v]\ \{Q\})}{\{H\}\ E[t]\ \{Q\}} \qquad \textit{WP-BIND}\ \frac{}{\mathit{wp}\, t\, (\lambda v.\, \mathit{wp}\, (E[v])\, Q) \;\vdash\; \mathit{wp}\, (E[t])\, Q}$$

3.3 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). In the Coq course [Charguéraud, 2021], I employ curried functions to minimize the boilerplate associated with manipulating list of arguments. In the original version of CFML, I was using curried functions, however the possibility for partial applications and over-applications, which are not tagged as such in the syntax, introduced significant overheads in the formalization of the reasoning rules. In CFML2, I switched to using native n-ary functions, whose syntax can be processed in a simpler and more efficient manner than curried functions.

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 3.3.1 (Reasoning rule for functions of arity 2)

$$\textit{APP2}\quad \frac{v_0 = \hat{\mu} f.\lambda x_1 x_2.t \qquad \{H\}\ ([v_2/x_2]\,[v_1/x_1]\,[v_0/f]\,t)\ \{Q\} \qquad \mathit{noduplicates}\,(f :: x_1 :: x_2 :: \mathit{nil})}{\{H\}\ (v_0\, v_1\, v_2)\ \{Q\}}$$

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 "$x_1 :: \ldots :: x_n :: \mathsf{nil}$". The term $v_0\,\overline{v}$ denotes the application of a value v_0 to a list of arguments $\overline{v}$ of the form "$v_1 :: \ldots :: v_n :: \mathsf{nil}$". The corresponding rule, which is used in the current version of CFML, is stated as follows.

Lemma 3.3.2 (Reasoning rule for n-ary functions)

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

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.

3.4 Treatment of Dynamic Checks

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

Lemma 3.4.1 (Evaluation rules and reasoning rule for assertions)

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

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.

The above reasoning rules for assertions were introduced in CFML at the time of my PhD thesis [Charguéraud, 2010].

3.5 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 "while t_1 do t_2".

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

Lemma 3.5.1 (Evaluation rule and reasoning rules for while loops)

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

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

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 3.5.1 (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.

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

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].

3.6 Arrays in an ML-like Language

In this section, we present specifications for operations on ML-style arrays. We introduce the representation predicate $\mathsf{Array}\, p\, L$ 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 $|L|$.

Let us start by explaining how the predicate $\mathsf{Array}\, p\, L$ may be defined in a foundational manner with respect to the memory model. An array consists of a header block, followed with a sequence of cells. The header block is described by a heap predicate written $\mathsf{ArrayHeader}\, p\, n$, where p denotes the address of the array and n its length. An individual array cell is described by a heap predicate written $\mathsf{Cell}\, p\, i\, v$ 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 $\mathsf{ArraySeg}\, p\, j\, L$ 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 $\mathsf{Array}\, p\, L$ describes both the array header and the *full* segment, which covers elements from index 0 to index $|L| - 1$, inclusive. In the formal definitions below, we write $L[i]$ for "List.nth $i\, L$" and $L[i := v]$ for "List.update $i\, v\, L$".

Definition 3.6.1 (Definitions of representation predicates for arrays)
The first two definitions are implementation-dependent and should not be revealed to the end user.

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

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 3.6.1 (Large-footprint specifications for array operations)

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

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

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

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

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, so as 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 3.6.3 (Borrowing of a cell) *Assume* $0 \leqslant i < |L|$.

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

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 3.6.4 (Small-footprint specifications for array operations, for segments)

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

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 3.6.5 (Splitting of array segments)

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

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 arrays is implemented in CFML since 2021, and is described in my course [Charguéraud, 2021].

3.7 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 3.7.1 (Alternative definitions for arrays in a C-like language)

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

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

Lemma 3.7.1 (Properties of the array representation predicate in a C-like language)

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

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

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

3.8 Records

In this section, I present 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.

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 3.8.1 (Definitions of representation predicates for records)

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

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 3.8.1 (Decomposition rules for the record representation predicate)

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

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

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

Lemma 3.8.3 (Large-footprint specifications for record operations)

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

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.

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

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 $\mathsf{Record}\, p\, ((k1, v1) :: (k2, v2) :: \mathsf{nil})$. 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 4

Omni-Big-Step Semantics

This chapter explains how to define foundational triples for the more general case of nondeterministic semantics. I present the definition of the *omni-big-step* judgment (Section 4.1), and detail its origins (Section 4.2). I next state the key properties of this judgment (Section 4.3), and prove in particular that it satisfies the *frame* property (Section 4.4). I then explain how to exploit this judgment to define the weakest-precondition predicate (Section 4.5) and Separation Logic triples (Section 4.6). Finally, I give an overview of other applications of omni-semantics (Section 4.7).

Soon after the publication of my ICFP'20 paper [Charguéraud, 2020], Ralf Jung asked me how I could extend my big-step-based definition of total correctness Separation Logic triples to handle the case of a nondeterministic semantics. I realized that, to generalize the standard big-step judgment for handling more than one execution path, one needs to relate an input configuration not to a single output configuration but to a set of output configurations, isomorphic to a postcondition. This observation had in fact already been made in prior work by Schäfer, Schneider and Smolka [2016], and in parallel work by Chlipala, Gruetter, and Erbsen [2021].

Interestingly, Schäfer et al. write: *The way we connect operational and axiomatic semantics bears a close resemblance to Charguéraud's work [2010] on characteristic formulas for program verification.* This sentence underlies how the judgment relating an input configuration to a postcondition stands half-way between a standard operational semantics and a set of axiomatic rules for reasoning about programs.

Schäfer et al.'s work [2016] targets a Hoare Logic, and does not discuss the frame rule. Erbsen et al.'s work [2021] exploits Separation Logic, but does so by including *explicit frames* in specifications. A key contribution I made was to show that the omni-big-step judgment inherently satisfies the frame rule. During the year 2021, I also formalized, numerous other properties of this judgment, and investigated the coinductive variant of the judgment and its applications to type soundness proofs.

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

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

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

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

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

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

Figure 4.1: Omni-big-step semantics (for terms in A-normal form)

In 2022, I joined Adam Chlipala, Samuel Gruetter, and Andres Erbsen on the writing of a journal paper investigating in depth the metatheory and the numerous applications of both small-step-style and big-step-style *omnisemantics* [Charguéraud et al., 2022]. This paper has been accepted for publication in the TOPLAS journal. The contents of the present chapter is an excerpt from that paper, focusing specifically on how to use the big-step judgment to define total correctness Separation Logic triples for nondeterministic semantics.

4.1 Definition of the Omni-Big-Step Judgment

The standard big-step judgment, written $t/s \Downarrow v/s'$, was defined in Figure 2.2. To introduce an interesting source of nondeterminism, we extend the language with a random number generator: $\mathsf{rand}\,n$, where n is a positive integer, evaluates to any integer in the range $[0, n)$. The big-step evaluation rule for this nondeterministic construct appears below.

$$\frac{0 \leqslant m < n}{(\mathsf{rand}\,n)/s \Downarrow m/s}\ \textsc{big-rand}$$

The corresponding omni-big-step semantics appears in Figure 4.1. Its evaluation judgment, written $t/s \Downarrow Q$, asserts that all possible evaluations starting from the configuration t/s reach final configurations that belong to the set Q. Observe how the standard big-step judgment $t/s \Downarrow$

v/s' describes the behavior of one possible execution of t/s, whereas the omni-big-step judgment describes the behavior of all possible executions of t/s. 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 → state → Prop" or by a predicate of type "(val × state) → 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 4.1.

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 $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$ in state s terminate and satisfy the postcondition Q. First of all, 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
$$\frac{t_1/s \Downarrow Q}{(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/s \Downarrow Q}$$

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

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.

4.2 History of the Omni-Big-Step Judgment

Omni-big-step semantics appear to have first been presented by Schäfer et al. [2016]. These authors present the notion on a nondeterministic source language of guarded commands, as well as a deterministic target language with named continuations. They also present characterizations of program equivalence and present a proof of equivalence with traditional small-step semantics, though only in the case of a deterministic semantics. Their work does not discuss Separation Logic, in particular the aspects related to the frame rule.

Omni-big-step semantics were later exploited by Erbsen et al. [2021] in the LightBulb project. (They call this style of semantics *CPS semantics*.) These authors consider an omni-big-step semantics for a high-level, core imperative language with external calls; and consider an *omni-small-step* semantics, applied to a low-level, RISC-V machine language. Omni-small-step semantics are out of the scope of the present manuscript, but are covered in depth in the omni-semantics paper [Charguéraud et al., 2022]. The LightBulb project provides end-to-end compiler-correctness results for terminating programs. This work employs Separation Logic reasoning rules (in weakest-precondition style), yet does not feature a generic frame rule. Instead, frames appear as explicit heap predicates in specifications.

In my Coq course, I introduced omni-big-step semantics in 2021-2022 for the purpose of deriving Separation Logic triples for a nondeterministic imperative λ-calculus. I proved that this semantics inherently satisfies the frame rule, avoiding the need to resort to the technique of the *baked-in frame rule*. Besides, I formalized in Coq the metatheory associated with the judgment, including equivalence with other operational semantics, both for terminating and for possibly-diverging programs—partial correctness is out of the scope of the current chapter.

Those results appear in the omnisemantics paper [Charguéraud et al., 2022], written jointly with the authors of the LightBulb project. The rest of this chapter contains an excerpt of that paper.

4.3 Properties of the Omni-Big-Step Judgment

In this section, we discuss some key properties of the omni-big-step judgment $t/s \Downarrow Q$. Recall that the metavariable Q denotes an of the set of possible final configurations.

Total correctness. The predicate $t/s \Downarrow Q$ captures total correctness in the sense that it captures the conjunction of termination (all executions terminate) and partial correctness (if an execution terminates, then its final state satisfies the postcondition Q). Let terminates(t, s) be a judgment capturing the fact that all executions of t/s terminate. This judgment can be defined

with respect to a small-step semantics, by the following two inductive rules.

$$\textsc{terminates-here}\quad \frac{}{\mathsf{terminates}(v,s)} \qquad\qquad \textsc{terminates-step}\quad \frac{\big(\exists t's'.\ t/s \longrightarrow t'/s'\big) \quad \big(\forall t's'.\ (t/s \longrightarrow t'/s') \ \Rightarrow\ \mathsf{terminates}(t',s')\big)}{\mathsf{terminates}(t,s)}$$

Recall that $t/s \Downarrow v/s'$ denotes the standard big-step evaluation judgment. We prove:

$$\textsc{omni-big-step-iff-terminates-and-correct}: \quad t/s \Downarrow Q \quad\Longleftrightarrow\quad \mathsf{terminates}(t,s) \ \wedge\ \big(\forall vs'.\ (t/s \Downarrow v/s') \ \Rightarrow\ (v,s') \in Q\big).$$

In particular, if we instantiate the postcondition Q with the *always-true* predicate, we obtain the predicate $t/s \Downarrow \{(v,s') \mid \mathsf{True}\}$, which captures only the termination property.

Remark: whereas the *inductive* interpretation of the evaluation rules definining the omni-big-step judgment yields a characterization of *total correctness*, the *coinductive* interpretation of the exact same set of rules yields a characterization of *partial correctness*. This aspect is discussed in [Charguéraud et al., 2022, §3.4].

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

$$\textsc{omni-big-consequence}: \quad t/s \Downarrow Q \quad\wedge\quad Q \subseteq Q' \quad\Rightarrow\quad t/s \Downarrow Q'$$

Strongest postcondition. If the omni-big-step judgment $t/s \Downarrow Q'$ holds for at least one set Q', then there exists a smallest possible set Q for which $t/s \Downarrow Q$ holds. This set corresponds to the strongest-possible postcondition Q, in the terminology of Hoare logic.

$$\mathsf{strongest\text{-}post}\ t\ s \quad\equiv\quad \bigcap_{Q \ \mid\ (t/s \Downarrow Q)} Q \quad=\quad \big\{(v,s') \ \big|\ \forall Q,\ (t/s \Downarrow Q) \Longrightarrow (v,s') \in Q\big\}$$

We prove that if $t/s \Downarrow Q$ holds for some Q, then $t/s \Downarrow (\mathsf{strongest\text{-}post}\ t\ s)$ holds.

No derivations for terms that may get stuck. The fact that $\mathsf{rand}\,0$ is a stuck term is captured by the fact that $(\mathsf{rand}\,0)/s \Downarrow Q$ does not hold for any Q. More generally, if one or more nondeterministic executions of t may get stuck, then we have: $\forall Q.\ \neg\ (t/s \Downarrow Q)$.

Relationship to standard big-step semantics. The following two results formalize the relationship between the omni-big-step judgment and the standard big-step judgment.

First, if $t/s \Downarrow Q$ holds, then any final configuration for which the standard big-step judgment holds necessarily belongs to the set Q.

$$\textsc{omni-big-and-big-inv}: \quad t/s \Downarrow Q \quad\wedge\quad t/s \Downarrow v/s' \quad\Rightarrow\quad (v,s') \in Q$$

Second, if $t/s \Downarrow Q$ holds, then there exists at least one evaluation according to the standard big-step judgment whose final configuration belongs to the set Q.

$$\textsc{omni-big-to-one-big}: \quad t/s \Downarrow Q \quad\Rightarrow\quad \exists vs'.\ t/s \Downarrow v/s' \ \wedge\ (v,s') \in Q$$

A corollary asserts that if $t/s \Downarrow Q$ holds with Q being a singleton set made of a unique final configuration v/s', then the standard big-step judgment holds for that configuration.

$$\textsc{omni-big-singleton}: \quad t/s \Downarrow \{(v,s')\} \quad\Rightarrow\quad t/s \Downarrow v/s'$$

Particular case of deterministic languages. In a deterministic language, an input configuration t/s may evaluate to at most one configuration v/s'. In such a case, the strongest postcondition is either empty or reduced to the singleton set $\{(v, s')\}$.

Nonempty outcome sets. Observe that the judgment $t/s \Downarrow Q$, as defined in Figure 4.1, can only hold for a nonempty set Q. When designing omni-big-step rules for a new language, one has to be careful not to accidentally include rules that allow derivations of empty outcome sets for some programs. To illustrate the matter, consider the term "rand 0". According to the standard big-step semantics, this term is stuck because the rule BIG-RAND requires a positive argument to rand. In the omni-big-step semantics, if we were to omit the premise $n > 0$ in the rule OMNI-BIG-RAND, we would be able to derive $(\mathsf{rand}\,0)/s \Downarrow Q$ for any s and Q. Indeed, the premise $\forall m.\ 0 \leqslant m < n \Rightarrow (m, s) \in Q$ becomes vacuously true when n is nonpositive.

A similar subtlety appears in the rule OMNI-BIG-REF, where the fresh location p must be picked fresh from the domain of s. This quantification could become vacuously true if the semantics allowed for infinite states or if the set of memory locations were finite. (We discuss in [Charguéraud et al., 2022, §6.5] the treatment of a language whose semantics account for a finite memory.)

The likelihood of inadequate formalization due to missing premises might be viewed as a potential weakness of omnisemantics. Yet, if needed, additional confidence can easily be restored at the cost of minor additional work: one may consider a standard small-step semantics as reference (i.e., as part of the trusted code base), then relate it to the corresponding omni-big-step semantics and use the latter to carry out big-step style, inductive proofs on nondeterministic executions.

4.4 Frame Property for the Omni-Big-Step Judgment

I next show 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 4.4.1 (Frame property for big-step omnisemantics)

$$\frac{t/s_1 \Downarrow Q \qquad s_1 \perp s_2}{t/(s_1 \uplus s_2) \Downarrow (Q \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))}\ \textsc{omni-big-frame}$$

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* $\mathsf{ref}\,v$. *The assumption is* $(\mathsf{ref}\,v)/s_1 \Downarrow Q$. *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* $(\mathsf{ref}\,v)/(s_1 \uplus s_2) \Downarrow (Q \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))$. *By* OMNI-BIG-REF, *we need to justify:* $\forall p \notin \mathrm{dom}\,(s_1 \uplus s_2).\ (Q \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))\,p\,((s_1 \uplus s_2)[p := v])$. *Consider a location* p *not in* $\mathrm{dom}\,s_1$ *nor in* $\mathrm{dom}\,s_2$. *The predicate* $(Q \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))\,p$ *is equivalent to* $(Q\,p) \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2)$. *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:* $((Q\,p) \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))\ ((s_1[p := v]) \uplus s_2)$. *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 "*$\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$*". 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* $(\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)/(s_1 \uplus s_2) \Downarrow (Q \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))$. *To that end, we invoke* OMNI-BIG-LET. *For its first premise, we prove* $t_1/(s_1 \uplus s_2) \Downarrow (Q_1 \mathbin{\text{⋆}} (\lambda s'.\ s' = s_2))$ *by exploiting the induction hypothesis*

applied to $t_1/s_1 \Downarrow Q_1$. *For the second premise, we have to prove* $\forall v' s''.\ (Q_1 \mathbin{\text{-}\!\!\star} (\lambda s'.\ s' = s_2))\ v'\ s'' \Rightarrow ([v'/x]\, t_2)/s'' \Downarrow (Q \mathbin{\text{-}\!\!\star} (\lambda s'.\ s' = s_2))$. *Consider a particular* v' *and* s''. *The assumption* $(Q_1 \mathbin{\text{-}\!\!\star} (\lambda s'.\ s' = s_2))\ v'\ s''$ *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* $([v'/x]\, t_2)/(s_1' \uplus s_2) \Downarrow (Q \mathbin{\text{-}\!\!\star} (\lambda s'.\ s' = s_2))$. *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.* □

4.5 Definition of the Weakest-Precondition Predicate

Recall from Section 2.6.3 that the weakest-precondition operator, written $\mathsf{wp}\, t\, Q$, computes the weakest predicate H for which the triple $\{H\}\ t\ \{Q\}$ holds. The interpretation of the inductively defined omni-big-step judgment $(t/s \Downarrow Q)$ matches, up to the order of arguments, the interpretation of the weakest-precondition operator. Formally:

$$\mathsf{wp}\, t\, Q\, s \quad \iff \quad t/s \Downarrow Q.$$

Thus, in a foundational approach, we can formally define wp as follows.

Definition 4.5.1 (Separation Logic WP in terms of the omni-big-step judgment)

$$\mathit{wp}\, t\, Q \quad \equiv \quad \lambda s.\ (t/s \Downarrow Q)$$

There remains to describe how the weakest-precondition-style reasoning rules can be derived from the omni-big-step evaluation rules. Recall that, in a foundational program logic, reasoning rules take the form of lemmas proved correct with respect to the definition of triples and with respect to the semantics of the language. Throughout this section and the next, we formulate rules by viewing postconditions as predicates of type val → state → Prop, as this presentation style is more idiomatic in program logics. We present reasoning rules using the horizontal bar; keep in mind that the statements are not inductive definitions but lemmas.

First of all, we establish structural rules, whose statement in weakest-precondition style is reproduced below.

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

The rule WP-CONSEQUENCE is an immediate consequence of OMNI-BIG-CONSEQUENCE (Section 4.3). The rule WP-FRAME is derived from OMNI-BIG-FRAME (Section 4.4) by the following reasoning. Consider a state s satisfying $(\mathsf{wp}\, t\, Q) \star H$. The goal is to prove $\mathsf{wp}\, t\, (Q \mathbin{\text{-}\!\!\star} H)\, s$. By definition of separating conjunction, the state s decomposes as $s_1 \uplus s_2$, with $s_1 \perp s_2$, and $\mathsf{wp}\, t\, Q\, s_1$ and $H\, s_2$. By definition, $\mathsf{wp}\, t\, Q\, s_1$ is equivalent to $t/s_1 \Downarrow Q$. Applying the lemma OMNI-BIG-FRAME gives: $t/(s_1 \uplus s_2) \Downarrow (Q \mathbin{\text{-}\!\!\star} (\lambda s'.\ s' = s_2))$. To conclude, we need to prove: $t/(s_1 \uplus s_2) \Downarrow (Q \mathbin{\text{-}\!\!\star} H)$. To that end, it suffices to verify that $(\lambda s'.\ s' = s_2)$ entails H. This entailment holds because s_2 satisfies H.

We then establish reasoning rules for term constructs. Consider for example a let-binding. Compare its weakest-precondition style reasoning rule, expressed in the form of an entailment, with the omni-big-step rule for let-bindings.

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

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

It may not be obvious at first sight how to relate the two. The Coq proof is actually a one-liner: `unfold wp; intros; intros h K; eapply omnibig_let; eauto`. Let us explain. To prove the rule WP-LET, let us consider a state s. We have to prove that $\mathsf{wp}\, t_1\, \big(\lambda v'.\, \mathsf{wp}\, ([v'/x]\, t_2)\, Q\big)\, s$ implies $\mathsf{wp}\, (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\, Q\, s$. By definition of wp, this is equivalent to proving that $t_1/s \;\Downarrow\; \big(\lambda v' s'.\, ([v'/x]\, t_2)/s' \;\Downarrow\; Q\big)$ implies $(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \;\Downarrow\; Q$. To establish this implication, we apply the rule OMNI-BIG-LET with the variable Q_1 instantiated as $\lambda v' s'.\, \big(([v'/x]\, t_2)/s' \;\Downarrow\; Q\big)$. With this instantiation, the second premise of OMNI-BIG-LET becomes a tautology. The first premise of that rule becomes: $t_1/s \;\Downarrow\; \big(\lambda v_1.\, ([v'/x]\, t_2)/s' \;\Downarrow\; Q\big)$, which matches the assumption at hand.

For most other term constructs, the wp rule is essentially a copy of the omni-big-step rule with arguments reordered. One interesting exception is that of loops. While-loops have not been discussed so far, but they appear in the language used our in case studies [Charguéraud et al., 2022, §6]. Typically, standard weakest-precondition rules for while loops are stated using loop invariants. In contrast, an omni-big-step rule essentially unfolds the first iteration of the loop, just like in a standard big-step semantics. From that unfolding rule, one can derive the loop-invariant-based rule by induction, in just a few lines of proof.

In summary, by considering a semantics expressed in omni-big-step style, one can derive practical reasoning rules, in most cases via one-line proofs. The construction of a program logic on top of an omni-big-step semantics is thus a major generalization over the use of a standard big-step semantics, which fall short in the presence of nondeterminism. It is also an improvement over the use of a small-step semantics, which require more work for deriving the reasoning rules, especially when trying to capture termination.

4.6 Definition of Triples w.r.t. Omni-Big-Step Semantics

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 4.6.1 (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:

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

Like in the case of weakest-preconditions, the rule of consequence and frame rule for triples follow directly from the properties OMNI-BIG-CONSEQUENCE and OMNI-BIG-FRAME of the omni-big-step judgment. Reasoning rules for term constructs are straightforward to derive. 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.`"

In summary, omni-big-step semantics allow for a direct definition of triples that inherently satisfy the frame rule. There is no need to apply the baked-in frame rule construction (Definition 2.4.2), and reasoning rules are derived by simpler and shorter proofs.

4.7 Other Applications of Omnisemantics

The omnisemantics paper [Charguéraud et al., 2022] covers numerous aspects beyond the set-up of total correctness triples on top of a big-step semantics.

- We explain how to capture partial correctness using coinductive interpretation. Partial correctness, in particular, can be used to define and reason about *type safety* of a program.
- We describe the *omni-small-step* judgment, written $t/s \longrightarrow P$, where P denotes a set of pairs each made of a term and a state. This judgment captures the set of configurations that are reachable in a single evaluation step from t/s. We then present the *eventually* judgment, written $t/s \longrightarrow^{\Diamond} P$, which may be exploited in inductive proofs.
- We present two novel proof techniques for establishing type safety for a type system with respect to a non-deterministic semantics. One is based on the omni-small-step semantics, the other on the omni-big-step semantics. We explain the benefits compared with standard proof techniques, in particular the fact that they avoid a quadratic case inspection.
- We discuss applications of the omni-semantics judgment for proving the correctness of compiler transformations via *forward simulations*. Prior work either had to face the complications of using *backward simulations*, or to work hard around the problem by artificially making semantics deterministic (as done, e.g., in the CompCert verified compiler [Leroy, 2009]). The proof techniques presented apply to both omni-big-step and omni-small-step semantics. Moreover, they have been shown useful to establish the correctness of a compilation pass from a high-level language whose semantics is specified in omni-big-step style, down to a lower-level language whose semantics is specified in omni-small-step style.

The aforementioned applications all exploit a key feature of omni-big-step semantics: the fact that they inherently deliver induction principles for reasoning about program executions.

Chapter 5

Characteristic Formulae

This chapter describes the technique of *characteristic formulae* for smoothly integrating Separation Logic in an interactive proof assistant. In Section 5.1, I explain the concept of characteristic formula, and in particular how it relates to the concept of weakest precondition. In Section 5.2 and Section 5.3, I show how to define the characteristic formulae generator as a function that effectively computes within Coq. In Section 5.4, I explain how to establish the soundness of this generator. In Section 5.5 and Section 5.6, I describe the set-up used for carrying interactive program verification in practice using characteristic formulae, through the use of tactics that allow for concise proof scripts.

The term *characteristic formula* was introduced in work by Hennessy and Milner [1985] on process calculi. The notion of characteristic formula was first applied to a first-order program logic by Berger et al. [2005]. In my PhD work, I introduced characteristic formulae for a higher-order Separation Logic Charguéraud [2010]. These characteristic formulae were generated by an external OCaml program in the form of Coq axioms—thus, they were not foundational.

Guéneau et al. [2017] ported my characteristic formulae approach to the CakeML verified compiler [Kumar et al., 2014], implemented in HOL. Armaël Guéneau, whom I had encouraged to go for an internship with Magnus Myreen, not only generalized on the way the characteristic formulae to support catchable exceptions and I/O, but also was able to establish the soundness of the characteristic formulae generator with respect to CakeML's semantics.

The characteristic formulae developed in my PhD thesis were presented as predicates that apply to both a precondition and a postcondition. In the years 2018-2019, I realized that they could be alternatively presented as predicates over postconditions only, that is, in *weakest-precondition style*. Moreover, I defined a Coq function that takes as input the deep embedding of an untyped term and computes, within Coq, its characteristic formula in weakest-precondition style (see Section 5.2). For establishing the soundness of this function with respect to the semantics, I developed a proof technique simpler than the one used by Armaël Guéneau (see Section 5.4).

Most of the contents this chapter corresponds to the chapter `WPgen` of my all-in-Coq course [Charguéraud, 2021]. As of 2022, the material has not yet been published.

5.1 Principle of Characteristic Formulae

Recall from Section 2.6 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.

In this chapter, I introduce a function to *effectively compute* 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 corrollary 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.

5.2 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 2.6.3), 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:	$(\mathsf{wp}\, t\, Q) \mathbin{\star} (Q \mathrel{-\!\!\star} Q') \vdash (\mathsf{wp}\, t\, Q')$	

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.

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

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.

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

We will make use of this alternative formulation later on, in Section 6.6.

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 5.3. 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 my PhD work [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 my prior work.

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 $(\mathsf{val} \to \mathsf{Hprop}) \to \mathsf{Hprop}$, and $\mathcal{G}$ denotes a formula that depends on one or several arguments of type val.

Definition 5.2.1 (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: "$\mathsf{cf}_E\, (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)$" as "$\mathsf{cf\_let}\, (\mathsf{cf}_E\, t_1)\, \big(\lambda X.\ \mathsf{cf}_{(x,X)::E}\, t_2\big)$". Furthermore, we introduce the Coq syntax "`Let x := F1 in F2`" for the predicate "$\mathsf{cf\_let}\, \mathcal{F}_1\, \mathcal{G}_2$". 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 5.5), 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 5.2.2 (Characteristic formulae generator) $\mathit{cf}\, t\, Q \equiv \mathit{cf}_{\mathit{nil}}\, t\, Q$ *with*

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

5.3 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
$$\frac{H \vdash \mathsf{framed}\, \mathcal{F}\, Q}{H \star H' \vdash \mathsf{framed}\, \mathcal{F}\, (Q \mathbin{\dot\star} H')}$$

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 $\exists Q'.\, (\mathsf{wp}\, t\, Q') \star (Q' \mathrel{-\!\!\dot\star} Q)$. We can mimic this definition by *defining* the assertion "framed $\mathcal{F}\, Q$" as "$\exists Q'.\, (\mathcal{F}\, Q') \star (Q' \mathrel{-\!\!\dot\star} Q)$". I am very grateful to Jacques-Henri Jourdan who, at a time when I was not yet familiar with the magic wand on postconditions, suggested to me that this definition of the framed predicate would *obviously* satisfy the rules CF-FRAMED-CONSEQ and CF-FRAMED-FRAME that I was aiming for.

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 3.1.2).

