Mechanized verification of the correctness and asymptotic complexity of programs: the right answer at the right time

Armaël Gueneau

Université de Paris
École doctorale 386 — Sciences Mathématiques de Paris Centre

Doctorat d'Informatique

Mechanized Verification of the Correctness and Asymptotic Complexity of Programs

The Right Answer at the Right Time

Armaël Guéneau

Thèse dirigée par François Pottier et Arthur Charguéraud

et soutenue le 16 décembre 2019 devant le jury composé de :

Jan Hoffmann	Assistant Professor, Carnegie Mellon University	Rapporteur
Yves Bertot	Directeur de Recherche, Inria	Rapporteur
Georges Gonthier	Directeur de Recherche, Inria	Examinateur
Sylvie Boldo	Directrice de Recherche, Inria	Examinatrice
Mihaela Sighireanu	Maître de Conférences, Université de Paris	Examinatrice
François Pottier	Directeur de Recherche, Inria	Directeur
Arthur Charguéraud	Chargé de Recherche, Inria	Co-directeur

Abstract

This dissertation is concerned with the question of formally verifying that the implementation of an algorithm is not only functionally correct (it always returns the right result), but also has the right asymptotic complexity (it reliably computes the result in the expected amount of time).

In the algorithms literature, it is standard practice to characterize the performance of an algorithm by indicating its asymptotic time complexity, typically using Landau's "big-O" notation. We first argue informally that asymptotic complexity bounds are equally useful as formal specifications, because they enable modular reasoning: a O bound abstracts over the concrete cost expression of a program, and therefore abstracts over the specifics of its implementation. We illustrate—with the help of small illustrative examples—a number of challenges with the use of the O notation, in particular in the multivariate case, that might be overlooked when reasoning informally.

We put these considerations into practice by formalizing the O notation in the Coq proof assistant, and by extending an existing program verification framework, CFML, with support for a methodology enabling robust and modular proofs of asymptotic complexity bounds. We extend the existing framework of Separation Logic with Time Credits, which allows to reason at the same time about correctness and time complexity, and introduce negative time credits. Negative time credits increase the expressiveness of the logic, and enable convenient reasoning principles as well as elegant specifications. At the level of specifications, we show how asymptotic complexity specifications using O can be integrated and composed within Separation Logic with Time Credits. Then, in order to establish such specifications, we develop a methodology that allows proofs of complexity in Separation Logic to be robust and carried out at a relatively high level of abstraction, by relying on two key elements: a mechanism for collecting and deferring constraints during the proof, and a mechanism for semi-automatically synthesizing cost expressions without loss of generality.

We demonstrate the usefulness and practicality of our approach on a number of increasingly challenging case studies. These include algorithms whose complexity analysis is relatively simple (such as binary search, which is nonetheless out of the scope of many automated complexity analysis tools) and data structures (such as Okasaki's binary random access lists). In our most challenging case study, we establish the correctness and amortized complexity of a state-of-the-art incremental cycle detection algorithm: our methodology scales up to highly non-trivial algorithms whose complexity analysis intimately depends on subtle functional invariants, and furthermore makes it possible to formally verify OCaml code which can then actually be used as part of real world programs.

Keywords Formal verification, Algorithms, Asymptotic Complexity, Interactive Proofs, Proof assistants, Program logics, Separation Logic, Resource analysis

Résumé

Cette thèse s'intéresse à la question de démontrer rigoureusement que l'implantation d'un algorithme donné est non seulement correcte (elle renvoie bien le bon résultat dans tous les cas possibles), mais aussi possède la bonne complexité asymptotique (elle calcule toujours ce résultat en le temps attendu).

Pour les chercheurs en algorithmique, caractériser la performance d'un algorithme se fait généralement en indiquant sa complexité asymptotique, notamment à l'aide de la notation "grand O" due à Landau. Nous détaillons, tout d'abord informellement, pourquoi de telles bornes de complexité asymptotiques sont également utiles en tant que spécifications formelles. La raison est que celles-ci permettent de raisonner de façon modulaire à propos du coût d'un programme : une borne en O s'abstrait de l'expression exacte du coût du programme, et par là des divers détails d'implantation. Par ailleurs, nous illustrons, à l'aide d'exemples simples, un certain nombre de difficultés liées à l'utilisation rigoureuse de la notation O, et qu'il est facile de négliger lors d'un raisonnement informel.

Ces considérations sont mises en pratique en formalisant d'une part la notation O dans l'assistant de preuve Coq, et d'autre part en étendant CFML, un outil existant dédié à la vérification de programmes, afin de permettre à l'utilisateur d'élaborer des démonstrations robustes et modulaires établissant des bornes de complexité asymptotiques. Nous étendons la logique de séparation avec crédits temps—qui permet de raisonner à la fois sur des propriétés de correction et de complexité en temps—avec la notion de crédits temps négatifs. Ces derniers augmentent l'expressivité de la logique, donnent accès à des principes de raisonnement commodes et permettent d'exprimer certaines spécifications de manière plus élégante. Au niveau des spécifications, nous montrons comment des bornes de complexité asymptotique avec O s'expriment en logique de séparation avec crédits temps. Afin d'être capable d'établir de telles spécifications, nous développons une méthodologie qui permet à l'utilisateur de développer des démonstrations qui soient à la fois robustes et menées à un niveau d'abstraction satisfaisant. Celle-ci s'appuie sur deux principes clefs : d'une part, un mécanisme permettant de collecter et remettre à plus tard certaines contraintes durant une démonstration interactive, et par ailleurs, un mécanisme permettant de synthétiser semi-automatiquement une expression de coût, et ce sans perte de généralité.

Nous démontrons l'utilité et l'efficacité de notre approche en nous attaquant à un certain nombre d'études de cas. Celles-ci comprennent des algorithmes dont l'analyse de complexité est relativement simple (par exemple, une recherche dichotomique, déjà hors de portée de la plupart des approches automatisées) et des structures de données (comme les "binary random access lists" d'Okasaki). Dans notre étude de cas la plus significative, nous établissons la correction et la complexité asymptotique d'un algorithme incrémental de détection de cycles publié récemment. Nous démontrons ainsi que notre méthodologie passe à l'échelle, permet de traiter des algorithmes complexes, donc l'analyse de complexité s'appuie sur des invariants fonctionnels subtils, et peut vérifier du code qu'il est au final possible d'utiliser au sein de programmes réellement utiles et utilisés.

Mots-clefs Preuves formelles, Algorithmes, Complexité asymptotique, Preuves interactives, Assistants de preuve, Logiques de programme, Logique de séparation

Remerciements

Mes premiers remerciements vont à mes encadrants, Arthur et François, qui ont su m'accueillir et m'accompagner tout au long de cette thèse. François a été celui qui m'a initié à la recherche, lors de mon tout premier stage de six semaines, avec les (heureuses) conséquences que l'on sait. Toujours disponible, François combine une grande curiosité intellectuelle et ouverture d'esprit avec une rigueur scientifique sans faille, deux qualités inestimables. Merci à Arthur, tout d'abord pour son implication malgré la distance géographique, et ensuite pour son optimisme et enthousiasme indéfectible. Arthur a toujours été présent pour me remonter le moral et m'aider dans les moments difficiles.

I would like to thank Jan Hoffmann and Yves Bertot for their meticulous proofreading and many comments on my manuscript. Je remercie également Georges Gonthier, Sylvie Boldo et Mihaela Sighireanu d'avoir accepté d'être membres de mon jury ; leur présence m'honore.

Merci à ceux qui ont contribué à ce qui est raconté dans ce manuscrit. Merci à Jacques-Henri Jourdan, dont la sagacité a été cruciale pour le succès de la vérification de l'algorithme de détection de cycles ; et merci à Glen Mével pour les discussions productives au sujet des O à plusieurs variables.

Cette thèse s'est nourrie des quelques années passées à l'Inria ; merci à tous ceux qui ont contribué à en faire un cadre de travail enrichissant. Merci à tous ceux de la "famille Gallium" étendue, aux conversations au coin café diverses et toujours intéressantes. Merci à Jonathan et Tahina pour m'avoir donné un aperçu de la belle vie le long de la côte nord-ouest des États-Unis. And thank you to my friendly German colleagues, Max, Simon and Manuel, for your enthusiasm, the time in Munich, and all the interesting discussions. Pour finir, merci au personnel administratif, assistants et assistantes d'équipe qui nous fournissent une précieuse tranquillité d'esprit.

Merci à ceux qui, en aval, ont permis que cette thèse advienne un jour. Merci à mes enseignants. Et merci plus particulièrement à Gabriel Scherer, de qui j'ai appris—alors lycéen—qu'il était possible de faire de la recherche en informatique, et qui m'a transmis le goût pour la programmation fonctionnelle.

Pour finir, merci beaucoup à tous celles et ceux qui m'ont entouré (humains et félins), mes amis, et ma famille.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Analysis of Execution Time . . . 3
- 1.3 Formalizing Asymptotic Complexity . . . 4
- 1.4 Time Credits . . . 5
- 1.5 Challenges . . . 6
- 1.6 Overview . . . 7
- 1.7 Applications and Case studies . . . 9
- 1.8 Research and Publications . . . 10

2 Background **12**
- 2.1 Interactive Proofs of Programs using Separation Logic . . . 12
  - 2.1.1 Example: Program Specifications using Separation Logic . . . 12
  - 2.1.2 Separation Logic: Semantics and Reasoning Rules . . . 16
  - 2.1.3 Proof Framework: Characteristic Formulae . . . 18
  - 2.1.4 CFML in Action: Characteristic Formulae in Practice . . . 20
- 2.2 Separation Logic with Time Credits . . . 26
  - 2.2.1 A Minimal Extension of Separation Logic . . . 27
  - 2.2.2 Example: Proving the Complexity of a Program . . . 30

3 Challenges in Reasoning with the O Notation **33**
- 3.1 A Flawed Proof of a O Bound for a Recursive Function . . . 33
- 3.2 A Flawed Proof of a O Bound in the Presence of Multiple Variables . . . 34
- 3.3 A Flawed Proof of a O Bound for a `for`-loop . . . 35
- 3.4 On Local Reasoning in the Presence of O . . . 36

4 Towards Practical Specifications for the Complexity of Programs **38**
- 4.1 Challenges with respect to Modularity . . . 39
  - 4.1.1 Specifying the Complexity of a Program, Formally . . . 39
  - 4.1.2 Modularity in Action: Composing Specifications . . . 41
  - 4.1.3 The Case of Cost Functions with Several Parameters . . . 43
- 4.2 Challenges with Abstract Cost Specifications . . . 45
  - 4.2.1 Why Cost Functions Must Be Monotonic . . . 46
  - 4.2.2 Why Cost Functions Must Be Nonnegative . . . 47
  - 4.2.3 Proving that Cost Functions are Monotonic and Nonnegative . . . 48
- 4.3 Summary: Constraints and Design Choices . . . 50

5 Formalizing the O Notation **52**
- 5.1 Domination Relation . . . 52
- 5.2 Filters . . . 53
  - 5.2.1 Definition . . . 53

5.2.2 Examples of Filters . . . 54
5.3 Properties of the Domination Relation . . . 57
5.3.1 Summation Lemmas . . . 58
5.3.2 Composition and Specialization . . . 61
5.4 Practical Multivariate O . . . 62
5.5 Tactics and Automation . . . 65

6 Program Logic for Asymptotic Complexity Reasoning 70
6.1 Example: Binary Search, Revisited . . . 70
6.2 Specifications with Asymptotic Complexity Claims . . . 74
6.3 Separation Logic with Possibly Negative Time Credits . . . 78
6.3.1 Motivation . . . 78
6.3.2 Model and Reasoning Rules . . . 82
6.3.3 Perspectives on Partial Correctness . . . 84
6.4 Proof Techniques for Credit Inference . . . 86
6.4.1 No Inference: Manual Splitting of Credits . . . 88
6.4.2 Using the Subtraction Rule . . . 90
6.4.3 Using Coq Evars as Placeholders . . . 93
6.4.4 Collecting Debts . . . 99
6.4.5 Summary . . . 101
6.5 Synthesizing and Solving Recurrence Equations . . . 102
6.6 More on Automation . . . 106

7 Example Gallery 110
7.1 Binary Search . . . 110
7.2 Dependent Nested Loops . . . 111
7.3 A Loop Whose Body has Exponential Cost . . . 111
7.4 Bellman-Ford's Algorithm . . . 112
7.5 Union-Find . . . 113
7.6 Higher-Order Interruptible Iterator . . . 113
7.7 Binary Random Access Lists . . . 116
7.7.1 Implementation . . . 116
7.7.2 Functional Invariants . . . 119
7.7.3 Complexity Analysis . . . 119

8 Case Study: A State-of-the-art Incremental Cycle Detection Algorithm 125
8.1 Specification of the Algorithm . . . 126
8.2 Overview of the Algorithm . . . 129
8.3 Informal Complexity Analysis . . . 131
8.4 Implementation . . . 134
8.4.1 Underlying Graph Interface . . . 134
8.4.2 Algorithm . . . 137
8.5 Data Structure Invariants . . . 140
8.5.1 Low-level Data Structure . . . 140
8.5.2 Functional Invariant . . . 140

8.5.3 Potential . . . 141
8.5.4 Advertised Cost . . . 142
8.6 Specifications for the Algorithm's Main Functions . . . 143
8.7 A Taste of the Formal Complexity Proof . . . 145
8.8 Future work . . . 154

9 Related Work **156**
9.1 Formalization of the O notation . . . 156
9.2 Approaches to Time Complexity Analysis . . . 158
9.2.1 A first intuition: ticking translation . . . 158
9.2.2 Separation Logic with Time Credits . . . 159
9.2.3 Hoare-style program logics . . . 160
9.2.4 Monadic shallow embeddings . . . 162
9.2.5 Extraction of recurrence relations . . . 163
9.2.6 Type-system based approaches . . . 164
9.2.7 Detection of performance anomalies . . . 166
9.3 Complexity-preserving compilation . . . 166
9.4 Formal Verification of Graph Algorithms . . . 167

10 Conclusion **168**

Chapter 1

Introduction

1.1 Motivation

A good algorithm must be correct. Yet, to err is human: algorithm designers and algorithm implementors sometimes make mistakes. Writing a program that appears to work is not so difficult; producing a fully-correct program has shown to be astonishingly hard. It is enough to get a single character wrong among millions lines of code to obtain a buggy program. When translating the description of a complex algorithm into executable code, a slight misunderstanding of the algorithm's design can lead to an incorrect implementation.

Although testing can detect mistakes, it cannot in general prove their absence. Thus, when high reliability is desired, algorithms should ideally be verified. A "verified algorithm" traditionally means an algorithm whose correctness has been verified. It is a package of three components: (1) an implementation of the algorithm, expressed in a programming language, describing the steps by which the algorithm computes a result; (2) a specification, expressed in a mathematical language, describing what result the algorithm is expected to compute; (3) a proof that the algorithm never crashes, always terminates, and always produces a result that the specification permits. All three components must be machine-readable, and the proof must be machine-checked.

However, a good algorithm must not just be correct: it must also be fast, and reliably so. Many algorithmic problems admit a simple, inefficient solution. Therefore, the art and science of algorithm design is chiefly concerned with imagining more efficient algorithms, which often are more involved as well. Due to their increased sophistication, these algorithms are natural candidates for verification. Furthermore, because the very reason for the existence of these algorithms is their alleged efficiency, not only their correctness, but also their complexity, should arguably be verified.

The goal of such an improved verification process is to get rid of both correctness bugs and complexity bugs—where the performance of a program is much poorer than expected in some scenarios. A complexity bug can have serious security implications, as it may be used to mount denial-of-service attacks. A well-known example involves the implementation of hash tables used (at the time the attack was discovered) by many web applications. In the worst case, a deterministic hash table has linear time complexity: it is just as inefficient as a linked list. This can be turned into an effective denial-of-service attack [cry]. Additionally, complexity bugs are typically hard to find through mere testing. Some have been detected during ordinary verification, as shown by Filliâtre and Letouzey [FL04], who find a violation of the balancing invariant in the balanced search tree implementation of OCaml's standard library, which could in principle lead to degraded performance. (This bug had remained previously undiscovered for many years, despite intensive use of the library.)

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

Figure 1.1: A flawed binary search. This code is provably correct and terminating, yet exhibits linear (instead of logarithmic) time complexity for some input parameters.

Nevertheless, ordinary verification alone cannot guarantee the absence of complexity bugs. As a practical example, consider the binary search implementation of Figure 1.1. Virtually every modern software verification tool allows proving that this OCaml code (or analogous code, expressed in another programming language) satisfies the specification of a binary search and terminates on all valid inputs. This code might pass a lightweight testing process, as some search queries will be answered very quickly, even if the array is very large. Yet, a more thorough testing process would reveal a serious issue: searching for a value that is stored in the second half of the range $[i, j)$ takes linear time. If such adverse queries are iterated millions of times over a large array containing millions of cells, termination could take months instead of a few seconds as expected. A recent line of research [NKP18, PZKJ17, WCF+18, LPSS18] is devoted to applying fuzzing techniques to find performance issues. This reveals a growing interest in the question of finding complexity bugs, and shows that this is still a challenging problem as of today.

The idea of providing a performance guarantee as part of a (sub-)program's specification goes back quite far in time. Wegbreit [Weg75] writes: "A possible prospect may be to proceed by analogy with program verification: to allow the addition to the program of performance specifications by the programmer, which the system then checks for consistency." In this work, our goal is not to establish bounds on the physical worst-case execution time, for reasons that we explain next (§1.2). Instead, we wish to specify an algorithm's performance using asymptotic complexity bounds, such as $O(n)$, and we wish to reproduce in our proofs the associated reasoning principles, as closely as possible. We are consequently interested in the "big-O notation", which denotes both a well-defined mathematical notion, and an informal method for analyzing the execution time of pseudo-code programs. This method is used in practice to reason about the complexity of algorithms. Informal paper proofs make use of and often abuse this notation, which includes an existential quantification on the multiplicative constant, and often, on the function subject to the asymptotic bound. In a paper proof, it is customary to leave as implicit the level at which such a quantification appears in the overall proof. Checking that such an informal approach is sound generally requires a more detailed inspection. This presents an additional challenge when trying to match a concrete implementation against its high-level complexity specification. Not only can a mistake be made while writing the code, but the proof of the algorithm's asymptotic complexity might be incorrect as well.

Finally, a good algorithm implementation must be reusable. Similarly, the specification and proof of correctness of a verified algorithm must be modular and reusable. The more effort we invest to make sure that a piece of software is correct, the more reusable we would like it to be.

The matter of this dissertation is the development of an approach for the modular verification of both the correctness and asymptotic complexity of programs. Given an algorithm, we wish to be able to relate a concrete implementation with a specification ensuring that it always returns the expected result, and that it does so in the expected amount of time. Implementation and specification must be related by a machine-checked proof, which should convey as closely as possible the high-level intuition behind it, for instance through the use of the well-known O notation.

1.2 Analysis of Execution Time

Following standard practice in the algorithms literature [Hop87, Tar87], we study the complexity of an algorithm based on an abstract cost model, as opposed to physical worst-case execution time. While bounds on physical execution time are of interest in real-time applications, they are difficult to establish and highly dependent on the compiler, the runtime system, and the hardware (as one must account for data and instruction caches, branch prediction, etc.). In contrast, an abstract cost model allows reasoning at the level of source code. Furthermore, on top of such an abstract cost model, we wish to establish *asymptotic* complexity bounds, such as $O(n)$, as opposed to concrete bounds, such as $3n + 5$.

In 1987, Tarjan [Tar87] writes:

> We chose to ignore constant factors in running time, so that our measure could be independent of any machine model and of the details of any algorithm implementation.

The same year, Hopcroft [Hop87] writes:

> I set out to demonstrate that [...] 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.

We work with a simple model of program execution, where certain operations, such as calling a function or entering a loop body, cost one unit of time, and every other operation costs nothing. Although one could assign a nonzero cost to each primitive operation, that would make no difference in the end: an asymptotic complexity bound is independent of the costs assigned to the primitive operations, and equally importantly, is robust in the face of minor changes in the implementation.

We argue that the use of asymptotic bounds is necessary for complexity analysis to be applicable at scale. At a superficial level, it reduces clutter in specifications and

proofs: $O(mn)$ is more compact and readable than $3mn + 2n \log n + 5n + 3m + 2$. At a deeper level, it is crucial for stating modular specifications, which hide the details of a particular implementation. Exposing the fact that a binary search algorithm costs $3 \log n + 4$ is undesirable: if a tiny modification of the implementation changes this cost to $3 \log n + 5$, then all direct and indirect clients of the algorithm must be updated, which is intolerable. Finally, recall that concrete bounds such as $3 \log n + 4$ refer to a source-level cost semantics. In some cases, a precise cost measure at the source-level is indeed meaningful (e.g. when measuring "gas" usage in smart contracts). However, when the source-level cost semantics is only related to concrete execution time up to an unknown hardware- and compiler-dependent constant factor, then for most practical purposes, no critical information is lost when concrete bounds such as $3 \log n + 4$ are replaced with asymptotic bounds such as $O(\log n)$.

1.3 Formalizing Asymptotic Complexity

There exists a wide variety of program verification techniques that can be used to establish that a given program satisfies its specification. Because this problem is undecidable in full generality, program verification approaches must include a way for the programmer to guide the verification process. In this thesis, we work in the setting of an interactive verification framework where the user provides information to the verification tool by using an interactive theorem prover. An interactive theorem prover (or proof assistant) allows one to state a theorem and to prove it interactively. In an interactive proof, one writes a sequence of statements or proof steps, while the proof assistant checks that each of these steps is legitimate. Interactive theorem provers have been successfully applied to formalize significant bodies of mathematical theories and verify the correctness of large programs.