Definition 5.3.1 (Predicate "framed")

$$\mathit{framed}\, \mathcal{F} \quad \equiv \quad \lambda Q.\ \ \exists Q'.\ (\mathcal{F}\, Q') \star (Q' \mathrel{-\!\!\dot\star} (Q \star \mathbb{T}))$$

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 5.3.2 (Properties of the predicate "framed")

$$\begin{array}{ll}
\textit{FRAMED-CONSEQ:} & Q \mathrel{\dot{\vdash}} Q' \;\Rightarrow\; \mathit{framed}\,\mathcal{F}\,Q \vdash \mathit{framed}\,\mathcal{F}\,Q' \\
\textit{FRAMED-FRAME:} & (\mathit{framed}\,\mathcal{F}\,Q) \star H \vdash \mathit{framed}\,\mathcal{F}\,(Q \star H) \\
\textit{FRAMED-ERASE:} & \mathcal{F}\,Q \vdash \mathit{framed}\,\mathcal{F}\,Q \\
\textit{FRAMED-MONO:} & (\forall Q.\ \mathcal{F}\,Q \vdash \mathcal{F}'\,Q) \;\Rightarrow\; \mathit{framed}\,\mathcal{F}\,Q \vdash \mathit{framed}\,\mathcal{F}'\,Q \\
\textit{FRAMED-WP:} & \mathit{framed}\,(\mathit{wp}\,t) = \mathit{wp}\,t \\
\textit{FRAMED-IDEM:} & \mathit{framed}\,(\mathit{framed}\,\mathcal{F}) = \mathit{framed}\,\mathcal{F}
\end{array}$$

5.4 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 "sound $t\,\mathcal{F}$", asserts that $\mathcal{F}$ is a logical formula stronger than the weakest-precondition of t.

Definition 5.4.1 (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 5.4.1 (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.

$$
\begin{array}{ll}
\textsc{sound-wp}: & \mathsf{sound}\, t\, (\mathsf{wp}\, t) \\
\textsc{sound-framed}: & \mathsf{sound}\, t\, \mathcal{F} \;\Rightarrow\; \mathsf{sound}\, t\, (\mathsf{framed}\, \mathcal{F}) \\
\textsc{sound-fail}: & \mathsf{sound}\, t\, \mathsf{cf\_fail} \\
\textsc{sound-val}: & \mathsf{sound}\, v\, (\mathsf{cf\_val}\, v) \\
\textsc{sound-app}: & \mathsf{sound}\, t\, (\mathsf{cf\_app}\, t) \\
\textsc{sound-if}: & \mathsf{sound}\, t_1\, \mathcal{F}_1 \;\wedge\; \mathsf{sound}\, t_2\, \mathcal{F}_2 \\
& \Rightarrow\; \mathsf{sound}\, (\mathsf{if}\, t_0\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2)\, (\mathsf{cf\_if}\, t_0\, \mathcal{F}_1\, \mathcal{F}_2) \\
\textsc{sound-let}: & \mathsf{sound}\, t_1\, \mathcal{F}_1 \;\wedge\; \big(\forall X.\, \mathsf{sound}\, ([X/x]\, t_2)\, (\mathcal{G}_2\, X)\big) \\
& \Rightarrow\; \mathsf{sound}\, (\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)\, (\mathsf{cf\_let}\, \mathcal{F}_1\, \mathcal{G}_2) \\
\textsc{sound-fix}: & \big(\forall F X.\, \mathsf{sound}\, ([X/x]\, [F/f]\, t)\, (\mathcal{G}\, F\, X)\big) \\
& \Rightarrow\; \mathsf{sound}\, (\mu f.\lambda x.t)\, (\mathsf{cf\_fix}\, \mathcal{G})
\end{array}
$$

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 5.4.1 on the empty environment.

Theorem 5.4.1 (Soundness of characteristic formulae)

$$cf\,t\,Q \vdash wp\,t\,Q$$

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 5.1).

5.5 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 chapters (files `Basic.v` and `Repr.v`) from my all-in-Coq course [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, Section 2.2.3). 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 the appendix of my ICFP'20 paper [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. Eventually, I may need to switch to an OCaml-based implementation.

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 5.6.

```
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 2.1.8. 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 2.1.3). 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.

5.6 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 chapter `WPgen.v` from my Coq course.

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.

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

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 $\mathsf{cf\_val}\,v$.

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

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 of all, `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 2.6.4 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, so as 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 6

Lifting: from Program Values to Logical Values

This chapter presents a technique, called *lifting*, for specifying program values using logical values, i.e. Coq values. I start by motivating this technique (Section 6.1). I then describe how to realize lifting using typeclasses (Section 6.2), how to define *lifted triples* (Section 6.3) and lifted representation predicates (Section 6.4). Next, I explain what makes it challenging to define, inside Coq, such a generator (Section 6.5). I then present an approach that relies on a characteristic formula generator implemented as an external program (Section 6.6). I describe the automated handling of record operations with respect to a lifted representation predicate (Section 6.7). Finally, I explain how to establish the soundness of these formulae in a foundational way (Section 6.8).

I exploited the lifting technique in the first version of CFML, developed during my PhD thesis [Charguéraud, 2010]. That work, however, was not foundational: characteristic formulae were generated as axioms, and OCaml values were translated into corresponding Coq values, without establishing in Coq a relation between those Coq values and the deep embedding of the source language. The work by Guéneau et al. [2017] on the CakeML compiler was the first to provide foundational characteristic formulae, however this work did not integrate the lifting technique. Instead, in CakeML, binary relations appear explicitly in every specification for relating program values with corresponding logical values.

In 2020, after developing the unlifted characteristic formula generator presented in the previous chapter, I investigated how to generalize this approach to take advantage of the lifting technique. After preliminary investigations, I concluded that it would be extremely difficult, if not impossible, to develop a Coq function for computing *lifted characteristic formulae*, for reasons that I explain in Section 6.5. In any case, the use of an external tool for generating Coq definitions remains required for translating user-defined algebraic data types into Coq inductive definitions.

Given a lifted characteristic formula generator implemented as an external tool, the key challenge was to find a way to justify, in a foundational manner, its soundness. In 2022, I developed a proof technique based on the *validation* approach. The insight is, for a given program, to formally relate the lifted formula generated by the external tool with the unlifted formula computed inside Coq. I describe this approach in Section 6.6. This recent technical contribution is yet to be submitted for publication.

6.1 Motivation for Lifting

Benefit #1: postconditions. Recall the specification of the ref operation for allocating a reference (Section 2.1.5).

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

There, the result value is described by a variable named r, of type val. Recall that val denotes the type of closed values in the deep embedding. The variable p, of type loc, describes the memory location of the freshly allocated cell. The equality $[r = p]$ hides a coercion. It actually stands for $[r = \texttt{val\_loc}\, p]$, where `val_loc` is the constructor for locations in the grammar of values.

Arguably, this specification is quite heavy-weighted. One would rather like to write the same specification by directly quantifying over "a result p of type loc", as shown below.

$$\{[\,]\}\ (\mathsf{ref}\, v)\ \{\lambda(p : \mathsf{loc}).\ (p \hookrightarrow v)\}$$

In the Coq course, to improve readability, I introduce a piece of syntactic sugar specifically for functions that return a location, allowing one to write: `Triple (ref v)[] (funloc p ⇒ p ↪ v)`. Yet, this ad-hoc approach based on a notation does not generalize well to other return types.

The aim of this section is to present a general approach to specifying values without having to go through the indirection of a variable of type val, such as the variable r above. The lifting technique introduces the notion of *lifted triples*, written `Triple` with a leading uppercase. With a lifted triple, one may write: `Triple (ref v)[] (fun p ⇒ p ↪ v)`, directly binding a variable p of type loc.

Lifted triples can be used to express postconditions that bind variables of any Coq type that can be mapped to a type of program values. For example, an operation that returns a pair, e.g. splitting on a list of pairs, can be specified using a function that binds a pair as postcondition. Such a specification can be written `Triple (list_split l)[] (fun '(l1,l2)⇒ ...)`, where the quote symbol is standard Coq syntax for binding tuples.

The mechanism of lifting is not restricted to built-in data types. It is extensible, and can be used for algebraic data types that are defined in a source ML program. Consider for example a tree data type.

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

To specify such trees, we introduce a corresponding Coq inductive data type, as shown below.

```
Inductive tree (A:Type) : Type :=
  | Leaf : tree A
  | Node : A → tree A → tree A → tree A.
```

Our lifting approach enables one to specify a function that produces a tree by binding in the postcondition a value of type `tree A`, for the appropriate type A. The specification then takes the form: `Triple (...)(...)(fun (t:tree A)⇒ ...)`.

Benefit #2: representation predicates. Lifting is useful not just for stating postconditions, but also for stating representation predicates. Consider for example the specification of `incr`.

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

There, $p \hookrightarrow n$ actually stands for $p \hookrightarrow (\texttt{val\_int}\ n)$, where the coercion `val_int` is the constructor for integers in the grammar of values. There are occasions where the need for writing this coercion (or at least taking its existence into account) shows up in unexpected ways. For example, the assertion $\exists n.\ p \hookrightarrow n \star [\mathsf{even}\, n]$ fails to type-check, because Coq infers from $p \hookrightarrow n$ that n is a value of type val. The work-around is either to write $\exists (n : \mathsf{int}).\ p \hookrightarrow n \star [\mathsf{even}\, n]$ with an explicit type annotation; or to write $\exists n.\ [\mathsf{even}\, n] \star p \hookrightarrow n$ with the integer predicate coming first; or to declare an "implicit type" asserting that every variable starting with the letter n has type int by default. Each of these three solutions is feasible but harms readability or does not scale up well. With lifting, on the contrary, we redefine p ↪ n in such a way as to mean "p points to the program value that corresponds to the logical value n". With that updated definition, which exploits typeclasses, the statement $\exists n.\ p \hookrightarrow n \star [\mathsf{even}\, n]$ typechecks as expected, with n inferred to be of type int.

The benefit of lifting is even clearer when considering representation predicates for polymorphic data structures. Consider for example the representation predicate for lists, written Mlist $L\, p$, introduced in Section 2.1.6. With the unlifted representation predicate Mlist, the logical value L denotes a list of values of type val. To specify that a particular list has contents $1 :: 2 :: 3 ::$ nil, one has to specify the list L either as "$\texttt{val\_int}\, 1 :: \texttt{val\_int}\, 2 :: \texttt{val\_int}\, 3 ::$ nil" or as "List.map $\texttt{val\_int}\ (1 :: 2 :: 3 ::$ nil$)$". In both cases, the coercion `val_int` gets in the way even further when trying to specify concrete operations on lists. With lifting, we can define a representation predicate over lists in such a way that the list L is a list made of the Coq values that corresponds to the relevant program values. Concretely, using the lifted definition of Mlist, we can write: Mlist $(1 :: 2 :: 3 ::$ nil$)\, p$.

Benefit #3: pattern matching. A third and major benefit of the lifting technique appears when reasoning about pattern matching. Consider the following OCaml code snippet.

```
match v with
| (2,true)::(3,b)::r -> t1
| _ -> t2
```

Without lifting, the hypothesis capturing the property that the value v matches the first pattern is expressed as follows.

```
∀(b:val) (r:val),
  v = val_constr "cons" [val_constr "pair" [val_int 2; val_bool true];
       val_constr "cons" [val_constr "pair" [val_int 3; b]; r ] ] → ...
```

With lifting, the same hypothesis can be expressed using typed Coq values as follows.

```
∀(b:bool) (r:list(int*bool)),
  v = (2,true) :: (3,b) :: r → ...
```

The hypothesis provided when using lifting is thus considerably more concise and easier to work with. Deriving the lifted hypothesis from the unlifted hypothesis *after* it has been generated is a fairly tedious task to perform by hand, and a challenging task to automate when the relevant type information is not available.

Summary. The lifting technique brings significant improvement on at least three aspects: (1) to simplify the writing of postconditions, (2) to simplify the manipulation of representation predicates and especially of polymorphic ones, and (3) to simplify the reasoning about data constructors. We next present the typeclass used to implement lifting.

6.2 A Typeclass for Encodable Coq Types

From this subsection onwards, we switch to using Coq syntax for stating definitions and lemmas, because it clarifies the statements in the presence of typeclasses arguments that are sometimes implicit and sometimes explicit.

The typeclass `Enc` characterizes the Coq types that correspond to a type from the programming language. A type `A` satisfying the typeclass "`Enc A`" comes with an *encoder*, defined as a function of type `A -> val`. It also comes with a proof of injectivity.

```
Class Enc (A:Type) : Type :=
  { enc : A → val;
    enc_inj : injective enc }.
```

Injectivity asserts that two distinct Coq values of an encodable type must correspond to two distinct program values. This requirement is exploited for example when reasoning about comparison operations. Injectivity may be exploited using the following lemma.

```
Lemma Enc_eq : ∀A (EA:Enc A) (V1 V2:A), (enc V1 = enc V2) ↔ (V1 = V2).
```

To every primitive type is associated an encoder. Consider for example primitive values of type `int`. A Coq integer n of type `int` (i.e., $\mathbb{Z}$), corresponds to the program value `val_int` n of type `val`. Thus, we define an instance of `Enc int`, whose encoder is the injective constructor `val_int`. Likewise, we introduce encoders for the types `unit`, `bool`, and `loc`.

```
Global Instance Enc_int : Enc int (* realized as [val_int] *)
Global Instance Enc_unit : Enc unit (* realized as [val_unit] *)
Global Instance Enc_bool : Enc bool (* realized as [val_bool] *)
Global Instance Enc_loc : Enc loc (* realized as [val_loc] *)
```

An interesting aspect is the treatment of first-class functions. A program function is *not* described in the logic as a Coq function. It would not be suitable to do so, because program functions are effectful whereas Coq functions are pure. Instead, program functions are described in the logic simply as a piece of syntax, that is, by their code. Concretely, a first-class function is specified using the type `val`, as it was already the case before the introduction of the lifting technique. For uniformity, we introduce an encoder for the type `val`, which consists of the identity function.

```
Global Instance Enc_val : Enc val (* realized as [fun (v:val) ⇒ v] *)
```

The interpretation of program functions as piece of syntax was already exploited to justify the soundness of CFML in my PhD Thesis [Charguéraud, 2010, Section 6.1]. There, the type `Func` was interpreted as the set of closed, syntactic function definitions, and the soundness of characteristic formulae was not mechanized in Coq. The key contribution of my recent research has been to show that this idea of interpreting functions as plain syntax (unlike other structured values, which are interpreted as the corresponding Coq values) can indeed be formalized as part of a *foundational* program logic.

Technically, the typeclass instance that serves as an encoder for function values, namely `Enc_val`, is defined as the identity operation. That said, from the perspective of the end-user, the definition of `Enc_val` needs not be revealed. Indeed, all the reasoning about program functions

is carried out by means of specification triples (or weakest-preconditions), which are provided via characteristic formulae, and which may be exploited for reasoning about function calls.

Another interesting aspect is the encoding of polymorphic data types. Let us describe a polymorphic encoder for pairs. Given two *encodable* types `A1` and `A2`, the encoder `enc_pair` converts a pair `(x1,x2)` of type `A1*A2` into the representation of a pair made of the encoding of `x1` and of the encoding of `x2`. In the definition shown below, `val_constr` is the Coq constructor for representing ML data constructors in our program syntax; it takes as argument the name of the data constructor, and a list of values representing the arguments to which the constructor is applied. The curly braces notation is Coq syntax for "maximally inserted arguments", meaning that the typeclass arguments are implicit and automatically inferred.

```
Definition enc_pair (A1:Type) {EA1:Enc A1} (A2:Type) {EA2:Enc A2} :=
  fun (p : A1*A2) : val ⇒
    let '(x1,x2) := p in
    val_constr "pair" (enc x1 :: enc x2 :: nil).
Global Instance Enc_pair (A1:Type) {EA1:Enc A1} (A2:Type) {EA2:Enc A2}
  : Enc (prod A1 A2) := ... (* realized as [enc_pair] *)
```

For recursive data types, definitions of encoders consist of recursive functions. For example, we show below the encoder for lists. Observe how the elements from the list are encoded with "`enc x`", which implicitly refers to the encoder `EA1` associated with the type `A1` of the elements.

```
Definition enc_list (A1:Type) {EA1:Enc A1} :=
  fix f (l:list A1) : val :=
    match l with
    | nil ⇒ val_constr "nil" nil
    | x::l' ⇒ val_constr "cons" ((enc x)::(f l')::nil)
    end.

Global Instance Enc_list : ∀(A1:Type) {EA1:Enc A1}, Enc (list A1) := ...
  (* realized as [enc_list] *)
```

CFML features a generator that takes as input an OCaml data type definition and generates the Coq typeclass definition for the corresponding encoder. Thus, in practice, the end-user never needs to worry about writing definitions of encoders.

6.3 Definition of Lifted Triples

A lifted triple takes the form `Triple t H Q`. As for unlifted triples, `t` is a term of type `trm` and `H` is a precondition of type `hprop`. The postcondition `Q`, however, no longer has type `val->hprop`. It now has type `A->hprop`, for some encodable type `A`. The predicate `Triple` admits two implicit arguments: a type `A`, and an encoder for that type, named `EA`, of type `Enc A`. Thus, in its explicit form, the predicate is written `@Triple t H A EA Q`, where the leading "`@`" symbol is Coq syntax for providing all arguments explicitly.

The lifted triple judgment is defined in terms of conventional triples. The definition involves a postcondition asserting that the output program value, named `v` (of type `val`) should correspond to the encoding of a Coq value, named `V` (of type `A`), such that `V` satisfies the postcondition `Q` (of type `A->hprop`).

```
Definition Triple (t:trm) (A:Type) {EA:Enc A} (H:hprop) (Q:A→ hprop) : Prop :=
  triple t H (fun (v:val) ⇒ ∃(V:A), [v = enc V] ⋆ Q V.
```

In our Coq formalization, we introduce a *postcondition transformer*, named `Post`, for interpreting a postcondition of type `A→ hprop` as a postcondition of type `val→ hprop`.

```
Definition Post (A:Type) {EA:Enc A} (Q:A→ hprop) : val→ hprop :=
  fun v ⇒ ∃V, \[v = enc V] ⋆ Q V.
```

Using `Post`, the definition of lifted triple may be reformulated more concisely.

```
Definition Triple (t:trm) (A:Type) {EA:Enc A} (H:hprop) (Q:A→ hprop) : Prop :=
  triple t H (Post Q).
```

6.4 Lifted Representation Predicates

Representation of lifted singleton heap predicates. The unlifted heap predicate for describing a singleton heap has the form $p \hookrightarrow v$, where v is a value of type val. In our Coq formalization, the corresponding predicate is named `hsingle v p`. The lifted heap predicate for singleton heap is written `Hsingle V p`, where `V` is a logical value of some encodable type `A`. This lifted predicate is defined in terms of the unlifted version, simply by asserting that, at location `p`, the heap contains the *encoding* of the logical value `V`, that is, the program value that corresponds to `V`.

```
Definition Hsingle (A:Type) {EA:Enc A} (V:A) (p:loc) : hprop :=
  hsingle p (enc V)
```

When we write `Hsingle V p`, the type argument `A` and the typeclass argument `EA` are implicit. They are automatically inferred from the type of the value `V`.

Likewise, we lift the representation predicate for record fields, written Field $p\,k\,v$ in Section 3.8.

```
Definition Hfield (A:Type) {EA:Enc A} (V:A) (p:loc) (f:field) : hprop :=
  hfield p f (enc V).
```

We are next seeking to define a lifted version of the representation predicate for record, written Record in Section 3.8. The challenge is to define it in a generic way, even though the types of the fields may vary from one record to the next. Our solution is based on the use of dependent pairs, made of an encodable type and a value of that type. Let us begin with the formalization of such dependent pairs, which, technically, consist of triples because they additionally carry an encoder.

Representation of heterogeneous lifted values. We let `dyn` denote a record made of a type, an encoder for that type, and a value of that type.

```
Record dyn := dyn_make {
  dyn_type : Type;
  dyn_enc : Enc dyn_type;
  dyn_value : dyn_type }.
```

A fundamental property is that a logical value of type `dyn`, which packs a value `V` of some encodable type `A`, can always be converted into a program value of type `val`, by applying to `V` the encoder associated with the type `A`. This conversion is implemented by the function `dyn_to_val` shown below. Recall that"`@`" is Coq syntax for disabling implicit arguments.

```
Definition dyn_to_val (d:dyn) : val :=
  @enc (dyn_type d) (dyn_enc d) (dyn_value d).
```

Representation of lifted records. We are now ready to define the lifted representation predicate for records, written `Record K p`, where `K` is a list of pairs, each made of a field name (internally, an offset) and an element of type `dyn`. Each element of type `dyn` describes the contents of a field as an encodable logical value. The predicate `Record` is defined as the iterated separating conjunction of the fields, each of them being described using the lifted predicate for record fields, namely `Hfield`.

```
Definition Fields : Type := list (field * dyn).

Fixpoint Record (K:Fields) (p:loc) : hprop :=
  match K with
  | nil ⇒ hheader 0 p
  | (f, dyn_make A EA V)::K' ⇒ (Hfield V l f) ⋆ (p ⇝ Record K')
  end.
```

Observe how the expressive power of Coq's dependent types is at play here: the pattern matching binds a type `A`, an encoder `EA` for that type, and a value `V` of that type. The encoder `EA` corresponds to a typeclass that appears as an implicit argument of the predicate `Hfield`.

Lifted representation predicate for mutable lists. We end this section with a presentation of the lifted version of the representation predicate MList $L\ p$ for mutable linked lists. Let us first recall its definition in the unlifted case, where the list `L` is described as a list of program values, of type `list val`. In the definition shown below, we use the syntax p ↪ { `head` := x; `tail` := y} for "`Record` ((`head`, x):: (`tail`, y):: `nil`)p".

```
(* Without lifting *)
Fixpoint MList (L:list val) (p:loc) : hprop :=
  match L with
  | nil ⇒ \[p = null]
  | x::L' ⇒ ∃q, p ↪ { head := x; tail := val_loc q}) ⋆ (MList L' q)
  end.
```

In the lifted version of `MList`, the elements are described by a logical list `L` of type `List A`, for some encodable type `A`. In the definition shown below, we write p ↪ { `head` := x; `tail` := q} for "`Record` ((`head`, `dyn_make` x):: (`tail`, `dyn_make` q):: `nil`)p". Recall that `dyn_make` x builds a value of type dyn; the type and its encoder are automatically inferred from the type of x.

```
(* With lifting *)
Fixpoint MList (A:Type) {EA:Enc A} (L:list A) (p:loc) : hprop :=
  match L with
  | nil ⇒ \[p = null]
  | x::L' ⇒ ∃q, p ↪ { head := x; tail := q} ⋆ (MList L' q)
  end.
```

As announced earlier, the benefits of the lifted representation predicate is to ease the reasoning about the contents of the list. Consider for example a function that increments all the integer values stored in a list. It is specified as follows, using a list of integers.

```
Lemma mlist_incr_spec : ∀(L:list int) (p:loc),
  Triple <{ mlist_incr p }>
    (MList L p)
    (fun (_:unit) ⇒ MList (List.map (fun n ⇒ n+1) L) p).
```

6.5 Attempt at a Lifted Characteristic Formulae Generator

I have explored the possibility of defining, inside Coq, a *lifted characteristic formulae generator*. Such a generator applies to a well-typed term t. The knowledge of the types of every subterm of t is required for producing the lifted characteristic formula of t. Moreover, the type of the formula produced as output *depends* on the type of the term t. Indeed, it has type `(T->hprop)->hprop`, where `T` denote the Coq type that corresponds to the type of t.

Formalizing such a generator involves (at least) three major challenges.

- First, it would require defining a function that translates ML types into the corresponding Coq types. It is not clear whether this can be achieved in a modular way, that is, for an extensible collection of ML types.
- Second, it would require a function that translates well-typed ML values into the corresponding Coq values. Such a dependently-typed function is challenging to define, and even more so if one wants to handle algebraic values of user-defined types. Polymorphic OCaml values are also particularly delicate to map to polymorphic Coq values, due to the need to cope with a variable number of type quantifiers.
- Third, it would require typing environments. The environment E used in the unlifted characteristic formula generator in the previous chapter needs to be refined into an environment that maps program variables to logical values *of the appropriate type*. The definition of the dependently typed generator must thus involve as extra argument a typing environment Γ, a proof that the environment E respects is compatible with Γ, and a proof that the term t provided is well-typed in Γ. These invariants need to be maintained throughout the recursive definition of the characteristic formulae generator, extending the environment each time a binder is traversed. Formalizing the lifted generator as a Coq function thus requires nontrivial dependently typed programming.

Considering all these highly technical—possibly insurmountable—obstacles, I chose to follow an alternative approach, based on the use of an external generator. Keep in mind that involving an external tool is required in any case for generating the inductive definitions that correspond to the user-defined, algebraic data types. Indeed, inductive definitions are not first class entities in Coq, so there is no hope whatsoever to have them be generated by a Coq function.

6.6 An External Characteristic Formulae Generator

The external generator consists of an OCaml program that performs the following steps.

1. The generator takes as input the name of an OCaml source file. It invokes the standard OCaml parser to parse the file and obtain the *parse tree*—an abstract syntax tree (AST).
2. On that parse tree, it invokes the OCaml typechecker to produce the *typed tree*. I use a version of the typechecker that I have patched to keep track of where each type variable is bound. Technically, we obtain an AST with explicit type quantifiers, in System-F style.
3. The generator produces a Coq source file. For each type definition from the OCaml source code, the Coq file contains a corresponding type definition. For each top-level value definition from the OCaml source, the Coq file contains one definition and one lemma, as described next.

For a top-level function definition let rec $f\,x\,=\,t$, a Coq constant named `f` of type `val` in introduced. The associated lemma follows the pattern presented in Section 5.2 (step 2). For a generator that does not exploit the *lifting* technique, it would take the form:

$$\forall X\,H\,Q.\ H \vdash (\mathsf{cf}_{(f,\mathtt{f})::(x,X)::E}\,t\,Q) \ \Rightarrow\ \{H\}\ (\mathtt{f}\ X)\ \{Q\}.$$

This statement is adapted to incorporate the lifting technique as described further in this section. For a value definition other than a function, the OCaml definition takes the form let $x = v$. A Coq constant named `x` is introduced, and the associated lemma asserts that `x = V`, where `V` is the Coq value that corresponds to the OCaml value `v`. Details on the lifting of values appear further on.

Until 2021, those produced lemmas were all admitted as axioms. Since 2022, I provide a set of tactics that can be used to prove such lemmas, by relating the lifted characteristic formula with the unlifted one computed inside Coq. I am currently working on tooling for automatically generating the relevant proof scripts. When this tooling is ready, all the lifted characteristic formulae that are generated will be justified in a foundational manner.

In what follows, I describe the key features of the generator. I try to remain at a sufficient high level to avoid entering all the technicalities of formalizing the notion of a typed AST, as I did in my PhD thesis. Thereafter, *CF* is an abbreviation for *characteristic formula*.

Lifting of values. The external generator includes a function that, given an OCaml value, computes the corresponding Coq value. For example, the OCaml value `(2,true)::(3,b)::r` is described by the Coq value `(2,true)::(3,b)::r`. This translation is straightforward—and that is the whole point of lifting, to avoid the encodings associated with the deep embedding. The only complication is for handling argument-free polymorphic data constructors such as `nil` or `None`. For such constructors, we need to decorate them with a type annotation to avoid situations where the generated Coq formula fails to typecheck due to missing type information.

A feature of the external CF generator is that it simplifies on-the-fly the treatment of calls to pure functions, such as total arithmetic operators, boolean operators, or simple list combinators (`rev`, `append`, etc). The use of such pure functions in programs is pervasive. For such function calls, there is no need to exploit the general CF for a function call. Indeed, it is possible to view the result of these operations directly as Coq values. For example, the pure OCaml term `3 + 2 * n`, where `n` is a program variable that admits the OCaml type `int`, can be interpreted as the Coq value `3 + 2 * n`, where `n` is a Coq variable that admits the Coq type `int`. Recognizing commonly-used pure functions is a straightforward addition to the CF generator, yet in practice it dramatically reduces the number of proof steps involved.

Lifted formulae. An unlifted CF admits the type `formula`, which describes a function that expects a postcondition of type `val→hprop`, and returns a heap predicate of type `hprop`. A lifted CF admits the type `Formula`, which describes a function that expects a type `A`, an encoder of type `Enc A`, a postcondition of type `A→hprop`, and returns a `hprop`.

```
Definition formula : Type := (val → hprop) → hprop.
Definition Formula : Type := ∀A (EA:Enc A), (A → hprop) → hprop.
```

Thereafter, I use the notation `^F Q` as a shorthand for `F _ _ Q`, that is, to apply a formula `F` to a postcondition `Q` while leaving the type and its encoder to be inferred by Coq's typechecker. Indeed, Coq unfortunately supports implicit arguments only for top-level constants, but not for local variables. Hence the need for a custom notation.

Lifted "framed" predicate. The construction of lifted formulae involves a lifted version of the framed predicate. Recall form Section 5.3 that this predicate is inserted at every node of a CF. The lifted definition of framed is defined on top of a slightly generalized version of framed, which handles postconditions over values of an arbitrary type `B` instead of imposing the type `val`.

```
Definition framed (A:Type) (F:(A→ hprop)→ hprop) : (A→ hprop)→ hprop :=
  fun Q ⇒ ∃Q', F Q' ⋆ (Q' -⋆ (Q ⋆ ⊤)).
```

The lifted predicate `Framed` may then be defined as follows.

```
Definition Framed (F:Formula) : Formula :=
  fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒ framed (@F A EA) Q.
```

Lifted CF for let-bindings. Consider a non-polymorphic let-binding of the form let $x = t_1$ in t_2. (Polymorphic let-bindings are discussed further on.) Let F1 be the CF produced for the subterm t_1, and G2 be a Coq function that, given a value X, returns the CF associated with the term $[X/x]\,t_2$. The CF produced for the let-binding is `CF_let F1 G2`. This combinator `CF_let` depends on A1, the Coq type that corresponds to the OCaml type of t_1.