While there is a fairly rich literature on automatically inferring complexity bounds, the idea of interactively verifying programmer-supplied bounds seems to have received relatively little attention. We consider the verification of the complexity of a program as an *extension* of the verification of its functional correctness. We believe that if one is willing to verify the functional correctness of a program (which we are!), then the extra effort involved in verifying its complexity often ought to be rather small. Furthermore, the complexity argument often relies on properties and invariants that are established during the proof of functional correctness: for instance, insertion in a balanced binary search tree has complexity $O(\log n)$ because the tree is balanced. This means that there is synergy between the two efforts. It also means that attempting to verify a program complexity without proving its correctness is doomed to failure in the general case. Tools for automatic complexity inference can produce complexity bounds, but usually have limited expressive power, both because they are automated and because they do not have access to the invariants that are explicit in a proof of functional correctness.

There are several approaches to program verification that could fit such a goal. One approach would be to consider tools that focus on automated reasoning (such as Why3 [FP13] or KeY [ABB+16]), which take as input a program annotated with specifications and invariants, and produce proof obligations that are delegated to an automated solver. Another approach is to leverage techniques based on interactive

proof assistants, which increased control and expressive power (but less automation). We follow the second approach, and base our work on the CFML tool [Cha10a, Cha13] introduced by Charguéraud for interactively verifying the functional correctness of OCaml programs. CFML is based on a representation of Separation Logic [Rey02] in the logic of Coq (technically speaking, it is a shallow embedding), and enables reasoning about higher-order heap-manipulating programs. CFML being embedded into the Coq proof assistant allows the full power of Coq to be used as its ambient logic.

There is no intrinsic limitation in the kind of asymptotic bounds that we can establish, because we work in an interactive setting. This is in contrast with automated complexity inference tools. In other words, we place emphasis on expressiveness, as opposed to automation. RAML [HDW17] is a state-of-the-art tool that focuses on automatic complexity inference. It is impressively able to infer amortized complexity bounds for OCaml programs in a press-button fashion, without requiring any input from the user. RAML supports (and is limited to) inferring "polynomial bounds that depend on the sizes of inductive data structures [...]. [It] can derive bounds for programs with exceptions, references and arrays. However, [...] it cannot derive bounds that depend on the sizes of data structures that are stored in arrays or references". This excludes for instance inferring a logarithmic bound for the complexity of binary search. In contrast, we support only a limited form of complexity inference, but we support establishing arbitrarily sophisticated complexity bounds—as long as the user is able to express the required mathematical analysis in the proof assistant.

1.4 Time Credits

In prior work [CP17b], Charguéraud and Pottier present a method for verifying that a program satisfies a specification that includes an explicit bound on the program's worst-case amortized time complexity. They use a marriage of Separation Logic with a simple notion of *time credit*, where the assertion \$1 represents a permission to perform one step of computation (say, calling a function, or entering a loop body) and is consumed when exercised. The assertion $\$n$ is a separating conjunction of n such time credits. This idea can be traced back to Atkey [Atk11], who introduces time credits in the setting of Separation Logic, and the work of Pilkiewicz and Pottier [PP11], who see them as linear capabilities in a type-and-capability system. The idea of considering time credits as a Separation Logic resource is particularly attractive because it allows (1) reasoning about functional correctness and time complexity together, (2) dealing with dynamic memory allocation, mutation, and sharing, and (3) carrying out non-trivial time complexity analyses, including amortized analyses. Charguéraud and Pottier implement Separation Logic with Time Credits as an extension of CFML, and use it to verify the correctness and time complexity of an OCaml implementation of the Union-Find data structure [CP17b]. However, their specifications involve *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 $f \in O(\alpha(n))$ (whose definition is not revealed), calling *find* requires and consumes $\$f(n)$.

The present thesis is a continuation of this pre-existing line of work. We work in the setting of Separation Logic with Time Credits, and use the corresponding extension of CFML as our verification framework. Our contribution is to show how in that setting one can establish asymptotic bounds for the complexity of programs instead of concrete ones, *in a modular way*. This presents several challenges, that we detail next.

1.5 Challenges

One might wonder whether establishing asymptotic bounds is simply a matter of using concrete bounds and re-packaging specifications after the fact by using asymptotic bounds. Once one has established that the complexity of *find* is bounded by $2\alpha(n) + 4$, it is indeed very easy to prove that it also admits the asymptotic bound $O(\alpha(n))$. However, this would not constitute a scalable proof methodology. It would mean that one has to reason with concrete cost expressions until the very end of the verification process, which would be very much inconvenient, not modular, and would not correspond to the standard practice where one reasons in term of asymptotic bounds. Additionally, as we have argued earlier, asymptotic bounds are required for specifications to be modular. After equipping a program specification with an asymptotic complexity bound, modular reasoning dictates that a client of this specification must rely only on the asymptotic bound exposed by the specification, and not on the concrete cost expression, which is abstracted away. One must work either directly at the level of Os or with abstract constants. Therefore, a challenge—studied more closely in Chapter 4—is that we need reasoning principles that not only allow establishing asymptotic bounds, but allow doing so in a modular fashion, where one can establish an asymptotic bound for an auxiliary function, and rely on it as part of the verification of a larger function.

In fact, reasoning and working with asymptotic complexity bounds is not as simple as one might hope. As demonstrated by several examples in Chapter 3, typical paper proofs using the O notation rely on informal reasoning principles which can easily be abused to prove a contradiction. Of course, using a proof assistant steers us clear of this danger, but implies that our proofs cannot be quite as simple and perhaps cannot have quite the same structure as their paper counterparts.

One challenge is to understand how asymptotic bounds involving several parameters should be handled. The complexity of many programs is best described as a function of several parameters. For instance, the complexity of a graph algorithm might depend on both the number of vertices and the number of edges of the input graph. The complexity of a function that concatenates two arrays (e.g. `Array.append`) will typically depend on the sizes of both input arrays. In paper proofs, it is customary to consider that multivariate asymptotic bounds hold when all of their parameters are large enough. However, it is common to call a program with multiple parameters and fix some inputs to be constants (or of bounded size). For instance, one might call "`Array.append [|1;2;3|] x`" or "`Array.append y [|'a';'b'|]`" as part of a larger program. This means that an asymptotic bound for `Array.append` previously established assuming that all parameters are "large enough" is not applicable in either case—therefore preventing modular reasoning.

An appropriate notion of multivariate O should instead allow reasoning about the complexity of a program even in the case where some arguments are fixed to constants.

Finally, a key challenge that we run against is the handling of existential quantifiers. According to what was said earlier, the specification of a sorting algorithm, say *mergesort*, should be, roughly: "there exists a cost function $f \in O(\lambda n.n \log n)$ such that *mergesort* requires $\$f(n)$, where n is the length of the input list." Therefore, the very first step in a naïve proof of *mergesort* must be to exhibit a witness for f, that is, a concrete cost function. An appropriate witness might be $\lambda n.2n \log n$, or $\lambda n.n \log n + 3$, who knows? This information is not available up front, at the very *beginning* of the proof; it becomes available only *during* the proof, as we examine the code of *mergesort*, step by step. It is not reasonable to expect the human user to guess such a witness. Instead, it seems desirable to delay the production of the witness and to *synthesize* a cost expression as the proof progresses. Furthermore, we wish to do so without loss of generality: the user must be able to step in and provide external insight or additional functional invariants whenever necessary. A challenge is to achieve a system in which automated synthesis and manual reasoning can be interleaved, without requiring the user to manually input verbose cost expressions.

1.6 Overview

Chapter 2 gives an introduction to Separation Logic, CFML, and their extension to Time Credits. The challenges that we exposed in the previous section are discussed in further details in Chapters 3 and 4.

Chapter 3 presents examples where an informal use of the O notation can be abused to prove a contradiction. These examples are instances where, in a formal setting, one needs to restrict these reasoning principles so that they remain sound. In particular, this chapter illustrates that it is hopeless to expect to be able to always reason about program complexity by composing asymptotic bounds "from the bottom up". In the case of a recursive program, one cannot prove a O bound directly "by induction". Instead, one must first show that "a cost function for the program satisfies some recursive equations"; then deduce an asymptotic bound on the solutions of these equations (for instance by using the Master Theorem [CLRS09]).

In Chapter 4, we ground the discussion in Separation Logic with Time Credits, and focus on the question of what a specification with asymptotic bounds should look like in that setting; how one would prove such specifications; and how one would use these as part of a larger proof. In particular, we show that abstracting over the concrete cost of a function by exposing only an asymptotic bound is in fact not sufficient to enable modular reasoning. In practice, it seems crucial to also expose that a cost function is nonnegative and monotonic.

The scene being set, Chapters 5 and 6 present our contributions from a methodological perspective. We give a summary of these technical contributions just next, in the rest of the section.

We apply these contributions to the methodology of formal proofs of complexity to the verification of actual programs. These include small and medium sized algorithms (Chapter 7), as well as our main challenging case study, the verification of a state-of-the-

art incremental cycle detection algorithm (Chapter 8). We describe these in the next section.

Let us first list the main technical contributions of this thesis.

Formal definition of O We formalize O as a binary *domination* relation between functions of type $A \to \mathbb{Z}$, where the type A is chosen by the user. Functions of several variables are covered by instantiating A with a product type. We contend that, in order to define what it means for $a \in A$ to "grow large", or "tend towards infinity", the type A must be equipped with a filter [Bou95], that is, a quantifier $\mathbb{U}a.P$. (Eberl [Ebe17] does so as well.) We propose a library of lemmas and tactics that can prove nonnegativeness, monotonicity, and domination assertions (Chapter 5).

In the case of programs involving straight-line code and loops, these lemmas allow establishing an asymptotic complexity bound for a complete program as the composition of the asymptotic bound of each individual operation. In particular, we provide lemmas that allow reasoning on the complexity of `for`-loops. Given an asymptotic bound on the body of a `for`-loop, it seems natural to bound the cost of the overall loop by the sum of the bound on its body. However, some restrictions apply, as we illustrate in Chapter 3. The summation lemmas that we present in Section 5.3.1 make these restrictions precise.

Filters for multivariate O We investigate the choice of a filter on a product type, so as to obtain a suitable notion of domination between functions with multiple parameters. Specifically, we look for a notion of multivariate O that (1) supports reasoning about the complexity of partially applied functions, and (2) allows deriving succinct complexity bounds (i.e. allow simplifying $O(mn + 1)$ into $O(mn)$). At the moment, we conclude that the most versatile solution is to reason on a case-by-case basis with respect to a notion of "strongest filter" such that some simplifications can be performed in the bound (Section 5.4).

Formal program specifications with asymptotic bounds We propose a standard style of writing specifications, in the setting of the CFML program verification framework, so that they integrate asymptotic time complexity claims (§6.2). We define a predicate, `specO`, which imposes this style and incorporates a few important technical decisions, such as the fact that every cost function must be nonnegative and nondecreasing (as argued in Section 4.2).

Separation Logic with Possibly Negative Time Credits We switch from $\mathbb{N}$ to $\mathbb{Z}$ in our accounting of Time Credits, and explain why this leads to a significant decrease in the number of proof obligations. In previous work [CP17b, GCP18], Time Credits are represented as elements of $\mathbb{N}$. In this approach, at each operation of (say) unit cost in the code, one must prove that the number of execution steps performed so far is less than the number of steps advertised in the specification. This proof obligation arises because, in $\mathbb{N}$, the equality $m + (n - m) = n$ holds if and only if $m \leq n$ holds. In contrast, in $\mathbb{Z}$, this equality holds unconditionally. Although natural numbers may seem to be perfectly adequate for counting execution steps, it turns out that representing costs as elements of $\mathbb{Z}$ can dramatically decrease the number of proof obligations (Section 6.3). Indeed, one

must then verify just once, at the end of a function body, that the actual cost is less than or equal to the advertised cost. The switch from $\mathbb{N}$ to $\mathbb{Z}$ requires a modification of the underlying Separation Logic, for which we provide a machine-checked soundness proof.

Local cost inference We propose a methodology, supported by a collection of Coq tactics, to prove program specifications with complexity bounds (Section 6.4). Our tactics help gradually synthesize cost expressions for straight-line code and conditionals, and help construct the recurrence equations involved in the analysis of recursive functions, while delaying their resolution. We present two main variants of this approach, one that relies on Coq metavariables (Section 6.4.3), and one based on our extension of Separation Logic with Time Credits and Debts and some custom reasoning rules.

We develop a mechanism to assist in the analysis of recursive functions: the user provides only a pattern for the cost function, and our system automatically gathers constraints over the free variables of that pattern (Section 6.5); the constraints may then be resolved after the analysis of the code per-se, possibly with the help of an external decision procedure (Section 6.6).

1.7 Applications and Case studies

One objective of the present thesis is to demonstrate that our techniques make it possible to mechanize challenging complexity analyses, and that such analyses can be carried out based on actual source code, as opposed to pseudo-code or an idealized mathematical model. For that purpose, we verify a number of case studies, from the scale of small illustrative examples to a larger challenging state-of-the-art algorithm, whose verification is one of our major contributions. As a byproduct of this work we provide standalone formally verified OCaml libraries, which are readily usable as part of larger OCaml developments.

We present several classic examples of complexity analyses in Chapter 7, including a simple loop in $O(n.2^n)$, nested loops in $O(n^3)$ and $O(nm)$, binary search in $O(\log n)$, and Union-Find in $O(\alpha(n))$. These examples mainly focus on the complexity analysis, and illustrate how we handle specific code patterns.

We verify an OCaml implementation of binary random access lists, a purely functional data structure due to Okasaki [Oka99] (Section 7.7). A binary random access list structure stores a sequence of elements, and provide efficient support for both list-like operations (adding or removing an element at the head of the sequence) and array-like operations (looking up and updating an element based on its index). All operations run in amortized time $O(\log n)$, where n denotes the number of elements stored in the structure. This medium-size case study illustrates the modular use of asymptotic complexity specifications. The implementation of several operations on the structure relies on internal auxiliary functions: these auxiliary functions are given an asymptotic complexity specification, which is then used in the proof of the main operations. Furthermore, we illustrate a situation where an auxiliary specification (involving a multivariate O bound) is established for the cost of an operation, and then specialized in order to obtain the final complexity bound.

As our main case study (Chapter 8), we formally verify the correctness and amortized complexity of an incremental cycle detection algorithm due to Bender, Fineman, Gilbert, and Tarjan [BFGT16, §2]. Using this algorithm, the amortized complexity of building a directed graph of n vertices and m edges, while incrementally ensuring that no edge insertion creates a cycle, is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. This is a significant verification effort: the design of the algorithm is subtle, and it is far from obvious by inspection of the code that it satisfies the advertised complexity bound. From a methodological perspective, this case study illustrates the use of fine-grained reasoning with time credits, involving the use of interruptible higher-order iterators (described separately in Section 7.6), and an amortization argument that relies on nontrivial functional invariants. Our formally verified implementation, which we package as a standalone OCaml library, has had some practical impact already. We recently integrated our verified algorithm as part of the implementation of the Dune build system [Str18], replacing an unverified implementation of the same algorithm. Our verified implementation is not only more trustworthy, but also appears to be up to 7 times faster than the previous one in a real-world scenario.

1.8 Research and Publications

Some of the work presented in this thesis has been published as part of two conference papers and a workshop presentation.

Research publications

- *A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification*,
  Armaël Guéneau, Arthur Charguéraud, François Pottier,
  European Symposium on Programming (ESOP), 2018.

- *Procrastination, a proof engineering technique*,
  Armaël Guéneau,
  FLoC Coq Workshop (extended abstract), 2018.

- *Formal Proof and Analysis of an Incremental Cycle Detection Algorithm*,
  Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, François Pottier,
  Interactive Theorem Proving (ITP), 2019.

In the paper that appeared at ESOP'18, we present our formal definition of O, the basis of our proof methodology, and some small examples. This paper forms the basis of Chapters 3, 4, 5, 7, and in a smaller proportion, Chapter 6. In the present thesis, these chapters have been significantly expanded compared to the original paper. Our presentation at the FLoC'18 Coq workshop introduces the `procrastination` Coq library, briefly described in Section 6.5 of this thesis. The paper that appeared at ITP'19 presents our main case study, the formalization of an incremental cycle detection algorithm, and includes most of the material appearing in Chapter 8. This paper also briefly introduces Separation Logic with Possibly Negative Time Credits, which we describe in more detail in Section 6.3 of this thesis.

The lemmas, theorems, tactics and example programs that appear in this thesis have been verified using Coq. They are distributed among three main developments, listed below.

Coq libraries and formalization

[Gué18c] The `procrastination` library, https://github.com/Armael/coq-procrastination.

[Gué18b] The `bigO` library, https://gitlab.inria.fr/agueneau/coq-bigO.

[GJCP19] Formalization of the incremental cycle detection algorithm, https://gitlab.inria.fr/agueneau/incremental-cycles.

[Cha19a] The CFML2 framework, https://gitlab.inria.fr/charguer/cfml2.

The `procrastination` library [Gué18c] provides support for interactively collecting and deferring constraints, and is used in particular for collecting recursive equations on the cost of recursive programs.

The `bigO` library [Gué18b] contains our formal definition of filters and the domination relation, as well as tactics for the synthesis of cost expressions. It includes the formalization of the small illustrative programs presented through the thesis, as well as those whose proof is detailed in Chapter 7. It depends on the `procrastination` library.

The third repository contains our verified implementation of the incremental cycle detection algorithm [GJCP19] described in Chapter 8. It depends on both the `procrastination` and `bigO` library.

The most up-to-date version of the formalization of the meta-theory of Separation Logic with Possibly Negative Time Credits is maintained as part of the CFML2 framework [Cha19a] by Arthur Charguéraud. The extension of CFML2 to Time Credits in $\mathbb{Z}$ is based on the work presented in this dissertation.

Chapter 2

Background

Separation Logic is a family of program logics pioneered by Reynolds [Rey02] that offer a natural framework for reasoning on heap-manipulating imperative programs. In this thesis, we work in a shallow-embedding of (higher-order) Separation Logic in Coq, materialized by the CFML tool [Cha10a, Cha13]. The fact that the Separation Logic is embedded into Coq conveniently allows us to use the powerful ambient logic of Coq as the meta-logic in which to express and establish functional correctness properties. Additionally, CFML is a verification framework: it provides tactics that allow the user to carry interactive proofs of OCaml programs.

In order to formally establish not just the correctness but also the complexity of a program, we work in Separation Logic with Time Credits, which extends ordinary Separation Logic with a resource (a "time credit") that represents a right to perform one step of computation. We follow the work of Charguéraud and Pottier [CP17b] who extend CFML with support for Time Credits. The resulting framework forms the basis for their verification of the asymptotic complexity of a Union-Find data structure, and we base our work upon this framework as well.

In this chapter, we illustrate the use of Separation Logic to specify and prove correct simple OCaml programs, including the number of computation steps taken. In a first section, we focus on ordinary Separation Logic and functional correctness properties. Then, we show how one can use Separation Logic with Time Credits to establish both the functional correctness and the (possibly amortized) complexity of a program.

This Chapter thus presents results that pre-exist the work of this thesis. The goal of the present chapter is to give the reader a taste of the logic we work with, and the state of affairs on which this work is based. For more information on CFML, the reader is referred to Charguéraud's paper [Cha13]; for more information on Separation Logic with Time Credits, we refer to Charguéraud and Pottier's article on their verification of Union-Find [CP17b], as well as Mével, Jourdan and Pottier's paper describing their implementation of Time Credits in Iris [MJP19].

2.1 Interactive Proofs of Programs using Separation Logic

2.1.1 Example: Program Specifications using Separation Logic

Let us start by illustrating how *program specifications* are expressed using Separation Logic. At the end of the day, the result of a verification effort typically materializes as the proof of a specification: in order to evaluate the work, one (only) needs to be able to read and understand the specification, in order to appreciate what exactly has been proved.

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

Figure 2.1: An OCaml implementation of binary search.

As a first example, let us consider an OCaml implementation of the binary search algorithm. The code appears in Figure 2.1, including an informal specification in comments. The `bsearch` program takes as input an array of integers `a` (which is assumed to be sorted), a value `v` which is the target of the search, and lower and upper bounds `i` and `j`. If `v` can be found between indices `i` (included) and `j` (excluded), then `bsearch` returns its index in the array; otherwise, it returns -1.

The Separation Logic specification for `bsearch` is as follows:

$$\begin{array}{l}\forall a\ xs\ v\ i\ j.\\ 0 \le i \le |xs| \wedge 0 \le j \le |xs| \wedge \text{sorted}\ xs \implies\\ \quad \{a \rightsquigarrow \mathsf{Array}\ xs\}\ \texttt{bsearch}\ a\ v\ i\ j\ \{\lambda k.\ a \rightsquigarrow \mathsf{Array}\ xs \star [\text{bsearch\_correct}\ xs\ v\ i\ j\ k]\}.\end{array}$$

The statement consists of a Hoare triple ($\{H\}\ t\ \{Q\}$), after quantification (in the logic) over suitable arguments (the array location a and integers v, i and j) as well as the logical model of the array xs (a list of integers). Notice that we associate Coq integers (with v, i, j) with OCaml integers. This is not an abuse of notation but a feature of CFML, which reflects pure OCaml values directly as corresponding values in Coq (more about that in Section 2.1.4). We require the indices i and j to be in bounds, and the contents of the array xs to be sorted. The Hoare triple which forms the conclusion of the statement relates an OCaml expression (here, a call to `bsearch`, "`bsearch` $a\ v\ i\ j$") with a pre-condition and post-condition describing the state of the heap before and after the execution, respectively. Both pre and post-conditions are expressed using Separation Logic assertions; the post-condition is additionally parameterized by the value returned by the function (here, named k).