```
Definition CF_let (F1:Formula) (A1:Type) {EA1:Enc A1} (G2:A1→ Formula)
 : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    ^F1 (fun (X:A1) ⇒ ^(G2 X) Q)).
```

An alternative definition of `CF_let` quantifies over the postcondition Q1 of t_1.

```
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    ∃(Q1:A1→ hprop), ^F1 Q1 ⋆ \[∀ (X:A1), Q1 X ⊢ ^(G2 X) Q]).
```

The interest of this second definition is that it is closer to the way we want to reason about a let-reasoning. Such reasoning is captured by the lemma that serves as basis for implementing the tactic `xlet`. This lemma, shown below, produces two subgoals: one for reasoning about t_1, and one for reasoning about t_2.

```
Lemma xlet_lemma : ∀(A1:Type) (EA1:Enc A1) (Q1:A1→ hprop) (H:hprop),
  ∀(F1:Formula) (G2:A1→ Formula) (A:Type) (EA:Enc A) (Q:A→ hprop),
  H ⊢ ^F1 Q1 →
  (∀ (X:A1), Q1 X ⊢ ^(G2 X) Q) →
  H ⊢ ^(@CF_let F1 A1 EA1 G2) Q.
```

The intermediate postcondition Q1 may be either supplied explicitly by the user, or in simple cases it may be inferred from the reasoning about t_1.

Lifted CF for sequences. Consider a sequence of the form t_1 ; t_2, where t_1 is a term of type unit. Let F1 and F2 be the CF produced for the subterms t_1 and t_2. The CF produced for the sequence is `CF_seq F1 F2`. The definition of the combinator `CF_seq` is slightly complicated by the fact that we wish to enforce that F1 applies to a postcondition of type `unit→ hprop`. Below, `tt` denotes the unit value in Coq, and `Q1 tt` is a heap predicate that describes the intermediate state between the evaluation of t_1 and t_2.

```
Definition CF_seq (F1 F2:Formula) : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    ∃(Q1:unit→ hprop), ^F1 Q1 ⋆ \[Q1 tt ⊢ ^F2 Q]).
```

The following lemma shows that the definition of `CF_seq` enables the tactic `xseq` to produce the two expected subgoals.

```
Lemma xseq_lemma : ∀(Q1:unit→ hprop) (H:hprop) (A:Type) (EA:Enc A) (Q:A→ hprop),
  ∀(F1 F2:Formula),
  H ⊢ ^F1 Q1 →
  (Q1 tt ⊢ ^F2 Q) →
  H ⊢ ^(CF_seq F1 F2) Q.
```

Lifted CF for values. Consider a value v. Let V denotes its encoding in Coq, and B denotes the type of V in Coq. The characteristic formula for this value expects as argument a postcondition Q of type A→ hprop, for some encodable type A. Intuitively, by virtue of typing, there should be no reason to consider specifications involving a type A that would differ from the type B associated with the value at hand. Nevertheless, the CF produced must admit the type `Formula`, and thus

apply to a postcondition of type A→ hprop for an arbitrary encodable type A. We handle the potential discrepancy by asserting that the postcondition Q should hold of a value V' of type A that admits the same encoding in the deep embedding as the value V of type B.

```
Definition CF_val (B:Type) {EB:Enc B} (V:B) : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    ∃(V':A), \[enc V' = enc V] ⋆ Q V').
```

In practice, we are only interested in proving *well-typed specifications*, in which the types A and B are equal. The lemma shown below shows that the expected reasoning rule holds in that case.

```
Lemma xval_lemma : ∀(A:Type) {EA:Enc A} (V:A) (H:hprop) (Q:A→ hprop),
  H ⊢ Q V →
  H ⊢ ^(CF_val V) Q.
```

Lifted CF for conditionals. Consider a well-typed conditional if v then t_1 else t_2, where v has type bool. The lifting of the value v produces a Coq boolean value, call it b. The CF for t_1 and t_2 is computed recursively, producing formulae F1 and F2. The CF produced for the conditional is CF_if b F1 F2, where CF_if is defined as shown below.

```
Definition CF_if (b:bool) (F1 F2:Formula) : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
            if b then ^F1 Q else ^F2 Q).
```

Compared with the unlifted definition (cf_if, introduced at the last step of Section 5.2), there is no need to existentially quantify over a boolean value b equal to the argument v of the conditional, because the argument v, which admits type bool, necessarily translates into a boolean value.

Lifted CF for applications. Consider a n-ary application of the form $f\ v_1 \ldots v_n$. It is described in the deep embedding as trm_apps (trm_val f)(List.map trm_val vs), where trm_apps is the term constructor for n-ary applications (it has type trm → list trm → trm).

Let VS denotes the list of Coq values that corresponds to the program values vs. This list admits type list dyn. In other words, each argument is packed with its type. The value f is a first-class function, so it remains described at type val. The same application $f\ v_1 \ldots v_n$ can be described in the deep embedding as Trm_apps f Vs, where the smart constructor Trm_apps is defined as shown below. This definition uses the conversion function from dyn_to_val, which encodes each argument V of type dyn into the corresponding program value.

```
Definition Trm_apps (f:val) (Vs:list dyn) : trm :=
  trm_apps (trm_val f) (List.map (fun V ⇒ trm_val (dyn_to_val V)) Vs).
```

The CF for an application is expressed in terms of the *lifted weakest-precondition predicate*, written Wp t, and defined as follows.

```
Definition Wp (t:trm) : Formula :=
  fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒ wp t (Post Q).
```

For an application Trm_apps f Vs, the generator produces the formula CF_app f VS, defined as shown below, in terms of the lifted Wp predicate.

```
Definition CF_app (A:Type) {EA:Enc A} (f:val) (Vs:list dyn) : Formula :=
  Framed (Wp (Trm_apps f Vs)).
```

Lifted CF for function definitions. Recall from Section 5.2 (step 2) that, given a function definition $\mu f.\lambda x.t$, the lifted CF generator defines $\mathsf{cf}_E\,(\mu f.\lambda x.t)$ as:

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

This definition generalizes to n-ary function. To keep the presentation simple, we consider a function of arity 2, written $\mu f.\lambda x_1 x_2.t$. The unlifted CF for $\mathsf{cf}_E\,(\mu f.\lambda x_1 x_2.t)$ is:

$$\begin{array}{l}\mathsf{framed}\,(\lambda Q.\\ \quad \forall F.\ [\forall X_1 X_2 H Q'.\ H \vdash \mathsf{cf}_{(f,F)::(x_1,X_1)::(x_2,X_2)::E}\,t\,Q' \ \Rightarrow\ \{H\}\ (F\,X_1\,X_2)\ \{Q'\}] \mathrel{-\!\!\star} Q\,F).\end{array}$$

What matters here is that the function F is viewed as an abstract entity by the user, who does not have access to the code that defines the function. The function is described extensionally, that is, by means of the behavior of its application to an arbitrary argument X. From this description, the user can prove a particular specification, in the form of a triple involving $F\,X$. Subsequently, this triple may be used to reason about particular functions calls of the form $F\,V$. In summary, when exploiting characteristic formulae with lifting, a function F is represented in the logic by the syntax of its definition. Yet, this representation is never revealed to the end-user; it only plays a role in the soundness proof.

Besides, three changes are involved in the process of lifting the CF of a function definition. First, the formula involves the construct `Trm_apps` for describing the application in terms of the *lifted* arguments. Second, the postcondition `Q'` admits a type of the form `A'→hprop`, thus we need to quantify the type `A'` and its encoder. Third, the typechecking of the body of the function may involve type variables, which we need to quantify. Let `A1`, `A2` and `A3` represent these variables. The hypothesis provided about the function in the CF is as shown below, where `F1` denotes the characteristic formula of the body t. The types of the arguments `X1` and `X2` (not shown below) are computed from the types of the arguments of the OCaml function. Likewise, the type of the postcondition `Q` depends on the return type of the OCaml function.

```
∀(A1:Type) {EA1:Enc A1} (A2:Type) {EA2:Enc A2} (A3:Type) {EA3:Enc A3} X1 X2 H Q,
  (H ⊢ ^F1 Q) →
  Triple (Trm_apps f ((dyn_make X1)::(dyn_make X2)::nil)) H Q
```

A subtlety is that the set of type variables to quantify (`A1`, `A2`, etc.) does not precisely match the type variables that appear in the polymorphic type of the function. On the one hand, additional type variables may be needed. For example, `let f x = (let y = ref [] in x)` is an OCaml function of type `A → A`. The typechecking of the empty list involves a local type `B`, which does not appear in the result type of the function. Both the variables `A` and `B` need to be quantified in the CF. On the other hand, there are type variables that appear in the OCaml type of the function but that need not be quantified in the CF. For example, `let f (g : 'a -> 'a)()= ()` is a function of type `('a → 'a)→ unit → unit`. The argument `g` is a function, described at type `val` in the CF. Thus, the variable `'a` has no counterpart in the CF. CFML computes the exact set of variables that need to be quantified.

Our implementation supports functions of arbitrary arity. Moreover, it supports mutually recursive functions. For those, we need to quantify a name for each of the function, then provide one hypothesis for describing the CF of each of the function independently.

Lifted CF for let-bindings on polymorphic values. Consider a term $\mathsf{let}\,x = v\,\mathsf{in}\,t$, where v corresponds to a polymorphic value. Note that we consider a value rather than a general term for the body to satisfy the *value restriction*. (We do not treat the *relaxed* value restriction implemented in OCaml.) If v_1 corresponds to a polymorphic function, then it is described in lifted CF at the

monomorphic type `val`, so there is no complication. The interesting case is when v is not a function, but a polymorphic constant such as the empty list, written [], or `None`.

We represent such polymorphic OCaml constants using the corresponding Coq constants. For example, the OCaml constant [], which admits the type 'a `list`, is represented in Coq as the value `fun` (A:`Type`)⇒ `@nil` A, of type ∀(A:`Type`), `list` A. As another example, a pair ([],[]) type-checked at type 'a `list` ∗ 'a `list` is represented in Coq as `fun` (A:`Type`)⇒ (`@nil` A, `@nil` A), which admits the Coq type ∀(A:`Type`), `list` A ∗ `list` A.

In what follows, we name `V` the logical value that corresponds to v. We name `A1` the polymorphic Coq type that corresponds to the OCaml type of `V`. The CF produced for let $x = v$ in t is of the form `CF_letval V G`, where `G` is a Coq function that, given a value `X`, returns the CF associated with the term $[X/x]\,t$. The definition of `CF_letval` provides the assumption `X = V`, which is an equality between two polymorphic constants of type `A1`.

```
Definition CF_letval (A1:Type) (V:A1) (G:A1→ Formula) : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    ∀(X:A1), \[X = V] -∗^(G X) Q).
```

The following lemma shows how this definition is exploited by the tactic `xletval`. For example, an OCaml binding "`let x = None`" gives rise in the program logic to the Coq hypothesis "`x = None`", where `x` has type "∀A, `option` A"—exactly what the user might *naturally* expect.

```
Lemma xletval_lemma :
  ∀(A:Type) (H:hprop) (G:A→ Formula) (V:A) (A1:Type) {EA1:Enc A1} (Q:A1→ hprop),
  (∀ (X:A1), X = V → (H ⊢ ^(G X) Q)) →
  H ⊢ ^(CF_letval V G) Q.
```

Lifted CF for pattern matching. For the purpose of generating CF, we view a pattern-matching construct with n branches as a cascade of n constructs that each test a single pattern. This cascade is terminated by an "assert false", which corresponds to the `Match_failure` exception in OCaml. With this view, we only have to consider terms of the form "match v with $(p \Rightarrow t_1) \mid t_2$". This term evaluates to t_1 if the value v matches the pattern p, and to t_2 otherwise.

Our CF generator processes such a term as follows. First, if the pattern contains wildcards, they are replaced with fresh variables. At that point, the pattern p contains a certain number of variables. For simplicity, assume the pattern binds three variables, named x_1, x_2 and x_3. The generator then computes the (closed) Coq value `V` that corresponds to the value v being matched, and compute the (non-closed) Coq value `W` that corresponds to the pattern p. Finally, the generator produces the formula shown below, where `F1` and `F2` denote the CF associated with t_1 and t_2, respectively, and where a Coq variable `Xi` (of the appropriate type) is introduced for each pattern variable x_i.

```
Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
      (∀ X1 X2 X3, \[V = W] -∗^F1 Q)
   ⩓ (\[ ∀X1 X2 X3, V ≠W] -∗^F2 Q)).
```

The first conjunct asserts that, if there is an instantiation of the variables `Xi` that makes the pattern `W` match the value `V`, then the user has to reason about the behavior of t_1. This behavior is described by the formula `F1`, which may refer to the variables `Xi`. The second conjunct asserts that, if no possible instantiation of the variables `Xi` can make the pattern `W` match the value `V`, then the user has to reason about the behavior of t_2, described by `F2`.

As mentioned earlier, the key benefits of lifted CF is that the equality `V = W` and the disequality `V ≠W` are stated directly in terms of Coq values, without any reference to constructors from the deep embedding.

Additional features for pattern matching. The "assert false" at the end of the cascade of patterns gives rise to the formula `CF_fail`, which requires the user to establish a contradiction.

```
Definition CF_fail : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    \[False]).
```

When a pattern matching is recognized as *exhaustive* by the OCaml typechecker, we can save the user the burden of proving that the catch-all branch is unreachable. In that catch-all branch, rather than generating `CF_fail`, which *requires* the user to prove False, we generate `CF_done`, which instead allows the user to *assume* False. This assumption makes it trivial to discard the proof obligation. The definition of `CF_done` is as follows.

```
Definition CF_done : Formula :=
  Framed (fun (A:Type) (EA:Enc A) (Q:A→ hprop) ⇒
    \[False] -⋆\[True]).
```

Our implementation supports *alias-patterns*, which bind additional names in the continuation of a branch. It also supports simple forms of *when-clauses*. We require the body of the when-clause to consist of an expression that can be directly translated into a Coq value, e.g., a boolean formula involving variables and pure arithmetic operations.

Treating the general case of when-clauses that perform effectful operations is left for future work. To appreciate the trickiness of the semantics at play, consider the fact that it took years for OCaml developers to realize that if a when-clause performs side-effects that modify the data under scrutiny by the pattern matching, then the resulting program could crash. As of writing, we are not aware of any proposal for describing an expressive yet safe semantics for effectful when-clauses.

6.7 Specifications for Operations on Lifted Records

In this section, I explain how to reason about a record read or write operations, with respect to either the lifted field representation predicate (`Hfield`) using small-footprint specifications, or with respect to the record representation predicate (`Record`) using large-footprint specifications. The technical difficulty associated with large-footprint specifications, which are proved correct with respect to the small-footprint specifications, is that they involve reading and writing Coq values in heterogeneous lists (of type `Fields`).

Small-footprint specifications for operations on lifted records. The small footprint specification for a read operation involves a heap predicate of the form `Hfield V p f`, describing a single field whose contents stores a value `V` of some encodable type `A`. The specification is expressed using a lifted triple whose postcondition describes a result of type `A`, rather than a result of type `val` as it was the case with unlifted triples.

```
Lemma Triple_get_field : ∀(p:loc) (f:field) (A:Type) {EA:Enc A} (V:A),
  Triple <{ val_get_field f p }>
    (Hfield V p f)
    (fun (r:A) ⇒ \[r = V] ⋆ Hfield V p f).
```

The specification of a write operation involves two values: `V1` denotes the value being written, and `V2` denotes the previous contents of the cell. For well-typed ML programs, those two values are described at a same type `A`. That said, Separation Logic supports reasoning about *strong updates*, i.e., write operations that modifies the type of the contents of a memory cell. The specification

for a strong update operation is relaxed in that it quantifies separately over the type of the old contents and that of the new contents.

```
Lemma Triple_set_field_strong : ∀(p:loc) (f:field),
  ∀(A1:Type) {EA1:Enc A1} (V1:A1) (A2:Type) {EA2:Enc A2} (V2:A2),
  Triple <{ val_set_field f p (enc V2) }>
    (Hfield V1 p f)
    (fun (_:unit) ⇒ Hfield V2 p f).
```

Large-footprint specifications for operations on lifted records. As explained in Section 3.8, it is very convenient for the user to be able to reason about record operations using directly the `Record` predicate, without having to first isolate the relevant field. We next present a Coq function called `record_get_spec` that, given a field `f` of a record at location `p`, *computes* the relevant lifted triple for a read operation on that field.

The auxiliary function `record_get_dyn` takes as argument `f`, a field name, and `K`, a description of lifted record fields. It returns the element of type `dyn` associated with field `f` in the association list `K`. It returns `None` if the field is missing—this case might arise if the record representation predicate had been split prior to reasoning on the read operation.

```
Fixpoint record_get_dyn (f:field) (K:Fields) : option dyn :=
  match K with
  | nil ⇒ None
  | (f',d)::K' ⇒
     if field_eq_dec f f'
       then Some d
       else record_get_dyn f K'
  end.
```

The function `record_get_spec` also takes `f` and `K` as argument. It invokes the auxiliary function `record_get_dyn`. If it obtains a `dyn` element of the form `dyn_make A EA V`, then `V` is the logical value that describes the program value being returned by the read operation. In this case, the function `record_get_spec` returns the statement of a lifted triple, whose precondition is `Record K p` and whose postcondition is `fun (R:A)⇒ [R = V] ∗ Record K p`. This postcondition asserts that the result value `R`, described as a logical value of type `A`, is equal to the contents of the fields, described by the logical value `V`.

```
Definition record_get_spec (f:field) (K:Fields) : option Prop :=
  match record_get_dyn f K with
  | None ⇒ None
  | Some (dyn_make A EA V) ⇒ Some (∀ (p:loc),
     Triple <{val_get_field f p}> (Record K p) (fun (R:A) ⇒ \[R = V] ∗ Record K p))
  end.
```

We accompany the function `record_get_spec` with a generic proof establishing that if this function succeeds in computing a lifted triple, then this lifted triple is correct. In the lemma below, `P` denotes the statement of that triple.

```
Lemma record_get_spec_correct : ∀(f:field) (K:Fields) (P:Prop),
  (record_get_spec f K = Some P) → P.
```

The function `record_get_spec` for computing the relevant lifted triple, as well as its accompanying correctness lemma, are automatically exploited by means of a CFML tactic that process record operations. The treatment of write operations, not shown, follows a similar pattern.

6.8 Validation of Lifted Characteristic Formulae

In the last section of this chapter, I explain how to formally justify the correctness of the lemmas generated by the external characteristic formula generated for each top-level definition.

Treatment of values other than functions. Consider a value definition other than a function. The OCaml definition is of the form let $x = v$. The Coq definition is `Definition x := V`, where `V` is the Coq value that corresponds to the OCaml value v. The generated Coq lemma, which asserts that `x = V`, trivially holds by definition of `x`. Yet, it does not connect `V` with v. To that end, we generate an additional lemma, stating the equality `enc V = v`, where `v` denotes the deep embedding of the value v. This equality is proved by the `reflexivity` tactic of Coq, which triggers the evaluation of the encoder function. This additional lemma would presumably be exploited by a framework that would connect CFML with a verified compiler.

Treatment of functions Consider now a top-level function definition let rec $f\ x = t$. As explained earlier, a Coq constant named `f` of type `val` is introduced. The associated lemma takes the form shown below, where `F1` denotes the lifted CF of the body `t`, where the types `T1` and `T` depend on the type of the function, and where `A1` is a type variable involved in the typechecking of t.

```
∀(A1:Type) {EA1:Enc A1} (X:T1) (H:hprop) (Q:T→ hprop),
  (H ⊢ ^F1 Q) →
  Triple (Trm_apps f ((dyn_make X)::nil)) H Q
```

Our approach is to prove this statement by exploiting the formally verified unlifted CF that computes inside Coq. This formula takes the form shown below, where `f1` denotes the unlifted CF of the body `t`. (Thereafter, variables denoting unlifted formulae are written in lowercase: `f1`, `g2`, etc.)

```
∀(x:val) (H:hprop) (Q:val→ hprop),
  (H ⊢ f1 Q) →
  triple (trm_apps f (x::nil)) H Q
```

To relate the two statements, we need to establish that a proof carried out by the user of the entailment `H ⊢ ^F1 Q` implies the validity of the entailment `H ⊢ f1 (Post Q)`. Recall that `Post Q` denotes the *unlifted* counterpart of `Q`. Thus, it suffices to establish: `^F1 Q ⊢ f1 (Post Q)`. The rest of this section explains how to establish an entailment of this form for relating a lifted CF with its corresponding unlifted CF, by following the structure of the code in a systematic manner.

The "Lifted" judgment. Let us first introduce the key judgment on which our proofs of correctness of lifted CF are based. The predicate `Lifted F1 f1` asserts that the lifted formula `F1` is *justified* by the unlifted formula `f1`.

```
Definition Lifted (F1:Formula) (f1:formula) : Prop :=
  ∀(A:Type) (EA:Enc A) (Q:A→ hprop), ^F1 Q ⊢ f1 (Post Q).
```

Key lemmas. We next present the key lemmas that enable relating a lifted CF with its corresponding unlifted CF. Currently, I apply these lemmas by hand, but I am working on automating the process. In a few lemmas, a side-condition of the form `isframed f` appear. This judgment asserts that `f` is a formula that is of the form `framed f'`. Whenever we apply these lemmas, the side-condition holds by construction of the CF generator, which applies the `framed` predicate at every node in the CF.

A first, auxiliary lemma, is involved in the proof of the remaining lemmas. Intuitively, it explains that the lifted `Framed` predicate matches the unlifted `framed` predicate.

```
Lemma Lifted_framed : ∀(F1:Formula) (f1:formula),
  Lifted F1 f1 →
  Lifted (Framed F1) (framed f1).
```

A second, auxiliary lemma is involved for handling applications. It relates the lifted `Wp` predicate with the unlifted wp predicate.

```
Lemma Lifted_wp : ∀(t:trm),
  Lifted (Wp t) (wp t).
```

Then, we have one lemma to handle each kind of term construct.

```
Lemma Lifted_val : ∀(A:Type) (EA:Enc A) (V:A) (v:val),
  v = enc V →
  Lifted (CF_val V) (cf_val v).

Lemma Lifted_let : ∀(F1:Formula) (f1:formula) (A1:Type) {EA1:Enc A1}
 ∀(G2:A1→ Formula) (g2:val→ formula),
  Lifted F1 f1 →
  (∀ (X:A1), Lifted (G2 X) (g2 (enc X))) →
  isframed f1 →
  Lifted (CF_let_trm F1 G2) (cf_let f1 g2).

Lemma Lifted_app : ∀(A:Type) {EA:Enc A} (f:val) (Vs:dyns) (vs:vals),
  List.map (fun V ⇒ trm_val (dyn_to_val V)) Vs = trms_vals vs →
  Lifted (@CF_app A EA f Vs) (cf_app f vs).

Lemma Lifted_if : ∀(F1 F2:Formula) (f1 f2:formula) (b:bool) (v:val),
  v = enc b →
  Lifted F1 f1 →
  Lifted F2 f2 →
  Lifted (CF_if b F1 F2) (cf_if v f1 f2).
```

Lemma for top-level functions. The most technical lemma is the one handling a top-level function definition. Let us first present a simplified version for the case of unary functions.

```
Lemma Triple_of_CF_and_Lifted_fix :
  ∀(H:hprop) (A:Type) (EA:Enc A) (Q:A→ hprop) (F1:Formula),
  ∀(F:val) (f x:var) (v:val) (t:trm),
  F = val_fix f x t →
  f ≠x →
  H ⊢ ^F1 Q →
  dyn_to_val V = v →
  Lifted F1 (cf ((f,F)::(x,v)::nil) t) →
  Triple (Trm_app F V) H Q.
```

The generalization to n-ary functions is more technical—reading it is optional.

```
Lemma Triple_of_CF_and_Lifted_fixs :
  ∀(H:hprop) (A:Type) (EA:Enc A) (Q:A→ hprop) (F1:Formula),
  ∀(F:val) (f:var) (xs:list var) (Vs:list dyn) (vs:list val) (t:trm),
  F = val_fixs f xs t →
  H ⊢ ^F1 Q →
  trms_to_vals (List.map (fun (V:dyn) ⇒ trm_val (dyn_to_val V)) Vs)
```

```
 = Some vs →
disjoint_vars (bind_var f) xs (List.length vs) →
Lifted F1 (cf (List.combine (f::xs) (F::vs)) t) →
Triple (Trm_apps F Vs) H Q.
```

Pure functions. One complication is related to the feature of our external CF generator of recognizing the application of basic pure functions. In the lifted CF generator, such an application is mapped directly to a value. In the unlifted CF generator, however, such an application is treated just like any other function call. To handle the places where such divergence appears between a lifted and an unlifted CF, we exploit the following lemma.

```
Lemma Lifted_inlined_fun : ∀(F1:Formula) (f:val) (vs:list val) (r:val),
  (triple (trm_apps f vs) \[] (fun x ⇒ \[x = r])) →
  Lifted F1 (cf_val r) →
  Lifted F1 (cf_app f vs).
```

The first premise of that lemma needs to be justified by exploiting the specification lemma for the pure OCaml operation named `f`.

Pattern matching. Regarding pattern matching, it is harder to capture the relationship between lifted and unlifted formulae in a systematic manner. This difficulty is due to at least two factors. First, patterns are associated with a variable number of independently quantified variables, each with its own type. Second, the justification that the equalities expressed at the level of Coq values matches the equalities expressed at the level of the deep embedding involves reasoning by case analysis (*inversion*). Furthermore, in the case of nested patterns, the case analysis steps need to be nested. I have developed a set of tactics to partially automate this process; the next step is to fully automate it.

Summary. The lifting technique enables the user to carry out proofs that manipulate Coq values. With this technique, the user never sees deeply embedded syntax. The user can work with typed values, and with representation predicates over typed values. Moreover, numerous proof steps involving pure functions can be eliminated by describing the results of pure operations using the corresponding Coq operations.

The translation of OCaml to Coq types, of OCaml to Coq values, and the production of lifted characteristic formulae is implemented by an external tool. The correctness of these lifted formulae can be justified by showing that they are formally related to the unlifted formulae computed inside Coq by a formally verified function. Put together, the generic proof and the per-program proofs justify the soundness of our lifted characteristic formulae in a foundational manner.

Chapter 7

Resource Analysis

In Section 7.1, I start by motivating formal analysis on program resource usage, and review prior work on resource analysis. I explain, in particular, that a uniform approach can be used to handle various kinds of resources. In Section 7.2, I present the *time credits* heap predicate and its properties. In Section 7.3, I explain how to realize time credits as an extension of Separation Logic with *ghost state*. In Section 7.4, I describe the statement of the soundness theorem, which asserts that time credits indeed capture amortized time bounds. In Section 7.5, I describe the—somewhat unexpected—interest of *negative* time credits. Finally, in Section 7.6, I present the case study that consists of the formalization of the amortized analysis of the classic Union-Find data structure, involving the inverse Ackermann function.

I started working on *time credits* with François Pottier in 2013, just after returning to Inria after my postdoc. We developed the metatheory and applied our approach to mechanize the amortized analysis of the Union-Find data structure. This work appeared at ITP'15 [Charguéraud and Pottier, 2015]. In 2017, we slightly simplified the proof [Charguéraud and Pottier, 2019] by following the proof technique by Alstrup et al.'s [2014] instead of the older proof technique presented in the *Introduction to Algorithms* textbook [Tarjan, 1999; Cormen et al., 2009]. We published that work in the Journal of Automated Reasoning (JAR).

Subsequently, together with François, we advised the PhD of Armaël Guéneau who, while working on the asymptotic notation, introduced *possibly negative time credits*. This key technical ingredient appears as a side contribution in our ITP'19 publication on the incremental cycle detection algorithm [Guéneau et al., 2019b], which is discussed in the next chapter. In 2021, I simplified the formalization of Separation Logic with possibly negative time credits. In this chapter, I present this simplified formalization, which is implemented in CFML.

The contents of the Section 7.1 is based on text that appears both in our ITP'15 paper on time credits [Charguéraud and Pottier, 2015] and in our POPL'23 paper [Moine et al., 2023] on space credits. The contents of Section 7.5 is adapted from the ITP'19 paper. The contents of Section 7.6 is a simplified excerpt from the JAR'19 paper.

7.1 Motivation and Related Work on Resource Analysis

A program whose functional correctness has been verified might nevertheless still contain complexity bugs: that is, its performance, in some scenarios, could be much poorer than expected. To illustrate the issue, consider the binary search implementation in Figure 7.1. A number of modern software verification tools could be used to prove that such a program satisfies the specification of a binary search, and that it terminates on any valid input. This code might even pass a lightweight testing process, as many queries will be answered very quickly, even if the array is very large. Yet, a more thorough testing process would reveal a serious issue: a search for a value that is stored near the end of the array takes not logarithmic time, but linear time!

The flaw in the implementation is that the occurrence of the variable `i` on the last line of Figure 7.1 should have been an occurrence of the variable `k`. It would be embarrassing if such faulty code was deployed, as it would aggravate benevolent users and possibly allow malicious users to mount denial-of-service attacks. *Formal asymptotic complexity analysis* could ensure the absence of such complexity bugs.

As illustrated by the flawed binary search example, complexity bugs can affect execution time. They could also concern space, including heap space, stack space, and disk space; or other resources, such as the network, energy, and so on. As we argue next, much of the work on formal resource analysis is independent of which resource is considered. In particular, we expect that the techniques presented in this chapter for establishing time bounds could be adapted to verify asymptotic bounds on the use of many other kinds of resources.

Reasoning about the use of a resource requires a "model" that tells when this resource is consumed or produced, and how much of it is consumed or produced. Such a model is usually an abstraction of some physical reality. For example, to obtain an asymptotic time bound, one can posit that every elementary instruction consumes one unit of time.[1] To obtain an asymptotic bound on stack space, one can posit that every (non-tail) function call consumes one unit of stack space, which is recovered when the function returns.[2] To derive a bound on heap space, when the language has an explicit deallocation instruction, one can posit that an allocation instruction consumes the requested amount of space and that a deallocation instruction recovers the space occupied by the heap block that is about to be deallocated. In all three cases, it is evident in the program where the resource of interest is consumed or produced.

In such settings, reasoning about resource consumption can be reduced to reasoning about safety. Indeed, one can construct a variant of the program that is instrumented with a *resource meter*, that is, a global variable whose value indicates what amount of the resource of interest remains available. In this instrumented program, one places assertions that cause a runtime failure if the value of the meter becomes negative. If one can verify that the instrumented program is safe, then one has effectively established a bound on the resource consumption of the original program.

The principle of a resource meter has been exploited in many papers, using various frameworks for establishing safety. For instance, Crary and Weirich [2000] exploit a dependent type system; Aspinall et al. [2007] exploit a VDM-style (Vienna Development Method) program logic; Carbonneaux et al. [2015] exploit a Hoare logic; He et al. [2009] exploit Separation Logic. The manner in which one reasons about the value of the meter depends on the chosen framework. In the most straightforward approach, the value of the meter is explicitly described in the pre- and postcondition of every function. This is the case, for instance, in He et al.'s work [2009], where

[1] Predicting physical execution time requires access to a compiled version of the program and an accurate model of the processor: see, e.g., Amadio et al. [2014].

[2] Computing a concrete bound, expressed in memory words, requires knowing the size of each stack frame [Amadio et al., 2014; Carbonneaux et al., 2014; Gómez-Londoño et al., 2020].