Generally speaking, Separation Logic assertions can be read in terms of ownership. For instance, the Separation Logic formula "$a \rightsquigarrow \mathsf{Array}\ xs$" asserts the unique ownership of an heap-allocated array, starting at location a, whose contents are described by the list xs. The operator $\star$ should be read as a conjunction; and $[\cdot]$ lifts an ordinary Coq proposition as a Separation Logic assertion. To express the fact that the value returned by `bsearch` is indeed correct, we rely on an auxiliary definition bsearch_correct, which expresses the functional correctness property of `bsearch`. Thereafter, we write "$xs[i]$", where xs is a Coq list and i an integer, to denote the i-th element of the list xs (or a dummy value if i is not a valid index).

Definition 2.1.1 (Functional correctness of `bsearch`)

$$\begin{array}{l} \text{bsearch\_correct } xs\ v\ i\ j\ k \triangleq \\ \quad (k = -1 \quad \wedge \quad \forall p.\ i \le p < j \implies xs[p] \neq v) \\ \vee \quad (i \le k < j \quad \wedge \quad xs[k] = v) \end{array}$$

From a high-level perspective, the specification for `bsearch` entails the following facts. Assuming that the preconditions are satisfied,

- `bsearch` will run and terminate without crashing (this is a consequence of our definition of Hoare triples, §2.1.2);
- `bsearch` will not modify the contents of the input array ("$a \rightsquigarrow \mathsf{Array}\, xs$" appears in both the pre and post-condition, with the same xs);
- `bsearch` will return a correct result, according to the bsearch_correct predicate.

A technical remark: one might wonder why the "pure" assumptions on i, j and xs are not part of the triple precondition, and instead are lifted as an implication at the meta-level (in the logic of Coq). Formally speaking, these two options are equivalent. That is, one could equivalently prove the following specification for `bsearch`:

$$\begin{array}{l} \forall a\ xs\ v\ i\ j. \\ \{a \rightsquigarrow \mathsf{Array}\, xs \star [0 \le i \le |xs| \wedge 0 \le j \le |xs| \wedge \text{sorted } xs]\} \\ \quad \texttt{bsearch}\ a\ v\ i\ j \\ \{\lambda k.\ a \rightsquigarrow \mathsf{Array}\, xs \star [\text{bsearch\_correct } xs\ v\ i\ j\ k]\}. \end{array}$$

In practice, we choose to lift pure preconditions at the meta-level as in the first specification, because it is more convenient. In particular, as a client of the specification, it saves us from manually extracting the pure facts to put them into the proof context (using rule EXTRACT-PROP).

Although `bsearch` in itself is not terribly interesting as a heap-manipulating program (it only reads from an array), it will serve later as a good example program with a non-trivial (logarithmic) complexity. Another example, perhaps more typical of introductions to Separation Logic, is a mutable queue data structure, whose main operations can be specified as follows:

$$\begin{array}{l} \{[\mathsf{True}]\}\ \texttt{create}\ ()\ \{\lambda q.\ \mathsf{Queue}\ q\ \mathsf{nil}\} \\ \{\mathsf{Queue}\ q\ L\}\ \texttt{is\_empty}\ q\ \{\lambda b.\ \mathsf{Queue}\ q\ L \star [b = \mathsf{true} \Leftrightarrow L = \mathsf{nil}]\} \\ \{\mathsf{Queue}\ q\ L\}\ \texttt{enqueue}\ v\ q\ \{\lambda().\ \mathsf{Queue}\ q\ (v :: L)\} \\ \{\mathsf{Queue}\ q\ L \star [L \neq \mathsf{nil}]\}\ \texttt{dequeue}\ q\ \{\lambda v.\ \exists L'.\ \mathsf{Queue}\ q\ L' \star [L = L' +\!\!+ v :: \mathsf{nil}]\} \end{array}$$

Note the use of "$\exists$" in the postcondition of `dequeue`, which corresponds to an existential quantification lifted to the level of Separation Logic assertions. These specifications rely on a Separation Logic predicate "$\mathsf{Queue}\ q\ L$", which asserts the ownership of a queue data structure at location q, containing values modeled by the Coq list L. Such a predicate can generally be treated as abstract by a client of the specification. The specific implementation of the Queue predicate depends on the OCaml implementation of the structure.

```
type 'a queue = {
  mutable front: 'a list;
  mutable back: 'a list;
}

let push (x: 'a) (q: 'a queue) =
  q.back <- x :: q.back

let pop (q: 'a queue) =
  match q.front with
  | [] ->
     begin match List.rev q.back with
     | [] -> assert false
     | x::f' ->
       q.front <- f';
       q.back <- [];
       x
     end
  | x::f' ->
    q.front <- f';
    x

let is_empty (q: 'a queue) =
  front = [] && back = []

let create () =
  { front = []; back = [] }
```

Figure 2.2: An OCaml implementation of mutable queues, as a pair of lists.

$$
\begin{array}{lcl}
[P] & \triangleq & \lambda h.\ h = \varnothing \wedge P \\
\mathrm{GC} & \triangleq & \lambda h.\ \mathsf{True} \\
l \hookrightarrow v & \triangleq & \lambda h.\ h = (l \mapsto v) \\
H_1 \star H_2 & \triangleq & \lambda h.\ \exists h_1\, h_2.\ h_1 \perp h_2 \wedge h = h_1 \uplus h_2 \wedge H_1\ h_1 \wedge H_2\ h_2 \\
\exists x.\ H & \triangleq & \lambda h.\ \exists x.\ H\ h
\end{array}
$$

Figure 2.3: Separation Logic assertions: syntax and interpretation.

There are several possible implementations that satisfy such a specification: a doubly linked list, a pair of lists, a dynamic array, etc. Figure 2.2 shows the code for a simple implementation that relies on a record—with mutable fields—holding two lists. The `enqueue` operation stores elements in the "back" list, in reverse order. The `dequeue` operation retrieves elements from the "front" list. If the "front" list is empty, `dequeue` reverses the "back" list to obtain a new "front" list. This implementation can be proved to satisfy the specifications above, by defining Queue as a mutable pair of two pure lists (we do not detail the proof here, it appears in [Cha19b, examples/BatchedQueue/BatchedQueue_proof.v]):

$$\mathsf{Queue}\ q\ L \triangleq \exists L_1\, L_2.\ q \rightsquigarrow \{|\mathrm{front}{=}L_1;\ \mathrm{back}{=}L_2|\} \star [L = L_1 +\!\!+ \mathrm{rev}\ L_2].$$

2.1.2 Separation Logic: Semantics and Reasoning Rules

In Separation Logic, an assertion H has type $\mathsf{Heap} \to \mathsf{Prop}$ and describes a heap fragment. Such an assertion has an ownership reading. If one knows (e.g., in a Hoare triple precondition) that H holds of a certain heap fragment, then one is the unique owner of this heap fragment. Ownership grants read-write access, while "others" have no access at all. A heap (or heap fragment) h is just a memory, that is, a finite map of locations to values: $\mathsf{Heap} \triangleq \mathsf{Memory}$. This definition is revisited in §2.2, where time credits are introduced.

Figure 2.3 shows the definition for the fundamental Separation Logic assertions. The assertion $[P]$ is true of an empty heap, provided the proposition P holds. (P has type Prop). This assertion is "pure" in that it represents pure information and no ownership. The assertion $[\,]$, a syntactic sugar for $[\mathsf{True}]$, denotes the empty heap.

The assertion GC (for "garbage collected") is true of any heap. It asserts the ownership of some heap fragment, but we know nothing about its contents. It is used in the interpretation of Separation Logic triples (Definition 2.1.2, below) to describe a heap fragment that the user has decided to discard.

The "points-to" assertion $l \hookrightarrow v$ is true of the singleton heap fragment $(l \mapsto v)$. It asserts the ownership of a unique memory cell at location l, currently containing the value v.

The separating conjunction $H_1 \star H_2$ is true of a heap h that can be split in two disjoint parts h_1 and h_2 which respectively satisfy H_1 and H_2. Its definition involves two auxiliary notions: $H_1 \perp H_2$ asserts that the heaps h_1 and h_2 have disjoint domains; $h_1 \uplus h_2$ denotes the union of two disjoint heaps. We also write $Q \star H'$ as sugar for $\lambda x.\ (Q\ x \star H')$.

Existential quantification, lifted to the level of Separation Logic assertions, is written $\exists x.\ H$.

We define entailment between assertions, written $H_1 \ \vDash\ H_2$, as follows:

$$H_1 \ \vDash\ H_2 \triangleq \forall h.\ H_1\ h \implies H_2\ h.$$

Entailment between postconditions, written $Q \ \vDash\ Q'$, is sugar for $\forall x.\ (Q\ x) \ \vDash\ (Q'\ x)$.

Ownership over an array or a record, as used in the previous section, can be defined in term of individual points-to assertions as an iterated separating conjunction (relying on the fact that memory locations are modeled by natural numbers):

$$\begin{array}{lcl} q \rightsquigarrow \{|\text{front}{=}v_1;\ \text{back}{=}v_2|\} & \triangleq & (q \hookrightarrow v_1) \star ((q+1) \hookrightarrow v_2) \\ a \rightsquigarrow \mathsf{Array}\ xs & \triangleq & \bigstar_{0 \le i < |xs|}\ (a+i) \hookrightarrow xs[i] \end{array}$$

A technical remark: the Separation Logic that we consider here is *linear*. That is, entailment does not allow discarding resources: $H \ \vDash\ [\,]$ holds if and only if the assertion H characterizes the empty heap. We say of an assertion that it is *affine* if it entails GC. The intuition is that affine assertions represent resources that can be disposed of automatically by the runtime. With the present definition of GC, every assertion is affine: $H \ \vDash\ \text{GC}$ holds of any H. Because the logic is linear, one cannot simply discard an affine assertion H at any point. One must first turn such an assertion into GC, then use the DISCARD-POST rule which allows getting rid of an GC assertion *during the proof of a Separation Logic triple.* This might seem like an unnecessary complication, but it anticipates on the fact that in the general case there might be assertions that are not affine (e.g. consider file handles that must be closed explicitly). In fact, this distinction will be important later on when we introduce possibly negative time credits (§6.3), where negative time credits are not affine.

A Separation Logic triple $\{H\}\ t\ \{Q\}$ is defined as follows. We consider *total correctness* triples: in a valid triple, the precondition must be strong enough to ensure termination.

Definition 2.1.2 (Separation Logic Triple) *A triple $\{H\}\ t\ \{Q\}$ is short for the proposition:*

$$\forall m\ H'.\ (H \star H')\ m \implies \exists v\ m'.\ \left\{ \begin{array}{l} t/m \Downarrow v/m' \\ (Q\ v \star H' \star \text{GC})\ m' \end{array} \right.$$

This semantic definition relies on the auxiliary judgment $t/m \Downarrow v/m'$, which corresponds to the big-step evaluation semantics of our programs. The judgment $t/m \Downarrow v/m'$ asserts that the evaluation of the term t in the initial memory m terminates, producing the value v and the final memory m'. We do not reproduce here the inductive rules defining this judgment, which are standard (see [CP17b]), and that we generalize in Section 2.2 to account for the cost of an evaluation.

Notice that, thanks to the universal quantification over an arbitrary heap predicate H', a triple applies to any heap which *extends* the heap described in precondition. Effectively, triples "build the frame rule (FRAME)" into their definition. This allows users to state "small-footprint" specifications, where only relevant parts of the heap are mentioned in the pre and post-condition.

$$\textsc{frame}\ \frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}} \qquad \textsc{consequence}\ \frac{H \Rrightarrow H' \qquad \{H'\}\ t\ \{Q'\} \qquad Q' \Rrightarrow Q}{\{H\}\ t\ \{Q\}} \qquad \textsc{discard-post}\ \frac{\{H\}\ t\ \{Q \star \mathrm{GC}\}}{\{H\}\ t\ \{Q\}}$$

$$\textsc{extract-prop}\ \frac{P \implies \{H\}\ t\ \{Q\}}{\{[P] \star H\}\ t\ \{Q\}} \qquad \textsc{extract-exists}\ \frac{\forall x.\ \{H\}\ t\ \{Q\}}{\{\exists x.\ H\}\ t\ \{Q\}} \qquad \textsc{val}\ \ \{[\,]\}\ v\ \{\lambda y.\ [y = v]\}$$

$$\textsc{if}\ \frac{b = \mathsf{true} \implies \{H\}\ t_1\ \{Q\} \qquad b = \mathsf{false} \implies \{H\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)\ \{Q\}} \qquad \textsc{let}\ \frac{\{H\}\ t_1\ \{Q'\} \qquad \forall x.\ \{Q'\,x\}\ t_2\ \{Q\}}{\{H\}\ (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)\ \{Q\}}$$

$$\textsc{app}\ \frac{v_1 = \mu f.\lambda x.t \qquad \{H\}\ ([v_1/f]\,[v_2/x]\,t)\ \{Q\}}{\{H\}\ (v_1\ v_2)\ \{Q\}} \qquad \textsc{ref}\ \ \{[\,]\}\ (\mathsf{ref}\ v)\ \{\lambda x.\ \exists l.\ [x = l] \star l \hookrightarrow v\}$$

$$\textsc{get}\ \ \{l \hookrightarrow v\}\ (\mathsf{get}\ l)\ \{\lambda x.\ [x = v] \star l \hookrightarrow v\} \qquad \textsc{set}\ \ \{l \hookrightarrow v'\}\ (\mathsf{set}\ (l,\ v))\ \{\lambda x.\ [x = ()] \star l \hookrightarrow v\}$$

Figure 2.4: Reasoning rules for ordinary Separation Logic

Additionally, post-conditions may only describe a fraction of the resources in the final heap, thanks to the use of GC. This validates the "garbage collection" rule DISCARD-POST which allows the user to discard resources that appear in post-condition.

The reasoning rules of Separation Logic appear in Figure 2.4. We do not detail these further, are they are standard. Our treatment of variables and functions (rules LET and APP) might seem somewhat mysterious, as it seems that we are mixing up program variables and metavariables. We direct the reader to the discussion in Section 3.4 of [CP17a] which specifically addresses this point. Each of these rules should be read as a separate lemma, whose validity is established with respect to Definition 2.1.2, leading to the following soundness theorem:

Theorem 2.1.3 (Soundness of Separation Logic) *Each of the reasoning rules in Figure 2.4 is valid with respect to the interpretation of triples given by Definition 2.1.2.*

The proof of this theorem has been machine-checked by Charguéraud and Pottier [CP17b], a modernized version of this proof also appears in CFML2 [Cha19a].

2.1.3 Proof Framework: Characteristic Formulae

This section gives only a high-level summary of how a framework based on characteristic formulae works, with a focus on CFML. For more details, we refer to Charguéraud's [Cha13] description of characteristic formulae and CFML, and the work by Guéneau, Myreen,

Kumar and Norrish [GMKN17] which shows how characteristic formulae can be implemented in the setting of the CakeML compiler and how to precisely link it to CakeML's mechanized semantics.

To prove that the implementation of an operation satisfies its specification, one needs to link concrete code with the rules of the logic (Figure 2.4), in one way or another. One possible approach would be to define the syntax of the programs directly in Coq (as a *deep embedding*), which enables directly reasoning about them. Another approach is to implement the rules of the logic in an external automated tool (such as Why3 [FP13] or KeY [ABB+16]) which mechanically applies the rules of the logic and produces verification conditions. We rely on CFML which lies somewhere in between in the design space. The CFML tool transforms an OCaml term t into a *characteristic formula*, that is, a higher-order logic formula written $[\![t]\!]$, which describes the semantics of t. More precisely, for any precondition H and postcondition Q, the proposition $[\![t]\!]\ H\ Q$ holds if and only if the Hoare triple $\{H\}\ t\ \{Q\}$ is valid. The characteristic formula $[\![t]\!]$ is automatically built from the term t by the CFML tool.

This machinery includes lifting OCaml types into Coq types, and reflecting OCaml values as Coq values. More concretely, in order to verify our `bsearch` program, we run CFML on an OCaml file `bsearch.ml`, which contains the implementation of `bsearch` (Figure 2.1). We obtain a Coq file, `bsearch_ml.v`, which contains a series of axioms. For each toplevel OCaml function, we get two axioms: one asserts that this function exists (and names it), while the other provides a means (based on this function's characteristic formula) of establishing a Hoare triple about this function. For instance, in the case of `bsearch`, the two following axioms are generated:

```
Parameter bsearch : func.
Parameter bsearch__cf :
  ∀a v i j : Z,
  ∀H : hprop,
  ∀Q : Z → hprop,
  ... (* bsearch's characteristic formula, applied to H and Q *) →
  app bsearch [a v i j] H Q.
```

The value `bsearch` is viewed in Coq as an object of abstract type `func`. OCaml integers are mirrored as integers in Coq. `hprop` is a shorthand for `Heap → Prop`: it is the type of a Separation Logic assertion.

The axiom `bsearch__cf` states that, to establish a Hoare triple of the form $\{H\}$ `bsearch` $a\ v\ i\ j\ \{Q\}$, it suffices to prove that the characteristic formula, applied to H and Q, holds (`app` is a shorthand for the Hoare triple of a function application).

In order to prove the specification for `bsearch` shown earlier, one has to apply the `bsearch__cf` axiom. We place the specification and its proof in a separate hand-written Coq file, `bsearch_proof.v`. CFML provides a library of tactics that allows us to carry out the proof of `bsearch` in a natural style, applying successively the reasoning rules of Separation Logic and proving Separation Logic entailments.

From a high-level perspective, our work is based on an *interactive* verification framework based on Separation Logic. It is not tied specifically to the verification approach used by CFML; our work would also likely apply to an approach based on a deep-embedding (e.g. Iris with time credits). However, understanding how to apply our work to automated verification tools remains future work.

2.1.4 CFML in Action: Characteristic Formulae in Practice

Let us now illustrate how the proof of `bsearch`'s specification can be carried out, using the tactics provided by CFML. In order to establish functional correctness, we rely on the following lemmas about `bsearch_correct` (Definition 2.1.1). Each lemma corresponds to one branch in the program; we do not detail their proof, as they only require standard Coq reasoning techniques.

```
Lemma bsearch_correct_out_of_bounds xs v i j :
  j ≤ i → bsearch_correct xs v i j (−1).

Lemma bsearch_correct_found xs v i j k :
  i ≤ k < j →
  v = xs[k] →
  bsearch_correct xs v i j k.

Lemma bsearch_correct_recurse_l xs v i j j' k :
  sorted xs →
  0 ≤ i →
  j ≤ length xs →
  i ≤ j' < j →
  v < xs[j'] →
  bsearch_correct xs v i j' k →
  bsearch_correct xs v i j k.

Lemma bsearch_correct_recurse_r xs v i i' j k :
  sorted xs →
  0 ≤ i →
  j ≤ length xs →
  i ≤ i' < j →
  xs[i'] < v →
  bsearch_correct xs v (i'+1) j k →
  bsearch_correct xs v i j k.
```

Remark that it is not strictly needed to state these auxiliary lemmas; one might prefer to inline their proof at their (single) use site in the proof of `bsearch_spec`. Relying on these lemmas, the complete proof of `bsearch` is shown below. This is only to give a rough idea of the size and the style of the proof: thereafter, we detail the proof step by step, and show the corresponding intermediate Coq goals.

In the specification below, the Coq notation "`app f [x y z] PRE (H) POST (Q)`" corresponds to a Hoare triple (in the specific case of a function application). It corresponds to "$\{H\}$ `f` $x\ y\ z$ $\{Q\}$" in mathematical notation. In the Coq goals that we display just next, the more general Coq notation "`PRE (H) POST (Q) CODE (c)`" appears. It should informally be understood as a Hoare triple "$\{H\}$ `c` $\{Q\}$", except that `c` stands for a *characteristic formula*, not directly an OCaml expression.

```
Lemma bsearch_spec : ∀a (xs : list int) (v : int) (i j : int),
  sorted xs → 0 ≤ i ≤ length xs → 0 ≤ j ≤ length xs →
  app bsearch [a v i j]
    PRE (a ⇝ Array xs)
    POST (fun (k:int) ⇒ a ⇝ Array xs ⋆ [ bsearch_correct xs v i j k ]).
Proof.
```

```
  introv. gen_eq n: (j−i). gen i j. induction_wf IH: (downto 0) n.
  intros i j Hn Hxs Hi Hj.
  xcf. xif. { xret. hsimpl. apply~ bsearch_correct_out_of_bounds. }
  xret; intro Hm. forwards~ Hmbound: middle_inbound Hm.
  xapps~. xrets. xif. { xrets. apply~ bsearch_correct_found. }
  xapps~. xif.
  { xapp~ (m − i). hsimpl. applys~ bsearch_correct_recurse_l m. }
  { xapp~ (j − (m+1)). hsimpl. applys~ bsearch_correct_recurse_r m. }
Qed.
```

Let us walk through the proof. The purpose of the first two lines of Coq tactics is to setup reasoning by (well-founded) induction on `j−i`. The tactics used there might seem unfamiliar, but they are not specific to CFML; they come from the TLC Coq library [Cha], which provides useful general-purpose Coq tactics.

```
  introv. gen_eq n: (j−i). gen i j. induction_wf IH: (downto 0) n.
  intros i j Hn Hxs Hi Hj.
```

After these steps, the context has been extended with the induction hypothesis `IH`, asserting that the spec holds for ranges $[i', j')$ of lesser width than $[i, j)$. (In what follows, we allow ourselves to elude part of the context for conciseness when displaying Coq goals).

```
  a : loc
  xs : list int
  v, n : int
  IH : ∀y : int,
       downto 0 y n →
       ∀i j : int,
       y = j − i →
       sorted xs → 0 ≤ i ≤ length xs → 0 ≤ j ≤ length xs →
       PRE a ⤳ Array xs
       POST fun k : int ⇒ a ⤳ Array xs ⋆ [bsearch_correct xs v i j k]
       CODE (app bsearch [a v i j])
  i, j : int
  Hn : n = j − i
  Hxs : sorted xs
  Hi : 0 ≤ i ≤ length xs
  Hj : 0 ≤ j ≤ length xs
  ============================
  PRE a ⤳ Array xs
  POST fun k : int ⇒ a ⤳ Array xs ⋆ [bsearch_correct xs v i j k]
  CODE (app bsearch [a v i j])
```

The first CFML-specific tactic that we use is the `xcf` tactic. As a general rule of thumb, tactics whose name start with an "`x`" (dubbed "`x`-tactics") are CFML-specific tactics that usually correspond to the application of one Separation Logic reasoning rule. The `xcf` tactic corresponds to an application of the APP rule (Figure 2.4).

```
  xcf.
```

From a user perspective, the effect of the `xcf` tactic is to replace `bsearch` with its body in the goal, in order to start the verification process. Technically speaking, it amounts to applying the `bsearch__cf` axiom that was generated by CFML beforehand. What appears in the `CODE ...` part of the goal is in fact `bsearch`'s characteristic formula.

As one can see, the characteristic formula has been derived from `bsearch`'s code in a syntax-directed fashion: it is here pretty-printed using Coq notations in a way that resembles the original program.

```
a : loc
xs : list int
v, n : int
IH : .....
i, j : int
Hn : n = j − i
Hxs : sorted xs
Hi : 0 ≤ i ≤ length xs
Hj : 0 ≤ j ≤ length xs
============================
PRE a ⇝ Array xs
POST fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i j k]
CODE (If_ (isTrue (j ≤ i)) Then Ret −1
      Else (Let m := Ret i + (j − i / 2) in
            Let x0__ := App Array_ml.get a m in
             Let x1__ := Ret_ (v = x0__) in
              If_ x1__ Then Ret m
              Else (Let x2__ := App Array_ml.get a m in
                    If_ (isTrue (v < x2__)) Then App bsearch a v i m; Else App
                        bsearch a v (m + 1) j;)))
```

At this point, the reasoning traverses the code structurally using relevant `x`-tactics. As can be read from the goal, `bsearch`'s implementation starts with an `if`. Consequently, we use the corresponding `xif` tactic, which corresponds to rule IF.

```
xif.
```

`xif` produces two sub-goals, one for each branch of the program. Let us now consider the first branch. In this case, we know additionally that `j ≤ i`, as given by the new hypothesis `C` that has been introduced by `xif`. The program simply returns -1 in that case (meaning that `v` cannot be found). The first subgoal is as follows:

```
a : loc
xs : list int
v, n : int
IH : .....
i, j : int
Hn : n = j − i
Hxs : sorted xs
Hi : 0 ≤ i ≤ length xs
Hj : 0 ≤ j ≤ length xs
C : j ≤ i
============================
PRE a ⇝ Array xs
POST fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i j k]
CODE (Ret (−1))
```

The "`Ret`" characteristic formula corresponds to a program that simply produces a value. It can be processed using the `xret` tactic, which combines an application of the VAL rule with the FRAME rule.

```
xret.
```

The goal produced by `xret` requires the user to prove that the pre-condition entails the post-condition, applied to the returned value:

```
...
============================
a ⤳ Array xs  ⊫  (a ⤳ Array xs ⋆ [bsearch_correct xs v i j (-1)]) ⋆ GC
```

CFML provides a specific tactic, called `hsimpl`, that is able to simplify heap entailments. Here, it is enough to call `hsimpl` to simplify the goal into:

```
bsearch_correct xs v i j (-1)
```

which can then be discharged using the lemma `bsearch_correct_out_of_bounds`.

The tilde at the end of some tactics (e.g., `apply~` below) comes from TLC: it indicates that some automation will be called to discharge proof obligations produced by the tactic. In this example, the automation simply consists of `eauto` extended with hints calling `omega` in order to discharge math-related obligations.

```
hsimpl. apply~ bsearch_correct_out_of_bounds.
```

Let us now focus on the second sub-goal, which consists of the rest of the program. In this sub-goal, we know that `i < j` holds, courtesy of the `xif` tactic.

```
...
C : i < j
============================
PRE a ⤳ Array xs
POST fun k : int ⇒ a ⤳ Array xs ⋆ [bsearch_correct xs v i j k]
CODE (Let m := Ret (i + (j - i / 2)) in
      Let x0__ := App Array_ml.get a m in
       Let x1__ := Ret_ (v = x0__) in
        If_ x1__ Then Ret m
        Else (Let x2__ := App Array_ml.get a m in
              If_ (isTrue (v < x2__)) Then App bsearch a v i m; Else App bsearch a v
                   (m + 1) j;))
```

One can notice that the translation from OCaml to characteristic formula introduced intermediate `let`-bindings. It is possible to manually call the corresponding `xlet` tactic, but fortunately it is also called automatically if needed. Consequently, to take care of the first `let`-binding, it is enough to again call `xret`:

```
xret; intro Hm.
```

The call to `xret` automatically calls `xlet` under the hood, which introduces a new binding in the environment, corresponding to the previously `let`-bound variable (rule LET). A new hypothesis (here named `Hm`) is also introduced; it corresponds to the body of the definition.

```
...
m : int
Hm : m = i + (j - i / 2)
============================
PRE a ⤳ Array xs
POST fun k : int ⇒ a ⤳ Array xs ⋆ [bsearch_correct xs v i j k]
```

```
CODE (Let x0__ := App Array_ml.get a m in
      Let x1__ := Ret_ (v = x0__) in
       If_ x1__ Then Ret m
       Else (Let x2__ := App Array_ml.get a m in
             If_ (isTrue (v < x2__)) Then App bsearch a v i m; Else App bsearch a v (
                 m + 1) j;))
```

The next `let`-binding corresponds to a call to the `Array.get` function for reading into an array (here, reading `a` at index `m`). The tactic for handling function calls is `xapp`, which automatically fetches a registered specification for the function being called. (In practice, this meta-level environment of specifications is implemented as a base of hints in Coq.) After the specification has been fetched, it is used through an application of the CONSEQUENCE and FRAME rules. Here, we use a variant of the tactic called `xapps`, which, when used under a `let`-binding, additionally substitutes the value of the `let` definition in the rest of the program.

```
xapps~.
```

Since the predicate "`a ⇝ Array xs`" can be found in the precondition, the result of reading into the array is `xs[m]`. (Recall that we write `xs[i]` to denote the `i`-th element of the list `xs`.) It is hence substituted in place of the variable `x0__` in the rest of the characteristic formula.

```
============================
PRE a ⇝ Array xs
POST fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i j k]
CODE (Let x1__ := Ret_ (v = xs[m]) in
       If_ x1__ Then Ret m
       Else (Let x2__ := App Array_ml.get a m in
             If_ (isTrue (v < x2__)) Then App bsearch a v i m; Else App bsearch a v (m
                 + 1) j;))
```

The last `let`-binding (corresponding to boolean "`v = xs[m]`") can be substituted away by using the tactic `xrets`, a variant of `xret`.

```
xrets.
```

A first conditional expression tests whether the value `v` has been found: this first case can be discharged easily using the lemma `bsearch_correct_found`.

```
xif. { xrets. apply~ bsearch_correct_found. }
xapps~.
```

Finally, if the value `v` has not been found, there remains to decide whether to recurse on the left half slice, or the right one.

```
...
Hmbound : i ≤ m < j
C0 : v ≠xs[m]
============================
PRE a ⇝ Array xs
POST fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i j k]
CODE (If_ (isTrue (v < xs[m])) Then App bsearch a v i m; Else App bsearch a v (m + 1) j
    ;)
```

```
xif.
```

Each branch consists of a (tail-)recursive call to `bsearch`. Since we are reasoning by induction: we know that recursive calls do satisfy the specification, as part of the induction hypothesis `IH`.

```
IH : ∀y : int,
     downto 0 y n →
     ∀i j : int,
     y = j − i →
     sorted xs →
     0 ≤ i ≤ length xs →
     0 ≤ j ≤ length xs →
     PRE a ⇝ Array xs
     POST fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i j k]
     CODE (app bsearch [a v i j])
...
Hmbound : i ≤ m < j
C1 : v < xs[m]
============================
PRE a ⇝ Array xs
POST fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i j k]
CODE App bsearch a v i m;
```

The `xapp` tactic then simply fetches the induction hypothesis from the context (we provide manually the value for "`y`"—the size of the segment).

```
xapp~ (m − i).
```

The only side-condition that remains is proving that the postcondition of the recursive call entails the postcondition of the whole function.

```
...
Hmbound : i ≤ m < j
C1 : v < xs[m]
============================
(fun k : int ⇒ a ⇝ Array xs ⋆ [bsearch_correct xs v i m k]) ⊫
(fun x : int ⇒ (a ⇝ Array xs ⋆ [bsearch_correct xs v i j x]) ⋆ GC)
```

This can be discharged simply by using the `hsimpl` tactic, and applying the lemma `bsearch_correct_recurse_l`.

```
hsimpl. applys~ bsearch_correct_recurse_l m.
```

The other branch is handled in a similar fashion, thus completing the proof:

```
xapp~ (j − (m+1)). hsimpl. applys~ bsearch_correct_recurse_r m.
```

Summary CFML provides tactics that enable stepping through the program that one wishes to verify, generating proof obligations in a step-by-step fashion.

We have seen that reasoning on recursive programs is done at the meta-level, using standard Coq principles for reasoning by induction. In other words, there is no built-in Separation Logic rule for recursive functions.

Each "`x`-tactic" roughly corresponds to one program construct (or alternatively, to one Separation Logic rule). Since the use of `x`-tactics seems to be directed by the syntax

of the program (and it mostly is), one might wonder why there is not a single tactic that would repeatedly call the correct `x`-tactic, saving the user a lot of work. It is possible to implement such a tactic—to some extent. However, this would not be in line with the overall style of CFML proofs. Because proofs are interactive in essence (and not fully automated), it is important to have fine grained control over the proof context. For instance, the choice of substituting a `let`-binding or keeping it as an equation is crucial for keeping the proof state understandable, as is manually providing intelligible names for the variables introduced. Using CFML, one has for instance the choice of using `xapps` instead of `xapp` (as demonstrated previously, `xapps` substitutes away a definition introduced by a `let`-binding, whereas `xapp` allows giving it a name). That choice requires human insight to keep proof obligations intelligible at all time.

2.2 Separation Logic with Time Credits

The previous section described Separation Logic as a framework in which to establish the *functional correctness* of heap-manipulating programs. In the present section, we show how Separation Logic can be extended with Time Credits, in order to additionally reason about *amortized time complexity*. The idea of using time credits as a Separation Logic resource goes back to Atkey [Atk11]; Pilkiewicz and Pottier also describe a similar approach [PP11]. More generally, the idea of associating credits (or potential) to data structures in order to devise amortized resource analyses goes back to the work of Hofmann and Jost [Hof99, HJ03]. Charguéraud and Pottier implement Separation Logic with Time Credits as an extension of a general purpose Separation Logic framework (CFML), and use it to verify the functional correctness and asymptotic complexity of an Union-Find data structure [CP17b].

Separation Logic with Time Credits forms the logical basis on which we build this thesis. It enjoys a number of features that makes it particularly interesting, and that we detail as follows.

Separation Logic with Time Credits is a minimal extension of Separation Logic. As mentioned previously, it has been implemented as an extension of CFML; Mével *et al.* also implemented Time Credits in the setting of the Iris Separation Logic [MJP19], which is also embedded in Coq. It seems reasonable to assume that such an extension could be implemented without too much effort in other existing interactive verification frameworks that use Separation Logic.

As argued by Atkey [Atk11], Separation Logic with Time Credits provides a convenient framework in which to reason about *amortized* complexity, in a way that fits very well with how ordinary Separation Logic proofs are carried out. In ordinary Separation Logic, it is customary to define (possibly inductive) predicates that describe the ownership of a data structure. Using time credits, one can simply "sprinkle" time credits in the definition of such a predicate, enabling amortized reasoning. Such a technique effectively implements the potential method of amortized complexity analysis [Tar85]: Separation Logic with Time Credits allows the user to easily relate the state of a data structure to some amount of time credits (its potential), which can be used to amortize the cost of expensive operations. We give an example in Section 2.2.2.

Time credits allow the user to rely on arbitrary invariants and functional correctness arguments in order to establish the complexity proof. In a sense, when using time credits, proving the time complexity of a program amounts to a proof of safety: one has to establish that the program always safely runs to completion, *provided a limited amount of time credits*. This intuition has been made precise by Mével *et al.* [MJP19]. Note that in this approach, functional correctness and asymptotic complexity are established together, as part of the same proof. This makes it easy to use intermediate invariants used for functional correctness to support the complexity analysis. In paper proofs, it is more common to establish the two separately. However, in a Hoare-style verification framework, it is unclear to us how one would go and step through the same code twice, while using in the second step intermediate invariants that have been established in the first step. We have observed that, in practice, it is in fact quite convenient to develop both correctness and complexity arguments in lock step.

2.2.1 A Minimal Extension of Separation Logic

We now describe how to extend Separation Logic (as described in Section 2.1.2) with time credits. To do this, we must revisit the definition of the type Heap of heap fragments. Instead of defining Heap as simply Memory, we let a heap h to be a pair (m, c) of a memory m and a natural number c, which represents a number of time credits that "we" own and are allowed to spend.

$$\mathsf{Heap} \triangleq \mathsf{Memory} \times \mathbb{N}$$

This new definition of Heap allows us to define the Separation Logic assertion $\$n$, which asserts the ownership of (exactly) n time credits.

$$\$n \triangleq \lambda(m, c).\ m = \varnothing \wedge c = n$$

The definitions of standard Separation Logic assertions (Figure 2.3) are unchanged, assuming we lift some of our notation to the level of our new Heap definition. Empty and singleton heaps must be understood as bearing no credits at all. Disjointness is simply lifted as disjointness on memories, and disjoint union must be extended to merge the amounts of credits. Formally speaking, we overload the notation for $\varnothing$, $l \mapsto v$, $\perp$ and $\uplus$ with new definitions. Below, the left-hand side refers to the new definition, while the right-hand side refers to the old definition.

$$\begin{array}{lcl}
\varnothing & \triangleq & (\varnothing, 0) \\
(l \mapsto v) & \triangleq & ((l \mapsto v), 0) \\
(m_1, c_1) \perp (m_2, c_2) & \triangleq & m_1 \perp m_2 \\
(m_1, c_1) \uplus (m_2, c_2) & \triangleq & (m_1 \uplus m_2, c_1 + c_2)
\end{array}$$

The definition of the separating conjunction $H_1 \star H_2$ does not change (we recall it below), but must be reinterpreted with regard to the new definitions of $\perp$ and $\uplus$.

$$H_1 \star H_2 \triangleq \lambda h.\ \exists h_1\, h_2.\ h_1 \perp h_2 \wedge h = h_1 \uplus h_2 \wedge H_1\ h_1 \wedge H_2\ h_2$$

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

$$\textsc{eval-let}\ \frac{t_1/m \Downarrow^{n_1} v_1/m' \qquad [v_1/x]\, t_2/m' \Downarrow^{n_2} v/m''}{(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2)/m \Downarrow^{(n_1+n_2)} v/m''} \qquad \textsc{eval-app}\ \frac{[\mu f.\lambda x.t/f]\,[v/x]\, t/m \Downarrow^n v'/m'}{((\mu f.\lambda x.t)\ v)/m \Downarrow^{(n+1)} v'/m'}$$

$$\textsc{eval-ref}\ \frac{l \notin \mathrm{dom}(m)}{(\mathsf{ref}\ v)/m \Downarrow^0 l/m \uplus (l \mapsto v)} \qquad \textsc{eval-get}\ \frac{l \notin \mathrm{dom}(m)}{(\mathsf{get}\ v)/m \uplus (l \mapsto v) \Downarrow^0 v/m \uplus (l \mapsto v)}$$

$$\textsc{eval-set}\ \frac{l \notin \mathrm{dom}(m)}{(\mathsf{set}\ (l,\, v))/m \uplus (l \mapsto v') \Downarrow^0 ()/m \uplus (l \mapsto v)}$$

Figure 2.5: Cost-instrumented big-step operational semantics

Based on these definitions, the following two fundamental reasoning rules on credits can be proved:

$$\$(n + n') = \$n \star \$n' \qquad \text{and} \qquad \$0 = [\,]$$

At this point, credits are purely an accounting device. One can carry them around, split them, join them, but they do not (yet) have any connection with the consumption of actual computational resources. There remains to revise the definition of Separation Logic triples (Definition 2.1.2) so as to connect the notion of credit with the number of computation steps that are taken by the program, therefore allowing credits to be interpreted as "time credits".

Before we introduce our revised definition of triples, we introduce a *cost-instrumented* evaluation judgment $t/m \Downarrow^n v/m'$, inductively defined in Figure 2.5. Such a judgment holds if the evaluation of the term t in the initial memory m terminates after n "steps" of computation, producing the value v and the final memory m'. In fact, the only "steps" that we count are function calls; we claim that it is a satisfying measure, even if somewhat abstract (see [CP17b, §2.7] for a discussion). As such, EVAL-APP is the only rule where the counter n is incremented. The previous evaluation judgment $t/m \Downarrow v/m'$, which does not carry a cost annotation, is defined as $\exists n.\ (t/m \Downarrow^n v/m')$.

Definition 2.2.1 displays the updated definition of triples.

Definition 2.2.1 (Triples in Separation Logic with Time Credits) *A triple* $\{H\}\ t\ \{Q\}$ *is short for the proposition:*

$$\forall m\, c\, H'.\ (H \star H')\ (m, c) \implies \exists v\, m'\, c'.\ \begin{cases} t/m \Downarrow^n v/m' \\ (Q\ v \star H' \star \mathrm{GC})\ (m', c') \\ c = n + c' \end{cases}$$

$$\textsc{app-pay}\quad \frac{v_1 = \mu f.\lambda x.t \qquad \{H\}\ ([v_1/f]\,[v_2/x]\,t)\ \{Q\}}{\{\$1 \star H\}\ (v_1\ v_2)\ \{Q\}}$$

Figure 2.6: Revised reasoning rules for Separation Logic with Time Credits.

There are three differences with Definition 2.1.2. First, the precondition $H \star H'$ and the postcondition $Q\,v \star H' \star \mathrm{GC}$ now apply to a pair of a memory and a number of credits, (m, c) and (m', c') respectively. Second, the evaluation judgment $t/m \Downarrow v/m'$ is replaced with the cost-instrumented judgment $t/m \Downarrow^n v/m'$. Finally, we need to express that each step of computation *consumes one time credit*. We therefore add the constraint $c = n + c'$, which captures the fact that the initial number of credits is exactly the sum of the number of computation steps taken, and the remaining number of credits. Note that the use of the GC predicate accounts for the possibility of discarding unused resources, including time credits. Even though we require the equality $c = n + c'$ to hold (instead of an inequality $c \geq n + c'$), time credits in a precondition do correspond to *upper bounds* on the execution cost, because the quantity c' might account for a number of discarded time credits. The fact that we are using a total program logic is mostly orthogonal to the introduction of time credits; see Section 6.3.3 for a more detailed discussion later on.

As a corollary, this interpretation of triples provides the following guarantee for a full program execution:

"If $\{\$n\}\ t\ \{\mathrm{GC}\}$ *holds, then* t *terminates safely in at most* n *computation steps."*

In other words, the number of time credits that one initially supplies in the precondition is an upper bound for the cost of running the program.

All the reasoning rules shown in Figure 2.4 except APP can be proved sound with respect to the revised definition of triples. We replace APP with APP-PAY (Figure 2.6) to account for the fact that one function call now costs one credit. The APP-PAY rule asserts that one must pay one time credit to "enter" the definition of a function and reason on its body. It is important to note that this rule is used exactly once in order to establish the specification of a function. When reasoning on a function call involving a known function, one does not use APP-PAY, but the specification that has been previously established for this function.

Theorem 2.2.2 (Soundness of Separation Logic with Time Credits) *Each of the reasoning rules in Figure 2.4 except* APP *is valid with respect to the interpretation of triples given by Definition 2.2.1. So is the rule* APP-PAY *in Figure 2.6.*

Again, the proof of this theorem has been formalized by Charguéraud and Pottier [CP17b].

Finally, the characteristic formulae framework has to be extended as well. This is done by introducing a new pseudo-instruction `pay`, which requires the user to pay for one time credit. It admits the following specification:

$$\{\$1\}\ \texttt{pay}()\ \{\lambda\_.\ [\,]\}$$

Then, the characteristic formulae generator of CFML is extended to insert a call to `pay()` at the beginning of the characteristic formula of the body of each function and at the head of every loop body.

2.2.2 Example: Proving the Complexity of a Program

Let us come back to the examples introduced in Section 2.1.1, and illustrate how one would specify and establish complexity bounds for these, using Separation Logic with Time Credits.

Binary Search

Binary search (Figure 2.1) enjoys logarithmic time complexity, with respect to the size of the input $(j - i)$. Because we are counting individual function calls, a more precise account is required. One can establish that "$\mathsf{If}\ j \leq i\ \mathsf{then}\ 1\ \mathsf{else}\ 3 \cdot \log_2(j-i) + 4$" is a sufficient amount of credits to verify a call to `bsearch` $a\ v\ i\ j$. In fact, since it is always sound to provide extraneous credits, this cost expression could be simplified to "$3 \cdot \log_2(j-i) + 4$" in a second step (assuming we define $\log_2 x$ to be equal to 0 for $x \leq 0$, following the Coq definition). However, the previous more complicated expression seems to be required for the induction to go through.