```
(* Requires t to be a sorted array of integers.
  Returns k such that i <= k < j and t.(k) = v
  or -1 if there is no such k. *)
let rec bsearch t v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = t.(k) then k
    else if v < t.(k) then bsearch t v i k
    else bsearch t v (i+1) j
```

Figure 7.1: A flawed binary search. This code is provably correct and terminating, yet exhibits linear (instead of logarithmic) time complexity for some input parameters.

two distinct meters are used to measure stack space and heap space.

In a more elaborate approach, which is made possible by Separation Logic, the meter is not regarded as an integer value, but as a bag of *credits* that can be individually *owned*. This removes the need to refer to the absolute value of the meter: instead, the specification of a function may indicate that this function requires a certain number of credits and produces a certain number of credits. Furthermore, because credits can be set aside for later use, this allows *amortized analysis*. Time credits, under various forms, have been used in several type systems [Danielsson, 2008; Hofmann and Jost, 2006; Hoffmann and Hofmann, 2010; Pilkiewicz and Pottier, 2011; McCarthy et al., 2016]. Atkey [2010; 2011] has argued in favor of viewing credits as predicates in Separation Logic. However, Atkey's work did not go as far as using time credits in an expressive framework.

The first practical, general-purpose program verification framework based on time credits in Separation Logic is the one that François Pottier and I developed in the years 2013-2014, as an extension of CFML. We illustrated the interest of such a framework by presenting the first mechanized proof of the amortized analysis for the Union-Find data structure [Charguéraud and Pottier, 2015; Charguéraud and Pottier, 2019]. This case study is presented in Section 7.6. Separation Logic with time credits has been subsequently exploited in other lines of work, including work in Iris on *time receipts* for establishing lower bounds on execution time [Mével et al., 2019], work in Iris on *transfinite time credits* for proving the termination of programs whose execution time cannot be bound upfront [Spies et al., 2021], and mechanized verification of data structures in Isabelle/HOL [Haslbeck and Nipkow, 2018; Haslbeck and Lammich, 2021].

Reasoning about heap space in the presence of explicit allocation and deallocation instructions can be achieved by extending traditional Separation Logic with *space credits*. To the best of our knowledge, such a variant of Separation Logic does not exist in the literature. However, Hofmann's work on the typed programming language LFPL [2000] can be viewed as a precursor of this idea: LFPL has explicit allocation and deallocation, which consume and produce values of a linear type, written $\diamond$, whose inhabitants behave very much like space credits.

Reasoning about heap space in the presence of a garbage collector is much more involved. In such a setting, there is no explicit deallocation instruction. Thus, it is not evident at which program points space can be reclaimed. Additional ingredients are needed for establishing space bounds for garbage-collected ML programs. We describe them in Chapter 9.

7.2 Principle of Time Credits

Time credits are described by a heap predicate written $\$n$. Intuitively, this predicate describes the *right to perform n steps of computation*, for a certain notion of *step* to be defined. To obtain asymptotic bounds, it is sufficent to count one step for each function call and for each loop iteration. Be aware that we do not aim for a worst-case execution time (WCET) analysis: we do not aim to provide bounds on actual physical execution time, but only asymptotic bounds.

To track in the program logic the number of computation steps, it suffices to instrument the source code by inserting, in a systematic manner, a call to a function called `pay` at the head of every function body and of every loop. (This is analogous to Danielsson's "tick" [2008].) The specification of the function `pay` requires in its precondition one time credit. Apart from that, a call to the function `pay` behaves like a no-op. Technically, `pay` satisfies the triple: $\{\$1\}\ \texttt{pay()}\ \{\lambda\_.\ []\}$.

When reasoning about a call to `pay`, the user has no choice but to (directly or indirectly) exploit the specification of `pay` and give away one time credit. Thus, because each call to `pay` consumes one time credit, the number of time credits available at the start of the program bounds the maximal number of steps that can be performed. In other words, the number of time credits mentioned in the precondition of the whole program gives an asymptotic upper bound on the execution time.

In practice, instead of exploiting directly the specification of `pay`, proofs can be streamlined using one the following reasoning rules, which apply to a call to `pay` that precedes a term t, which corresponds to a function or loop body.

SEQ-PAY
$$\frac{\{H\}\ t\ \{Q\}}{\{H \star \$1\}\ (\mathsf{pay}() ; t)\ \{Q\}}$$

SEQ-PAY-TRANS
$$\frac{H \vdash \$1 \star H' \qquad \{H'\}\ t\ \{Q\}}{\{H\}\ (\mathsf{pay}() ; t)\ \{Q\}}$$

WP-SEQ-PAY
$$\frac{}{\$1 \star (\mathsf{wp}\, t\, Q) \ \vdash\ \mathsf{wp}\, (\mathsf{pay}() ; t)\, Q}$$

When using characteristic formulae as defined in Section 5.2 and Section 5.6, the relevant definitions to handle the `pay` function are as follows.

$$\begin{array}{lcl} \mathsf{cf}_E\, (\mathsf{pay}() ; t) & \equiv & \mathsf{cf\_pay}\, (\mathsf{cf}_E\, t) \\ \mathsf{cf\_pay}\, \mathcal{F} & \equiv & \mathsf{framed}\, (\lambda Q.\ \$1 \star \mathcal{F}\, Q) \end{array} \qquad \frac{H \ \vdash\ \$1 \star H' \qquad H' \ \vdash\ \mathcal{F}\, Q}{H \ \vdash\ \mathsf{cf\_pay}\, \mathcal{F}\, Q}\ \text{CF-PAY}$$

Time credits are provided as part of the precondition of a term. These credits may be used to justify computation steps performed by that term and, in particular, by the functions that this term calls. A function might consume fewer time credits than it receives in its precondition. Time credits may indeed be *saved for future use*, by being stored as part of the representation predicate of a data structure. Such a representation predicate, storing saved credits, typically appears in the postcondition of the function. Credits saved in a representation predicate may be extracted when reasoning about another function call whose precondition includes this representation predicate. This pattern of saving credits in heap predicates, to accumulate them on certain operations and spend them subsequent operations, allows for *amortized analysis*.

In the original presentations of time credits [Charguéraud and Pottier, 2015; Charguéraud and Pottier, 2019], a number n that appears in the time credits predicate $\$n$ was restricted to be a natural number: $n \in \mathbb{N}$. Subsequent work [Guéneau et al., 2019b; Guéneau, 2019] argues for the benefits of allowing *possibly negative time credits*: $n \in \mathbb{Z}$. We will come back to these benefits in Section 7.5. More recently, I encountered a tree based data structure whose analysis, based on geometric series, requires the introduction of rational numbers: $n \in \mathbb{Q}$. It is not clear to me whether any data structure could require the manipulation of non-rational pieces of time credits, i.e., require $n \in \mathbb{R}$. Thus, for now, I will stick to using rational numbers in the formalization.

The formalization of time credits consists of three components. First, credits are defined as heap predicate over a piece of *ghost state* (Section 7.3). Second, lemmas for rearranging time credits are established (see below). Third, the specification of the pay function is stated. This function may be related to a cost-instrumented semantics of the programming language (Section 7.4). We begin with the properties of possibly negative time credits. They are stated below and explained next.

Lemma 7.2.1 (Properties of time credits) *The properties below hold for any $n, m \in \mathbb{Q}$. They may be proved correct with respect to the realization of time credits described further on.*

1.	$\$0 = [\,]$	*zero credit is equivalent to nothing at all*
2.	$\$(m+n) = \$m \star \$n$	*credits are additive*
3.	$n \geqslant 0 \Rightarrow \mathit{affine}\,(\$n)$	*only nonnegative credits can be discarded*

The first property asserts that zero credits are equivalent to the empty assertion. The second property explains how time credits may be split and joined. The equality may be equivalently reformulated as follows.

$$\$\,n \;=\; \$\,m \star \$\,(n-m)$$

This equality is typically exploited when one has n credits at hand, and need to justify a call to a function that consumes m credits. The equality introduces the $n - m$ credits that remains.

In most cases, when exploiting the above equality, n is no less than m, meaning that one has more credits at hand enough credits to cover the coming expense. Yet, this inequality does not have to hold. If it does not, exploiting the equality introduces a *debt* that corresponds to the negative amount $n - m$. In particular, the split rule can be instantiated to create time credits out of thin air. The statement below allows to forge n credits, by producing at the same time a debt expressed as $-n$ credits.

$$[\,] \;=\; \$\,n \star \$\,(-n)$$

Negative credits do not compromise soundness, assuming that they cannot be discarded. As put by Tarjan [1985], "*we can allow borrowing of credits, as long as any debt incurred is eventually paid off*".

The third property above captures the fact that a nonnegative number of time credits may be freely discarded—it corresponds to an affine predicate in the sense of Section 3.1. On the contrary, a negative number of time credits cannot be discarded—such an assertion must be treated linearly. Use cases for negative time credits are discussed further on (Section 7.5).

7.3 Realizing Time Credits as Ghost State

We now explain how *time credits* can be realized as heap predicates using *ghost state*. The Iris framework [Jung et al., 2018b] provides generic tooling for constructing pieces of ghost state in a modular way, and for integrating that ghost state into a Separation Logic. Yet, Iris provides an *affine* logic, which would be unsound in the presence of negative time credits. As explained earlier in Section 3.1.1, the extensions of Iris that have been proposed for handling linearity [Tassarotti et al., 2017a; Bizjak et al., 2019] do not provide any obvious means of representing time credits, which are either linear or affine depending on whether they are negative or not.

In this section, we give a two-layer presentation of ghost state, somewhat in the fashion of Iris yet without the modularity aspects to keep things simple. On the one hand, we give an axiomatization of the properties that a piece of ghost state must satisfy, accompanied by generic definitions

for Separation Logic heap predicates that must hold for any piece of ghost state. On the other hand, we give a realization of that axiomatization for the specific case of time credits. Time credits provide a minimalistic yet interesting example of ghost state, thus our construction may also be useful for pedagogical purpose. To enforce that only nonnegative credits can be discarded, we exploit our generic treatment of affine predicates introduced in Section 3.1.

The following table axiomatizes the concepts involved for representing heaps that might include a piece of ghost state. For example, we will later interpret heaps, which were originally defined as predicate over states (Section 2.2.2), as predicates over pairs made of a state and of a rational number. The type heap should be accompanied by several operators. The constant hempty denotes the empty heap. The *compatibility* operator, written hcompat h_1 h_2, generalizes the disjointness relation. Two heaps are compatible if one can build a non-conflicting union of their contents. The notion of compatibility is standard in Separation Logic: it appears in *partial commutative monoids (PCMs)* and is used, e.g., in the traditional models of fractional permissions. The *union* operator, written hunion h_1 h_2, applies to any two compatible heaps. The *heap-affine* predicate, written haffine h, characterizes affine predicate as in Section 3.1.2.

Definition 7.3.1 (Axiomatization of heaps that possibly include ghost state)

heap	:	*Type*	type of heaps
hempty	:	*heap*	empty heap
hcompat	:	*heap* → *heap* → *Prop*	compatibility relation
hunion	:	*heap* → *heap* → *heap*	union for compatible heaps
haffine	:	*heap* → *Prop*	characterization of affine heaps

The above entities must satisfy the properties that appear in the next table. The first part of the table asserts that compatibility is symmetric and that it is well-behaved with respect to empty heaps and unions of heaps. The second part of the table asserts that the union operator (on its domain, i.e., on compatible heaps) and the empty heap together form a commutative monoid. The third part of the table corresponds to the requirements for the heap-affine predicate (Definition 3.1.1).

Definition 7.3.2 (Axiomatization of operations on heaps featuring ghost state)

hcompat h *hempty*	compatibility with empty
hcompat h_1 h_2 ⇔ *hcompat* h_2 h_1	symmetry of compatibility
hcompat h_1 h_2 ⇒ *hcompat* (*hunion* h_1 h_2) h_3 ⇔ (*hcompat* h_1 h_3 ∧ *hcompat* h_2 h_3)	distrib. compatibility/union
hunion h *hempty* $=$ h	union with empty heaps
hunion h_1 h_2 $=$ *hunion* h_2 h_1	commutativity of union
hunion h_1 (*hunion* h_2 h_3) $=$ *hunion* (*hunion* h_1 h_2) h_3	associativity of union
haffine hempty	affinity of the empty heap
hcompat h_1 h_2 ⇒ *haffine* h_1 ∧ *haffine* h_2 ⇒ *haffine* (*hunion* h_1 h_2)	distrib. affinity/union

Given a definition of heaps satisfying the above axiomatization, one can define the core Separation Logic operators in a systematic way. In CFML, this construction takes the form of a Coq functor. The corresponding definitions, shown below, refine Definition 2.2.3.

Definition 7.3.3 (Core heap predicates, revisited for abstract heap data type)

$[\,]$	$\lambda h.\ h = \mathit{hempty}$
$[P]$	$\lambda h.\ h = \mathit{hempty} \wedge P$
$H_1 \star H_2$	$\lambda h.\ \exists h_1 h_2.\ \mathit{hcompat}\, h_1\, h_2 \wedge h = \mathit{hunion}\, h_1\, h_2 \wedge H_1\, h_1 \wedge H_2\, h_2$
$\exists x.\, H$	$\lambda h.\ \exists x.\ H\, h$
$\forall x.\, H$	$\lambda h.\ \forall x.\ H\, h$

We are now ready to instantiate our functor to derive a *Separation Logic with time credits*. To that end, we instantiate the type heap to be predicates over pairs made of a state and of a rational number of credits. The empty heap is made of an empty state and zero credits. Compatibility simply asserts that the two underlying states have disjoint domains. The union operator constructs the (disjoint) union of the two underlying states, and sums up the two amounts of credits at hand. The heap-affine predicate asserts that affine heaps are exactly those that carry a nonnegative number of credits—other heaps must be treated linearly.

Definition 7.3.4 (Realization of heaps for time credits)

heap	$\equiv$	$\mathit{state} \times \mathbb{Q}$	type of heaps
hempty	$\equiv$	$(\varnothing, 0)$	empty heap
$\mathit{hcompat}\,(s_1, n_1)\,(s_2, n_2)$	$\equiv$	$s_1 \perp s_2$	compatibility relation
$\mathit{hunion}\,(s_1, n_1)\,(s_2, n_2)$	$\equiv$	$(s_1 \uplus s_2, n_1 + n_2)$	union for compatible heaps
$\mathit{haffine}\,(s, n)$	$\equiv$	$n \geqslant 0$	affinity for nonnegative credits

With respect to our model of heaps as pairs of a state and a number of credits, we can define the heap predicates for representing time credits. The predicate $\$\,m$ characterizes a pair made of an empty state and of exactly m credits. We also need to update the definition of the singleton heap predicate, which characterizes a singleton state, to assert that it carries zero credits. The corresponding definitions appear next.

Definition 7.3.5 (Definition of time credits and singleton heap predicates)

$\$\,m$	$\equiv$	$\lambda(s, n).\ n = m \wedge s = \varnothing$	time credits
$p \mapsto v$	$\equiv$	$\lambda(s, n).\ n = 0 \wedge s = (p \to v) \wedge p \neq \mathit{null}$	singleton heap predicate

We have found that using instead the definitions shown below lead to proofs that can be carried out at a slightly higher abstraction level. Those alternative definitions are used in CFML, yet are not essential for the rest of the discussion.

Definition 7.3.6 (Alternative definition of time credits and singleton heap predicates)

$\mathit{hcredits}\, n$	$\equiv$	$(\varnothing, n)\ :\ \mathit{heap}$	empty state with credits
$\$\,n$	$\equiv$	$\lambda h.\ h = \mathit{hcredits}\, n$	time credits
$\mathit{hstatepred}\, K$	$\equiv$	$\lambda(s, n).\ n = 0 \wedge K\, s$	heap predicate from state pred.
$p \mapsto v$	$\equiv$	$\mathit{hstatepred}\,(\lambda s.\, s = (p \to v) \wedge p \neq \mathit{null})$	singleton heap predicate

7.4 Soundness of Time Credits with respect to the Semantics

So far, we have viewed the function `pay` as an abstract function satisfying $\{\$1\}$ `pay()` $\{\lambda\_.\ [\,]\}$. This function can be viewed as an external system call. The fact that the input program is correctly instrumented with the appropriate calls to `pay` is either part of the trusted code base, or should be

carried out by a separate, formally verified tool. In the journal paper on time credits [Charguéraud and Pottier, 2019], we aimed for a more foundational approach. To that end, we established a formal link between the spending of time credits and the execution of steps in the semantics. In what follows, we present a cost-instrumented semantics, cost-aware reasoning rules, and a definition of triples that relates the two.

The cost-instrumented semantics judgment is written $t/s \Downarrow^n v/s'$, where n denotes the number of computation steps involved in the evaluation. Note that this judgment corresponds to a simplified form of trace semantics: the only events that we care about are function calls, and the only trace property that we care about is the number of such events. We next show two key evaluation rules that are part of the definition of that judgment. The rule BIGN-APP adds one unit to the number of steps. The rule BIGN-LET sums up the number of steps performed by each of its two subterms.

BIGN-APP
$$\frac{v_1 = \hat{\mu} f.\lambda x.t \qquad ([v_2/x]\,[v_1/f]\,t)/s \Downarrow^n v'/s'}{(v_1\, v_2)/s \Downarrow^{(n+1)} v'/s'}$$

BIGN-LET
$$\frac{t_1/s \Downarrow^{n_1} v_1/s' \qquad ([v_1/x]\,t_2)/s' \Downarrow^{n_2} v/s''}{(\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2)/s \Downarrow^{(n_1+n_2)} v/s''}$$

Our cost-aware program logic does not assume programs to be instrumented with calls to pay. Instead, we replace the reasoning rule for applications with a variant that enforces the spending of one time credit. Concretely, we keep all the standard reasoning rules from Section 2.4.3, except that we replace APP with APP-PAY. The only change is the appearance of $\$1$ in the precondition. This credit is effectively consumed when applying the rule for processing a function call.

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

APP-PAY
$$\frac{v_1 = \hat{\mu} f.\lambda x.t \qquad \{H\}\ ([v_2/x]\,[v_1/f]\,t)\ \{Q\}}{\{\$\,1 \star H\}\ (v_1\, v_2)\ \{Q\}}$$

There remains to give a definition of triples relating the reasoning rules to the semantics. First, let us recall the definition of triples for a plain Separation Logic. Definitions 2.4.2 and 2.4.3 provide two equivalent definitions for triples (with respect to a semantics deterministic up to allocations). Let us give a third, equivalent formulation, that is well-suited for an extension accounting for time credits. Because we need an affine logic to be able to discard spare credits, let us also include in the definition triple an affine top predicate ($\mathbb{T}$), as done in Definition 3.1.4. Our baseline is thus the following definition of affine triples.

Definition 7.4.1 (Alternative presentation of partially-affine Separation Logic triples)

$$\{H\}\, t\, \{Q\} \;\equiv\; \forall s H'.\ (H \star H')\, s \;\Rightarrow\; \exists v s'.\ \begin{cases} t/s \Downarrow v/s' \\ (Q\, v \star H' \star \mathbb{T})\, s' \end{cases}$$

Let us now refine the above definition to account for time credits. Recall that heap predicates are now predicates over pairs of a state and a number of credits. In the definition shown below, n denotes the number of steps of computation performed and c denotes the number of credits covered by the precondition. The variable c' denotes the number of credits covered by the postcondition $(Q\, v)$ *and* by the affine top predicate ($\mathbb{T}$). Thus, c' may be greater than the number of credits actually mentioned in the postcondition. Our definition enforces the constraint $c = n + c'$, which means that the credits available initially in the precondition are distributed among (1) the credits consumed by the computation steps, plus (2) the credits that remain in the postcondition, plus (3) a nonnegative number of spare credits that have been discarded.

Definition 7.4.2 (Separation Logic triples with support for time credits)

$$\text{SEP+CREDITS}\ \{H\}\,t\,\{Q\} \quad\equiv\quad \forall s H' c.\ (H \star H')\,(s,c) \quad\Rightarrow\quad \exists v s' n c'.\ \begin{cases} t/s \Downarrow^n v/s' \\ (Q\ v \star H' \star \mathbb{T})\ (s',c') \\ c = n + c' \end{cases}$$

In the case of a full program execution described by a triple of the form $\{\$m\}\ t\ \{\lambda\_.\ [\,]\}$, our definition asserts that there exist c, v, s', n and c' such that $t/\varnothing \Downarrow^n v/s'$ and $m = c$ and $c = n + c'$ and $c' \geqslant 0$. These arithmetic constraints imply $n \leqslant m$. In other words, a program proved correct with m time credits terminates after at most m computation steps. In summary, our formalization gives a foundational interpretation of time credits in terms of a cost-aware semantics.

7.5 Possibly Negative Time Credits

As explained earlier in Section 7.2, whereas the original presentation of time credits considered credits in $\mathbb{N}$ [Charguéraud and Pottier, 2015], subsequent work [Guéneau et al., 2019b; Guéneau, 2019] introduced *possibly negative time credits* in $\mathbb{Z}$—or in $\mathbb{Q}$, if we were to consider the generalization necessary for fine-grained amortized analyses. We next describe the two most important motivations for possibly negative credits.

Simplifying proof obligations associated with the spending of credits. Consider a situation where n credits are at hand and if one wishes to step over an operation whose cost is m. As explained in Section 7.2, the relevant reasoning rule is: $\$n \;=\; \$(n-m) \star \$m$. Yet, when credits are expressed in $\mathbb{N}$, this rule only holds under the side condition $m \leqslant n$. This side condition gives rise to a proof obligation at every place where time credits need to be spent.

Even worse, the size of such proof obligations tend to grow bigger as we make progress through the code. To see why, assume that n credits are initially at hand, and suppose that we need to step over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$. Then, k distinct proof obligations arise: $n - m_1 \geqslant 0$ and $n - m_1 - m_2 \geqslant 0$ and and so on, until $n - m_1 - m_2 - \ldots - m_k \geqslant 0$. In fact, these proof obligations are redundant: the last one alone implies all the previous ones. Unfortunately, in an interactive proof assistant such as Coq, it is not easy to take advantage of this fact and present only the last proof obligation to the user.

When credits are in $\mathbb{Z}$, the rule $\$n \;=\; \$(n-m) \star \$m$ has no side condition. Thus, stepping over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$ gives rise to no proof obligation at all until reaching the end of the sequence. Then, at the end of the sequence, $n - m_1 - m_2 - \ldots - m_k$ credits remain, which the user typically wishes to discard (or rather, *must* discard, in order to match the targeted postcondition). Discarding those credits by means of rule DISCARD-PRE (Section 3.1.3) requires a proof of: affine $(\$n - m_1 - m_2 - \ldots - m_k)$. Recall from Section 7.2 that only nonnegative credits are affine, as captured by the property $n \geqslant 0 \Rightarrow \text{affine}\,(\$n)$. Thus, to discard the remaining credits, one must discharge the proof obligation: $n - m_1 - m_2 - \ldots - m_k \geqslant 0$. As expected, this inequality captures the fact that the amount of credits that are spent must be less than the amount of credits that are initially available. In summary, switching from $\mathbb{N}$ to $\mathbb{Z}$ greatly reduces the number of proof obligations that appear about credits.

The case of a sequence of k operations is actually a simple pattern. In the proof of an incremental cycle detection algorithm (Section 8.7), we have encountered a more complex situation. There, instead of looking at a straight-line sequence of operations, one is looking at a loop, whose body is a sequence of operations, and which itself is followed with another sequence of operations. In our proof carried out with credits in $\mathbb{Z}$, we only needed to discharge a single proof obligation at the

very end. Credits in $\mathbb{N}$ would have required not just a much larger number of proof obligations, but also a nontrivial strengthening of the loop invariant.

Specification of function whose cost depends on the output value. Consider a function called `index_of_first_nonzero` that takes as argument an array of integers that contains at least one non-zero value (as captured by the predicate contains-a-nonzero), and that returns the index of the first non-zero value found in the array. The implementation of this function traverses the array and stops at the first non-zero value, found at some index i. Thus, the cost of this function is $1 + i$, where 1 pays for the function call and i pays for reading the i first cells of the array.

When credits are expressed in $\mathbb{N}$, the cost of a function must be covered by time credits visible in the precondition. Yet, this cost depends on the result value i, which is only bound in the postcondition. To nevertheless state a correct specification, one needs to characterize, in the logic, what the output value i is *going to be* in terms of the contents of the array. Then, the cost of the function is expressed as $i + 1$, and the output of the function is specified to be equal to i. The corresponding specification appears next, where L denotes the elements in the array.

$$
\begin{array}{l}
\mathsf{contains\text{-}a\text{-}nonzero}\, L \;\wedge\; 0 \leqslant i < |L| \;\wedge\; L[i] \neq 0 \;\wedge\; \big(\forall j.\, 0 \leqslant j < i \Rightarrow L[j] = 0\big) \;\Rightarrow \\
\{\mathsf{Array}\, L\, p \;\star\; \$(i+1)\} \\
(\texttt{index\_of\_first\_nonzero}\, p) \\
\{\lambda i'.\; \mathsf{Array}\, L\, p \;\star\; [i' = i]\}
\end{array}
$$

In order to invoke the above specification, one must instantiate the specification with a suitable i. Exhibiting the value of i in the proof essentially requires implementing a Coq function that computes the first index of a nonzero element in a list L. In other words, in order to instantiate the specification of the function `index_of_first_nonzero`, one needs to implement a purely functional counterpart of it in the logic.

Alternatively, in a logic where Hilbert's epsilon operator is available (e.g., in Coq extended with strong classical logic), one can rewrite the above specification using a min operator in a more succinct way, as shown below. One can instantiate this specification slightly more easily. Yet, reasoning about the min operator still introduces a fair amount of clutter in proofs.

$$
\begin{array}{l}
\mathsf{contains\text{-}a\text{-}nonzero}\, L \;\Rightarrow \\
\mathsf{let}\, i = \mathsf{min}_{\mathbb{N}}(\lambda\, i.\; L[i] \neq 0)\, \mathsf{in} \\
\{\mathsf{Array}\, L\, p \;\star\; \$(i+1)\} \\
(\texttt{index\_of\_first\_nonzero}\, p) \\
\{\lambda i'.\; \mathsf{Array}\, L\, p \;\star\; [i' = i]\}
\end{array}
$$

In contrast, with time credits in $\mathbb{Z}$, we can state a significantly simpler specification: a triple of the form $\{\$n \;\star\; H\}\; t\; \{\lambda r.\;\; H'\}$ can be equivalently stated as: $\{H\}\; t\; \{\lambda r.\;\; \$(-n) \;\star\; H'\}$, that is, with the opposite number of credits in the postcondition. (Reciprocally, credits that appear in the postcondition may be instead subtracted in the precondition.) The specification of `index_of_first_nonzero` can now be expressed with a specification that can be smoothly instantiated in proofs. The properties of the output value i, and the cost of the function expressed as $i + 1$, both appear in the postcondition.

$$
\begin{array}{l}
\mathsf{contains\text{-}a\text{-}nonzero}\, L \;\Rightarrow \\
\{\mathsf{Array}\, L\, p\} \\
(\texttt{index\_of\_first\_nonzero}\, p) \\
\{\lambda i.\; \mathsf{Array}\, L\, p \;\star\; \$(-i-1) \;\star\; [0 \leqslant i < |L| \;\wedge\; L[i] \neq 0 \;\wedge\; \big(\forall j.\, 0 \leqslant j < i \Rightarrow L[j] = 0\big)]\}
\end{array}
$$

In summary, possibly negative time credits ease the statement of credits-based specifications for functions whose cost is output-sensitive. Armaël Guéneau's PhD thesis (§6.3.1) [2019] contains another similar example, based on a function that traverses consecutive segments of an array.

7.6 Formal Analysis of the Union-Find Data Structure

Union Find and its complexity analysis. The Union-Find data structure, also known as a disjoint set forest [Galler and Fischer, 1964; Cormen et al., 2009], maintains a collection of disjoint sets and keeps track in each set of a distinguished element, known as the representative of this set. It supports the following operations: `make` creates a new element, which forms a new singleton set; `find` maps an element to the representative of its set; `union` merges the sets associated with two elements.

Union-Find is among the simplest classic data structures, yet requires one of the most challenging complexity analyses. Tarjan [1975] and Tarjan and van Leeuwen [1984] prove, for an implementation of disjoint set forests exploiting path compression and linking-by-rank, that the worst-case time required by a sequence of m operations involving at most n elements is $O(m \cdot \alpha(n))$. There, the function α is the inverse of Ackermann's function. It grows so slowly that $\alpha(n)$ does not exceed 5 in practice. The analysis of Union-Find has been progressively simplified over the years [Kozen, 1992], ultimately resulting in a readable 2.5-page proof that appears in Tarjan's online course notes [1999]. It also appears in the textbook *Introduction to Algorithms* [Cormen et al., 2009], where it occupies about 8 small pages.

In that textbook presentation, the parameter n must be a priori fixed, and represents an upper bound on the number of elements that can ever appear in the data structure. In more recent work, Kaplan et al. [2002] and Alstrup et al. [2014] establish (among other results) a more precise and more pleasant bound: each operation has worst-case amortized complexity $O(\alpha(n))$, where n is the number of elements in the data structure at the time this operation is performed.[3] Our journal publication [Charguéraud and Pottier, 2019] formalizes Alstrup et al.'s proof [2014], which removes the need to fix the maximal number of elements in advance, and leads to specifications simpler and easier to use.

Mechanized Proofs of Union Find. The functional correctness of implementations Union Find had been established before [Conchon and Filliâtre, 2007; Chlipala et al., 2009a; Lammich and Meis, 2012], but without considering bounds on the execution time. Our work [Charguéraud and Pottier, 2015] has been the first to provide a mechanized proof of the mathematical analysis of Union-Find, and the first to formalize time bounds for a concrete, executable implementation of that data structure.

Before presenting the formal specifications, let us make two comments about the implementation. First, we represent the reverse forest using pointers. Our approach would also apply for verifying an array-based implementation. The vast majority of the formal development would be shared between the two implementations: only a small fraction of the proofs are specific to a particular implementation. Second, we assume that ranks are represented using arbitrary-precision integers. Doing so enables us to ignore the possibility of integer overflow. To handle machine integers, one could introduce additional assumptions of the form $n < 2^{2^{62}}$ to justify that ranks, which are at most $\log_2 n$, can be represented without overflow in a 64-bit OCaml program. Alternatively, one could attempt to avoid the introduction of such assumptions by following Clochard

[3] In fact, they prove even tighter results, stated in terms of the sizes of equivalence classes, as opposed to the size of the entire data structure. For our purposes, taking n to be the current number of elements in the data structure seems good enough, and allows exposing a specification that seems as simple as one might hope for.

et al.'s [2015] "proof-of-work" argument, and argue that creating $2^{2^{62}}$ elements is impossible in practice, by sheer lack of time.

Specification. I here present slighly simplified specifications compared with the ones that appear in the publication [Charguéraud and Pottier, 2015], ignoring the extra feature that provides the ability to associate a value to every representative.

The formal specification of the Union-Find interface appears in Figure 7.2. The specification begins with the declaration of a representation predicate, `UF`. This predicate is abstract: the client is not supposed to know how it is defined. The Separation Logic assertion `UF` $D\,R$ claims the existence (and unique ownership) of a Union-Find data structure, whose current state is summed up by the parameters D and R. The parameter D gives the *domain* of the data structure, that is, the set of all elements. The parameter R maps every element to its *representative*.

Since R maps an element to its representative, we expect it to be an idempotent function of the set D into itself. Furthermore, although this is in no way essential, we decide that R should be the identity outside D. These properties are captured by the theorem named `UF_properties`. Its statement is of the form `UF` $D\,R \vdash$ `UF` $D\,R \star [P]$, which means that, if one owns `UF` $D\,R$, then the properties described by P hold.

The next theorem, `UF_create`, asserts that out of nothing one can create an empty Union-Find data structure. Its statement is, again, an entailment. Creating an empty Union-Find data structure is indeed a "ghost" operation: the OCaml code does not explicitly offer such an operation.

Next comes the specification of the OCaml function `make`. The precondition is the conjunction of `UF` $D\,R$, which describes the pre-state, and of $\$\,3$, which indicates that `make` works in constant time. In the postcondition, x denotes the element returned by `make`. The postcondition describes the post-state via the assertion `UF` $D'\,R$, where D' is $D \cup \{x\}$, as the domain has grown. The postcondition also asserts that x is new, that is, distinct from all previous elements. The new element is its own representative: by our convention, R must be the identity outside D, so $R\,x$ must be x.

The next theorem provides a specification for `find`. The argument x must be a member of D. In addition to `UF` $D\,R$, the precondition requires $2\alpha(|D|) + 4$ credits. This reflects the amortized cost of `find`. The formal definition of `alpha`, the inverse of Ackermann's function, may be found in [Charguéraud and Pottier, 2019, §6.4]. The postcondition asserts that `find` returns an element y such that $R\,x = y$. In other words, `find` returns the representative of x. Furthermore, the postcondition asserts that `UF` $D\,R$ still holds. Even though path compression may update internal pointers, the parameters D and R, which represent the client's view of the state, are unchanged.

The precondition of `union` requires `UF` $D\,R$ together with $4\alpha(|D|) + 12$ time credits. The postcondition indicates that `union` returns an element z, which is either x or y, and has the effect of updating the data structure to `UF` $D\,R'$, where R' maps to z every element that was equivalent to x or y.[4] This means that the equivalence classes of x and y are merged and that, out of the two pieces of user data associated with these classes, an arbitrary one is retained.

Amortized analysis. The representation predicate `UF` includes, as part of its definition, a certain number of time credits, written $\$\,\Phi$. Their number corresponds to the current potential of the data structure, in Tarjan's terminology [1985]. These credits are "saved inside the data structure", so to speak. They can be used to pay in part for an operation and therefore decrease its apparent cost. Conversely, if one chooses to advertise an apparent cost that is greater than the operation's

[4] The construct `If P then e1 else e2`, where `P` is in Prop, is a nonconstructive conditional. It is defined using the law of excluded middle and Hilbert's ϵ operator.