We therefore prove by induction (on the width of the segment $[i, j)$) the following specification for `bsearch`, which establishes both functional correctness and time complexity:

$$\begin{array}{l}\forall a\ xs\ v\ i\ j.\\ 0 \leq i \leq |xs| \wedge 0 \leq j \leq |xs| \wedge \text{sorted}\ xs \implies\\ \quad \{a \rightsquigarrow \mathsf{Array}\ xs \star \$(\mathsf{If}\ j \leq i\ \mathsf{then}\ 1\ \mathsf{else}\ 3 \cdot \log_2(j-i) + 4)\}\\ \qquad\qquad \texttt{bsearch}\ a\ v\ i\ j\\ \qquad \{\lambda k.\ a \rightsquigarrow \mathsf{Array}\ xs \star [\text{bsearch\_correct}\ xs\ v\ i\ j\ k]\}.\end{array}$$

The characteristic formula generated for `bsearch` using CFML updated with support for time credits can be read from the Coq goal at the beginning of the proof (similarly to what appears in Section 2.1.4):

```
...
============================
PRE a ⤳ Array xs ⋆ $(If j ≤ i then 1 else 3 * Z.log2 n + 4)
POST fun _ : int ⇒ a ⤳ Array xs
CODE Pay_ ;;
     (If_ (isTrue (le j i)) Then Ret -1
      Else (Let m := Ret i + (j - i / 2) in
            Let x0__ := App Array_ml.get a m in
             Let x1__ := Ret_ (v = x0__) in
              If_ x1__ Then Ret m
              Else (Let x2__ := App Array_ml.get a m in
                    If_ (isTrue (lt v x2__)) Then App bsearch a v i m;
                    Else App bsearch a v (m + 1) j;)))
```

The characteristic formula only differs from the ordinary one (in Section 2.1.4) by the initial call to `pay()` (here noted as `Pay_`). Additionally, one has to take into account

the fact that the `Array.get` primitive that reads from an array is now equipped with a specification that also require one time credit to execute. (This means that, in fact, `Array.get` is viewed here as a function, not as a primitive operation.)

We will not go into the details of how exactly one would carry the proof of the specification above. At this point, we only have rudimentary techniques available to handle time credits, and such a proof would be quite tedious. One contribution of this thesis is actually to come up with better proof techniques (see Chapter 6). In short, the main two complexity-related proof obligations that arise from stepping through the proof are the following:

$$\mathsf{If}\ (i + \tfrac{j-i}{2} \le i)\ \mathsf{then}\ 1\ \mathsf{else}\ (3 \cdot \log_2(\tfrac{j-i}{2}) + 4) \quad \le \quad 3 \cdot \log_2(j-i) + 1$$
$$\mathsf{If}\ (j \le i + \tfrac{j-i}{2} + 1)\ \mathsf{then}\ 1\ \mathsf{else}\ (3 \cdot \log_2(j - i + \tfrac{j-i}{2} + 1) + 4) \quad \le \quad 3 \cdot \log_2(j-i) + 1$$

These proof obligations are purely mathematical, and do not refer to the program source code or characteristic formula in any way.

Amortized Mutable Queues

The above specification for `bsearch` establishes a worst-case complexity bound: all time credits that appear in the precondition are spent during the execution of `bsearch`; no time credit appears in the postcondition.

The implementation of mutable queues using a pair of lists (Figure 2.2, introduced earlier in §2.1.1) is a good example on which to illustrate *amortized* complexity specifications. Queues implemented as a (mutable) pair of lists enjoy worst-case constant time `enqueue` operations: it is enough to add the element to the "back" list. A `dequeue` operation consists in removing the head element from the "front" list. In the case of an empty "front" list, a `dequeue` operation additionally requires reversing the "back" list in order to obtain a new "front" list.

Fortunately, it is possible to amortize the cost of reversing the list. Indeed, a `dequeue` operation necessarily follows at least one `enqueue` operation, and costs linearly in the number of `enqueue` operations since the last call to `dequeue`. This means that the overall cost of a sequence of `enqueue` and `dequeue` operations is linear in the number of operations. In other words, both `enqueue` and `dequeue` operations on the structure therefore enjoy an *amortized* constant time complexity.

In order to formalize that intuition, the key idea is to associate with the Queue data structure a number of time credits, that will be eventually used to pay for reversing the "back" list. Therefore, we revise the definition of the Queue predicate, in order to include as many time credits as there are elements in the "back" list:

$$\mathsf{Queue}\ q\ L \triangleq \exists L_1\ L_2.\ q \rightsquigarrow \{|\mathrm{front}{=}L_1;\ \mathrm{back}{=}L_2|\} \star [L = L_1 +\!\!+ \mathrm{rev}\ L_2] \star \$(|L_2|).$$

As operations on Queue take as precondition a predicate of the form Queue $q\ L$ and return in postcondition a predicate of the form Queue $q\ L'$, the time credits "stored" in Queue are threaded through the calls to operations on the structure, increasing or decreasing depending on the size of the internal "back" list L_2.

It is now possible to prove specifications for the various operations on Queue that all advertise a constant cost: amortization is abstracted away by the definition of Queue.

The functions `create` and `is_empty` do not modify the queue: they only require one time credit to pay for calling the function. The `enqueue` operation requires two time credits: one to pay for calling `enqueue` itself, and one that is returned as part of the updated Queue predicate. Since one element has been added to the "back" list, one extra time credit must be kept to later pay for traversing that element while reversing the list. Finally, the `dequeue` operation has an advertised cost of only one credit. If `dequeue` needs to reverse the "back" list, the cost of reversing the list can be paid by using the credits accumulated in the input Queue predicate.

$\{\$1\}$ `create` $()$ $\{\lambda q.\ \mathsf{Queue}\ q\ \mathsf{nil}\}$
$\{\mathsf{Queue}\ q\ L \star \$1\}$ `is_empty` q $\{\lambda b.\ \mathsf{Queue}\ q\ L \star [b = \mathsf{true} \Leftrightarrow L = \mathsf{nil}]\}$
$\{\mathsf{Queue}\ q\ L \star \$2\}$ `enqueue` $v\ q$ $\{\lambda().\ \mathsf{Queue}\ q\ (v :: L)\}$
$\{\mathsf{Queue}\ q\ L \star [L \neq \mathsf{nil}] \star \$1\}$ `dequeue` q $\{\lambda v.\ \exists L'.\ \mathsf{Queue}\ q\ L' \star [L = L' +\!\!+ v :: \mathsf{nil}]\}$

Note that it would be possible to prove an alternative version of these specifications, where the cost of calling `dequeue` is amortized as well: one could advertise a cost of \$3 for `enqueue`, and `dequeue` would require no credit at all. This alternative presentation proves useful for verifying algorithm whose analysis involves arguments of the form "there can not be more `enqueue` operations than `dequeue` operations".

Note that the predicate Queue predicate is *not duplicable*, which ensures single-threaded use of the queue; this is crucial to guarantee that the amortization argument makes sense.

Conclusion

Separation Logic with Time Credits provides a powerful framework in which to establish both the functional correctness and amortized complexity of heap-manipulating programs.

However, program specifications using bare time credits are not *modular*. Indeed, they require exporting a precise amount of time credits, where one is generally only interested in the asymptotic behavior, expressed using the well-known big-O notation. Thereafter, we discuss in Chapter 3 some challenges associated with the O notation in the setting of program verification; then, in Chapter 4, we address the question of how modular complexity specifications can be expressed in the setting of Separation Logic with Time Credits.

It is also not clear at this point how proofs using time credits should be carried out. Atkey mentions the use of a mechanism that automatically infers constant amounts of credits through a proof [Atk11]; but it is unclear how such a mechanism would apply in an interactive setting. In Chapter 6, we detail the reasoning principles and proof techniques that we developed in order to make proofs involving time credits practical.

Chapter 3

Challenges in Reasoning with the O Notation

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. This style of reasoning leads to believe that reasoning about the complexity of a program can always be performed by composing O bounds from the bottom up, by inferring O bounds for the leaves of the program, and composing them to finally obtain a O bound for the whole program.

In this chapter, we demonstrate that this intuition does not in fact hold. More generally, we show that assertions of the form "this code has asymptotic complexity $O(n)$" are too informal: they do not have sufficiently precise meaning, and can be easily abused to produce flawed paper proofs.

As we shall see, these assertions are imprecise for several reasons. When writing "$O(n)$", the relationship between "n" and the parameters of the program is left implicit. Additionally, since a O bound is asymptotic, it refers to "big enough" values of n, which appears to be a somewhat fuzzy notion, in particular in the presence of a O bound with multiple variables (e.g. $O(mn)$). Finally, when writing "this code has complexity [...]", this assertion informally refers to a program function, or a code block, but does not make explicit under which context. Indeed, when reasoning modularly about programs, one cannot make a closed world assumption: this particular piece of program might depend on some parameters, or call auxiliary functions—those are usually left implicit.

3.1 A Flawed Proof of a O Bound for a Recursive Function

A first striking example illustrates how one might "prove" that a recursive function has complexity $O(1)$, whereas its actual cost is $O(n)$.

Code:

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

Correct claim: The OCaml function `waste` has asymptotic complexity $O(n)$.

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

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

- **Case** $n \leq 0$: `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, $O(1)$. □

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.

3.2 A Flawed Proof of a O Bound in the Presence of Multiple Variables

Code:

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

let f n = g (0, n)
```

Correct claim: The OCaml function `f` has asymptotic complexity $O(n)$.

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

Flawed proof:

- `g`(m, n) involves mn inner loop iterations, thus costs $O(mn)$.
- The cost of `f`(n) is the cost of `g`$(0, n)$, plus $O(1)$. As the cost of `g`(m, n) is $O(mn)$, we find, by substituting 0 for m, that the cost of `g`$(0, n)$ is $O(0)$. There follows that `f`(n) costs $O(1)$. □

In this example, the auxiliary function `g` takes two integer arguments m and n and involves two nested loops, over the intervals $[1, n]$ and $[1, m]$. Its asymptotic complexity is $O(n + mn)$, which, *under the hypothesis that m is large enough*, can be simplified to $O(mn)$. The reasoning, thus far, is correct. The flaw lies in our attempt to substitute 0 for m in the bound $O(mn)$. 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 "`g`(m, n) costs $O(mn)$ when n and m are sufficiently large", one *cannot* deduce anything about the cost of `g`$(0, n)$. To repair this proof, one must take a step back and prove that `g`(m, n) has asymptotic complexity $O(n + mn)$ *for sufficiently large n and for every m*. This fact *can* be instantiated with $m = 0$, allowing one to correctly conclude that `g`$(0, n)$ costs $O(n)$.

This example also raises the more general question: what is the most general specification for the complexity of a program, when using O bounds with multiple parameters? One would like to, at the same time, give a general and succinct asymptotic bound, without having to anticipate the fact that one might want to use it by specializing one or several of the parameters. We come back to this question in §4.1.3 and later §5.4.

3.3 A Flawed Proof of a O Bound for a `for`-loop

An additional example of tempting yet invalid reasoning appears thereafter. We borrow it from Howell [How08].

Code:

```
1    let h (m, n) =
2      for i = 0 to m−1 do
3        let p = (if i = 0 then pow2 n else n∗i) in
4        for j = 1 to p do () done
5      done
```

Correct claim: The OCaml function `h` has asymptotic complexity $O(2^n + nm^2)$.

Incorrect claim: The OCaml function `h` has asymptotic complexity $O(nm^2)$.

Flawed proof:

- Let us first consider the body of the outer loop (lines 3-4) in isolation: it has asymptotic cost $O(ni)$. Indeed, as soon as $i > 0$ holds, the inner loop performs ni constant-time iterations. The case where $i = 0$ does not matter in an asymptotic analysis.
- The cost of `h`(m, n) is the sum of the costs of the iterations of the outer loop:

$$\sum_{i=0}^{m-1} O(ni) \;=\; O\left(n \cdot \sum_{i=0}^{m-1} i\right) \;=\; O(nm^2).$$

□

This flawed proof exploits the dubious idea that "the asymptotic cost of a loop is the sum of the asymptotic costs of its iterations". In more precise terms, the proof relies on the following implication, where $f(m, n, i)$ represents the true cost of the i-th loop iteration and $g(m, n, i)$ represents an asymptotic bound on $f(m, n, i)$:

$$f(m,n,i) \in O(g(m,n,i)) \quad \Rightarrow \quad \sum_{i=0}^{m-1} f(m,n,i) \in O\left(\sum_{i=0}^{m-1} g(m,n,i)\right)$$

As pointed out by Howell, this implication is in fact invalid. Here, $f(m, n, 0)$ is 2^n and $f(m, n, i)$ when $i > 0$ is ni, while $g(m, n, i)$ is just ni. The left-hand side of the above implication holds, but the right-hand side does not, as $2^n + \sum_{i=1}^{m-1} ni$ is $O(2^n + nm^2)$, not $O(nm^2)$. The Summation lemma presented later on in this paper (Lemma 5.3.10) rules out the problem by adding the requirement that f be a nondecreasing function of the loop index i. We discuss in depth later on (§4.2.1) why cost functions should and can be monotonic.

3.4 On Local Reasoning in the Presence of O

This last example is perhaps most compelling for readers who are used to formal reasoning on programs in the style of Hoare logic. In this style, a main principle is that one is able to reason about a complex function by proving intermediate specifications (e.g. for locally defined functions) as intermediate proof steps, and later use these local specifications to prove a top-level specification for the whole function. However, this (informal) reasoning principle seems to be incompatible with program specifications that use O.

Consider for instance the OCaml function `f` defined below. To prove a given specification about `f`, one proceeds by reasoning on its body, given fixed, abstract values of the parameters. More specifically, for some fixed `n`, one might locally prove some specification about the locally-defined function `g`, and then use it in the rest of the body of `f`, to eventually establish a top-level specification about `f`.

```
let f n =
  let g () =
    for i = 1 to n do () done
  in
  g ()
```

When reasoning about the asymptotic complexity of `f`, one would like to be able to prove locally an asymptotic bound for `g` using O, and afterwards use it to derive a O bound for `f`. However, naively following the same reasoning style does not work. One can indeed prove locally a O bound for `g`, but this bound will depend on the given value of the parameter `n` which appears in the local context. In particular, one can successfully prove that, locally, `g` runs in $O(1)$. Indeed, for any (fixed!) value of `n`, calling `g()` takes a constant amount of time. But it would be clearly bogus to use this fact to conclude that since `f` simply calls `g()`, then `f` runs in $O(1)$ *whenever* `n` *goes to infinity*.

The fact that `g` is $O(1)$ for any fixed `n` is technically correct. On paper, this seems like a rather strange statement. Instead, one is expected to make explicit the dependency on `n` in the asymptotic bound, and state that `g` has complexity $O(n)$. Unfortunately, during the proof of `f`, this statement is equivalent to "`g` is $O(1)$", because `n` is a constant! In fact, any specification about `g` that has been proved as part of the proof of `f` cannot be used in the rest of the proof. This comes from the fact that O hides an existential quantification, whose witness might depend on any constant appearing in the local context (e.g. `n`), making it useless in a more general context (e.g. where `n` is not fixed but goes to infinity). This is similar to the faulty proof of O "by induction" of §3.1.

This seems to indicate that O bounds are always implicitly stated "at the top-level", even when talking about local functions or blocks of code. This requirement can be made explicit by "lambda-lifting" the program, i.e. applying a mechanical transformation that extracts local functions into top-level definitions. The snippet below illustrates the transformation: it shows an equivalent definition of `f`, after lifting `g` as a top-level definition, that is now explicitly parameterized by `n`. In a Hoare-style framework, this forces a O specification for `g` to explicitly depend on `n`, and makes it usable in the proof of `f`.

```
let g n () =
  for i = 1 to n do () done
let f n =
```

```
g n ()
```

The challenge is as follows: in a Hoare-style verification framework, we would like to be able to give O specifications about locally-defined functions as usual, without having to manually extract the local functions as top-level definitions.

Conclusion

The examples that we have presented 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. Sections 3.1 and 3.4 demonstrate that proving a O bound for a program does not amount to composing O bounds from its sub-programs in a bottom-up fashion. Sections 3.2 and 3.3 demonstrate that extra care and side-conditions are required when composing O bounds with multiple parameters.

In the rest of this thesis, we do assign O specifications to functions, because we believe that this style is elegant, modular and scalable (Chapter 4). We give a formal account of O and associated reasoning principles, in order to clear up the subtleties of the multivariate case (Chapter 5). However, composition principles on O alone do not constitute a workable program logic in which one can reason about the complexity of programs—in particular in the case of recursive functions (§3.1). We design a program logic and proof techniques that enable reasoning about the cost of a function body, even in the recursive case (Chapter 6). We are able to synthesize a cost expression (or recurrence equations) for the function, then check that this expression is indeed dominated by the asymptotic bound that appears in the specification. The techniques that we introduce allow the proof to be independent from the low-level details of the cost expression, and let the user perform approximations or introduce abstraction barriers in order to simplify the complexity bounds when required.

Chapter 4

Towards Practical Specifications for the Complexity of Programs

In Chapter 3 we show that although program specifications using O seem to convey an intuitive meaning, the devil is in the details. The O notation itself—as a relation between mathematical functions—is sometimes involved in fuzzy reasonings. Worse, statements of the form "`p` is $O(n)$"—where `p` is a computer program—are treated in an even more informal fashion.

In this chapter, we take a step back and address the following question: what might be a program specification which corresponds to the informal "`p` is $O(n)$", but is both *rigorously defined* and *practical*? We ground the discussion in the formal setting of Separation Logic with Time Credits (as introduced in §2.2). This already provides a logic in which it is possible to rigorously express and prove specifications about the (possibly amortized) complexity of programs, and departs from the high-level but possibly bogus reasoning principles discussed in the previous chapter. We then argue that for a program specification to be practical, it should result in a trade-off between precision and modularity. A specification should provide enough information so that it is usable by a client, but it should also abstract over the specific details of the implementation, in order to make modular reasoning feasible.

We argue that the natural style of specification exposing an explicit number of credits is not *practical*, because it is not modular. We argue that despite the challenges exposed in Chapter 3, asymptotic complexity specifications using O is still our best bet. We then show how modular asymptotic specifications using O can be expressed in the setting of Separation Logic with Time Credits.

Disclaimer

Our formal account of the O notation is introduced in the next chapter (§5). In this chapter, whenever we write "$f \in O(g)$", it should be understood in a somewhat informal sense, intended to correspond to how the notion is used in practice in pen and paper proofs. If f and g are mathematical functions of type $\mathbb{Z} \to \mathbb{Z}$, then we expect it to coincide with the following definition:

$$f \in O(g) \triangleq \exists c.\, \exists n_0.\, \forall n \geq n_0.\, |f(n)| \leq c \cdot |g(n)|.$$

To make a clear distinction, our "formal O" (Definition 5.1.1) is written as "$f \preceq_F g$", and in particular relies on an extra parameter F (denoting a *filter*).

Running Example

A running example in this chapter is the `length` function on lists.

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

As detailed in §2.2, to reason about its complexity we consider an instrumented version of the code, where one call to `pay ()` has been inserted after entering the function.

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

It is clear that `length` has a linear time complexity with respect to the size of its input list. We can even be more precise here: on this particular instrumented version of the code, a call to `length l` requires exactly $|l| + 1$ time credits (recall that one time credit is required per call to `pay()`).

4.1 Challenges with respect to Modularity

4.1.1 Specifying the Complexity of a Program, Formally

We can give a first formal specification for the complexity of `length` that uses an *explicit number of credits*, in Separation Logic with Time Credits. Note that we can specify at the same time that `length` is functionally correct, i.e. it indeed returns the length of its input list.

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

This specification guarantees that `length` runs in linear time, but does not allow predicting how much real time is consumed by a call to `length`. Thus, this specification is already "asymptotic" in a certain sense, i.e., up to a constant factor. 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 requirement, 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. Note that one might be tempted to say that this new specification is "more general", but that is not true; it is actually "less general" (i.e. a consequence of the initial specification). It is "more abstract" and "less precise".

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 $a \cdot |l| + b$. On this particular example, these two cost expressions are equivalent. However, in the more general case, cost expressions with "abstract multiplicative constants" can quickly become unwieldy (for instance, $a \cdot n^3 + b \cdot n^2 + c \cdot n \log n + d \cdot n + e$), whereas one is typically only interested in the asymptotic behavior ($O(n^3)$).

It is preferable to state at a more abstract level that the cost is asymptotically dominated by $\lambda n.n$: such a style is more readable and scales to situations where multiple parameters and complex cost expressions are involved. Thus, we propose the following statement:

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

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.

One could imagine defining a notation of the form $\{\$O(|l|)\}$ (`length` l) $\{...\}$ for statements of the same form as above, but we believe that it would be deceptive with respect to the nature of O, which denotes a relation between functions (or equivalently, a set of functions), not a first class resource that can be part of a triple precondition. Such a notation would also be limited to first-order triples, and would not extend to specifications for higher-order functions.)

Thereafter, we refer to the function *cost* as the *concrete cost* of `length`, as opposed to the *asymptotic bound*, represented here by the function $\lambda n.\, n$. The specification asserts that there exists such a concrete cost function *cost*, which is $O(n)$, and such that $\mathit{cost}(|l|)$ credits suffice to justify the execution of `length` l. The fact that n represents $|l|$ is expressed by applying *cost* to $|l|$ in the precondition. Thus, $\mathit{cost}(|l|)$ is an upper bound on the actual number of `pay()` instructions that are executed at runtime. Note that we rely for now on the "usual" notion of O on functions from $\mathbb{Z}$ to $\mathbb{Z}$, and give a formal definition later in Chapter 5.

In general, it is up to the user to choose what the parameters of the cost analysis should be and what these parameters represent.

The example of the Bellman-Ford algorithm (§7) illustrates this.

4.1.2 Modularity in Action: Composing Specifications

A program specification is *modular* if it provides a layer of abstraction isolating from irrelevant details of the program implementation. In other words, a client of the specification should be robust with respect to benign changes in the specified program. In our setting, we are interested in the *asymptotic* complexity of programs: specifications should be robust with respect to changes that do not impact the asymptotic behavior of a program.

Specifications and proofs of specifications are therefore tightly coupled: the modularity of a specification can only be evaluated by using it in the proof of another specification.

Thereafter, we make the discussion of Section 4.1.1 more concrete by illustrating how one might prove specifications for (simple) programs that use the `length` function, depending on the style of specification chosen for it.

Example: a straight-line function

Consider the straight-line (toy) function `length2`, which calls `length` two times and sums up the results:

```
let length2 l =
  length l + length l
```

Specifications using an explicit number of credits. In this case, we consider the specification for `length` that states that it requires exactly $|l| + 1$ time credits. To state a specification for `length2` in the same style, one must guess up front that a sufficient number of credits is $2 \cdot |l| + 3$. We get the following specification:

$$\forall(A : \mathsf{Type})(l : \mathsf{list}\, A).\ \{\, \$(2 \cdot |l| + 3) \,\}\ (\texttt{length2}\ l)\ \{\lambda y.\, [\, y = 2 \cdot |l| \,]\}$$

The outline of a proof of this specification would be:

Calling "`length2 l`" costs:

- 1 (for entering the function)
- $(|l| + 1)$ (for calling `length l`)
- $(|l| + 1)$ (for calling `length l`)

It remains to show that: $1 + (|l| + 1) + (|l| + 1) \leq 2 \cdot |l| + 3$. Because the inequality holds trivially, we are done. Recall that we only count functions calls, and consider `(+)` to be a primitive operation.

The main difficulty of the proof is perhaps the manual accounting of credits that must be performed simply to be able to write the specification, by adding up the cost of all sub-functions. The rest of the proof is mechanical. More importantly, this proof outline highlights the fact that the specification as well as the proof of `length2` are here tied to

the exact number of credits required by `length`. Changing it to e.g. $|l| + 2$ would require updating possibly both the specification and the proof of `length2`. In practice, reasoning on bigger programs would involve going back and forth between the specification and the proof, to adjust the constants in the specification while developing the proof.

Intermediate definitions. One idea to alleviate the issue of manual accounting of credits is to introduce intermediate definitions. For instance, one might reformulate the specifications above as follows:

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

$$\mathit{length2\_cost}(n) \triangleq 2 \cdot \mathit{length\_cost}(n) + 1$$
$$\forall(A : \mathsf{Type})(l : \mathsf{list}\, A).\ \{\, \$\mathit{length2\_cost}(|l|)\,\}\ (\texttt{length2}\ l)\ \{\lambda y.\, [\, y = 2 \cdot |l|\,]\}$$

This idea is interesting to keep in mind. However, as is, it seldom constitutes an improvement. It indeed mildly alleviates the issue of accounting credits when stating a specification and proving it, but only pushes it for later. To get any information about the complexity of a program (e.g. is *length2_cost* an affine function?), one has to transitively unfold all the intermediate definitions involved before obtaining a closed cost expression. Performing approximations on the cost function is also harder in this setting (for example, bounding the cost of a loop by the number of iterations times a bound on the cost of the body, instead of expressing it as an iterated sum).

Specifications using abstract constants. Let us move to the more abstract specification style, in which "`length l`" is advertised to cost $a \cdot |l| + b$, *for some constants* a *and* b. We can then state that "`length2 l`" should cost $a' \cdot |l| + b'$ for some a' and b'.

$$\exists(a', b' : \mathbb{Z}).\ \forall(A : \mathsf{Type})(l : \mathsf{list}\, A).\ \{\$(a' \cdot |l| + b')\}\ (\texttt{length2}\ l)\ \{\lambda y.\, [\, y = 2 \cdot |l|\,]\}$$

Note that at this stage, the need for unique names (a vs. a', b vs. b') reveals the fact that during the proof of `length2` we need to talk about "the a and b" from the specification of `length`, in order to construct a' and b'. We will clear things up in Section 6.2.

An outline of the proof would then be:

Let us delay the choice of a' and b', and take a and b to be the constants from the specification of `length`. Then, calling "`length2 l`" costs:

- 1 (for entering the function)
- $(a \cdot |l| + b)$ (for calling `length l`)
- $(a \cdot |l| + b)$ (for calling `length l`)

The overall cost is equal to $2a \cdot |l| + 2b + 1$. Choosing a' to be $2a$ and b' to be $2b + 1$ completes the proof.

The existential quantification on a' and b' makes it possible to abstract away the specification from irrelevant details of the implementation. This is unlike the previous specification, where one could for instance read from the specification that `length` is

called twice by `length2`. In fact, `length` and `length2` now have the same specification complexity-wise, expressing the fact that their cost is an affine function of $|l|$.

Specifications using asymptotic complexity bounds. Finally, specifications that use asymptotic complexity bounds do abstract over the whole cost expression, but export an asymptotic bound for it. This style of specifications is in fact quite similar to the idea of introducing intermediate definitions for the cost functions—except that the definitions are made abstract, and only their asymptotic behavior is published to the client.

Unsurprisingly, in this setting the specification for `length2` is very similar to the specification for `length`:

$$\exists \mathit{length2\_cost} : \mathbb{Z} \to \mathbb{Z}.\\ \begin{cases} \mathit{length2\_cost} \in O(\lambda n.n) \\ \forall (A : \mathsf{Type})(l : \mathsf{list}\, A).\ \{\$\mathit{length2\_cost}(|l|)\}\ (\texttt{length2}\ l)\ \{\lambda y.\, [\, y = 2 \cdot |l|\,]\} \end{cases}$$

Or informally, "`length2` is $O(|l|)$". It is tempting to jump the gun and sketch a proof by reasoning at the level of O, stating that the cost of `length2 l` is $1 + O(|l|) + O(|l|)$, which is equal to $O(|l|)$. However, formally speaking, the proof must be performed in two separate phases. First, a cost expression is computed by analyzing the code of `length2`; this cost expression provides an expression for *length2_cost*. Then, an asymptotic bound is proved for this cost expression.

Let us delay the choice of *length2_cost*, and take *length_cost* to be the cost function from the specification of `length`. Let us proceed with the proof of the Hoare-triple for `length2`. Calling "`length2 l`" costs:

- 1 (for entering the function)
- $\mathit{length\_cost}(|l|)$ (for calling `length l`)
- $\mathit{length\_cost}(|l|)$ (for calling `length l`)

The overall cost is equal to $1 + 2 \cdot \mathit{length\_cost}(|l|)$. Choosing *length2_cost* to be $\lambda n.\, 1 + 2 \cdot \mathit{length\_cost}(n)$ completes the proof of the Hoare triple. To complete the whole proof, there is one proof obligation left: $\mathit{length2\_cost} \in O(\lambda n.\, n)$, which unfolds to $\lambda n.\, 1 + 2 \cdot \mathit{length\_cost}(n) \in O(\lambda n.\, n)$. This immediately follows from the specification of `length`, which states that $\mathit{length\_cost} \in O(\lambda n.\, n)$.

4.1.3 The Case of Cost Functions with Several Parameters

Similarly to the single-parameter case, it is possible to prove and compose specifications that use abstract constants (of the form $a \cdot n + b \cdot m + c$). As was previously mentioned, the shortcomings are verbosity and a relative lack of modularity.

The case of asymptotic specifications using O bounds is more interesting. Recall the example function introduced in §3.2:

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

```
let f n = g (0, n)
let h m = g (m, 0)
```

A cost function for `g` depends on both n and m, and therefore lives in $\mathbb{Z}^2 \to \mathbb{Z}$. How should one specify the complexity of `g` using a O bound over $\mathbb{Z}^2$ (i.e. a "multivariate O")? In fact, how should one interpret multivariate O bounds, and what are the implications with respect to the modularity of specifications? Single-variable O bounds are well understood, but it seems there is some leeway in how multivariate O bounds can be interpreted. Does the bound have to hold when *all* parameters go to infinity, or maybe only at least some of them?

If one interprets multivariate O bounds as requiring all parameters to go to infinity, then $O(mn)$ is an appropriate asymptotic bound for specifying the complexity of `g`. However, such a specification is not re-usable: in particular, it is not applicable to analyze the asymptotic complexity of `f`, which calls `g` with a fixed value for m.

Instead, one could decide to require a two-variables bound $O(p(m,n))$ to hold as soon as "n goes to infinity and m can be set to any non-negative integer". Under that assumption, an appropriate bound for `g` is $O(mn+n)$ (not $O(mn)$!—if m is zero then the cost of `g` is $O(n)$, not $O(0)$). Then, this (quite ad-hoc) interpretation of O can be used to derive that `f` runs in $O(n)$—it is more modular than the previous one in that sense.

We see that the choice of interpretation has a direct impact on the *modularity* of specifications. Unfortunately, the informal syntax of the O notation does not clearly distinguish between a bound $O(mn)$ (where both m and n go to infinity) and $O(mn+n)$ (where only n goes to infinity). In our formal definition of O (Section 5), the difference is made explicit. Our formal account of the O notation is parameterized by a *filter*, which characterizes how "going to infinity" should be interpreted for each of the parameters. In the uni-variate case, the choice of filter is canonical and therefore not an issue, but in the multi-variate case it does matter, in particular when considering specializing some arguments to constants. The two aforementioned asymptotic bounds for `g` correspond to two different choices of filters: for one filter $O(mn+n)$ can be simplified to $O(mn)$, but not for the other one.

Let us consider a perhaps less artificial example: the `Array.append` function, which concatenates two arrays. Because `Array.append` needs to copy both of its arguments into the result array, its complexity depends on the size of both input arrays. Informally, "`Array.append` runs in $O(m+n)$", where m, n are the sizes of its first and second argument, respectively.

```
(* In Array.mli *)
val append : 'a array -> 'a array -> 'a array

let f1 a = Array.append [|1; 2|] a
let f2 a = Array.append a [|0|]
```

As with the previous example `g`, a formal specification for `Array.append` using the O notation over $\mathbb{Z}^2$ should allow specializing either of its arguments to a constant. For instance, one should be able to deduce from the specification that both functions `f1` and `f2` (as defined above) run in linear time.

Interestingly, it is also possible to give a formal specification for `Array.append` using a single-variable O. One can state that `Array.append` runs in $O(t)$, where t is taken to be

"$m + n$", i.e. the sum of the sizes of the two input arrays. This specification conveniently allows specializing either m or n to be constants, as long as their sum goes asymptotically to infinity... This is another, subtly different, interpretation of the informal "$O(m + n)$" statement. Note that we anticipate that such a specification style might be too coarse grained to be a good idea in practice, and would lead to specifications that compose poorly. Indeed, consider the case of the sequence of a function running in "$O(u)$", with $u = m \cdot n$, followed by a function running in "$O(t)$", with $t = m + n$. The cost of the sequence would need to be expressed as a function of $m + n$ and $m \cdot n$, which seems inconvenient.

At this point, we reach the following temporary conclusion: in the case of several parameters, an informal O statement can be interpreted in several subtly different ways. Formally, this imprecision is accounted for by different choices of *filters*, a parameter of our formal definition of O. Our program specifications have an additional degree of freedom—the choice of a filter—which makes the filter an integral part of the specification proved for a program. In other words, we have not hardwired in the framework a choice of filter that ought to be good for any specification. In Section 5.4, we discuss choices of filters that we believe cover most practical use-cases. Nevertheless, the choice of a filter is ultimately left to the user that states and proves the specification, allowing for special-purpose filters to be used when required in specific cases.

With respect to the examples at hand, let us slightly anticipate, and say that it is always possible to pick a filter that allows specializing any number of parameters—possibly at the price of a more verbose (precise) asymptotic bound. Filters solve an additional challenge, which is to express asymptotic bounds that involve *arbitrary dependencies between the parameters*: for instance, in the specification of the Bellman-Ford (§7.4) algorithm, we use a filter that restricts the domain to pairs (m, n) such that $m \leq n^2$, an inequality which always holds whenever m and n denote the number of edges and vertices of a graph, respectively. With this filter, $\lambda(m, n).\, mn$ is dominated by $\lambda(m, n).\, n^3$.

Our formal account of O and filters is fully detailed in Section 5. The discussion on filters with multiple parameters is resumed in its full details in Section 5.4, after the necessary definitions have been introduced.

4.2 Challenges with Abstract Cost Specifications

In Section 4.1.1 we argue in favor of *abstract* cost specifications rather than *concrete* ones, for the sake of modular reasoning. More specifically, we argue for specifications where the expression of the cost function is abstracted away, and only an asymptotic bound is exposed. In this section, we show that this specification style needs some fine-tuning: exporting a couple additional side-conditions is in fact required to make it usable in practice.

Let us recall the specification for `length` proposed at the end of Section 4.1.1.

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

We claim that this specification is not usable *as is*, because it is too abstract with regard to the function *cost*. The solution is to expose additional properties that the *cost*

function satisfies, that would otherwise be hidden behind the existential quantifier. We identify below two properties (monotonic, nonnegative) that appear to be key for the compositionality of specifications that use asymptotic complexity bounds.

4.2.1 Why Cost Functions Must Be Monotonic

One key reason why cost functions should be monotonic has to do with the "avoidance problem". Let us illustrate the problem. The cost of the program "`let x = e1 in e2`" must not refer to the variable `x`, which is a local variable. Moreover, the cost of such a program (`let x = e1 in e2`) is computed as the cost of `e1` plus the cost of `e2`. However, because `e2` may typically refer to `x`, the cost of `e2` might depend on `x` as well. Therefore, we have a problem: the cost expression for `e2` cannot be used directly to express the cost of the whole program.

More generally, the problematic cost expression is of the form $E[|x|]$, where $|x|$ represents some notion of the "size" of the data structure denoted by x, and E is an arithmetic context, that is, an arithmetic expression with a hole. Fortunately, it is often possible to provide an upper bound on $|x|$. This upper bound can be exploited if the context E is monotonic, i.e., if $x \leq y$ implies $E[x] \leq E[y]$. Because the hole in E can appear as an actual argument to an abstract cost function, we must record the fact that this cost function is monotonic.

As a more concrete example, let us consider the following OCaml function, which counts the positive elements in a list of integers. It does so, in linear time, by first building a sublist of the positive elements, then computing the length of this sublist.

```
let count_pos l =
  (* pay (); *)
  let l' = List.filter (fun x -> x > 0) l in
  List.length l'
```

How would one go about proving that this code actually has linear time complexity? On paper, one would informally argue that the cost of the sequence `pay()`, `filter`, `length` is $O(1) + O(|l|) + O(|l'|)$, then exploit the inequality $|l'| \leq |l|$, which follows from the semantics of `filter`, and deduce that the cost is $O(|l|)$.

In a formal setting, though, the problem is not so simple. Assume that we have two specification lemmas for `List.length` and `List.filter`, which describe the behavior of these OCaml functions and guarantee that they have linear-time complexity. For brevity, let us write just g and f for the cost functions associated with the specifications of `List.length` and `List.filter`, respectively. Also, at the mathematical level, let us write l^+ for the sublist of the positive elements of the list l. It is easy enough to check that the cost of the expression "`pay(); let l' = ... in List.length l'`" is $1 + f(|l|) + g(|l'|)$. The problem, now, is to *find an upper bound* for this cost *that does not depend on* l', a local variable, and to verify that this upper bound, *expressed as a function of* $|l|$, is dominated by $\lambda n.\, n$. Indeed, this is required in order to establish an asymptotic complexity specification about `count_pos`.

What might this upper bound be? That is, which functions *cost* of $\mathbb{Z}$ to $\mathbb{Z}$ are such that:

(A) $1 + f(|l|) + g(|l'|) \leq cost(|l|)$ (in the scope of the local variable l')

(B) $cost \in O(\lambda n.\, n)$

Three potential answers come to mind:

1. Within the scope of l', the equality $l' = l^+$ is available, as it follows from the postcondition of `filter`. Thus, within this scope, $1 + f(|l|) + g(|l'|)$ is provably equal to $\mathit{let}\ l' = l^+\ \mathit{in}\ 1 + f(|l|) + g(|l'|)$, that is, $1 + f(|l|) + g(|l^+|)$. This remark may seem promising, as this cost expression does not depend on l'. Unfortunately, this approach falls short, because this cost expression cannot be expressed as the application of a closed function *cost* to $|l|$. Indeed, the length of the filtered list, $|l^+|$, is not a function of the length of l. In short, substituting local variables away in a cost expression does not always lead to a usable cost function.

2. The inequality $|l^+| \leq |l|$ holds in general. Within the scope of l', it follows that $|l'| \leq |l|$, because $l' = l^+$. Thus, inequality (A) can be proved, provided we take:

$$\mathit{cost} = \lambda n.\ \max_{0 \leq n' \leq n}\ 1 + f(n) + g(n')$$

   Furthermore, for this definition of *cost*, the domination assertion (B) holds as well. The proof relies on the fact that asymptotic bounds for g also hold for $\widehat{g}$ under some assumptions, where $\widehat{g}$ is $\lambda n.\ \max_{0 \leq n' \leq n}\ g(n')$ (see Section 4.2.3). Although this approach seems viable, and does not require the function g to be monotonic, it is a bit more complicated than we would like.

3. Let us now assume that the function g is monotonic, that is, nondecreasing. As before, within the scope of l', the inequality $|l'| \leq |l|$ is available. Thus, the cost expression $1 + f(|l|) + g(|l'|)$ is bounded by $1 + f(|l|) + g(|l|)$. Therefore, inequalities (A) and (B) are satisfied, provided we take:

$$\mathit{cost} = \lambda n.\ 1 + f(n) + g(n)$$

Between approaches 2 and 3, we believe that approach 3 is the simplest and most intuitive, because it allows us to easily eliminate l', without giving rise to a complicated cost function, and without the need for a running maximum.

However, this approach requires that the cost function g be monotonic. This explains why we build a monotonicity condition in our program specifications (Section 6.2). Another motivation for doing so is the fact that some lemmas (such as Lemma 5.3.10, which allows reasoning about the asymptotic cost of an inner loop) also have monotonicity hypotheses.

One may be worried that there exist concrete cost functions that are not monotonic. We detail in Section 4.2.3 how one can solve this issue by using a "running maximum" function in place of a non-monotonic concrete cost function.

4.2.2 Why Cost Functions Must Be Nonnegative

A cost function describes the number of steps a program takes: on the domain of the parameters of the program, it is necessarily nonnegative. We could embed this fact in the type of cost functions and set their codomain to be $\mathbb{N}$, at the price of possibly using coercions between $\mathbb{Z}$ and $\mathbb{N}$ in the concrete expression of cost functions. In our framework,

we choose the codomain of cost functions to be $\mathbb{Z}$ (as can be seen in the asymptotic specification introduced in Section 4.1.1). This is for practical reasons: integers coming from programs can be negative; we model them as mathematical integers in $\mathbb{Z}$. To avoid many tedious conversions between $\mathbb{Z}$ and $\mathbb{N}$, we decide to only handle integers in $\mathbb{Z}$ throughout the framework.

In that setting, the fact that a cost function is nonnegative on some domain is lost after the specification abstracts over the cost function. As an illustrative example, let us consider the following program:

```
let g () =
  if Random.bool () then f () else ()
```

Let us call *cost_f* the cost function for the program `f`. It seems natural to state that $\lambda().\ 1 + \mathit{cost\_f}()$ is a good enough cost function for `g`, since the "else" branch has cost zero. However, if we do not know that $\mathit{cost\_f}()$ is nonnegative, this cost function is incorrect: the cost function for `g` must be instead $\lambda().\ 1 + \max(\mathit{cost\_f}(), 0)$. Indeed, the cost of an "`if`" conditional is the maximum of the cost of each branch. If $\mathit{cost\_f}()$ is not known to be nonnegative, then $1 + \max(\mathit{cost\_f}(), 0)$ cannot be simplified into $1 + \mathit{cost\_f}()$.

To recover somewhat intuitive reasoning principles, we require cost functions exported by specifications to be nonnegative everywhere.

4.2.3 Proving that Cost Functions are Monotonic and Nonnegative

The reader may be worried that the monotonicity and nonnegativity requirements are too strong. In practice, there might exist concrete cost functions that are not monotonic. This may be the case, in particular, of a cost function f that is obtained as the solution of a recurrence equation. Similarly, concrete cost functions might be negative outside of their domain of interest. Consider the concrete cost function for `length`: $\lambda n.\ 2n + 3$. It is positive on the domain of interest (nonnegative values of n, since in the specification "n" corresponds to the length of a list), but negative for all values of n smaller than -2.

For most practical purposes, the monotonicity and nonnegativity requirements are in fact not an issue. Under some reasonable assumptions, it is always possible to make the concrete cost function monotonic and nonnegative *a posteriori*, without polluting the proof of the specification. For instance, the proof of the `length` function can be carried using the intuitive concrete cost expression $\lambda n.\ 2n + 3$. However, the specification requires providing some cost function which is $O(n)$, monotonic, and *nonnegative everywhere*. Fortunately, it is possible to instantiate the specification with the cost function $\lambda n.\ \max(2n + 3, 0)$ at the very end of the proof, in a systematic fashion, that can mostly be made transparent to the user.

This relies on the following principle: if the cost function appears only in the precondition of the specification (an assumption that holds for all but exotic specifications), then one can always derive a specification with a greater cost function. On the previous example, $\lambda n.\ \max(2n + 3, 0)$ is always greater than $\lambda n.\ 2n + 3$ point-wise, so it can be used to derive a new specification for `length`. In other words, the following rule is derivable in Separation Logic with Time Credits.

$$\bar{f}(x) \triangleq \max(0, f(x))$$

(1) $\forall x.\ \bar{f}(x) \geq 0$
(2) $\forall x.\ f(x) \leq \bar{f}(x)$
(3) $\forall g.\ f \in O(g) \implies \bar{f} \in O(g)$

Figure 4.1: Definition and properties of $\bar{f}$

$$\frac{n \in D \implies \{\$f(n) \star P\}\ e\ \{Q\} \qquad \forall x \in D, f(x) \leq f'(x)}{n \in D \implies \{\$f'(n) \star P\}\ e\ \{Q\}}$$

WeakenCostF

The WeakenCostF rule asserts that it is always sound to provide more time credits than necessary in the precondition: it is equivalent to not using them and throwing them away afterwards. Technically, the rule is a consequence of the splitting rule for time credits, the frame rule and the garbage collection rule which allows throwing away affine resources. This principle allows us to prove a Separation Logic specification for a program using some convenient function f, and then "patch" the function afterwards, as long as the modifications only increase the value of the cost function on the domain of interest D.

Obtaining a nonnegative function from a given cost function f can be done by considering the function $\max(f, 0)$, written $\bar{f}$. Figure 4.1 gives the definitions and key properties of $\bar{f}$. For any function f, the function $\bar{f}$ is nonnegative (1), always greater or equal than f (2), and exhibits the same asymptotic behavior as f (3).

To obtain a monotonic function, one can consider the family of "running maxima", written $\widehat{f}$. For any values x_0 and x, $\widehat{f}(x_0)(x)$ computes the maximum value of f between x_0 and x. Figure 4.2 gives the definitions and properties of $\widehat{f}$. For any function f and starting value x_0, the function $\widehat{f}(x_0)$ is monotonic (1), greater than f after x_0 (2), and exhibits the same asymptotic behavior as f (3, with a side-condition).

Both $\bar{f}$ and $\widehat{f}(x_0)$ can be used as the f' function of rule WeakenCostF. In the case of $\widehat{f}(x_0)$, one has to prove additionally that the domain D of relevant inputs is included in the interval $[x_0, +\infty)$. Indeed, an iterated maximum can only be defined from a given starting point. This requirement is again not an issue in practice, because parameters of cost functions typically represent the size of input data structures, and therefore live in $[0, +\infty)$.

Both $\bar{f}$ and $\widehat{f}$ preserve asymptotic bounds. This means one can prove an asymptotic bound about the convenient function cost function f, and transfer it (almost) for free on the "patched" function. There is a side-condition when $\widehat{f}$ is involved (Figure 4.2, property 3): the asymptotic bound must eventually be monotonic. All common asymptotic bounds, e.g. n, n^2, $n \log n$, 2^n, ..., satisfy this requirement.

$$\widehat{f}(x_0)(x) \triangleq \max_{x_0 \le z \le x} f(z) \quad \text{if } x \ge x_0, \qquad f(x_0) \quad \text{otherwise}$$

(1) $\forall x_0.\quad \widehat{f}(x_0)$ is monotonic

(2) $\forall x_0\ x.\ x \ge x_0 \implies f(x) \le \widehat{f}(x_0)(x)$

(3) $\forall x_0\ g.\ f \in O(g) \wedge g$ is ultimately monotonic $\implies \widehat{f}(x_0) \in O(g)$

Figure 4.2: Definition and properties of $\widehat{f}$

4.3 Summary: Constraints and Design Choices

In this chapter, we ask the question "What is a good specification for the complexity of a program?". It is tempting to answer that it must be the *most general* specification. However, this idea quickly falls short: formally speaking, the most general specification for a program provides no abstraction and exposes all the inner details of a program. We explain in Section 4.1.1 that this not desirable, because it is *not modular*. Hence, our criterion for "good specifications" is a bit more subtle than finding the most general specifications. Instead, it consists in finding a good balance between specifications that do not abstract anything (and therefore are not modular), and specifications that abstract everything (and therefore are not usable).

With respect to program complexity, it seems that this balance is met when using the O notation, which is widely known and used in particular in (informal) analysis of the complexity of algorithms. Unfortunately, even though the O notation appears to be very convenient when writing informal proofs, this is less true in a formal setting. Specifically, the O notation does not provide a level of abstraction that is sufficient in itself, and can be used consistently throughout the analysis of a program. To the question "Is it always possible to prove a O specification by composing other O specifications in a bottom-up fashion?", Chapter 3 answers: "No." (because of recursive functions, for instance). To establish a program specification that uses O, on must first reason at a lower level of abstraction about a concrete cost function, possibly defined using other cost functions. Only afterwards can a bound be proven and exported as part of the specification. This is illustrated in Section 4.1.2.

When proving program specifications, one must then reason about concrete cost functions. Yet, at the specification level, asymptotic complexity bounds are still our best bet: as discussed in Section 4.1.1, complexity specifications that either are fully explicit or use abstract constants are not satisfactory.

Then, we must ensure that we state asymptotic complexity specifications that are abstract enough but still usable. It must be possible to verify some (possibly complex) program that uses `List.length` by relying on an asymptotic complexity specification published for `List.length` (for instance in a "standard library" of program specifications). Having to unfold the definition of the concrete cost function of `List.length` in the proof would be a failure in the modularity of our specifications.

For that purpose, our program specifications are of the form "$\exists\, cost.\ \ldots$": the expression of the cost function is made abstract by the system. As detailed in Section 4.2, additional properties about *cost* need to be exported for such a specification to be usable

(monotonicity and nonnegativity). In the case of a cost function with several parameters, there is an additional degree of freedom in the choice of the *filter* parameter of the $O()$ notation. As explained in Section 4.1.3, this choice also has an influence on the modularity or re-usability of the specification.

In the end, reasoning about the complexity of a program is a two step process. First, one reasons about the program itself, matching it against a *cost function*, which may be defined with the help of other abstract cost functions. Then, one reasons on the cost function, in order to establish an asymptotic bound and other side-conditions that are eventually exported as part of the specification.

Chapter 5

Formalizing the O Notation

The asymptotic complexity specifications that we advertised in the previous chapter rely on a well-known notion of asymptotic domination between functions. Our uses of the "O notation" have been left informal up to this point. In this chapter, we give a formal account for the O notation, focusing on its use for the analysis of the asymptotic complexity of programs.

Because the O notation serves as an abstraction barrier in our program specifications, we need to formalize not only domination properties between well-known mathematical functions, but also reasoning principles on O that match common patterns of program composition. For instance, we need to be able to deduce asymptotic bounds for the complexity of a function implemented by composing, specializing or iterating other functions in a for-loop.

On a related note, even though the O notation and its properties are well-understood in the case of functions of one variable, the performance of many algorithms is best described as a function of several parameters. For example, the asymptotic behavior of many graph algorithms depends both on the number m of edges and the number n of vertices of the input graph. Many textbooks omit an explicit definition of domination between functions of several variables. Instead, they implicitly assume that there exists a notion of domination that satisfies a number of desirable properties. As shown by Howell [How08], the devil is in the details: if stated without care, the desired properties can be inconsistent. We therefore study the question of what should a definition of multivariate O be, in the context of program verification.

5.1 Domination Relation

In many textbooks, the fact that f is bounded above by g asymptotically, up to a 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. 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 [AD04]; this is also the notation we used informally in the previous chapters. Yet, in the rest of this document, we will prefer to think directly in terms of a *domination* preorder between functions. (In Coq, the two are essentially equivalent, since a set is modeled as a predicate.) 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 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* [Bou95]. 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 $|f(x)| \leq c\,|g(x)|$ 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, $|f(x)| \leq c\,|g(x)|$"

In mathematical notation, we would like to write: $\exists c.\ \mathbb{U}x.\ |f(x)| \leq c\,|g(x)|$. 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 (§5.2) and sum up what has been explained so far:

Definition 5.1.1 (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:*

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

5.2 Filters

The definitions and lemmas that appear in this section have been formalized in Coq as part of our `bigO` library [Gué18b, src/Filter.v].

5.2.1 Definition

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 [Bou95] 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.

[1] At this time, we require the codomain of f and g to be $\mathbb{Z}$. Following Avigad and Donnelly [AD04], 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 of this special case, however, requiring the inequality to hold everywhere is usually too strong.

Definition 5.2.1 (Filter) *A* filter [Bou95] *on a type A is an object $\mathbb{U}$ of type $\mathcal{P}(\mathcal{P}(A))$ that enjoys the following three properties, where $\mathbb{U}x.P$ is short for $\mathbb{U}\,(\lambda x.P)$:*

$$\begin{array}{lcll} (1) & (P_1 \Rightarrow P_2) \Rightarrow \mathbb{U}x.P_1 \Rightarrow \mathbb{U}x.P_2 & & \textit{(covariance)} \\ (2) & \mathbb{U}x.P_1 \wedge \mathbb{U}x.P_2 \Rightarrow \mathbb{U}x.(P_1 \wedge P_2) & & \textit{(stability under binary intersection)} \\ (3) & \mathbb{U}x.\mathit{True} & & \textit{(stability under 0-ary intersection)} \end{array}$$

These properties 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 (2) and (3), 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 (3) together imply $\forall x.P \Rightarrow \mathbb{U}x.P$.

One can additionally require a filter to be *proper* (Bourbaki [Bou95] directly defines filters as being proper filters). Proper filters satisfy the property that if P holds when x is large enough, then P should hold of some x. In classical logic, it would be equivalent to $\neg(\mathbb{U}x.\mathrm{False})$.

Definition 5.2.2 (Proper Filter) *A* proper filter *on a type A is a filter $\mathbb{U}$ on A that satisfies the additional property:*

$$\mathbb{U}x.P \Rightarrow \exists x.P \quad \textit{(nonemptiness)}$$

In our formalization, we originally defined filters as being proper filters, following Bourbaki. However, we observed that—for our program verification purposes at least—we never actually needed the nonemptiness property, and having to establish it was the source of a few tedious side-conditions when constructing filters. It seems more practical to define filters following Definition 5.2.1 (as in the formalizations of filters in Isabelle/HOL [HIH13] or Coquelicot [BLM15]), and only when needed assert that a given filter must be proper as an additional property on the side (as in mathcomp-analysis [ACR18]).

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.

5.2.2 Examples of Filters

When $\mathbb{U}$ is a *universal filter*, $\mathbb{U}x.Q(x)$ is (by definition) equivalent to $\forall x.Q(x)$. Thus, a predicate Q is "ultimately true" if and only if it is "everywhere true". In other words, the universal quantifier is a filter (we write it $\top$ because of Definition 5.2.7 below).

Definition 5.2.3 (Universal filter) *Let T be a type. Then $\lambda Q.\forall x.Q(x)$ is a filter on T.*

At the opposite end of the spectrum, when $\mathbb{U}$ is the *trivial filter*, $\mathbb{U}x.Q(x)$ holds for any property Q.

Definition 5.2.4 (Trivial filter) *Let T be a type. Then $\lambda Q.\forall x.\mathit{True}$ is a filter on T.*

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

Definition 5.2.5 (Order filter) *Let $(T, \leq)$ be a nonempty ordered type, such that every two elements have an upper bound. Then $\lambda Q.\exists x_0.\forall x \geq x_0.\ Q(x)$ is a filter on T.*

The order filter associated with the ordered type $(\mathbb{Z}, \leq)$ 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 5.1.1 with the filtered type $\mathbb{Z}$, we recover the classic definition of domination between functions of $\mathbb{Z}$ to $\mathbb{Z}$:

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

Another filter that is useful to consider is the restriction of an existing filter to some sub-domain characterized by a predicate P.

Definition 5.2.6 (Induced filter) *Let A be a filtered type, and P a predicate on A. Then $\lambda Q.\ \mathbb{U}_A x.(P(x) \Rightarrow Q(x))$ is a filter on A, that we write $(A \mid P)$.*

It is also interesting to note that filters on a given type T form a complete lattice (these filters are not necessarily *proper*, in particular the bottom element of the lattice—the trivial filter—is not a proper filter). The definitions presented here follow the formalization of filters from Isabelle/HOL [HIH13].

Definition 5.2.7 (Partial order on filters, supremum, infimum) *Let T be a type. Then, let us define:*

- *The partial order on filters $F \sqsubseteq G \triangleq \forall Q.\ \mathbb{U}_G Q \Rightarrow \mathbb{U}_F Q$.*

  $F \sqsubseteq G$ means that the filter F is finer *than the filter G; alternatively, it means that G is* more demanding *than F.*

- *The top filter $\top \triangleq \lambda Q.\ \forall x.Q(x)$ (the universal filter)*

- *The bottom filter $\bot \triangleq \lambda Q.\ \forall x.\mathit{True}$ (the trivial filter)*

- *Supremum $F \vee G \triangleq \lambda Q.\ \mathbb{U}_F Q \wedge \mathbb{U}_G Q$*

- *Infimum $F \wedge G \triangleq \lambda Q.\ \exists R_1 R_2.\ \mathbb{U}_F R_1 \wedge \mathbb{U}_G R_2 \wedge (\forall x.\ R_1(x) \wedge R_2(x) \implies Q(x))$*

- *Supremum of a class of filters S: $\bigvee S \triangleq \lambda Q.\ \forall F.\ S(F) \implies \mathbb{U}_F Q$*

- *Infimum of a class of filters S: $\bigwedge S \triangleq \bigvee(\lambda F.\ \forall G.\ S(G) \implies F \sqsubseteq G)$*

Lemma 5.2.8 (Lattice of filters) *Let T be a type. Then, filters on T form a complete lattice with respect to Definition 5.2.7.*

Lemma 5.2.9 (Contravariance of domination) *Let F_1, F_2 be filters on a type T. Then $f \preceq_{F_2} g$ and $F_1 \sqsubseteq F_2$ imply $f \preceq_{F_1} g$.*

We now turn to the definition of filters on a product type $A_1 \times A_2$. Such filters are a pre-requisite for defining domination between functions of several variables.

Assuming A_1 and A_2 are filtered types, one can consider the following *product filter* construction. Note that this is a generic construction that works for any two filtered types.

Definition 5.2.10 (Product filter) *Let A_1 and A_2 be filtered types. Then*

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

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 [Bou95, Chapter 1, §6.7], the infinite rectangles form a *basis* of the product filter. Another intuition is that, according to a product filter, the two components *independently* go to infinity. 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.

Another way of defining a filter on $A_1 \times A_2$ is by considering a *norm* on elements of that type. For instance, on $\mathbb{Z}^2$, one can consider the norm $\|(x, y)\|_1 \triangleq |x| + |y|$ or the norm $\|(x, y)\|_\infty \triangleq \max(|x|, |y|)$. The exact choice of the norm on $\mathbb{Z}^n$ is in fact irrelevant for asymptotic analysis, since all norms are equivalent up to a multiplicative constant. Thereafter, we use the ∞-norm because it seems slightly simpler to handle. Then, one can define the filter of elements whose norm "goes to infinity". In fact, this construction generalizes to any type A equipped with a "measure" function of type $A \to \mathbb{Z}$.

As a preliminary definition, let us define the *inverse image* of a filter by a function. If f is a function of type $A \to B$, and B is a filtered type, then the inverse image of B by f defines a filter on A, for which elements a of A go to infinity whenever $f(a)$ goes to infinity in B.

Definition 5.2.11 (Inverse image filter) *Let A be a type, B a filtered type, and f a function of type $A \to B$. Then*

$$\lambda Q.\ \exists R.\ \mathbb{U}_B b.\ R(b) \wedge (\forall a.\ R(f(a)) \implies Q(a))$$

is a filter on A, that we write $f^{-1}(B)$.

We can now define the *norm filter* on $\mathbb{Z}^2$ as the inverse image of the filtered type $\mathbb{Z}$ by the norm $\|\cdot\|_\infty$.

Definition 5.2.12 (Norm filter) *The norm filter on type $\mathbb{Z}^2$, written $\|\mathbb{Z}^2\|$, is defined as $\|\cdot\|_\infty^{-1}(\mathbb{Z})$.*

By unfolding Definition 5.2.5 with the filtered type $\mathbb{Z}$ and Definition 5.2.11, we obtain a more natural characterization of the norm filter:

$$\begin{aligned}\mathbb{U}_{\|\mathbb{Z}^2\|}(x,y).\, Q((x,y)) &\iff \exists x_0.\, \forall x, y.\, \|(x,y)\|_\infty \geq x_0 \implies Q((x,y)) \\ &\iff \exists x_0.\, \forall x, y.\, |x| \geq x_0 \vee |y| \geq x_0 \implies Q((x,y)).\end{aligned}$$

Intuitively, for a predicate Q on $\mathbb{Z} \times \mathbb{Z}$ to be ultimately true w.r.t the norm filter, it has to be "true everywhere except on a ball around the origin".

Compared to the product filter $\mathbb{Z} \times \mathbb{Z}$, $\|\mathbb{Z}^2\|$ is strictly finer: we have $\|\mathbb{Z}^2\| \sqsubseteq \mathbb{Z} \times \mathbb{Z}$, but not $\mathbb{Z} \times \mathbb{Z} \sqsubseteq \|\mathbb{Z}^2\|$.

5.3 Properties of the Domination Relation

Many properties of the domination relation can be established with respect to an arbitrary filtered type A. Here are some of the most useful lemmas; there are many more. The definition of the domination relation and the lemmas are part of our our `bigO` Coq library [Gué18b, src/Dominated.v]. As before, f, g and h range over $A \to \mathbb{Z}$. The operators $f + g$, $\max(f, g)$ and $f.g$ denote pointwise sum, maximum, and product, respectively.

Lemma 5.3.1 (Distributivity over Sum) *$f \preceq_A h$ and $g \preceq_A h$ imply $f + g \preceq_A h$.*

Lemma 5.3.2 (Sum) *Assume g_1 and g_2 are ultimately nonnegative, that is, $\mathbb{U}_A\, x.\, f(x) \geq 0$ and $\mathbb{U}_A\, x.\, g(x) \geq 0$ hold. Then, $f_1 \preceq_A g_1$ and $f_2 \preceq_A g_2$ imply $f_1 + f_2 \preceq_A g_1 + g_2$.*

Lemma 5.3.3 (Sum and Max Are Alike) *Assume f and g are ultimately nonnegative. Then, we have $\max(f, g) \preceq_A f + g$ and $f + g \preceq_A \max(f, g)$.*

Lemma 5.3.4 (Multiplication) *$f_1 \preceq_A g_1$ and $f_2 \preceq_A g_2$ imply $f_1.f_2 \preceq_A g_1.g_2$.*

Lemma 5.3.5 (All Constants Are Alike) *If $c_2 \neq 0$, then $\lambda x.c_1 \preceq_A \lambda x.c_2$.*

Lemma 5.3.6 (Constant Functions) *If $g \nearrow_A^{\mathbb{Z}}$, then $\lambda x.c \preceq_A g$.*

The lemma above states that a constant function is dominated by any function g that grows large enough. This is the meaning of the side condition $g \nearrow_A^{\mathbb{Z}}$, which is defined as follows:

Definition 5.3.7 (Limit) *The assertion $p \nearrow_A^B$ is defined as follows:*

$$p \nearrow_A^B \quad \equiv \quad \forall Q,\ (\mathbb{U}_B\, b.\, Q(b)) \Rightarrow \mathbb{U}_A\, a.\, Q(p(a))$$

By definition, $p \nearrow_A^B$ holds if and only if any predicate Q that is ultimately true of b is ultimately true of $p(a)$. In other words, whatever height one wishes b to reach, $p(a)$ is able to reach, when a grows large. Instantiating both A and B with $\mathbb{Z}$, we find that $p \nearrow_{\mathbb{Z}}^{\mathbb{Z}}$ means that p diverges, that is, $p(x)$ tends towards infinity when x tends towards infinity. In other words, this assertion provides a generalized notion of limit between filtered types.