```
Definition UF ( D : set elem) (R : elem → elem) : heap → Prop := ..
(* abstract for the client: implementation not revealed *)

Theorem UF_properties : ∀D R, UF D R ⊢ UF D R ⋆ \[
    (∀ x, R (R x) = R x) ∧
    (∀ x, x ∈ D → R x ∈ D) ∧
    (∀ x, x ∉ D → R x = x) ].

Theorem UF_create : \[] ⊢ UF ∅ (fun x ⇒ x).

Theorem make_spec : ∀D R V,
  TRIPLE (make ())
    PRE (UF D R ⋆ $3)
    POST (fun x ⇒ UF (D ∪ {x}) R ⋆ \[ x ∉ D ]).

Theorem find_spec : ∀D R x, x ∈ D →
  TRIPLE (find x)
    PRE (UF D R ⋆ $(2 * alpha (card D) + 4))
    POST (fun y ⇒ UF D R ⋆ \[ R x = y ]).

Theorem union_spec : ∀D R x y, x ∈ D → y ∈ D →
  TRIPLE (union x y)
    PRE (UF D R ⋆ $(4 * alpha (card D) + 12))
    POST (fun z ⇒
     UF D (fun w ⇒ If (R w = R x ∨R w = R y) then z else R w) ⋆ \[ z = R x ∨z = R y ])
          .
```

Figure 7.2: Specification of Union-Find

actual cost, then the extra credits provided by the caller can be "saved", that is, used to justify an increase of Φ.

When we formally verify the code, we are required to prove that, for each operation, the initial potential Φ, plus the number of credits brought by the caller, covers the new potential Φ', plus the number of credits consumed during the operation.

$$\Phi + \text{advertised cost of operation} \geqslant \Phi' + \text{actual cost of operation}$$

For instance, the specification of `find` requires $2\alpha(|D|) + 4$ credits, where $|D|$ denotes the current number of elements in the data structure. This is its apparent cost. Its actual cost, i.e., the number of (abstract) computation steps that it actually performs may be lower or higher. The reasoning associated with the verification of `find` is as follows. The precondition of `find` includes $2\alpha(|D|) + 4$ credits from the apparent cost plus the Φ credits from the potential. To establish the postcondition of `find`, one must show that, after spending as many time credits as computation steps executed by `find`, there remains Φ' credits, where Φ' denotes the potential of the updated internal state of the data structure—the internal state may have changed as a result of a path compression operation. In summary, the bound that explicitly appears in the precondition of `find`, namely $2\alpha(|D|) + 4$, is a worst-case *amortized* time complexity bound.

This concludes the presentation of this case study. In the next chapter, I explain how to state the time bound in the form $O(\alpha(|D|))$, that is, using the big-O notation to abstract away from implementation details.

Chapter 8

Big-O Notation for Time Bounds

I start by explaining the motivation for using the big-O notation in formal proofs (Section 8.1), and explain the challenges associated with the big-O notation, especially in the multivariate case (Section 8.2). I then review prior work on mechanizing the big-O notation (Section 8.3), and present our solution to formalizing the mathematical concept using filters and domination relations (Section 8.4). I next describe our approach to integrating dominating functions in formal specifications (Section 8.5). Finally, I present a number of small case studies representative of various types of big-O bounds (Section 8.6), as well as a major case study that consists in the formal verification of functional correctness and asymptotic time bounds for a state-of-the-art incremental cycle detection algorithm (Section 8.7).

The contents of this section corresponds to the PhD work of Armaël Guéneau, co-advised by François Pottier and myself. The technical text of this section is a mildly revised version of the text that appeared in our ESOP'18 paper [Guéneau et al., 2018]. The notion of *filters* had been already mechanized in other contexts; the Coq definitions that we use were contributed by François Pottier. The incremental cycle detection case study presented at the end of the section corresponds to our ITP'19 publication [Guéneau et al., 2019b], work for which we were joined by Jacques-Henri Jourdan. All the work presented here has been formalized in Coq as an extension of CFML.

8.1 Motivation for the Asymptotic Notation

The use of asymptotic complexity in the analysis of algorithms (on paper), initially advocated by Hopcroft and by Tarjan [Hopcroft, 1987; Tarjan, 1987], has been widely successful and is nowadays standard practice. It may be worth recalling that this idea has not always been taken for granted. Hopcroft [1987] gives the following historical account.

> *During the 1960s, research on algorithms had been very unsatisfying. A researcher would publish an algorithm in a journal along with execution times for a small set of sample problems, and then several years later, a second researcher would give an improved algorithm along with execution times for the same set of sample problems. The new algorithm would invariably be faster, since in the intervening years, both computer performance and programming languages had improved. [...]*

> *I set out to demonstrate that a theory of algorithm design based on worst-case asymptotic performance could be a valuable aid to the practitioner. The idea met with much resistance. People argued that faster computers would remove the need for asymptotic efficiency. Just the opposite is true, however, since as computers become faster, the size of the attempted problems becomes larger, thereby making asymptotic efficiency even more important.*
>
> *In early 1970, I took a year-long sabbatical at Stanford University, where I met and shared an office with Robert Tarjan, a second-year graduate student. The research recognized by the 1986 Turing Award took place during that period of collaboration.*

In our formalization of the Union-Find data structure [Charguéraud and Pottier, 2019], specifications involved *concrete* cost functions. For instance, the precondition of the function *find* indicates that calling *find* requires and consumes $\$(2\alpha(n) + 4)$, where n is the current number of elements in the data structure, and where α denotes an inverse of Ackermann's function. We would prefer the specification to give the *asymptotic* complexity bound $O(\alpha(n))$, which means that, for *some* function $f \in O(\alpha(n))$, calling *find* requires and consumes $\$f(n)$.

At a superficial level, the use of asymptotic bounds reduces clutter in specifications and proofs: $O(mn)$ is more compact and readable than $3mn + 2n \log n + 5n + 3m + 2$. We argue, however that the use of asymptotic bounds, such as $O(\alpha(n))$, is more than a syntactic convenience. We claim that it is necessary for complexity analysis to be applicable at scale. Indeed, it is crucial for stating modular specifications, which hide the details of a particular implementation. Exposing the fact that *find* costs $2\alpha(n) + 4$ is undesirable: if a tiny modification of the Union-Find module changes this cost to $2\alpha(n) + 5$, then all direct and indirect clients of the Union-Find module must be updated, which is intolerable.

We would like to stress an important practical limitation of asymptotic bounds: a loop that counts up from 0 to 2^{60} has complexity $O(1)$, even though it typically won't terminate in a lifetime. Although this is admittedly a potential problem, traditional program verification falls prey to analogous pitfalls: for instance, a program that attempts to allocate and initialize an array of size (say) 2^{48} can be proved correct, even though, on contemporary desktop hardware, it will typically fail by lack of memory. In spite of these limitations, we believe that there is value in formal asymptotic analysis.

8.2 Challenges with Big-O

When informally reasoning about the complexity of a function, or of a code block, it is customary to make assertions of the form "this code has asymptotic complexity $O(1)$", "that code has asymptotic complexity $O(n)$", and so on. Yet, these assertions are too informal: they do not have sufficiently precise meaning, and can be easily abused to produce flawed paper proofs.

A striking example appears in Figure 8.1, which shows how one might "prove" that a recursive function has complexity $O(1)$, whereas its actual cost is $O(n)$. The flawed proof exploits the (valid) relation $O(1) + O(1) = O(1)$, which means that a sequence of two constant-time code fragments is itself a constant-time code fragment. The flaw lies in the fact that the O notation hides an existential quantification, which is inadvertently swapped with the universal quantification over the parameter n. Indeed, the claim is that "there exists a constant c such that, for every n, `waste`(n) runs in at most c computation steps". However, the proposed proof by induction establishes a much weaker result, to wit: "for every n, there exists a constant c such that `waste`(n) runs in at most c steps". This result is certainly true, yet does not entail the claim.

Incorrect claim: The OCaml function `waste` has asymptotic complexity $O(1)$.

```
let rec waste n =
  if n > 0 then waste (n-1)
```

Flawed proof:
Let us prove by induction on n that `waste`(n) costs $O(1)$.

- **Case** $n \leqslant 0$: `waste`(n) terminates immediately. Therefore, its cost is $O(1)$.
- **Case** $n > 0$: A call to `waste`(n) involves constant-time processing, followed with a call to `waste`$(n-1)$. By the induction hypothesis, the cost of the recursive call is $O(1)$. We conclude that the cost of `waste`(n) is $O(1) + O(1)$, that is, $O(1)$.

Figure 8.1: A flawed proof that `waste`(n) costs $O(1)$, when its actual cost is $O(n)$.

Incorrect claim: The OCaml function `f` has asymptotic complexity $O(1)$.

```
let g (n, m) =
  for i = 1 to n do
    for j = 1 to m do () done
  done
let f n = g (n, 0)
```

Flawed proof:

- $\mathrm{g}(n, m)$ involves nm inner loop iterations, thus costs $O(nm)$.
- The cost of `f`(n) is the cost of $\mathrm{g}(n, 0)$, plus $O(1)$. As the cost of $\mathrm{g}(n, m)$ is $O(nm)$, we find, by substituting 0 for m, that the cost of $\mathrm{g}(n, 0)$ is $O(0)$. Thus, `f`(n) is $O(1)$.

Figure 8.2: A flawed proof that `f`(n) costs $O(1)$, when its actual cost is $O(n)$.

An example of a different nature appears in Figure 8.2. There, the auxiliary function g takes two integer arguments n and m and involves two nested loops, over the intervals $[1, n]$ and $[1, m]$. Its asymptotic complexity is $O(n+nm)$, which, *under the hypothesis that* m *is large enough*, can be simplified to $O(nm)$. The reasoning, thus far, is correct. The flaw lies in our attempt to substitute 0 for m in the bound $O(nm)$. Because this bound is valid only for sufficiently large m, it does not make sense to substitute a specific value for m. In other words, from the fact that "$\mathrm{g}(n, m)$ costs $O(nm)$ when n and m are sufficiently large", one *cannot* deduce anything about the cost of $\mathrm{g}(n, 0)$. To repair this proof, one must take a step back and prove that $\mathrm{g}(n, m)$ has asymptotic complexity $O(n + nm)$ *for sufficiently large* n *and for every* m. This fact *can* be instantiated with $m = 0$, allowing one to correctly conclude that $\mathrm{g}(n, 0)$ costs $O(n)$.

Howell [2008] presents a slightly more technical counterexample to illustrate how one can abuse the argument that "the asymptotic cost of a loop is the sum of the asymptotic costs of its iterations". In may be found in the ESOP'18 paper [Guéneau et al., 2018, §2].

All these examples show that the informal reasoning style of paper proofs, where the O notation is used in a loose manner, is unsound. One cannot hope, in a formal setting, to faithfully mimic this reasoning style. In our work, we do assign O specifications to functions, because we

believe that this style is elegant, modular and scalable. However, during the analysis of a function body, we generally have to abandon the O notation. Instead, we first synthesize a cost expression for the function body, then check that this expression is indeed dominated by the asymptotic bound that appears in the specification.

8.3 Prior Work on Formal Definitions for Big-O

The O notation and its siblings are documented in numerous textbooks, e.g. [Graham et al., 1994; Brassard and Bratley, 1996; Cormen et al., 2009]. Such textbooks provide a rigorous definition for the notation O in the case where a single variable is involved. Then, further on, they exploit the notation with multiple variables, as in, e.g., $O(nm)$. Yet, these textbooks omit to give a definition for how to interpret the multivariate case. In fact, they fail to even *mention* the fact that there might be complications with the design of a multivariate definition.

We have found only one textbook that draws attention to the subtleties of the multivariate case: that of Howell [2012]. He points out that one cannot take for granted that the properties of the O notation, which in the univariate case are well-known, remain valid in the multivariate case. Howell [2008] published a technical report, whose abstract reads as follows.

> *We show that it is impossible to define big-O notation for functions on more than one variable in a way that implies the properties commonly used in algorithm analysis. We also demonstrate that common definitions do not imply these properties even if the functions within the big-O notation are restricted to being strictly nondecreasing. We then propose an alternative definition that does imply these properties whenever the function within the big-O notation is strictly nondecreasing.*

Concretely, Howell states several properties which, at first sight, seem natural and desirable, then proceeds to show that they are inconsistent—no definition of the O notation can satisfy them all. He then proposes a candidate notion of domination between functions whose domain is $\mathbb{N}^k$. His notation, $f \in \hat{O}(g)$, is defined as the conjunction of $f \in O(g)$ and $\hat{f} \in O(\hat{g})$, where the function $\hat{f}$ is a "running maximum" of the function f, and is by construction monotonic. He shows that this notion satisfies all the desired properties, provided some of them are restricted by additional side conditions, such as monotonicity requirements.

In our work, we go slightly further than Howell, in that we consider functions whose domain is an arbitrary filtered type A, rather than necessarily $\mathbb{N}^k$. We give a standard definition of O and verify all of Howell's properties, again restricted with certain side conditions. We find that we do not need $\hat{O}$, which is fortunate, as it seems difficult to define $\hat{f}$ in the general case where f is a function of domain A. The monotonicity requirements that we impose are not exactly the same as Howell's, but we believe that the details of these administrative conditions do not matter much, as all the functions that we manipulate in practice are everywhere nonnegative and monotonic.

Avigad and Donnelly [2004] formalize the O notation in Isabelle/HOL. They consider functions of type $A \to B$, where A is arbitrary and B is an ordered ring. Their definition of "$f = O(g)$" requires $|f(x)| \leqslant c|g(x)|$ for every x, as opposed to "when x is large enough". Thus, they get away without equipping the type A with a filter. The price to pay is an overly restrictive notion of domination, except in the case where A is $\mathbb{N}$, where both $\forall x$ and $\mathbb{U}x$ yield the same notion of domination—this is Brassard and Bratley's "threshold rule" [1996]. Avigad and Donnelly suggest defining "$f = O(g)$ eventually" as an abbreviation for $\exists f', (f' = O(g) \wedge \mathbb{U}x.f(x) = f'(x))$. In our eyes, this is less elegant than parameterizing O with a filter in the first place.

Eberl [2017] formalizes the Akra–Bazzi method [Akra and Bazzi, 1998; Leighton, 1996], a generalization of the well-known Master Theorem [Cormen et al., 2009], in Isabelle/HOL. He creates

a library of Landau symbols specifically for this purpose. Although his paper does not mention filters, his library in fact relies on filters, whose definition appears in Isabelle's Complex library. Eberl's definition of the O symbol is identical to ours. That said, because he is concerned with functions of type $\mathbb{N} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$, he does not define product filters, and does not prove any lemmas about domination in the multivariate case. Eberl sets up a decision procedure for domination goals, like $x \in O(x^3)$, as well as a procedure that can simplify, say, $O(x^3 + x^2)$ to $O(x^3)$.

Boldo et al. [2013] use Coq to verify the correctness of a C program which implements a numerical scheme for the resolution of the one-dimensional acoustic wave equation. They define an ad hoc notion of "uniform O" for functions of type $\mathbb{R}^2 \to \mathbb{R}$, which we believe can in fact be viewed as an instance of our generic definition of domination, at an appropriate product filter. Subsequent work on the Coquelicot library for real analysis [Boldo et al., 2015] includes general definitions of filters, limits, little-o and asymptotic equivalence. A few definitions and lemmas in Coquelicot are identical to ours, but the focus in Coquelicot is on various filters on $\mathbb{R}$, whereas we are more interested in filters on $\mathbb{Z}^k$.

8.4 Formalization of Big-O

In many textbooks, the fact that f is bounded above by g asymptotically, up to constant factor, is written "$f = O(g)$" or "$f \in O(g)$". However, the former notation is quite inappropriate, as it is clear that "$f = O(g)$" cannot be literally understood as an equality. Indeed, if it truly were an equality, then, by symmetry and transitivity, $f_1 = O(g)$ and $f_2 = O(g)$ would imply $f_1 = f_2$. The latter notation makes much better sense: $O(g)$ is then understood as a set of functions. This approach has in fact been used in formalizations of the O notation [Avigad and Donnelly, 2004]. Yet, we prefer to think directly in terms of a *domination* preorder between functions. Thus, instead of "$f \in O(g)$", we write $f \preceq g$.

Although the O notation is often defined in the literature only in the special case of functions whose domain is $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$, we must define domination in the general case of functions whose domain is an arbitrary type A. By later instantiating A with a product type, such as $\mathbb{Z}^k$, we get a definition of domination that covers the multivariate case. Thus, let us fix a type A, and let f and g inhabit the function type $A \to \mathbb{Z}$.[1]

Fixing the type A, it turns out, is not quite enough. In addition, the type A must be equipped with a *filter* [Bourbaki, 1995]. To see why that is the case, let us work towards the definition of domination. As is standard, we wish to build a notion of "growing large enough" into the definition of domination. That is, instead of requiring a relation of the form $|f(x)| \leqslant c\,|g(x)|$ to be "everywhere true", we require it to be "ultimately true", that is, "true when x is large enough".[2] Thus, $f \preceq g$ should mean, roughly: "up to a constant factor, ultimately, $|f|$ is bounded above by $|g|$." That is, somewhat more formally: "for some c, for every sufficiently large x, $|f(x)| \leqslant c\,|g(x)|$"

In mathematical notation, we would like to write: $\exists c.\ \mathbb{U}x.\ |f(x)| \leqslant c\,|g(x)|$. For such a formula to make sense, we must define the meaning of the formula $\mathbb{U}x.P$, where x inhabits the type A. This is the reason why the type A must be equipped with a filter $\mathbb{U}$, which intuitively should be thought of as a quantifier, whose meaning is "ultimately". Let us briefly defer the definition of a filter and sum up what has been explained so far:

[1] At this time, we require the codomain of f and g to be $\mathbb{Z}$. Following Avigad and Donnelly [2004], we could allow it to be an arbitrary nondegenerate ordered ring. We have not yet needed this generalization.

[2] When A is $\mathbb{N}$, provided $g(x)$ is never zero, requiring the inequality to be "everywhere true" is in fact the same as requiring it to be "ultimately true". Outside this special case, however, requiring the inequality to hold everywhere is usually too strong.

Definition 8.4.1 (Domination) *Let A be a filtered type, that is, a type A equipped with a filter $\mathbb{U}_A$. The relation $\preceq_A$ on $A \to \mathbb{Z}$ is defined as follows:*

$$f \preceq_A g \quad \equiv \quad \exists c.\, \mathbb{U}_A\, x.\, |f(x)| \leqslant c\, |g(x)|$$

Whereas $\forall x.P$ means that P holds of *every* x, and $\exists x.P$ means that P holds of *some* x, the formula $\mathbb{U}x.P$ should be taken to mean that P holds of every *sufficiently large* x, that is, P *ultimately* holds. The formula $\mathbb{U}x.P$ is short for $\mathbb{U}\ (\lambda x.P)$. If x ranges over some type A, then $\mathbb{U}$ must have type $\mathcal{P}(\mathcal{P}(A))$, where $\mathcal{P}(A)$ is short for $A \to \mathrm{Prop}$. To stress this better, although Bourbaki [1995] states that a filter is "a set of subsets of A", it is crucial to note that $\mathcal{P}(\mathcal{P}(A))$ is the type of a quantifier in higher-order logic.

Definition 8.4.2 (Filter) *A* filter [Bourbaki, 1995] *on a type A is an object $\mathbb{U}$ of type $\mathcal{P}(\mathcal{P}(A))$ that enjoys the following four properties, where $\mathbb{U}x.P$ is short for $\mathbb{U}\ (\lambda x.P)$:*

(1)	$(P_1 \Rightarrow P_2) \Rightarrow \mathbb{U}x.P_1 \Rightarrow \mathbb{U}x.P_2$	*(covariance)*
(2a)	$\mathbb{U}x.P_1 \wedge \mathbb{U}x.P_2 \Rightarrow \mathbb{U}x.(P_1 \wedge P_2)$	*(stability under binary intersection)*
(2b)	$\mathbb{U}x.\mathit{True}$	*(stability under 0-ary intersection)*
(3)	$\mathbb{U}x.P \Rightarrow \exists x.P$	*(nonemptiness)*

Properties (1)–(3) are intended to ensure that the intuitive reading of $\mathbb{U}x.P$ as: "for sufficiently large x, P holds" makes sense. Property (1) states that if P_1 implies P_2 and if P_1 holds when x is large enough, then P_2, too, should hold when x is large enough. Properties (2a) and (2b), together, state that if each of $P_1, \ldots, P_k$ independently holds when x is large enough, then $P_1, \ldots, P_k$ should simultaneously hold when x is large enough. Properties (1) and (2b) together imply $\forall x.P \Rightarrow \mathbb{U}x.P$. Property (3) states that if P holds when x is large enough, then P should hold of some x. In classical logic, it would be equivalent to $\neg(\mathbb{U}x.\mathrm{False})$.

In the following, we let the metavariable A stand for a *filtered type*, that is, a pair of a carrier type and a filter on this type. By abuse of notation, we also write A for the carrier type. (In Coq, this is permitted by an implicit projection.) We write $\mathbb{U}_A$ for the filter.

When $\mathbb{U}$ is the *order filter* associated with the ordering $\leqslant$, the formula $\mathbb{U}x.Q(x)$ means that, when x becomes sufficiently large with respect to $\leqslant$, the property $Q(x)$ becomes true.

Definition 8.4.3 (Order filter) *Let $(T, \leqslant)$ be a nonempty ordered type, such that every two elements have an upper bound. Then $\lambda Q.\exists x_0.\forall x \geqslant x_0.\ Q(x)$ is a filter on T.*

The order filter associated with the ordered type $(\mathbb{Z}, \leqslant)$ is the most natural filter on the type $\mathbb{Z}$. Equipping the type $\mathbb{Z}$ with this filter yields a filtered type, which, by abuse of notation, we also write $\mathbb{Z}$. Thus, the formula $\mathbb{U}_{\mathbb{Z}}\, x.Q(x)$ means that $Q(x)$ becomes true "as x tends towards infinity".

By instantiating Definition 8.4.1 with the filtered type $\mathbb{Z}$, we recover the classic definition of domination between functions of $\mathbb{Z}$ to $\mathbb{Z}$:

$$f \preceq_{\mathbb{Z}} g \iff \exists c.\, \exists n_0.\, \forall n \geqslant n_0.\, |f(n)| \leqslant c\, |g(n)|$$

We now turn to the definition of a filter on a product type $A_1 \times A_2$, where A_1 and A_2 are filtered types. Such a filter plays a key role in defining domination between functions of several variables. The following *product filter* is the most natural construction, although there are others:

Definition 8.4.4 (Product filter) *Let A_1 and A_2 be filtered types. Then*

$$\lambda Q.\exists Q_1, Q_2.\ \begin{cases} & \mathbb{U}_{A_1}\, x_1.\, Q_1 \\ \wedge & \mathbb{U}_{A_2}\, x_2.\, Q_2 \\ \wedge & \forall x_1, x_2.\, Q_1(x_1) \wedge Q_2(x_2) \Rightarrow Q(x_1, x_2) \end{cases}$$

is a filter on the product type $A_1 \times A_2$.

To understand this definition, it is useful to consider the special case where A_1 and A_2 are both $\mathbb{Z}$. Then, for $i \in \{1, 2\}$, the formula $\mathbb{U}_{A_i}\, x_i.\ Q_i$ means that the predicate Q_i contains an infinite interval of the form $[a_i, \infty)$. Thus, the formula $\forall x_1, x_2.\ Q_1(x_1) \wedge Q_2(x_2) \Rightarrow Q(x_1, x_2)$ requires the predicate Q to contain the infinite rectangle $[a_1, \infty) \times [a_2, \infty)$. Thus, a predicate Q on $\mathbb{Z}^2$ is "ultimately true" w.r.t. to the product filter if and only if it is "true on some infinite rectangle". In Bourbaki's terminology [Bourbaki, 1995, Chapter 1, §6.7], the infinite rectangles form a *basis* of the product filter.

We view the product filter as the default filter on the product type $A_1 \times A_2$. Whenever we refer to $A_1 \times A_2$ in a setting where a filtered type is expected, the product filter is intended.

We stress that there are several filters on $\mathbb{Z}$, including the universal filter and the order filter, and therefore several filters on $\mathbb{Z}^k$. Therefore, it does not make sense to use the O notation without specifying which filter one considers. Consider again the function $\mathrm{g}(n, m)$ in Figure 8.2. One can prove that $\mathrm{g}(n, m)$ has complexity $O(nm + n)$ with respect to the standard filter on $\mathbb{Z}^2$. With respect to *this filter*, this complexity bound is equivalent to $O(mn)$, as the functions $\lambda(m, n).mn+n$ and $\lambda(m, n).mn$ dominate each other. Unfortunately, this *does not allow* deducing anything about the complexity of $\mathrm{g}(n, 0)$, since the bound $O(mn)$ holds only when n and m grow large. An alternate approach is to prove that $\mathrm{g}(n, m)$ has complexity $O(nm + n)$ with respect to a stronger filter, namely the product of the standard filter on $\mathbb{Z}$ and the universal filter on $\mathbb{Z}$. With respect to *that filter*, the functions $\lambda(m, n).mn + n$ and $\lambda(m, n).mn$ are *not* equivalent. This bound *does allow* instantiating m with 0 and deducing that $\mathrm{g}(n, 0)$ has complexity $O(n)$.

Several useful properties of the domination relation with respect to a given filter, such as the summation lemma, may be found in [Guéneau, 2019, §3.4].

8.5 Using Big-O Notation in Specifications

As a running example, consider a function that computes the length of a list:

```
let rec length l =
  match l with
  | [] -> 0
  | _ :: l -> 1 + length l
```

About this function, one can prove the following statement:

$$\forall (A : \mathsf{Type})(l : \mathsf{list}\ A).\ \ \{\, \$(|l| + 1)\,\}\ (\texttt{length}\ l)\ \{\lambda y.\, [\, y = |l|\,]\}$$

The postcondition $\lambda y.\, [\, y = |l|\,]$ asserts that the call `length` l returns the length of the list l. The precondition $\$(|l| + 1)$ asserts that this call requires $|l| + 1$ credits.

The above specification guarantees that `length` thus runs in linear time. It does not allow predicting how much real time is consumed by a call to `length`. Thus, this specification is already rather abstract. Yet, it is still too precise. Indeed, we believe that it would not be wise for a list library to publish a specification of `length` whose precondition requires exactly $|l| + 1$ credits. Indeed, there are implementations of `length` that do not meet this specification. For example, the tail-recursive implementation found in the OCaml standard library, which in practice is more efficient than the naïve implementation shown above, involves exactly $|l| + 2$ function calls, therefore requires $|l| + 2$ credits. By advertising a specification where $|l| + 1$ credits suffice, one makes too strong a guarantee, and rules out the more efficient implementation.

After initially publishing a specification that requires $\$(|l| + 1)$, one could of course still switch to the more efficient implementation and update the published specification so as to require

$\$(|l| + 2)$ instead of $\$(|l| + 1)$. However, that would in turn require updating the specification and proof of every (direct and indirect) client of the list library, which is intolerable.

To leave some slack, one should publish a more abstract specification. For example, one could advertise that the cost of `length` l is an affine function of the length of the list l, that is, the cost is $a \cdot |l| + b$, for some constants a and b:

$$\exists(a, b : \mathbb{Z}).\ \forall(A : \mathsf{Type})(l : \mathsf{list}\ A).\ \ \{\$(a \cdot |l| + b)\}\ (\texttt{length}\ l)\ \{\lambda y.\, [\, y = |l|\,]\}$$

This is a better specification, in the sense that it is more modular. The naïve implementation of `length` shown earlier and the efficient implementation in OCaml's standard library both satisfy this specification, so one is free to choose one or the other, without any impact on the clients of the list library. In fact, any reasonable implementation of `length` should have linear time complexity and therefore should satisfy this specification.

That said, the style in which the above specification is written is arguably slightly too low-level. Instead of directly expressing the idea that the cost of `length` l is $O(|l|)$, we have written this cost under the form $a \cdot |l| + b$. It is preferable to state at a more abstract level that $cost$ is dominated by $\lambda n.n$: such a style is more readable and scales to situations where multiple parameters and nonstandard filters are involved. Thus, we propose the following statement:

$$\exists cost : \mathbb{Z} \to \mathbb{Z}.\ \begin{cases} cost\ \preceq_{\mathbb{Z}}\ \lambda n.\, n \\ \forall(A : \mathsf{Type})(l : \mathsf{list}\ A).\ \ \{\$cost(|l|)\}\ (\texttt{length}\ l)\ \{\lambda y.\, [\, y = |l|\,]\} \end{cases}$$

Thereafter, we refer to the function $cost$ as the *concrete cost* of `length`, as opposed to the *asymptotic bound*, represented here by the function $\lambda n.\, n$. This specification asserts that there exists a concrete cost function $cost$, which is dominated by $\lambda n.\, n$, such that $cost(|l|)$ credits suffice to justify the execution of `length` l. Thus, $cost(|l|)$ is an upper bound on the actual number of `pay` instructions that are executed at runtime.

The above specification informally means that `length` l has time complexity $O(n)$ where the parameter n represents $|l|$, that is, the length of the list l. The fact that n represents $|l|$ is expressed by applying $cost$ to $|l|$ in the precondition. The fact that this analysis is valid when n grows large enough is expressed by using the standard filter on $\mathbb{Z}$ in the assertion $cost\ \preceq_{\mathbb{Z}}\ \lambda n.\, n$.

In general, it is up to the user to choose what the parameters of the cost analysis should be, what these parameters represent, and which filter on these parameters should be used. The example of the Bellman-Ford algorithm (Section 8.6) illustrates this.

The specifications presented in the previous section share a common structure. We define a record type that captures this common structure, so as to make specifications more concise and more recognizable, and so as to help users adhere to this specification pattern.

This type, `specO`, is defined in Figure 8.3. The first three fields in this record type correspond to what has been explained so far. The first field asserts the existence of a function `cost` of A to $\mathbb{Z}$,

```
Record specO (A : filterType) (le : A → A → Prop)
             (bound : A → Z) (P : (A → Z) → Prop)
  := { cost : A → Z;
       cost_spec : P cost;
       cost_dominated : dominated A cost bound;
       cost_nonneg : ∀x, 0 ⩽ cost x;
       cost_monotonic : monotonic le Z.le cost; }.
```

Figure 8.3: Definition of `specO`.

where A is a user-specified filtered type. The second field asserts that a certain property `P cost` is satisfied; it is typically a Separation Logic triple whose precondition refers to `cost`. The third field asserts that `cost` is dominated by the user-specified function `bound`. The last two fields, which restrict cost functions to be nonnegative and monotonic, are included for technical reasons that are explained in [Guéneau, 2019, §4.4 and §4.5].

Using this definition, our proposed specification of `length` is stated in concrete Coq syntax as follows:

```
Theorem length_spec:
  specO Z_filterType Z.le (fun n ⇒ n) (fun cost ⇒
    ∀A (l:list A), TRIPLE (length l)
      PRE ($ (cost |l|))
      POST (fun y ⇒ \[ y = |l| ]))
```

The key elements of this specification are `Z_filterType`, which is $\mathbb{Z}$, equipped with its standard filter; the asymptotic bound `fun n ⇒ n`, which means that the time complexity of `length` is $O(n)$; and the Separation Logic triple, which describes the behavior of `length`, and refers to the concrete cost function `cost`.

One key technical point is that `specO` is a strong existential, whose witness can be referred to via to the first projection, `cost`. For instance, the concrete cost function associated with `length` can be referred to as `cost length_spec`. Thus, at a call site of the form `length xs`, the number of required credits is `cost length_spec |xs|`.

To prove a specification lemma, such as `length_spec`, one must construct a `specO` record. By definition of `specO` (Figure 8.3), this means that one must exhibit a concrete cost function $cost$ and prove a number of properties of this function, including the fact that, when supplied with $\$(cost\ \ldots)$, the code runs correctly (`cost_spec`) and the fact that $cost$ is dominated by the desired asymptotic bound (`cost_dominated`).

Thus, the very first step in a naïve proof attempt would be to *guess* an appropriate cost function for the code at hand. However, such an approach would be painful, error-prone, and brittle. It seems much preferable, if possible, to enlist the machine's help in *synthesizing* a cost function *at the same time as we step through the code*—which we have to do anyway, as we must build a Separation Logic proof of the correctness of this code. Armaaël Guéneau developed a framework for user-guided synthesis of cost functions [Guéneau, 2019, §5].

8.6 Small Case Studies

Binary Search. We prove that binary search has time complexity $O(\log n)$, where $n = j - i$ denotes the width of the search interval $[i, j)$. The code is as in Figure 7.1, except that the flaw is fixed by replacing `i+1` with `k+1` on the last line. We synthesize the following recurrence equation on the cost function f:

$$f(0) + 3 \leqslant f(1) \quad \wedge \quad \forall n \geqslant 0.\ 1 \leqslant f(n) \quad \wedge \quad \forall n \geqslant 2.\ f(n/2) + 3 \leqslant f(n)$$

To derive the asymptotic cost, we apply the substitution method and search for a solution of the form $\lambda n.\ \mathit{if}\ n \leqslant 0\ \mathit{then}\ 1\ \mathit{else}\ a \log n + b$, which is dominated by $\lambda n.\ \log n$. Substituting this shape into the above constraints, we find that they boil down to $(4 \leqslant b) \wedge (0 \leqslant a \wedge 1 \leqslant b) \wedge (3 \leqslant a)$. Finally, we guess a solution, namely $a := 3$ and $b := 4$.

Dependent Nested Loops. Many algorithms involve dependent nested `for` loops, that is, nested loops, where the bounds of the inner loop depend on the outer loop index, as in the following simplified example:

```
for i = 1 to n do
  for j = 1 to i do () done
done
```

For this code, the cost function $\lambda n.\ \sum_{i=1}^{n}(1 + \sum_{j=1}^{i} 1)$ is synthesized. There remains to prove that it is dominated by $\lambda n.n^2$. We could recognize and prove that this function is equal to $\lambda n.\frac{n(n+3)}{2}$, which clearly is dominated by $\lambda n.n^2$. This works because this example is trivial, but, in general, computing explicit closed forms for summations is challenging, if at all feasible.

A higher-level approach is to exploit the fact that, if f is monotonic, then $\sum_{i=1}^{n} f(i)$ is less than $n \cdot f(n)$. Applying this lemma twice, we find that the above cost function is less than $\lambda n.\sum_{i=1}^{n}(1+i)$ which is less than $\lambda n.n(1+n)$ which is dominated by $\lambda n.n^2$. This simple-minded approach, which does not require a *summation lemma*, is often applicable.

A Loop Whose Body Has Exponential Cost. This example illustrates a situation where to exploit the *summation lemma*—see [Guéneau, 2019, Lemma 5.3.8]. In this example, the loop body is just a function call:

```
for i = 0 to n-1 do b(i) done
```

Thus, the cost of the loop body is not known exactly. Instead, let us assume that a specification for the auxiliary function b has been proved and that its cost is $O(2^i)$, that is, $\texttt{cost b} \preceq_{\mathbb{Z}} \lambda i.\ 2^i$ holds. We then wish to prove that the cost of the whole loop is also $O(2^n)$.

For this loop, the cost function $\lambda n.\ \sum_{i=0}^{n}(1 + \texttt{cost b}\ (i))$ is automatically synthesized. We have an asymptotic bound for the cost of the loop body, namely: $\lambda i.\ 1 + \texttt{cost b}\ (i) \preceq_{\mathbb{Z}} \lambda i.\ 2^i$. The side conditions of the summation lemma are met: in particular, the function $\lambda i.\ 1+\texttt{cost b}\ (i)$ is monotonic. The lemma yields $\lambda n.\ \sum_{i=0}^{n}(1 + \texttt{cost b}\ (i)) \preceq_{\mathbb{Z}} \lambda n.\ \sum_{i=0}^{n} 2^i$. Finally, we have $\lambda n.\ \sum_{i=0}^{n} 2^i = \lambda n.\ 2^{n+1} - 1 \preceq_{\mathbb{Z}} \lambda n.\ 2^n$.

The Bellman-Ford Algorithm. We verify the asymptotic complexity of an implementation of Bellman-Ford algorithm, which computes shortest paths in a weighted graph with n vertices and m edges. The algorithm involves an outer loop that is repeated $n-1$ times and an inner loop that iterates over all m edges. The specification asserts that the asymptotic complexity is $O(nm)$:

$$\exists cost : \mathbb{Z}^2 \to \mathbb{Z}.\ \begin{cases} cost \preceq_{\mathbb{Z}^2} \lambda(m,n).\, nm \\ \{\$cost(\#edges(g), \#vertices(g))\}\ (\texttt{bellmanford}\ g)\ \{\ldots\} \end{cases}$$

By exploiting the fact that a graph without duplicate edges must satisfy $m \leqslant n^2$, we prove that the complexity of the algorithm, viewed as a function of n, is $O(n^3)$.

$$\exists cost : \mathbb{Z} \to \mathbb{Z}.\ \begin{cases} cost \preceq_{\mathbb{Z}} \lambda n.\, n^3 \\ \{\$cost(\#vertices(g))\}\ (\texttt{bellmanford}\ g)\ \{\ldots\} \end{cases}$$

To prove that the former specification implies the latter, one instantiates m with n^2, that is, one exploits a *composition lemma* [Guéneau, 2019, Lemma 5.3.16]. In practice, we publish both specifications and let clients use whichever one is more convenient.

Union-Find. In our original formalization of Union-Find [Charguéraud and Pottier, 2019], we proved that the (amortized) concrete cost of `find` is $2\alpha(n) + 4$, where n is the number of elements. With a few lines of proof, we can now derive a specification where the cost of `find` is expressed under the form $O(\alpha(n))$:

```
specO Z_filterType Z.le (fun n ⇒ alpha n) (fun cost ⇒
  ∀D R V x, x ∈ D → triple (UnionFind_ml.find x)
```

```
PRE (UF D R V ⋆ $(cost (card D)))
POST (fun y ⇒ UF D R V ⋆ \[ R x = y ])).
```

Union-Find is a mutable data structure, whose state is described by the abstract predicate UF D R V. In particular, the parameter D represents the domain of the data structure, that is, the set of all elements created so far. Thus, its cardinal, `card D`, corresponds to n. This case study illustrates a situation where the cost of an operation depends on the current state of a mutable data structure.

8.7 Formal Analysis of Incremental Cycle Detection

This section discusses our mechanized proof [Guéneau et al., 2019b] of an *incremental* cycle detection algorithm for directed graphs [Bender et al., 2016]. The aim of such an algorithm is to receive queries for adding vertices and edges (arcs) to a graph, and to detect the first edge whose insertion creates a cycle.

In terms of applications, Haeupler et al. [2008] write: "*This problem arises in incremental circuit evaluation, pointer analysis, management of compilation dependencies, and deadlock detection.*" Let us also mention two specific applications in the field of programming languages. First, Jacques-Henri Jourdan [2016] has deployed an (as-yet-unverified) incremental cycle detection algorithm in the kernel of the Coq proof assistant [The Coq development team, 2020], where it is used to check the satisfiability of universe constraints [Sozeau and Tabareau, 2014, §2]. Second, the Dune build system [Jane Street, 2018] needs an incremental cycle detection algorithm in order to reject circular build dependencies as soon as possible.

Bender, Fineman, Gilbert, and Tarjan [Bender et al., 2016] proposed an algorithm that provides an asymptotic improvement over the best previously-known bounds. The complexity of this algorithm for building a directed graph of n vertices and m edges, while incrementally ensuring that no edge insertion creates a cycle, is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Although the implementation of this algorithm is relatively straightforward, its design is subtle, and it is far from obvious, by inspection of the code, that the advertised complexity bound is respected.

We actually consider a slightly enhanced variant of Bender et al.'s algorithm. To handle the insertion of a new edge, the original algorithm depends on a runtime parameter, which limits the extent of a certain backward search. This parameter influences only the algorithm's complexity, not its correctness. Bender et al. show that setting it to $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm allows achieving the advertised complexity. This means that, in order to run the algorithm, one must anticipate the *final* values of m and n. This seems at least awkward, or even impossible, if one wishes to use the algorithm in an online setting, where the sequence of operations is not known in advance. Instead, we proposed a modified algorithm, where the extent of the backward search is limited by a value that depends only on the *current* state. Our modified algorithm has the same complexity as the original algorithm and is a genuine online algorithm.

Pseudocode. The pseudocode appears in Figure 8.4. Observe that the algorithm is *from first principles*: it relies solely on carefully controlled depth-first searches, and labelling of vertices.

When the user requests the creation of an edge from v to w, finding out whether this operation would create a cycle amounts to determining whether a path already exists from w to v. A naïve algorithm could search for such a path by performing a forward search, starting from w and attempting to reach v. To achieve improved bounds, Bender et al.'s algorithm (and our variant) exploit 3 key ingredients.

The first ingredient is to associate, to every vertex v, a positive integer level $L(v)$. The following invariant is maintained: L forms a pseudo-topological numbering. That is, "no edge goes

To insert a new edge from v to w and detect potential cycles:
- If $L(v) < L(w)$, insert the edge (v, w), declare success, and exit
- Perform a backward search:
  - start from v
  - follow an edge (backward) only if its source vertex x satisfies $L(x) = L(v)$
  - if w is reached, declare failure and exit
  - if $L(v)$ edges have been traversed, interrupt the backward search
- If the backward search was not interrupted, then:
  - if $L(w) = L(v)$, insert the edge (v, w), declare success, and exit
  - otherwise set $L(w)$ to $L(v)$
- If the backward search was interrupted, then set $L(w)$ to $L(v) + 1$
- Perform a forward search:
  - start from w
  - upon reaching a vertex x:
    - if x was visited during the backward search, declare failure and exit
    - if $L(x) \geqslant L(w)$, do not traverse through x
    - if $L(x) < L(w)$, set $L(x)$ to $L(w)$ and traverse x
- Finally, insert the edge (v, w), declare success, and exit

Figure 8.4: Pseudocode for our online variant of Bender et al.'s cycle detection algorithm.

down": if there is an edge from v to w, then $L(v) \leqslant L(w)$ holds. The presence of levels can be exploited to accelerate a search: for instance, during a forward search whose purpose is to reach the vertex v, any vertex whose level is greater than that of v can be disregarded. The price to pay is that the invariant must be maintained: when a new edge is inserted, the levels of some vertices must be adjusted.

A second key ingredient of the algorithm is that it not only performs a forward search, but begins with a backward search that is both *restricted* and *bounded*. It is restricted in the sense that it searches only one level of the graph: starting from v, it follows only *horizontal* edges, that is, edges whose endpoints are both at the same level. Therefore, all the vertices that it discovers are at level $L(v)$. It is bounded in the sense that it is interrupted, even if incomplete, after it has processed $L(v)$ edges.[3]

A third key ingredient of the algorithm is the manner in which levels are updated so as to maintain the invariant when a new edge is inserted. Bender et al. adopt the policy that the level of a vertex can never decrease. Thus, when an edge from v to w is inserted, all the vertices that are accessible from w must be promoted to a level that is at least the level of v. In principle, there are many ways of doing so. Bender et al. proceed as follows: if the backward search was not interrupted, then w and its descendants are promoted to the level of v; otherwise, they are promoted to the next level, $L(v) + 1$. In the latter case, $L(v) + 1$ is possibly a new level. We see that such a new level can be created only if the backward search has not completed, that is, only if there exist at least F edges at level $L(v)$. In short, a new level may be created only if the previous level contains sufficiently many edges. This mechanism is used to control the number of levels. Bounding this number is key to establishing the bound $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. I wrote a high-level explanation of why this bound holds in [Guéneau et al., 2019b, §6 and Appendix A].

[3] Whereas we interrupt the backward search after processing $L(v)$ edges, Bender et al.'s original algorithm interrupts it after processing $\min(m^{1/2}, n^{2/3})$ edges, where m and n are upper bounds on the *final* numbers of edges and vertices in the graph, and not the *current* values. Requiring m and n to be known ahead of time mean that Bender et al.'s algorithm is not truly online.

COMPLEXITY
nonnegative $\psi \ \wedge$ monotonic $\psi \ \wedge$
$\psi \preceq_{\mathbb{Z}\times\mathbb{Z}} \lambda(m,n).\,(m \cdot \min(m^{1/2}, n^{2/3}) + n)$

INITGRAPH
$\exists k.\ \{\$k\}$ init_graph() $\{\lambda g.\text{IsGraph}\ g\ \varnothing\}$

ADDVERTEX
$\forall g\ G\ v.$
let $m, n :=$|edges G|, |vertices G| in
$v \notin$ vertices $G \implies$
$\left\{ \begin{array}{l} \text{IsGraph}\ g\ G\ \star \\ \$(\psi\,(m, n+1) - \psi\,(m,n)) \end{array} \right\}$
(add_vertex $g\ v$)
$\left\{ \begin{array}{l} \lambda().\,\text{IsGraph}\ g\ (G + v) \\ \end{array} \right\}$

ADDEDGE
$\forall g\ G\ v\ w.$
let $m, n :=$|edges G|, |vertices G| in
$v, w \in$ vertices $G \ \wedge\ (v,w) \notin$ edges $G \implies$
$\left\{ \begin{array}{l} \text{IsGraph}\ g\ G\ \star \\ \$(\psi\,(m+1, n) - \psi\,(m,n)) \end{array} \right\}$
(add_edge_or_detect_cycle $g\ v\ w$)
$\left\{ \begin{array}{l} \lambda res.\ \text{match}\ res\ \text{with} \\ \quad \mid \text{Ok} \Rightarrow \text{IsGraph}\ g\ (G + (v,w)) \\ \qquad\qquad \star[\forall x.\ x \longrightarrow^{+}_{G+(v,w)} x] \\ \quad \mid \text{Cycle} \Rightarrow [w \longrightarrow^{*}_{G} v] \end{array} \right\}$

Figure 8.5: Specification of an incremental cycle detection algorithm.

Formal specification. We next present the key elements of the specifications, which formally establishes that the incremental cycle detection algorithm indeed correctly detects cycles, and moreover admits the complexity bound $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Figure 8.5 shows the formal specification. It consists of three public operations: init_graph, which creates a fresh empty graph, add_vertex, which adds a vertex, and add_edge_or_detect_cycle, which either adds an edge or report that this edge cannot be added because it would create a cycle. Specifications are expressed using the representation predicate IsGraph $g\ G$, which asserts that, at address g in memory, one finds a well-formed data structure that represents the mathematical graph G. Internally, this assertion stores a certain number of time credits—the potential. The specification consists of four statements.

COMPLEXITY asserts the existence of a function ψ, whose exact definition is not exposed, but that satisfies the expected asymptotic bound: $\psi(m,n) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. (The relation $\preceq_{\mathbb{Z}\times\mathbb{Z}}$ is a domination relation between functions of type $\mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$.) Intuitively, $\psi(m,n)$ is meant to represent the advertised cost of a sequence of n vertex creation and m edge creation operations. In other words, it is the number of credits that one must pay in order to create n vertices and m edges.

INITGRAPH states that the function call init_graph() creates a valid data structure, which represents the empty graph $\varnothing$, and returns its address g. Its cost is k, where k is an unspecified constant; in other words, its complexity is $O(1)$.

ADDVERTEX states that add_vertex requires a valid data structure, described by the assertion IsGraph $g\ G$, and returns a valid data structure, described by IsGraph $g\ (G + v)$. Here, $G + v$ denotes the result of extending the mathematical graph G with a new vertex v and $G + (v,w)$ for the result of extending G with a new edge from v to w. In addition, add_vertex requires $\psi(m, n+1) - \psi(m,n)$ credits. These credits are not returned: they do not appear in the postcondition. They either are actually consumed or become stored inside the data structure for later use. Thus, one can think of $\psi(m, n+1) - \psi(m,n)$ as the amortized cost of add_vertex.

ADDEDGE states that the cost of add_edge_or_detect_cycle is $\psi(m+1, n) - \psi(m,n)$. This postcondition distinguishes two cases. If the operation returns Ok, then the graph has been suc-

cessfully extended with a new edge from v to w. The assertion $\forall x.\ x \longrightarrow^{+}_{G+(v,w)} x$ asserts that the extended graph does not contain any cycle. If, however, the operation returns Cycle, then the edge from v to w is not added because the graph G already contains a path from w to v. In the latter case, the data structure is invalidated: the assertion $\mathrm{IsGraph}\, g\ G$ is not returned. Thus, in that case, no further operations on the graph are allowed. (An additional rollback mechanism would be needed to allow resuming after a cycle detection.)

By combining the three triples in Figure 8.5, a client can verify that a call to init_graph, followed with an arbitrary interleaving of n calls to add_vertex and m successful calls to the function add_edge_or_detect_cycle, satisfies the specification $\{\$(k + \psi(m, n))\}\ \ldots\ \{\top\}$, where k is the cost of init_graph. Indeed, the cumulated cost of the calls to the functions add_vertex and add_edge_or_detect_cycle forms a telescopic sum that adds up to $\psi(m, n) - \psi(0, 0)$, which itself is bounded by $\psi(m, n)$.

Closing words on formal big-O bounds. This case study shows that our program logic allows to simultaneously verify the correctness and complexity of a production-ready implementation of a state-of-the-art algorithm—our verified implementation runs in production in the Dune build system [Jane Street, 2018] since 2019.

Our public specification exposes an asymptotic complexity bound: no literal constants appear in it. There, the use of the big-O notation in *specifications* is a significant improvement compared with concrete bounds, which are verbose, reveal implementation detail, and lack modularity. We remark, however, that it is sometimes difficult to use something that resembles the O notation for reasoning about the costs involved in the *implementation* of a complex algorithm or data structure. Indeed, it may be the case that an existential quantifier must be hoisted very high, so that its scope encompasses not just a single statement, but possibly a group of definitions, statements, and proofs.

Chapter 9

Space Bounds for Garbage-Collected Heap Space

This chapter presents the key ideas associated with the set up of a Separation Logic with *space credits* that allows establishing space bounds for programs equipped with a garbage collector. Doing so involves, in particular, reasoning about *roots* and about *unreachability*. To begin with, we need to formalize the notion of roots, in a way that matches the definition used in practice by compilers (Section 9.1). We distinguish, for the purpose of inductive reasoning about programs, the notion of *visible roots* from that of *invisible roots* (Section 9.2). I will not present the reasoning rules of the logic, but only explain the intuition behind the key logical deallocation rule (Section 9.3) and a novel *Stackable* assertion (Section 9.4). I then explain how to set up a semantics that account for the *physical* garbage collection steps (Section 9.5), and how to state the soundness theorem that expresses the bound on the maximal space usage (Section 9.6). Finally, I present an example application involving a modular construction of a stack of stacks (Section 9.7).

The contents of this section corresponds to the beginning of the PhD Alexandre Moine, co-advised by François Pottier and myself. The text of this section is a mildly revised version of the text that appeared in our POPL'23 paper [Moine et al., 2023]. Our program logic builds on prior work by Jean-Marie Madiot and François Pottier, published at POPL'22 [Madiot and Pottier, 2022]. In that prior work, they tackled the problem of space bounds for an artificial language in which the notion of *roots considered by the garbage collector* is made trivial. Our main contributions were to generalize the work to target a standard ML-style language satisfying the *free variable rule* (used to determine the roots), to add reasoning rules for function closures, and to cover more challenging examples. Each of these aspects required significant technical additions. Alexandre Moine formalized all the program logic and the accompanying examples in Coq using the Iris framework [Jung et al., 2018b], which provides fractional permissions and advanced forms of ghost state that we rely upon.

9.1 Reachability, Roots, and The Free Variable Rule

An allocation consumes a number of space credits, depending on the size of the requested block. These credits can be reclaimed upon deallocation. Yet, in the absence of a memory deallocation

instruction in the language, deallocation points are not explicit. A tracing garbage collector (GC) can be invoked at arbitrary points in time, and may deallocate any subset of the *unreachable blocks*. An unreachable block is a block that is not *reachable* from any *root* via a *path* in the heap. Thus, deallocation takes the form of a *logical operation*: it is up to the person who verifies the program to decide at which program points an unreachable memory block should be reclaimed. The GC may physically deallocate a block before or after the point where the user chooses to logically deallocate this block.

The heap-allocated data forms a graph, where *allocated blocks* correspond to the vertices, and where *pointers* stored in the blocks correspond to edges. Certain blocks have their addresses registered as *roots*. To enable modular reasoning about unreachability, we exploit *pointed-by* assertions [Kassios and Kritikos, 2013]. Such assertions record the *predecessors* of a memory block. They may be divided into *fractions*, allowing for modular reasoning about predecessors. At some point through the reasoning about a program, it becomes possible to assert that a memory block has no heap predecessors. If, furthermore, the block is not a root, then this block can be logically deallocated.

A central question is to formalize the notion of *root*. What is a root? How can this concept be reflected in a small-step, substitution-based operational semantics? A commonly agreed-upon answer is given by the *free variable rule* (FVR) [Felleisen and Hieb, 1992; Morrisett et al., 1995]. Technically, this rule states that *a root is a memory location ℓ such that ℓ occurs in the term that is undergoing reduction*. In slightly more informal words, ℓ is a root if and only if it appears possible that ℓ might be used in the future, based on the existence of a path from the current program point to a program point where ℓ is used. The FVR represents a conservative approximation of the locations that will be accessed in the future: indeed, depending on which branches are taken, it may turn out that ℓ is in fact never accessed.

Our starting point is an operational semantics where the FVR is built in. We propose a program logic that is sound with respect to this semantics, and we use this logic to establish worst-case space complexity bounds. To obtain a binary program that respects the complexity bounds established using our logic, one needs a compiler (and runtime system) that respect the FVR. As a prominent example, the CakeML compiler [Tan et al., 2019], provably respects the FVR.[1] More precisely, Gómez-Londoño et al. [2020] prove that CakeML respects the operational semantics of its source language, and respects a cost model that is defined at the level of an intermediate language named DataLang.

Our work is complementary with that of CakeML: whereas its authors prove that the CakeML compiler respects this cost model, we propose a program logic that allows establishing space complexity bounds, based on a similar cost model. Adapting in the future our program logic to DataLang would allow obtaining an end-to-end guarantee, that is, establishing a space complexity bound about a source CakeML program and deriving a bound about the compiled program.

9.2 Visible and Invisible Roots

One may wonder why the FVR is so named, since its statement does not contain the word "variable". The answer lies in the gap between the programmer's point of view and the semantic point of view. A programmer may like to think that the roots are *variables*. When the programmer focuses on a certain program point, corresponding to a subterm t, a variable x that occurs free in t can be regarded by the programmer as a root at this program point—whence the name of

[1] As far as we know, many real-world implementations of garbage-collected languages, such as OCaml, SML, Scala, Java, and many more, are meant to respect the FVR. Unfortunately, this intention is often undocumented.

```
1  let rec rev_append(xs, ys) =
2    if is_nil(xs) then ys else
3      let x = head(xs) in
4      let xs' = tail(xs) in
5      let ys' = cons(x, ys) in
6      rev_append(xs', ys')
```

Figure 9.1: An implementation of linked list reversal

the "free variable rule". In contrast, in the operational semantics, there are no variables: they are substituted away and replaced with closed values. Thus, in the operational semantics and in our reasoning rules, the roots are *memory locations*. When we write that "the address x is a root" at a certain program point, we mean that, once this program point is reached, the memory location with which the variable x has been replaced is a root.

Let us illustrate reasoning about roots via the example of the function `rev_append` (Figure 9.1). This function expects two linked lists and returns a linked list. A call to the function `rev_append(xs, ys)` returns a list whose elements are the elements of `xs` in reverse order followed with the elements of `ys`. This code is expressed in an untyped language using ML syntax. For simplicity, we do not use pattern matching; instead, we use the auxiliary functions `is_nil`, `head`, `tail`, and `cons`, whose definitions are omitted. A linked list is represented as a heap block whose first field holds the integer tag 0 or 1. If the tag is 0, then there are no more fields; if the tag is 1, then there are two more fields, holding the head and tail of the list. We now wish to explain which locations are roots, at each program point in `rev_append`, according to the FVR. Before doing so, however, we must point out that, when one reasons about `rev_append` in isolation, its calling context is unknown. By inspecting the code of this function, one can tell that certain memory locations are roots at certain points; we refer to these as the *visible roots*. However, in addition, every caller along the unknown call chain may have retained certain memory locations. One can think of them as locations that appear "in the stack". From a semantic point of view, these locations occur in the evaluation context, so, according to the FVR, they are also roots. We refer to them as the *invisible roots*. The set of all roots is the union of the sets of visible roots and invisible roots. These sets may overlap.

At the entry point of `rev_append` (at the beginning of line 2), the locations `xs` and `ys` are visible roots, because the variables `xs` and `ys` occur free in the code that remains to be executed (that is, the whole function body). Upon entering the `else` branch, on line 3, `xs` and `ys` are still roots. At the beginning of line 5, after reading the "head" and "tail" fields of the first list cell, two more variables (namely, `x` and `xs'`) are visible roots, but `xs` is no longer one, as it does not occur on lines 5–6. A somewhat subtle phenomenon takes place at this point: the location `xs` may or may not be an invisible root. If it is *not* an invisible root, which means that no caller has retained the address of the list `xs`, then this address is not a root at all, which means that the first list cell can be reclaimed at this program point by the GC. Otherwise, this cell cannot be reclaimed. On line 5, a fresh cell, named `ys'`, is allocated. At the beginning of line 6, `ys` is no longer a visible root, but `ys'` is one. The location `ys` remains reachable via `ys'`, thus the list `ys` cannot be deallocated. Finally, on line 6, a tail-recursive call is made. The locations `xs'` and `ys'` cease to be roots for this instance of `rev_append`, but immediately become roots for the new instance of `rev_append`.

What is the (heap) space complexity of `rev_append`? Two distinct answers can be given. On the one hand, without any assumption about the calling context, one can state that the space complexity is linear in the length of the list `xs`. This is due to the allocation of a new cell at line 5. On the other hand, under the assumption that the address `xs` is *not* retained by the calling context

(that is, `xs` is not an invisible root), `rev_append` runs in constant heap space.[2] Indeed, in that case, the cost of allocating a new cell at line 5 can be compensated by deallocating the cell `xs`, which is no longer a root, also at line 5. There is no guarantee that the GC *will* deallocate the cell `xs` at this point, but it *can* do so, which is what matters.

9.3 Logical Deallocation and its Requirements

Suppose one wishes to verify the claim that `rev_append` runs in constant space, under the assumption that `xs` is not an invisible root—that is, under the assumption that `rev_append`, when called, holds the unique pointer onto its argument `xs`.

A key step in this proof takes place at the beginning of line 5. There, one must apply a logical deallocation rule to the list cell `xs`, so as to recover a number of space credits, which can then be used to pay for the allocation of a new cell on the same line. Our logical deallocation rule requires proving that `xs` has no predecessors (in the heap) and is not a (visible or invisible) root. More specifically, its requirements are as follows:

- As in traditional Separation Logic [Reynolds, 2002], a full *points-to assertion* for the memory block at address `xs` is required. This assertion is obtained by unfolding the representation predicate for lists, used to express assumptions about the lists `xs` and `ys`.
- As in Madiot in Pottier's system [2022], a full *pointed-by assertion* for `xs`, carrying an empty multiset of predecessors, is required. This assertion too is obtained by unfolding the representation predicate for lists.
- A proof that `xs` is *not a visible root* is required. To establish this fact, one first computes the visible roots at the beginning of line 5: they are the addresses `x`, `xs'`, and `ys`. Then, one must prove that the address `xs` is not a member of this set. This check is not syntactic: proving that the address `xs` is distinct from the addresses `x`, `xs'`, and `ys` requires Separation Logic reasoning. For instance, proving that `xs` and `ys` are distinct addresses follows from the presence of separate list assertions about `xs` and `ys`.
- A proof that `xs` is *not an invisible root* is required. In other words, a proof that no direct or indirect caller has retained the address `xs` is required. Here, the only way of proving this property is to make it an assumption, that is, to let it appear in the precondition of the function `rev_append`. We express this assumption using our novel *Stackable* assertions.

Another key step in the proof takes place at the recursive call `rev_append(xs', ys')` on line 6. To prove that this call is permitted, one must prove that the precondition of `rev_append`, instantiated with the actual parameters `xs'` and `ys'`, is satisfied. Thus, according to the last bullet point above, one must prove that `xs'` is not an invisible root. In other words, one must prove that the cell that follows the cell `xs` in the linked list is not an invisible root. Where might this evidence come from? The most natural answer, we argue, is to bake it in the definition of the list representation predicate: the definition of a valid linked list must state that a cell that is the destination of a link is never an invisible root.

In summary, we have outlined the requirements of our logical deallocation rule and explained the need for a new Separation Logic assertion, which guarantees that a memory location ℓ is not an invisible root. This assertion, written $\mathit{Stackable}\ \ell\ 1$, is described next.

[2] Because `rev_append` is tail-recursive, it runs in constant stack space as well, but that is another story. In this paper, we are not concerned with stack space usage.

9.4 Reasoning about Invisible Roots

To understand how one might keep track in Separation Logic of which memory locations are or are not invisible roots, one must first have a clear picture of what this means and at what points in a proof a location *becomes* or *ceases to be* an invisible root.

A proof in Separation Logic is carried out under an unknown context. That is, one reasons about a term t without knowing in what evaluation context K this term is placed. There are specific points in the proof where this unknown context grows and shrinks. As an archetypical example, consider the sequencing construct $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$. To reason about this construct, one first focuses on the term t_1, thereby temporarily forgetting the frame $\mathsf{let}\, x = [] \,\mathsf{in}\, t_2$, which is pushed onto the unknown context. After the verification of t_1 is completed, this focusing step is reversed: the frame $\mathsf{let}\, x = [] \,\mathsf{in}\, t_2$ is popped and one continues with the verification of t_2. These focusing and defocusing steps are described by the "bind" rule of Separation Logic [Jung et al., 2018a, §6.2].

An invisible root is a memory location that occurs in the unknown context K. When this context grows and shrinks, the set of invisible roots grows and shrinks as well. More specifically, when the user of the program logic focuses on t_1, a location ℓ that occurs in the frame $\mathsf{let}\, x = [] \,\mathsf{in}\, t_2$ (that is, a location that occurs in t_2) becomes an invisible root: it is "pushed onto the stack", so to speak. (This location may have been an invisible root already, prior to this focusing step.) This is undone when this focusing step is reversed: this location is "popped off the stack".

To keep track in Separation Logic, on a per-location basis, of whether a location may be or definitely is not an invisible root, we propose the following discipline.

- We introduce an assertion $\mathit{Stackable}\ \ell\ p$, where p is a rational number such that $0 < p \leqslant 1$. The presence of a fraction allows $\mathit{Stackable}$ assertions to be split and joined.
- The assertion $\mathit{Stackable}\ \ell\ 1$ appears when a fresh memory block is allocated at address ℓ, and is eventually consumed when this block is logically deallocated.
- When ℓ is "pushed onto the stack" in an application of the "bind" rule, an assertion of the form $\mathit{Stackable}\ \ell\ p$ is consumed, where the choice of p is up to the user; when ℓ is later "popped off the stack", as part of the same application of the "bind" rule, this assertion reappears.

One can see that "pushing a location ℓ onto the stack" requires an assertion $\mathit{Stackable}\ \ell\ p$. Thus, this fractional assertion can be intuitively regarded as a *permission* to push ℓ onto the stack, whence the name $\mathit{Stackable}$. Because this assertion is splittable, it allows pushing ℓ onto the stack as many times as one wishes. One can also see intuitively that if the full assertion $\mathit{Stackable}\ \ell\ 1$ is at hand, then no fraction of it has been consumed, so ℓ currently is not "on the stack", that is, not an invisible root. Thus, $\mathit{Stackable}\ \ell\ 1$ serves as a witness that ℓ currently is not an invisible root. It is one of the key novel requirements of our logical deallocation rule.

The presentation of the reasoning rules of our program logic may be found in [Moine et al., 2023]. The paper also includes predicates for reasoning about function closures, which, via their environment, may hold pointers to private or shared memory blocks. Additional technical details on the reasoning rules are beyond the scope of the present manuscript. I nevertheless wish to explain the statement of the soundness theorem, stated with respect to a semantics that explicitly account for a garbage collector. The remaining of this section includes a presentation of that semantics (Section 9.5), followed with the presentation of the soundness theorem (Section 9.6).

$$\textsc{HeadAlloc}\quad \frac{\ell \notin dom(\sigma) \qquad \sigma' = [\ell := ()^n]\sigma \qquad size(\sigma') \leqslant S}{\mathsf{alloc}\ n \,/\, \sigma \longrightarrow \ell \,/\, \sigma'}$$

Other rules for $t \,/\, \sigma \longrightarrow t' \,/\, \sigma'$ are not shown.

$$\textsc{StepHead}\quad \frac{t \,/\, \sigma \longrightarrow t' \,/\, \sigma'}{t \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, \sigma'} \qquad \textsc{StepCtx}\quad \frac{t \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, \sigma'}{K[t] \,/\, \sigma \xrightarrow{\text{step}} K[t'] \,/\, \sigma'} \qquad \textsc{Edge}\quad \frac{\sigma(\ell) = \vec{w} \qquad \vec{w}(i) = \ell'}{\ell \rightsquigarrow_\sigma \ell'}$$

$$\textsc{GC}\quad \frac{dom(\sigma') = dom(\sigma) \qquad \forall \ell \in dom(\sigma').\ \begin{cases} \sigma'(\ell) = \sigma(\ell) \\ \vee\ \sigma'(\ell) = \lightning \ \wedge\ \neg\,(\exists r \in R,\ \ r \rightsquigarrow_\sigma^* \ell) \end{cases}}{R \vdash \sigma \xrightarrow{\text{gc}} \sigma'}$$

$$\textsc{RedGC}\quad \frac{locs(t) \vdash \sigma \xrightarrow{\text{gc}} \sigma'}{t \,/\, \sigma \xrightarrow{\text{step} \cup \text{gc}} t \,/\, \sigma'} \qquad \textsc{RedStep}\quad \frac{t \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, \sigma'}{t \,/\, \sigma \xrightarrow{\text{step} \cup \text{gc}} t' \,/\, \sigma'}$$

Figure 9.2: Semantics with a garbage collector

9.5 Semantics Aware of Garbage Collection

I next present our semantics that accounts for garbage collection steps. The garbage collector is viewed as a nondeterministic operation that might collect any unreachable block. It may do so at any time, that is, in between every two small-step reductions. The semantics is parameterized by a bound S on the maximal size of the heap. The small-step evaluation rule for allocation requires in its premise that the size of the extended heap does not exceed S. This bound S also appears in the final soundness theorem. To define the semantics, we introduce five judgments, whose rules appear in Figure 9.2.

The *head reduction* relation, written $t \,/\, \sigma \longrightarrow t' \,/\, \sigma'$, corresponds to the standard small-step reductions. We only show one rule, namely HeadAlloc. This rule asserts that an allocation instruction that attempts to exceed this limit cannot take a step: this is expressed by the premise $size(\sigma') \leqslant S$. The freshly allocated space is initialized with unit values, written $()^n$.

The *reduction under context* relation, written $t \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, \sigma'$, allows one head reduction step under an evaluation context K. It is defined by the rules StepHead and StepCtx, which allow performing head reduction steps and reduction steps under evaluation contexts, respectively.

The *edge* relation, written $\ell \rightsquigarrow_\sigma \ell'$, asserts the existence of an edge in the memory graph associated with σ from the block at location ℓ towards the block at location ℓ'. This relation is defined by the rule Edge. There, $\vec{w}$ denotes the contents of the fields of the block at location ℓ. The premise $\vec{w}(i) = \ell'$ indicates that the i-th field of the block stores the location ℓ'. The reflexive and transitive closure of the edge relation, written $\ell \rightsquigarrow_\sigma^* \ell'$, asserts the existence of a path in the store σ from ℓ to ℓ'. Blocks reachable from a root may not be deallocated.

The *garbage collection* relation $R \vdash \sigma \xrightarrow{\text{gc}} \sigma'$ is defined by the rule GC. It captures when it is possible for a store σ to evolve to a store σ' through a GC step that respects a set of roots R. During such a GC step, any location ℓ that is unreachable from every root $r \in R$ may be deallocated. This is reflected by setting $\sigma'(\ell)$ to the token $\lightning$, which is the marker for deallocated blocks. Other blocks remain unchanged.

The *main reduction* relation is written $t \,/\, \sigma \xrightarrow{\text{step} \cup \text{gc}} t' \,/\, \sigma'$. It is defined by the rules RedGC and RedStep, which allow performing garbage collection steps and reductions under context, respectively. In RedGC, the parameter R, which represents the set of roots that the GC must respect, is

$$\{\diamond A\} \quad (\mathit{create}\ []) \quad \left\{\lambda \ell.\ \begin{matrix} \mathit{Stack}\ []\ \ell \\ \ell \hookleftarrow \varnothing \end{matrix}\right\}$$

$$\langle\{\ell\}\rangle \begin{Bmatrix} \ulcorner |L| < C \urcorner \\ \mathit{Stack}\ L\ \ell \\ \diamond B \\ v \hookleftarrow \varnothing \end{Bmatrix} \quad (\mathit{push}\ [v; \ell]) \quad \{\lambda\_.\ \mathit{Stack}\ (v :: L)\ \ell\}$$

$$\langle\{\ell\}\rangle \{\ \mathit{Stack}\ (v :: L)\ \ell\} \quad (\mathit{pop}\ [\ell]) \quad \left\{\lambda v. \begin{matrix} \mathit{Stack}\ L\ \ell \\ \diamond B \\ v \hookleftarrow \varnothing \end{matrix}\right\}$$

$$\begin{pmatrix} \mathit{Stack}\ L\ \ell \\ \ell \hookleftarrow \varnothing \end{pmatrix} \quad \Rrightarrow^{\top}_{\varnothing} \quad \begin{pmatrix} \diamond(A + B \times |L|) \\ \circledast_{v \in L}(v \hookleftarrow \varnothing) \end{pmatrix}$$

Figure 9.3: A (slightly simplified) interface for possibly-bounded stacks

instantiated with $\mathit{locs}(t)$, the set of all locations that occur in the term t. This expresses the free variable rule (Section 9.1).

9.6 Soundness Theorem with Space Bounds

We are now ready to state the soundness theorem associated with our program logic.

A configuration $t_1 \,/\, \sigma_1$ is *final* if the term t_1 is a value. A configuration $t_1 \,/\, \sigma_1$ is *reducible* if, after the garbage collection of zero, one or several blocks, the configuration can take a proper reduction step: that is, if there exist two stores σ_1' and σ_2 and a term t_2 such that $\mathit{locs}(t_1) \vdash \sigma_1 \xrightarrow{\text{gc}} \sigma_1'$ and $t_1 \,/\, \sigma_1' \xrightarrow{\text{step}} t_2 \,/\, \sigma_2$ hold. A configuration is *stuck* if it is neither final nor reducible. A program t is *safe* if, for any execution prefix described by $t \,/\, \varnothing \xrightarrow{\text{step} \cup \text{gc}}{}^{*} t_1 \,/\, \sigma_1$, it is the case that the configuration $t_1 \,/\, \sigma_1$ is either final or reducible—therefore not stuck.

In our setting, the notion of a stuck configuration is more subtle than usual. On the one hand, the limit on the size of the heap may block allocations. On the other hand, garbage collection steps may reduce the size of the heap. Thus, a program is stuck if, no matter how much memory the GC is able to reclaim, it cannot avoid growing the heap beyond the size S. In other words, a program is stuck if its *live heap size* is about to exceed S. In the contrapositive form, if a program is safe, then its live heap size never exceeds S.

Our soundness theorem states that if a program can be verified, using our program logic, under an allowance of S space credits, then this program is safe.

Theorem 9.6.1 (Soundness with space bounds) *If $\{\diamond S\}\ t\ \{\lambda\_.\ \ulcorner \mathit{True} \urcorner\}$ holds, then t is safe.*

Therefore, if a program can be verified under S space credits, then its live heap size never exceeds S. This result holds for every S. Thus, the space bounds that are established via our program logic are indeed correct. The proof of the theorem is described at a high-level in in [Moine et al., 2023, Appendix]; further details may be found in the Coq formalization.

9.7 Case Study: Stacks of Stacks

To illustrate the ability of our program logic to establish nontrivial space bounds for practical data structures, we present specifications for three implementations of stacks, which have a common behavior, but different space usage. The first implementation demonstrates a use of our linked lists. The second implementation relies on a mutable array, and demonstrates that if one omits to overwrite an array slot when a value is popped off the stack, then a memory leak appears and the code cannot be verified. The third implementation is a generic construction of a stack as a stack

of stacks. It demonstrates modular reasoning as well as an amortized space complexity analysis that exploits rational number of space credits.

We show, in particular, how to instantiate this generic construction to obtain a linked list of fixed-capacity arrays, a.k.a. *chunked stacks*. This data structure is a practical, time-efficient and space-efficient implementation of stacks. Compared with stacks implemented using linked lists, chuncked stacks are much more compact, moreover they avoid numerous indirections in traversals. Compared with stacks implemented using vectors, chunked stacks are also much more compact, moreover they avoid disruptive resize operations.

Figure 9.3 presents a common interface for all our stacks. In the pre- and postconditions, the vertically stacked items are implicitly separated by a star symbol. Before explaining all the symbols that appear in the figure, let us point out that this interface gives a slightly simplified specification, specialized to the case where the stack holds the unique pointer onto each of its elements. The specification covering the general case, where elements might also be pointed at from outside the stack, may be found in [Moine et al., 2023, §8]. In recent work, we have also refined the specification and proofs to verify that push and pop operations indeed execute in amortized constant time.

The specification consists of three triples, for the functions $create$, $push$ and pop, and of one *ghost update*, which describes the logical deallocation of a stack. Two of the triples feature a leading annotation $\langle\, \{\ell\} \,\rangle$. At a high-level, this annotation indicates that the caller to the function keeps at hand the pointer ℓ onto the stack. It is an additional feature of our logic that saves the need to thread assertions of the form $Stackable\ \ell\ p$ throughout the proof.

In contrast, the fact that v does not appear in this leading annotation, together with the precondition $v \hookleftarrow \varnothing$ in $push$ indicates that the caller to the function is expected to transfer the *unique pointer* onto the value v to the stack. Dually, the postcondition $v \hookleftarrow \varnothing$ in pop asserts that the caller receives a *unique pointer* onto the value v extracted from the stack. Similarly, the assertion $\ell \hookleftarrow \varnothing$ in the postcondition of $create$ asserts that the location ℓ returned is a unique pointer onto the fresh stack that has just been allocated.

The representation predicate $Stack\ L\ \ell$ is standard: it asserts that at address ℓ there is a valid stack whose elements are described by the mathematical list L. This list is empty in $create$, it grows by one element in $push$, and shrinks by one element in pop. Our interface for possibly-bounded stacks is parameterized with a capacity C, which is either an integer or $+\infty$. The pure precondition $\ulcorner |L| < C \urcorner$ asserts that the stack size should not exceed C. This requirement is trivially satisfied if C is $+\infty$.

Our interface is also parameterized with two constants. First, A denotes the number of credits required to allocate an empty stack. The space credits assertion $\diamond A$ appears in the precondition of $create$. Second, B denotes the number of credits required for a push operation. The assertion $\diamond B$ appears in the precondition of $push$, which consumes space, and in the postcondition of pop, which frees space.

The ghost update operation at the bottom of Figure 9.3 describes the logical deallocation of the stack. We emphasize that the deallocation of a stack is an implicit operation at runtime: there is no code for it. Nevertheless, a deallocation lemma must be included in the stack API and must be established inside the abstraction boundary of stacks. This logical deallocation operation consumes the ownership of a unique pointer ℓ onto a stack, and the representation predicate for the stack. It returns not just the number A space credits consumed at stack creation, but also B space credits for each of the elements that remain in the deallocated stack. Thus, in total, $A + B \times |L|$ space credits can be reclaimed for a stack with contents L. Furthermore, it returns the ownership of the unique pointers associated with each of the elements remaining in that stack. These assertions may be further exploited for reclaiming the space associated with those values.

Our three implementations of stacks (whose code is not shown here) differ in their space complexity. Each of them is verified with respect to a particular instantiation of the parameters A, B, and C.

Our first implementation consists of a mutable reference to a linked list. The reference occupies 2 words (one for the header and one for the contents), an empty list occupies 1 word (we represent empty lists using an allocated block, but we plan to later refine the formalization so that empty list are represented as non-allocated constants), and each list cell occupies 3 words (for storing the header, a value, and a tail pointer). This implementation satisfies our common interface for the parameters $A = 3$, $B = 3$, and $C = +\infty$. Thus, a stack implemented as a plain linked list has infinite capacity and requires $3n + 3$ words in memory for storing n elements.

Our second implementation consists of a record where one field holds the logical size of the stack and one field holds a pointer to an array of fixed capacity T. Every unused cell in this array is filled with a unit value. This implementation satisfies our interface with creation cost $A = T+4$ (3 words for the 2-field record and its header, and $T + 1$ for the array of size T and its header), insertion cost $B = 0$, and bounded capacity $C = T$. Thus, this implementation provides stacks of bounded capacity T; such a stack requires $T + 4$ words in memory, regardless of the number n of elements that it stores, where $n \leqslant T$.

Our third implementation is generic: it is a functor that expects two implementations of stacks, say *X-stacks* and *Y-stacks*, and produces a new implementation, say *Z-stacks*. A Z-stack is implemented as a record made of (1) a nonempty *Y-stack* storing the elements at the top of the stack, (2) a *X-stack* of full *Y-stacks*, storing all the remaining elements, and (3) an optional, empty *Y-stack*, called the "spare". Keeping a spare Y-stack at hand rather than immediately deallocating the top Y-stack when it becomes empty is necessary to achieve $O(1)$ amortized time bounds. Figure 9.4 shows the code for the key operations. There, we use OCaml syntax for the presentation; we verified untyped code written in Iris' *HeapLang* language, deeply embedded in Coq.

To simplify the explanations, we assume that Y-stacks are bounded—an assumption that our formalization does not make. Let us write $X.A$ and $X.B$ and $X.C$ for the space complexity parameters of *X-stacks*, and likewise for *Y-stacks*. We formally establish that our *Z-stacks* have creation cost $A = X.A + 2 \cdot Y.A + 4$, have insertion cost $B = Y.B + (Y.A + X.B)/Y.C$, and have capacity $C = X.C \times (1 + Y.C)$. The insertion cost is of particular interest. An empty *Y-stack* is allocated and pushed on the *X-stack* only every $Y.C$ *push* operations on the *Z-stack*: this explains the fractional cost $(Y.A + X.B)/Y.C$. Obtaining this bound requires rational space credits and an amortized analysis. Moreover, it involves defining a suitable potential function and saving space credits in the definition of *Stack* for Z-stacks.

By applying the functor to our previous two implementations of stacks as arrays and stacks as linked lists, we obtain a linked list of fixed-capacity arrays, a.k.a. *chunked stacks*. Recall that the space usage of a stack satisfying our interface is $Bn+A$, where n denotes the number of elements. By suitably instantiating the parameters A and B associated with our functor, we derive from our formal analysis that a linked list of fixed-capacity arrays of capacity T require $(1+\frac{7}{T})\cdot n+(2T+15)$ words. Asymptotically, chunked stacks are thus not far from optimal in terms of space usage. For example, for $T = 128$, the asymptotic space usage is less than $1.055n$, that is, 5.5% above optimal. In comparison, the asymptotic space usage for linked lists is $3n$, and that for vectors is $4n$ (or $2n$ if pop operations are disallowed). In summary, our program logic supports the formal verification of nontrivial space bounds for a practical, state-of-the-art data structure.

```
let empty () =
  let front = Y.empty in
  { front = front;
    tail = X.empty;
    spare = front; }

(* The equality [s.spare = s.front] means "s has no spare" *)

let push v s =
  let front = s.front in
  if Y.is_full front then begin
    let newfront =
      if front == s.spare then begin
        let newfront = Y.empty() in
        s.spare <- newfront;
        newfront
      end else
        spare
      in
    s.front <- newfront;
    X.push front s.tail;
    Y.push v newfront
  end else
    Y.push v front

let pop s =
  let front = s.front in
  let x = Y.pop front in
  if Y.is_empty front then begin
    let tail = s.tail in
    if not (X.is_empty tail) then begin
      let newfront = X.pop tail in
      s.spare <- front;
      s.front <- newfront;
    end
  end;
  x
```

Figure 9.4: Functor for building a X-stack of Y-stacks, where X and Y denote two abstract stack implementations, of either bounded or unbounded capacity. Untyped code written in ML syntax.

Chapter 10

A Survey of Separation Logic for Sequential Programs

This chapter focuses on research on Separation Logic, excluding work on concurrent Separation Logic. In Section 10.1, I start by listing the ingredients that were already present in the seminal papers on Separation Logic [O'Hearn et al., 2001; Reynolds, 2002]. In Section 10.2, I try to trace the origin of every other ingredient. In Section 10.3, I give a tour of the work that involves mechanized presentations of Separation Logic. Finally, in Section 10.4, I review existing course notes on Separation Logic.

For a broader survey of Separation Logic, I 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].

The survey is, up to minor updates, what was published in my ICFP'20 paper [Charguéraud, 2020, §10]. I wish to thank once more Andrew Appel, Lars Birkedal, Adam Chlipala, Magnus Myreen, Gerwin Klein, Peter Lammich, Xavier Leroy, François Pottier, and Zhong Shao, who kindly answered my questions about historical and technical details associated with the Separation Logic literature. The chapter ends with a discussion focused on the handling of partial correctness and proofs of termination (Section 10.5).

10.1 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 2.2.2;
- the small footprint specifications for primitive state-manipulating operations,
- the definition of Mlist, stated by pattern-matching over the list structure like in Definition 2.1.2;
- the characterization of the magic wand operator via characterizations (1), (3) and (4) from Definition 2.5.1, 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 2.4.1) 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 2.1). 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 2.2.1, 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 2.1).

10.2 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 2.4.2. 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 2.5.4 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 2.5.2 appear to be a (minor) contribution of Charguéraud [2020].

Regarding while loops, the possibility to frame over the remaining iterations (Section 3.5) 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] 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.

10.3 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. [2009b] 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 (Section 3.1) [Guéneau et al., 2019a].

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, 2018b], 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 (Section 2.6) and function specifications are stated as in Lemma 2.6.5, although using syntactic sugar to make specifications resemble conventional triples. Iris is defined as a fully-affine logic, with an affine entailment. Tassarotti et al. [2017b] 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].

10.4 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. I am aware of two other mechanized Separation Logic tutorials that target a λ-calculus based language, with immutable variables and return values for terms.

The Iris tutorial by Birkedal and Bizjak [2018] presents the core ideas of Iris' concurrent Separation Logic [Krebbers et al., 2017; Jung et al., 2018b]. 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., 2009b]. 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.

I wrote an all-in-Coq course entitled *Foundations of Separation Logic* [Charguéraud, 2021]. This course is distributed as Volume 6 of the *Software Foundations Series*, edited by Benjamin C. Pierce. The current version of this course covers most of the material from Chapters 2, 3, 4 and 5 of the present manuscript. The course moreover includes two introductory chapters with verification of example programs via CFML-style proofs.

10.5 Partial Correctness and Termination

In this last section, I compare the treatment of termination in CFML and other frameworks.

Partial correctness, modalities and step-indexing—and their absence in CFML. A partial-correctness triple asserts that, under the precondition, if the term terminates, then its output satisfies the postcondition. A partial-correctness triple says nothing about termination, and says nothing about programs that diverge. Refinements of partial-correctness triples exist for *reactive* programs, which perform infinite sequence of I/O operations. For such programs, triples are

extended with means of specifying the set of valid traces of interaction.

Partial-correctness rules may be useful for reasoning about reactive programs, which do not terminate. Partial-correctness rules may also useful for reasoning about concurrent programs whose progress or termination may depend on subtle ways on the fairness of the scheduler. In that case, establishing a partial-correctness result provides a way to decouple the reasoning on functional correctness from the reasoning on progress. For some applications, a mix of partial and total-correctness may be relevant. For example, Erbsen et al.'s work [2021] formalizes the code of an embedded system, whose main routine consists of an infinite loop, but where the contents of that loop consists of a terminating procedure is entirely verified using total-correctness reasoning.

When considering partial-correctness triples, one can establish a triple for the diverging term `f 0` where `f` is defined as `let rec f x = f x`. The partial-correctness reasoning rule for recursive functions provides, for reasoning about the body, a hypothesis describing recursive calls. Concretely, to prove that `f 0` admits a particular partial-correctness behavior, it is sound to assume that any occurrence of `f 0` appearing in the body of the definition of `f` does satisfy that behavior.

Intuitively, the soundness of partial-correctness reasoning can be justified as follows: if the program terminates, then one could have established a total-correctness triple using a proof carried by induction over the length of the execution trace; and if the program diverges, then the partial-correctness triples asserts nothing, so the triple holds. Technically, partial-correctness reasoning rules can be formalized and justified by means of the *later* modality, as done e.g. in Iris [Jung et al., 2015, 2018b], where that modality is defined with respect to the *step-indexing*.

In summary, Iris provides support for reasoning about partial correctness of sequential and concurrent programs, with proofs involving modalities, based on a framework grounded on step-indexing. In contrast, CFML provides support for reasoning about total correctness of sequential programs, with proofs carried out by means of standard logical induction, without need for any modality nor any form of step-indexing.

There are situations where it might be interesting to combine Iris and CFML proof style. Indeed, a concurrent program may include numerous functions involving only sequential, terminating code. Maybe we could support reasoning about those subcomponents using totally-correctness, modality-free reasoning, and then be able to lift the results into the partial-correctness triples that may be exploited for reasoning about the parts of the code involving concurrency.

About transfinite proofs for reasoning about termination Reasoning about termination in the context of a step-indexed program logic has shown to be technically challenging, with the need to introduced generalized notion of *transfinite step-indices* and *transfinite time credits* [Spies et al., 2021]. In CFML, however, proofs of termination, including for programs involving recursive functions and while loops, are established without the need to introduce any explicit transfinite features. How is that possible? Let us try to give some intuition.

Time credits have been first introduced as an extension to CFML for establishing amortized complexity bounds [Charguéraud and Pottier, 2015; Charguéraud and Pottier, 2019]. Concretely, whereas a plain CFML proof would already establish only termination, a proof in CFML with credits would establish an explicit complexity bound. Subsequently, Mével et al. [2019] formalized time credits in Iris. Whereas Iris provides only partial correctness results, a proof in Iris with time credits provides a proof of termination with an explicit complexity bound.

Reasoning in Iris with time credits applies to all programs that *do* admit a complexity bound. Yet, there exists interesting programs that *do* terminate but *do not* admit a complexity bound. Consider the following example program: `let n = rand()in for i = 1 to n do ()done`. Assume a semantics with idealized integers, and assume that the random number generator executes in constant time, and returns an unbounded integer. (No hardware can meet these constraints, but

let us ignore that aspect, which is orthogonal to the point we are trying to illustrate.)

This program is a classic example illustrating the notion of *infinitely-branching nondeterminism*. In particular, it is mentioned in Spies et al. [2021, §5.1]. The point of this program is that it terminates, no matter the value returned by the random-number generator. This program can be trivially proved to terminate in CFML. Yet, this program does not admit any time bound. Thus, it cannot be verified by means of (original) time credits. The point of the research by Spies et al. [2021] is to develop a version of Iris with a generalized form of time credits that could allow proving the termination of *sequential* programs such as the one above. To achieve that, they introduce the notions of *transfinite step-indices* and *transfinite time credits*. So, how does CFML get away without transfinite features?

The short story is that CFML leverages Coq's logic and that the proof trees built during a CFML proof are themselves transfinite objects. Concretely, the proof of the example program above involves a premise with the statement: for any value of n, the term `for i = 1 to n do ()done` does terminate. This universal quantification involves an infinitely-branching proof term, yielding (as far as I understand) the expressive power of transfinite reasoning.

A follow-up question: for reasoning about terminating, sequential programs, what are the benefits of Iris's transfinite time credits compared with CFML (as of 2022)? One decisive argument is the support for Iris' invariants, which allow hiding pieces of internal mutable state from specifications. Spies et al. [2021] wrote the following paragraph in their related work section.

> Yoshida et al. [63] and Charguéraud [17] introduce program logics capable of handling liveness reasoning, even in higher-order stateful settings. The fundamental difference to our work is that all of these logics are not step-indexed. In this paper, we have focused on enabling liveness in a step-indexed setting, allowing us to use features like Löb induction, guarded recursion, and impredicative invariants. It was precisely the combination of these features that allowed us to prove a generic specification for `memo_rec` [(a combinator for recursive memoization)] and then instantiate it for multiple clients. To the best of our knowledge, verifying `memo_rec` generically is not possible in the above logics.

As I have shown, CFML *can* be used to reason about a memoization combinator, as well as a fixed-point combinator using a knot in the store (a.k.a. Landin's knot).[1]

Yet, because CFML does not feature Iris' style *invariants*, the internal mutable state associated with the memoization function or the fixed point combinator cannot be hidden from the client. The resulting specifications are therefore not pretty, and I did not include them in any of my publications. In other words, these two programs and their clients *can* be verified using CFML, yet only with *unsatisfying* specifications that do not hide from the client the existence of a piece of internal mutable state.

Here again, there might be interesting research directions in trying to combine the best of the features of CFML and Iris.

[1] These proofs do not appear in any of my research paper; they may be found in the folders `examples/LandinsKnot` and `examples/Memoization` of the deprecated CFML repository (https://gitlab.inria.fr/charguer/cfml).

Chapter 11

Perspectives

I next list a few directions that I would like to investigate in future work.

Combining CFML and Iris. On the one hand, CFML provides characteristic formulae, moreover with lifting, and provides customizable affinity for heap predicates. On the other hand, Iris [Jung et al., 2018b] provides modular constructions for describing ghost state, and means of reasoning about concurrent programs. A natural question to ask is: *can we combine the benefits of the two frameworks into one?* I plan to investigate this question together with several of my colleagues, who are experts in both CFML and Iris. Besides, regarding termination proofs for sequential programs, it would be interesting to investigate how proofs carried out by induction with respect to the omni-big-step judgment (Chapter 4) compare with proofs carried out with respect to *transfinite time credits* [Spies et al., 2021], and whether there are interesting connections between the two forms of proofs (Section 10.5).

Integration with formally verified compilers. An interactive framework for verifying code is only one element of the chain. Down the chain, the verified code needs to be compiled. A short-term project is to connect the semantics of CFML with that of CakeML [Kumar et al., 2014], a verified ML compiler, and with that of CompCert [Leroy, 2009], a verified C compiler. A useful addition to CFML would be to add support for reasoning about machine integers, and about floating-point programs by leveraging the Gappa tool [Boldo et al., 2009; Boldo and Melquiond, 2017]. Higher-up in the chain, we may be interested in providing a concise surface language for end-users to read and write specifications. I am a member of the Gospel project, which investigates the design of such a surface specification language with applications to OCaml. I have co-authored a preliminary paper published at FM'19 [Charguéraud et al., 2019]. Further research on Gospel is funded by an ANR (national) project, lead by François Pottier, since October 2022.

Code optimization with formal guarantees. Before the code is compiled, we may be interested in optimizing it. The cost of verifying an unoptimized code is already quite high. Verifying a manually-optimized code can be prohibitive. These observations motivated me to investigate means of verifying unoptimized code and then deriving optimized code, with formal guarantees. Concretely, the idea is to describe optimizations as code transformations that carry through program invariants, in such a way that program invariants remain available at every step for justifying the correctness of the optimizations being performed. In particular, this approach enables one to write and verify code in a modular way, and then apply optimizations across abstraction barriers. The OptiTrust project aims at producing such a programmer-guided, trustworthy, source-to-

source transformation framework. Research on OptiTrust has previously been funded by Inria, and is funded since October 2022 by an ANR project, for which I am the principal investigator.

Verification of parallel programs. My work has focused mainly on the verification of sequential programs. There is a long line of work on the development of Concurrent Separation Logic, culminating with the development of the Iris framework [Jung et al., 2018b]. Reasoning about concurrency is usually quite complex; it is not unusual to see developments with thousands of lines of Coq script for just a couple dozen of lines of code. I would be particularly interested in focusing on the verification of *parallel* programs. Parallel programs are expressed using high-level programming constructs, such as *fork-join*, *async-finish*, and *parallel-for* loops. They may also leverage concurrent data structures (viewed as black boxes), as well as well-delimited, restricted forms of concurrent programming patterns. I think that it would be particularly interesting to derive high-level reasoning principles for parallel programs. Regarding applications, I have in mind the verification of state-of-the-art multicore algorithms, such as those developed by Guy Blelloch and his colleagues [Shun et al., 2012; Anderson et al., 2022; Acar and Blelloch, 2022]. I will certainly be tempted to also prove the correctness and efficiency of the *work-efficient, parallel unordered depth-first search* algorithm that I have developed with Umut Acar and Mike Rainey and published at SuperComputing'15 [Acar et al., 2015].

Scaling up to larger programs. CFML has been used to reason so far to reason about a couple thousand lines of OCaml code. I would be interested in verifying an advanced data structure library such as Sek [Charguéraud and Pottier, 2021; Moine et al., 2022], which provides a *transient* sequence data structure, i.e., one that features both an ephemeral and a persistent interface, as well as conversion functions between the two. This library consists of nearly 10k lines of code. Beyond this particular library, a large-scale verification project would presumably involve leveraging several such libraries. The long-term question that I have in mind is: *what would it take to support the interactive verification of a hundred thousand lines of code?* In that endeavour, several critical obstacles need to be overcome.

First, we need to be able to leverage a robust, well-organized, easy-to-use, library of standard mathematics. None of the existing proof assistants provide a sufficiently-complete mathematical library for carrying out verification of common programs. Many results are available only in some proof assistants but not others—and porting results from one prover to another is a daunting task. Many other results have simply never been formalized. The design of an ideal library is certainly an enormous task. Yet, it is a necessary one.

Second, we need to overcome performance bottlenecks. In CFML, easy gains could be made by abandoning Ltac and reimplementing entailment simplifying tactics in OCaml. Coq-related bottlenecks are more challenging. For example, optimizing (the *elaboration* phase of) type-checking could be achieved by introducing caching mechanisms for typeclass resolution. As another example, optimizing user development time could be achieved by making the proof assistant more *incremental*. Yet, such optimizations have far-reaching consequences, and are tremendously hard to implement in Coq's legacy code base.

Third, we need to decrease the cost of verifying code. When verifying a program in practice, there is an incompressible effort associated with coming up with the appropriate invariants. There is also a large share of the work that consists of proving a large number of relatively "trivial" facts. These facts are trivial in the sense that we would "reasonably" expect an automated tool to verify these facts using either an appropriate decision procedure, or using brute-force with limited exploration depth. To increase the fraction of proof obligations that can be automatically discharged, we need to better integrate state-of-the-art work on *hammers*, on *SMT-solvers*,

as well as on domain-specific decision procedures. Also very important are *simplification* procedures, generalizing Coq's *autorewrite* tactic to help the user avoid numerous bookkeeping steps for normalizing expressions involving, e.g., arithmetic operators, polynomials, sets, etc.

Unlike the other research directions, the three items listed above have the specificity that they involve far too much work for a small team of researchers. Developing libraries of mathematics, high-performance theorem provers, and fast-and-robust proof automation does require a community-wide effort. I plan to contribute, in particular, by providing benchmarks that consist of example proofs and collection of proof obligations that arise from the interactive verification of data structure and algorithms. Just like the SMT-Comp benchmarks have had a great impact on the field of SMT provers, I speculate that new benchmarks could motivate and guide research and development of novel techniques for improving practical tools for interactive program verification.

Further production of teaching material. Following up on the writing of *Separation Logic Foundations*, Volume 6 of the *Software Foundations* series, I am looking forward to writing a second all-in-Coq book. This second book will focus on the *practice* of interactive program verification using Separation Logic. The book by Nipkow et al. [2021] presents formal specifications for purely functional algorithms and data structures, but it—deliberately—does not focus on the presentation of the interactive Isabelle/HOL proofs. The all-in-Coq book by Appel [2022], Volume 3 of *Software Foundations*, covers fewer data structures, but goes much more in-depth in the explanations of interactive Coq proofs.

Regarding interactive verification of imperative data structure and algorithms, the only book available to date is that by Appel et al. [2022] (Volume 5 of *Software Foundations*). This volume, shorter than the others from the series, aims at verifying idiomatic C code directly. I believe that aiming for a cleaner, ML-style language can reduce the amount of technicalities involved in the proofs, and thereby allows to go further in terms of contents. I plan to cover the verification of numerous classical imperative data structures and algorithms, a large number of which I (or my collegues) have already verified using CFML. The book will also cover specification techniques, including the treatment of first-order and higher-order iterators [Pottier, 2017; Moine et al., 2022], and of polymorphic containers that store mutable data structures—the *recursive ownership pattern* described in my CPP'16 paper [Charguéraud, 2016].

I believe that interactive program verification tools have reached—or almost reached—a sufficient degree of maturity to be usable for teaching at the undergrad level. The production of teaching material thus appears essential: *what will be the impact of interactive verification tools if there are too few engineers able to use them?*

Bibliography

Umut A. Acar and Guy E. Blelloch. Algorithms: Parallel and Sequential, may 2022. URL https://www.umut-acar.org/algorithms-book. Draft book developed for a course at Carnegie Mellon University.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. A Work-Efficient Algorithm for Parallel Unordered Depth-First Search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337236. doi: 10.1145/2807591.2807651. URL https://doi.org/10.1145/2807591.2807651.

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

Mohamad A. Akra and Louay Bazzi. On the Solution of Linear Recurrence Equations. *Comp. Opt. and Appl.*, 10(2):195–210, 1998. URL https://doi.org/10.1023/A:1018373005182.

Stephen Alstrup, Mikkel Thorup, Inge Li Gørtz, Theis Rauhe, and Uri Zwick. Union-Find with Constant Time Deletions. *ACM Transactions on Algorithms*, 11(1):6:1–6:28, 2014. URL http://doi.acm.org/10.1145/2636922.

Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis*, volume 8552 of *Lecture Notes in Computer Science*, pages 1–18. Springer, August 2014. URL http://dx.doi.org/10.1007/978-3-319-12466-7_1.

Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. The Problem-Based Benchmark Suite (PBBS), V2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 445–447, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392044. doi: 10.1145/3503221.3508422. URL https://doi.org/10.1145/3503221.3508422.

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. *Verified Functional Algorithms*, volume 3 of *Software Foundations*. Electronic textbook, 2022. URL http://softwarefoundations.cis.upenn.edu. Version 1.5.2.

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.

Andrew W. Appel, Lennart Beringer, and Qinxiang Cao. *Verifiable C*, volume 5 of *Software Foundations*. Electronic textbook, 2022. URL http://softwarefoundations.cis.upenn.edu. Version 1.2.1.

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, 2007. URL https://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/publikationen-pdfs/j25-ProgramLogisResources.pdf.

Robert Atkey. Amortised Resource Analysis with Separation Logic. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010. URL http://personal.cis.strath.ac.uk/~raa/amortised-sep-logic.pdf.

Robert Atkey. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science*, 7 (2:17), 2011. URL http://bentnib.org/amortised-sep-logic-journal.pdf.

Jeremy Avigad and Kevin Donnelly. Formalizing O Notation in Isabelle/HOL. In *International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 357–371. Springer, July 2004. URL https://www.andrew.cmu.edu/user/avigad/Papers/bigo.pdf.

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.

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, 2016. URL https://doi.org/10.1145/2756553.

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.

Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 280–293, September 2005. URL http://doi.acm.org/10.1145/1086365.1086401.

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.

Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier, 2017.

Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*, pages 59–74. Springer, 2009.

Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013. URL https://hal.inria.fr/hal-00649240.

Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A User-Friendly Library of Real Analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, March 2015. URL https://hal.inria.fr/hal-00860648.

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.

Nicolas Bourbaki. *General Topology, Chapters 1–4*. Springer, 1995. URL https://doi.org/10.1007/978-3-642-61701-0.

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.

Gilles Brassard and Paul Bratley. *Fundamentals of algorithmics*. Prentice Hall, 1996.

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, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation (PLDI)*, pages 270–281, June 2014. URL http://flint.cs.yale.edu/flint/publications/veristack.pdf.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Programming Language Design and Implementation (PLDI)*, pages 467–478, June 2015. URL https://www.cs.yale.edu/homes/hoffmann/papers/amort_imp15.pdf.

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. Program Verification Through Characteristic Formulae. In *International Conference on Functional Programming (ICFP)*, pages 321–332, September 2010. URL http://www.charguer aud.org/research/2010/cfml/main.pdf.

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 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 and François Pottier. Sek, 2021. URL https://gitlab.inria.fr/fpottier/sek/.

Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. GOSPEL - Providing OCaml with a Formal Specification Language. In *Formal Methods (FM)*, volume 11800 of *Lecture Notes in Computer Science*, pages 484–501. Springer, October 2019. URL https://hal.inria.fr/hal-02157484v2.

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. Higher-order representation predicates in separation logic. In *Certified Programs and Proofs (CPP)*, pages 3–14, January 2016. URL https://hal.inria.fr/hal-01408670.

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.

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*, 62(3): 331–365, March 2019. URL http://cambium.inria.fr/~fpottier/publis/charguer aud-pottier-uf-sltc.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 *International Conference on Functional Programming (ICFP)*, pages 79–90, September 2009a. URL http://ynot.cs.harvard.edu/papers/icfp09.pdf.

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, 2009b. Association for Computing Machinery. ISBN 9781605583327. doi: 10.1145/1596550.1596565. URL https://doi.org/10.1145/1596550.1596565.

Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In *Verified Software: Theories, Tools and Experiments*, volume 9593 of *Lecture Notes in Computer Science*, pages 94–109. Springer, July 2015. URL https://hal.inria.fr/hal-01162661.

Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *ACM Workshop on ML*, pages 37–46, October 2007. URL https://www.lri.fr/~filliatr/puf/.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11866.

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.

Karl Crary and Stephanie Weirich. Resource bound certification. In *Principles of Programming Languages (POPL)*, pages 184–198, January 2000. URL http://www.cs.cornell.edu/talc/papers/resource_bound/res.pdf.

Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Principles of Programming Languages (POPL)*, January 2008. URL http://www.cse.chalmers.se/~nad/publications/danielsson-popl2008.pdf.

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.

Manuel Eberl. Proving Divide and Conquer Complexities in Isabelle/HOL. *Journal of Automated Reasoning*, 58(4):483–508, 2017. URL https://www21.in.tum.de/~eberlm/divide_and_conquer_isabelle.pdf.

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration Verification Across Software and Hardware for a Simple Embedded System. *PLDI'21*, 2021. https://doi.org/10.1145/3453483.3454065.

Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992. URL https://www2.ccs.neu.edu/racket/pubs/tcs92-fh.pdf.

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.

Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964. URL http://doi.acm.org/10.1145/364099.364331.

Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do you have space for dessert? A verified space cost semantics for CakeML programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):204:1–204:29, 2020. URL https://doi.org/10.1145/3428272.

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.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley, 1994. URL http://www-cs-faculty.stanford.edu/~knuth/gkp.html.

Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. PhD thesis, Université de Paris, December 2019. URL https://tel.archives-ouvertes.fr/tel-02437532.

Armaël Guéneau. *Mechanized verification of the correctness and asymptotic complexity of programs : the right answer at the right time*. PhD thesis, 2019. URL http://www.theses.fr/2019UNIP7110. Thèse de doctorat dirigée par François Pottier et Arthur Charguéraud; Informatique Université Paris Cité 2019.

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, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, April 2018. URL http://cambium.inria.fr/~fpottier/publis/gueneau-chargeraud-pottier-esop2018.pdf.

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 2019a. URL http://cambium.inria.fr/~fpottier/publis/gueneau-jourdan-charguéraud-pottier-2019.pdf.

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

Informatik. URL http://cambium.inria.fr/~fpottier/publis/gueneau-jourdan-charguéraud-pottier-2019.pdf.

Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen 0001, and Robert Endre Tarjan. Faster Algorithms for Incremental Topological Ordering. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 421–433. Springer, 2008. ISBN 978-3-540-70574-1.

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.

Maximilian P. L. Haslbeck and Tobias Nipkow. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10805 of *Lecture Notes in Computer Science*, pages 155–171. Springer, April 2018. URL https://www21.in.tum.de/~nipkow/pubs/tacas18.pdf.

Guanhua He, Shengchao Qin, Chenguang Luo, and Wei-Ngan Chin. Memory Usage Verification Using Hip/Sleek. In *Automated Technology for Verification and Analysis (ATVA)*, volume 5799 of *Lecture Notes in Computer Science*, pages 166–181. Springer, October 2009. URL https://dro.dur.ac.uk/6241/.

Matthew Hennessy and Robin Milner. Algebraic Laws for Nondeterminism and Concurrency. *J. ACM*, 32 (1):137–161, jan 1985. ISSN 0004-5411. doi: 10.1145/2455.2460. URL https://doi.org/10.1145/2455.2460.

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.

Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, March 2010. URL http://www.cs.yale.edu/homes/hoffmann/papers/aapoly_conference.pdf.

Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. URL http://www.dcs.ed.ac.uk/home/mxh/nordic.ps.gz.

Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, March 2006. URL https://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/publikationen-pdfs/c36-typebasedamortisedheap-space.pdf.

John E. Hopcroft. Computer Science: The Emergence of a Discipline. *Communications of the ACM*, 30(3):198–202, 1987. URL https://doi.org/10.1145/214748.214750.

Rodney R. Howell. On Asymptotic Notation with Multiple Variables. Technical Report 2007-4, Kansas State University, January 2008. URL http://people.cs.ksu.edu/~rhowell/asymptotic.pdf.

Rodney R. Howell. Algorithms: A Top-Down Approach, July 2012. URL http://people.cs.ksu.edu/~rhowell/algorithms-text/text/. Draft.

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.

Jane Street. Dune: A composable build system, 2018. URL https://dune.build/.

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.

Jacques-Henri Jourdan. New implementation of cycle detection in univ.ml, 2016. https://github.com/coq/coq/pull/90.

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:e20, 2018a. URL https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf.

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, 2018b. URL https://doi.org/10.1017/S0956796818000151.

Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Symposium on Discrete Algorithms (SODA)*, pages 19–28, January 2002. URL http://dl.acm.org/citation.cfm?id=545381.545384.

Ioannis T. Kassios and Eleftherios Kritikos. A Discipline for Program Verification Based on Backpointers and Its Use in Observational Disjointness. In *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 149–168. Springer, March 2013. URL https://doi.org/10.1007/978-3-642-37036-6_10.

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.

Dexter C. Kozen. *The design and analysis of algorithms*. Texts and Monographs in Computer Science. Springer, 1992. URL http://www.cs.cornell.edu/~kozen/papers/daa.pdf.

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.

Peter Lammich and Rene Meis. A Separation Logic Framework for Imperative HOL. *Archive of Formal Proofs*, 2012. URL http://afp.sourceforge.net/entries/Separation_Logic_Imperative_HOL.shtml.

Tom Leighton. Notes on better Master theorems for divide-and-conquer recurrences, 1996. URL http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf.

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.

Jean-Marie Madiot and François Pottier. A Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages*, 6(POPL), January 2022. URL http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf.

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.

Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming*, volume 9613 of *Lecture Notes in Computer Science*, pages 144–162. Springer, March 2016. URL https://www.eecs.northwestern.edu/~robby/pubs/papers/flops2016-mfnff.pdf.

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. Specification and Verification of a Transient Stack. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 82–99, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391825. doi: 10.1145/3497775.3503677. URL https://doi.org/10.1145/3497775.3503677.

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.chargueraud.org/research/2022/space_with_gc/space_with_gc.pdf.

J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 66–77, June 1995. URL https://www.cs.cmu.edu/~rwh/papers/gc/fpca95.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.

Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. Functional Algorithms, Verified, 2021. URL https://functional-algorithms-verified.org/.

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.

Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*, January 2011. URL http://cambium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity.pdf.

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.

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, page 68–70, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312134. doi: 10.1145/2312005.2312018. URL https://doi.org/10.1145/2312005.2312018.

Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In *Interactive Theorem Proving (ITP)*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, July 2014. URL https://www.irif.fr/~sozeau/research/publications/Universe_Polymorphism_in_Coq.pdf.

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.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2, 2019. URL https://cakeml.org/jfp19.pdf.

Robert E. Tarjan. Disjoint Set Union. Class notes, 1999. URL http://www.cs.princeton.edu/courses/archive/spr00/cs423/handout3.pdf.

Robert E. Tarjan and Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245–281, April 1984. URL http://dx.doi.org/10.1145/62.2160.

Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2): 215–225, April 1975. URL http://www.csd.uwo.ca/~eschost/Teaching/07-08/CS445a/p215-tarjan.pdf.

Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985. URL http://dx.doi.org/10.1137/0606031.

Robert Endre Tarjan. Algorithmic Design. *Communications of the ACM*, 30(3):204–212, 1987. URL https://doi.org/10.1145/214748.214752.

Joseph Tassarotti, Ralf Jung, and Robert Harper. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In Hongseok Yang, editor, *Programming Languages and Systems*, pages 909–936, Berlin, Heidelberg, 2017a. Springer Berlin Heidelberg. ISBN 978-3-662-54434-1.

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 2017b. URL https://iris-project.org/pdfs/2017-esop-refinement-final.pdf.

The Coq development team. *The Coq Proof Assistant*, 2020. URL http://coq.inria.fr/.

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.