This small set of lemmas is enough to analyze the complexity of many *straight-line* programs. The cost of a sequence of operations is the sum of the cost of the operations, which can be bounded using Lemmas 5.3.2 and 5.3.1. The cost of a conditional branch is reflected into a max, which can be handled using Lemma 5.3.3 (and a few other derived lemmas). Additionally, Lemma 5.3.4 corresponds to Howell's Property 5 [How08]. As noted by Howell, this lemma can be useful in the analysis of a loop when the cost of a loop body is independent of the loop index. (Whereas Howell states this property on $\mathbb{N}^k$, our lemma is polymorphic in the type A.)

5.3.1 Summation Lemmas

In full generality, the analysis of programs involving for-loops requires lemmas that allow bounding iterated sums, where the i-th term of the sum represents the cost of the i-th iteration of the loop. We have a few variants of such lemmas.

Let us start with the simple case where the cost of the i-th iteration only depends on the index i. Then, if one can bound the cost of each individual loop iteration, the sum of the bounds is itself a bound on the cost of the whole loop.

Lemma 5.3.8 (Summation (single parameter case)) *Let f, g range over $\mathbb{Z} \to \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$. Assume the following two properties:*

1. $\mathbb{U}_{\mathbb{Z}} i.\ f(i) > 0$.
2. $\mathbb{U}_{\mathbb{Z}} i.\ g(i) > 0$.

Then,

$$f \preceq_{\mathbb{Z}} g$$

implies

$$\left(\lambda n.\ \sum_{i=i_0}^{n} f(i)\right) \preceq_{\mathbb{Z}} \left(\lambda n.\ \sum_{i=i_0}^{n} g(i)\right).$$

Side-conditions 1 and 2 are used to rule out the pathological case where f or g would eventually be equal to zero, meaning that the sums would in fact consist only of the initial terms of the functions. It seems likely that the lemma could in fact be relaxed to remove side-condition 1, but we have not done so in the Coq development at the moment.

Lemma 5.3.8 enables relatively fine-grained reasoning, because it allows bounding separately each term of the sum. However, in many cases, such level of precision is not needed: it is enough to coarsely bound the sum by its last term, times the number of terms. One can then use the following simpler lemma:

Lemma 5.3.9 (Coarse Summation (single parameter case)) *Let f range over $\mathbb{Z} \to \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$. Assume the following two properties:*

1. $\mathbb{U}_{\mathbb{Z}} i.\ f(i) > 0$.
2. *f is ultimately nondecreasing.*

Then,

$$\left(\lambda n.\ \sum_{i=i_0}^{n} f(i)\right) \preceq_{\mathbb{Z}} \lambda n.\ n \cdot f(n).$$

This lemma additionally requires the cost of the loop f to be ultimately nondecreasing—this requirement is typically easy to satisfy since we require cost functions to be monotonic (§4.2.1).

Lemmas 5.3.8 and 5.3.9 are restricted to the simple case where the cost of a loop iteration only depends on the loop index. Below, we describe variants of these lemmas that work in the more general case, where the cost of a loop iteration may depend on other parameters. In that case, the cost of the loop body and its asymptotic bound (f and g) are multivariate functions. We handle both the case where the asymptotic bound on the loop body is expressed using a product filter and the case where it is expressed using a norm filter on $\mathbb{Z}^k$.

Lemma 5.3.10 (Summation (product filter)) *Let f, g range over $A \to \mathbb{Z} \to \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$. Assume the following three properties:*

1. $\mathbb{U}_A\, a.\ \forall i \geq i_0.\ f(a)(i) \geq 0$.
2. $\mathbb{U}_A\, a.\ \forall i \geq i_0.\ g(a)(i) \geq 0$.
3. *for every a, the function $\lambda i. f(a)(i)$ is nondecreasing on the interval $[i_0, \infty)$.*

Then,

$$\lambda(a, i). f(a)(i) \ \preceq_{A \times \mathbb{Z}}\ \lambda(a, i). g(a)(i)$$

implies

$$\left(\lambda(a, n).\ \sum_{i=i_0}^{n} f(a)(i)\right) \ \preceq_{A \times \mathbb{Z}}\ \left(\lambda(a, n).\ \sum_{i=i_0}^{n} g(a)(i)\right).$$

Lemma 5.3.10 uses the product filter on $A \times \mathbb{Z}$ in its hypothesis and conclusion. It corresponds to Howell's property 2 [How08]. The variable i represents the loop index, while the variable a collectively represents all other parameters in scope, so the type A is usually instantiated with a tuple type (an example appears in §7.3).

Let us explain informally the reasons for conditions 1 – 3. In what follows, we show counter-examples that invalidate the conclusion of the lemma if one of the conditions is omitted.

Condition 1, in combination with condition 3, prevents erroneous reasoning as illustrated in Section 3.3, where the function f would include initial terms that are ignored by the asymptotic bound, but would dominate when included in the iterated sum.

Lemma 5.3.11 (Condition 1 is necessary) *Lemma 5.3.10 deprived of condition 1 is wrong.*

Proof Let us take f to be $(\lambda(a,i).\ \mathsf{If}\ i=0\ \mathsf{then}\ -2^a\ \mathsf{else}\ 0)$, g to be $(\lambda(a,i).\ 0)$, and the initial index i_0 to be 0. Then, g satisfies condition 2, and f satisfies condition 3: on the interval $[0,+\infty)$, f is indeed nondecreasing. We have $f \preceq_{\mathbb{Z}\times\mathbb{Z}} g$ (in particular they are both equal to 0 when $a \geq 0$ and $i \geq 1$). However, $\lambda(a,n).\ \sum_{i=0}^{n} f(a)(i)$ (which is equal to $\lambda(a,n).\ -2^a$) is not dominated by $\lambda(a,n).\ \sum_{i=0}^{n} g(a)(i)$ (which is equal to $\lambda(a,n).0$).

Lemma 5.3.12 (Condition 3 is necessary) *Lemma 5.3.10 deprived of condition 3 is wrong.*

Proof Then, let us consider the function f defined as $(\lambda(x,i).\ 2^x$ if $i=0$, i otherwise). Perhaps surprisingly, f is dominated by $\lambda(x,i).\ i$ for large enough x and i. However, $\lambda(x,n).\ \sum_{i=0}^{n} f(x,i)$ is equal to $\lambda(x,n).\ 2^x + \sum_{i=0}^{n} i$, which is not dominated by $\lambda(x,n).\ \sum_{i=0}^{n} i$. Note that the same counter-example applies if condition 3 is relaxed to "*for every a, the function $\lambda i.f(a)(i)$ is ultimately nondecreasing*". In other words, f does really need to be nondecreasing starting from the summation index.

Condition 2 ensures that the values of g do not cancel each other out. Indeed, the domination relation between f and g compares absolute values, which could involve both positive and negative values. When considering iterated sums, the values of g could cancel each other when summed together.

Lemma 5.3.13 (Condition 2 is necessary) *Lemma 5.3.10 deprived of condition 2 is wrong.*

Proof Let us take f to be $\lambda(a,i).\ 1$ and g to be $\lambda(a,i).\ (-1)^{(i \bmod 2)}$. Then, f satisfies condition 1 and condition 3, and we have $f \preceq_{A\times\mathbb{Z}} g$ (they are equal in norm). However, $\lambda(a,n).\ \sum_{i=0}^{n} f(a)(i)$ (which is equal to $\lambda(a,n).\ n$) is not dominated by $\sum_{i=0}^{n} g(a)(i)$ (which is bounded by 1).

Similarly to Lemma 5.3.9, a general coarse summation lemma can be stated as follows:

Lemma 5.3.14 (Coarse Summation (product filter)) *Let f range over $A \to \mathbb{Z} \to \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$. Assume the following two properties:*

1. $\mathbb{U}_A\, a.\ \forall i \geq i_0.\ f(a)(i) \geq 0$.
2. *for every a, $\lambda i.f(a)(i)$ is nondecreasing on the interval $[i_0,\infty)$.*

Then,

$$\left(\lambda(a,n).\ \sum_{i=i_0}^{n} f(a)(i)\right) \preceq_{A\times\mathbb{Z}} \lambda(a,n).\ n\cdot f(a)(n).$$

For convenience, we also prove a few easy corollaries of these lemmas, where the end index of the summation is given by a function, or where the body of the summation is allowed to depend on the end index.

In our few case studies that involve for-loops, we illustrate the use of Lemma 5.3.9 in the "Dependent nested loops" example (§7.2), and Lemma 5.3.8 in the next example (§7.3, "A loop whose body has exponential cost"). The proof of the Bellman-Ford example (§7.4) also makes use of the general versions of both lemmas (Lemma 5.3.10 and 5.3.14).

Lemma 5.3.15 (Summation (norm filter)) *Let f, g range over $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$. Assume the following property:*

$$\mathbb{U}_{\|\mathbb{Z}^2\|}\,(a, i).\ g(a)(i) > 0.$$

Then,

$$\lambda(a, i).f(a)(i) \ \preceq_{\|\mathbb{Z}^2\|}\ \lambda(a, i).g(a)(i)$$

implies

$$\left(\lambda(a, n).\ \sum_{i=i_0}^{n} f(a)(i)\right) \ \preceq_{\|\mathbb{Z}^2\|}\ \left(\lambda(a, n).\ \sum_{i=i_0}^{n} g(a)(i)\right).$$

Lemma 5.3.15 states a similar summation lemma for asymptotic bounds expressed using the norm filter on $\mathbb{Z}^2$. (This lemma can be generalized to a lemma on $\mathbb{Z}^n$, for instance by encoding tuples of integers as integers, using the fact that there is a bijection between $\mathbb{Z}$ and $\mathbb{Z}^n$.)

5.3.2 Composition and Specialization

An important property is the fact that function composition is compatible, in a certain sense, with domination. This allows transforming the parameters under which an asymptotic analysis is carried out. This can include instantiating some parameters, as illustrated by the Bellman-Ford example (Section 7.4), where a specification with a $O(\lambda n.\ n^3)$ complexity bound is deduced from a more general specification with a $O(\lambda(m, n).\ n \cdot m)$ complexity bound (m and n are the number of edges and vertices of the input graph, and therefore $m \leq n^2$ holds).

In the following, let A and B be filtered types. Let f and g inhabit $B \to \mathbb{Z}$ and let p inhabit $A \to B$, so that $f \circ p$ and $g \circ p$ have type $A \to \mathbb{Z}$. Recall that $p \nearrow_A^B$ (Definition 5.3.7) expresses that the function p "grows large" in B for large enough inputs in A. The statement of the Composition lemma is as follows:

Lemma 5.3.16 (Composition) *If $p \nearrow_A^B$ then $f \preceq_B g$ implies $f \circ p \preceq_A g \circ p$.*

In the above lemma, the hypothesis $f \preceq_B g$ controls the behavior of f and g when b grows large, while the goal $f \circ p \preceq_A g \circ p$ is to control their behavior when a grows large, where the parameters a and b are related by $b = p(a)$. Thus, we need $p(a)$ to grow large when a grows large, which is given by the side condition $p \nearrow_A^B$.

In the Bellman-Ford example, we lose one dimension by instantiating m with n^2: the function p is $\lambda n.(n^2, n)$, of type $\mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$. In the proof of Binary Random Access Lists (Section 7.7), the composition lemma is used crucially in the analysis of the `lookup` function, to go from a $O(\lambda(k, l).\ k + l)$ complexity bound to a $O(\lambda n.\ \log n)$ bound, by composing with $\lambda n.(0, \log(2n + 1))$ (i.e. taking k to be 0 and the length of the spine l

to be logarithmic in the number of elements n – this is guaranteed by the functional invariants of the structure). One could wonder why we bothered proving an asymptotic bound involving two variables k and l only to specialize k to zero afterwards. The reason is that we first need to verify an internal function that operates inductively on a *list segment*. The two parameters k and l which characterize the list segment both change through the induction, hence the need for a two-variables specification. Finally, at the toplevel, the internal function is called on a complete list, for which the value of k is known to be zero.

Notice that we instantiated k with a constant value, which does not in fact "grow large" in the usual sense. This is possible because the specification of the first complexity bound $O(\lambda(k,l).\ k+l)$ anticipated that one might want to specialize k to a constant, and established the bound with respect to the slightly unusual filter $(\top \mid \lambda x.\, x \geq 0) \times \mathbb{Z}$. In this filter, the second parameter must "grow large" in the usual sense, but the first parameter can be any nonnegative value.

5.4 Practical Multivariate O

In Section 3.2, we illustrate a specific challenge for the complexity analysis of functions with multiple parameters: while it seems natural to state an asymptotic complexity specification for a program assuming that all inputs "grow large" (in the common sense), this prevents deducing anything about the complexity of the program when some of its arguments are of constant size. Section 4.1.3 hints at the fact that the re-usability of multivariate O specifications depends on the chosen filter. Because the meaning of "growing large" is actually provided by the filter parameter of an asymptotic bound, whether it is possible to specialize such a bound with constant parameters depends on whether the filter allows it.

At the same time, we are interested in O bounds that are more concise than concrete cost expressions. If the concrete cost of a program is $\lambda n.\ 3n+4$, one should be able to simplify this to an asymptotic bound $O(\lambda n.\ n)$. All simplifications are not valid however: it depends on the filter used. For instance, one cannot simplify $O(\lambda(m,n).\ mn+1)$ to $O(\lambda(m,n).\ mn)$ when using a filter that allows either m or n to be specialized to zero.

In this section, we adress the remaining question: what is a multivariate filter that allows specializing some parameters to constant values, and also allows simplifying asymptotic bounds? Or, alternatively: in the specification of a multivariate program, which filter should one use for the specification to be the most reusable?

Mixed products of order and universal filters Consider again the OCaml function `g` introduced in Section 3.2. A concrete cost function for `g` is $cost\_g(m,n) \triangleq mn+n+1$. One can prove that $cost\_g$ is dominated by $\lambda(m,n).\ mn$ with respect to the standard filter on $\mathbb{Z}^2$ ("order filter $\times$ order filter"):

$$cost\_g \preceq_{\mathbb{Z}^2} (\lambda(m,n).\ mn).$$

However this does not entail any asymptotic bound about $\lambda n.\ cost\_g(0,n)$, and therefore does not allow deducing anything about the complexity of the OCaml function `f n = g (0, n)`.

A first possible approach is to *anticipate* that one might want to specialize m to 0, and prove that *cost_g* is dominated by $\lambda(m,n).\ mn+n$ with respect to a stronger filter, for instance the product of the standard filter on $\mathbb{Z}$ and the universal filter on $\mathbb{Z}^+$:

$$\mathit{cost\_g} \preceq_{\mathbb{Z}\times(\top\mid\lambda x.\ x\geq 0)} (\lambda(m,n).\ mn+n).$$

This bound *does allow* instantiating m with 0 and deducing that $\lambda n.\ \mathit{cost\_g}(0,n) \preceq_{\mathbb{Z}} \lambda n.\ n$, by using the composition lemma (Lemma 5.3.16) with p equal to $\lambda n.\ (0,n)$. Similarly, in the proof of Binary Random Access Lists mentioned earlier (§5.3.2), the verification of the `lookup` function proceeds in two steps, anticipating in the first step that the second step will need to specialize one parameter to 0. This makes sense since the intermediate step is internal to the proof. In the general case, however, it is hard to anticipate which of the arguments of a function one might want to specialize to small values.

Universal filter Alternatively, on can consider the universal filter (on the domain of the arguments, typically). This naturally allows any of the arguments to be instantiated freely, but departs from the usual interpretation that O, as expressing a bound that holds for "sufficiently large inputs"; instead, the bound must hold on all inputs.

Using such a filter, it is still possible to perform *some* simplifications in a bound. For instance, by restricting m and n to be nonnegative, one can get rid of the multiplicative factors in a cost function of the form $3m^2+2n^2+5mn+1$. Additionally, the "mn" term can also be eliminated in this case, independently of the filter used: $\lambda(m,n).\ mn \preceq_A \lambda(m,n).\ m^2+n^2$ holds for any filter A (equivalently, it holds for the filter $\top$).[3]

$$\begin{aligned}
&\lambda(m,n).\ 3m^2+2n^2+5mn+1\\
&\quad\preceq_{(\top\mid\lambda(m,n).\ m\geq 0\wedge n\geq 0)}\\
&\lambda(m,n).\ m^2+n^2+mn+1\\
&\quad\preceq_{(\top\mid\lambda(m,n).\ m\geq 0\wedge n\geq 0)}\\
&\lambda(m,n).\ m^2+n^2+1
\end{aligned}$$

Consider the `Array.append` function, as discussed in Section 4.1.3. The concrete cost function *cost_append* of `Array.append` is of the form $\lambda(m,n).\ 4m+4n+6$ (the exact value of the constants depend on the implementation; but the point is precisely that they do not matter). With respect to the filter $(\top \mid \lambda(m,n).\ m\geq 0 \wedge n\geq 0)$, one can prove that `Array.append` has complexity $O(m+n+1)$:

$$\mathit{cost\_append} \preceq_{(\top\mid\lambda(m,n).\ m\geq 0\wedge n\geq 0)} \lambda(m,n).\ m+n+1.$$

Additionally, derived bounds can be obtained where either m or n are specialized to any nonnegative value.

Bounding the complexity of `Array.append` by $\lambda(m,n).\ m+n+1$ is nonetheless not completely satisfactory. Such a bound accounts for the possibility where *both* m and n are instantiated with 0. This is not particularly interesting: it is clear

[3] We have $(n-m)^2 \geq 0$, i.e. $n^2+m^2 \geq 2mn$. Qed.

that "`Array.append [||] [||]`" (or "`Array.append [|1;2;3|] [|4;5|]`") is a constant time operation—it is a closed expression without any parameters—without having to specialize the asymptotic bound for *cost_append*. Instead, one would like to establish the more natural "$O(m+n)$" bound: $\mathit{cost\_append} \preceq \lambda(m,n).\ m+n$.

This is not possible with the filter at hand, since the following *does not hold:*

$$\lambda\_.\ 1 \preceq_{(\top \mid \lambda(m,n).\ m\geq 0 \wedge n \geq 0)} \lambda(m,n).\ m+n.$$

Quantifying on the side-conditions An alternative approach is to make the specification parametric on any filter that satisfies a set of useful side-conditions. This quantification on filters can itself be expressed as a filter, as the *supremum* of a class of filters. Recall that we write $\bigvee S$ for the supremum filter of the class of filters S (Definition 5.2.7).

In particular, one can prove that `Array.append` has complexity $O(m+n)$, with respect to the following filter:

$$G \triangleq \bigvee \left(\lambda F. \begin{array}{c} \lambda\_.\ 1 \preceq_F \lambda(m,n).\ m+n \\ \wedge\ \mathbb{U}_F(m,n).\ (m \geq 0 \wedge n \geq 0) \end{array} \right)$$

Intuitively speaking, G is the strongest (the most demanding) filter that validates the assertions $\lambda\_.\ 1 \preceq_G \lambda(m,n).\ m+n$, $\mathbb{U}_G(m,n).\ m \geq 0$ and $\mathbb{U}_G(m,n).\ n \geq 0$.

Using Lemma 5.2.9 ("$\preceq$ is contravariant"), this filter allows specializing m or n, because both $\mathbb{Z} \times (\top \mid \lambda n.\ n \geq 0)$ (which allows specializing n) and $(\top \mid \lambda m.\ m \geq 0) \times \mathbb{Z}$ (which allows specializing m) are less than G in the order between filters.

There is clearly a trade-off here: compared with the previous bound $O(m+n+1)$, we have a simpler bound $O(m+n)$, but a more complicated filter. This makes life easier for the user proving the specification (they can assume the filter satisfies the side-conditions they migt desire), and then it is up to the client of the specification to provide a more concrete filter (corresponding to their use of the specification), and prove that their filter satisfies the side-conditions.

This approach seems interesting, but perhaps somewhat awkward to carry out, as it really relies on using a whole class of filters, instead of a clearly defined one. For instance, we have no corresponding summation lemmas for such filters (and it is unclear how such lemmas would look like), preventing their use to reason modularly about the complexity of loops.

Norm filter The use of the norm filter on $\mathbb{Z}^2$ provides a more satisfying solution to our problem. Firstly, this filter allows eliminating constant terms, that is, we have for instance:

$$(\lambda(m,n).\ m+n+1) \preceq_{\|\mathbb{Z}^2\|} (\lambda(m,n).\ m+n)$$

and (recall that $\mathit{cost\_g}(m,n) \triangleq mn+n+1$):

$$\mathit{cost\_g} \preceq_{\|\mathbb{Z}^2\|} (\lambda(m,n).\ mn+n).$$

Consequently, according to the norm filter, `Array.append` has complexity $\mathrm{O}(m+n)$, which is what one would expect. Additionally, the norm filter does allow specializing

some parameters to constant values. From the following asymptotic bound on *cost_g*:

$$cost\_g \preceq_{\|\mathbb{Z}^2\|} (\lambda(m,n).\ mn+n).$$

we can derive a bound on $\lambda n.\ cost\_g(0,n)$ simply by using the composition lemma (Lemma 5.3.16) with p equal to $\lambda n.\ (0,n)$ (indeed, for a big enough n, the pair $(0,n)$ is equally big in norm).

$$\begin{array}{lll}(\lambda n.\ cost\_g(0,n)) & \preceq_{\mathbb{Z}} & (\lambda n.\ 0\cdot n+n)\\ & \preceq_{\mathbb{Z}} & (\lambda n.\ n)\end{array}$$

In the technical development presented in this thesis, we have mostly used product filters to account for multivariate functions, because at first it seemed like a more generic notion. However, it appears that using a norm filter on a domain of the type $\mathbb{Z}^k$ is more compositional, since it supports specializing bounds with constant parameters. In future work, one should revisit the case studies involving multivariate asymptotic bounds to use a norm filter.

5.5 Tactics and Automation

Our formalization of filters and domination forms a stand-alone Coq library [Gué18b]. In addition to many lemmas about these notions, the library proposes automated tactics that can prove goals of the following forms:

$$\begin{array}{cccc}\mathbb{U}_A\, x.\ c \le f(x) & \text{(nonnegativeness)} & f \nearrow_A^B & \text{(limit)}\\ \forall x,y.\ x\le y \Rightarrow f(x)\le f(y) & \text{(monotonicity)} & f \preceq_A g & \text{(domination)}\end{array}$$

These tactics currently support functions built out of variables, constants, sums and maxima, products, powers, logarithms. They were sufficient for our needs—but it would be possible to go further, for instance along the lines of Eberl's decision procedure for Landau symbols [Ebe17]. We detail below a number of design decisions (some of which might be worth revisiting) and technical challenges that were involved in the development of the library.

Proofs involving filters When carrying proofs involving filters, it is common to have to prove that some property Q holds ultimately, assuming that a collection of assumptions $P_1,\ldots,P_n$ hold ultimately. It is then enough to prove Q assuming all of $P_1,\ldots,P_n$. We implement a couple of simple tactics that turn goals of the form:

$$\mathbb{U}_A P_1 \to \ldots \to \mathbb{U}_A P_n \to \mathbb{U}_A Q \qquad \mathbb{U}_A P_1 \to \ldots \to \mathbb{U}_A P_n \to Q$$

into (respectively):

$$P_1 \to \ldots \to P_n \to Q \qquad \mathbb{U}_A(P_1 \wedge \ldots \wedge P_n) \to Q$$

These simple-minded tactics can be implemented in a straightforward way using the *covariance* and *stability under binary intersection* properties of filters. They are already quite useful in practice. It is worth noting that Affeldt, Cohen and Rouhling [ACR18] develop with `near=>` a set of more sophisticated tactics that follow similar motivations.

Proofs of domination properties To establish the various lemmas of the library, one needs to first prove domination properties ("$f \preceq_A g$") directly from the definition. This requires providing upfront an adequate value for the multiplicative constant c (recall that the definition of $f \preceq_A g$ is $\exists c.\, \mathbb{U}_A x.\, |f(x)| \leq c|g(x)|$).

Guessing c beforehand can become relatively tedious. For instance, the proof of Lemma 5.3.10 ("Summation") requires splitting iterated sums into several parts, and the constant needs to be adjusted so that each part gets bounded accordingly later in the proof. Worse, manually providing a value for c requires unfolding the definition of product and order filters, thus reasoning at a lower level of abstraction than one would desire. In our mechanized proof of the lemma, we alleviate the first issue by using the "big enough" tactic of Cohen [Coh12, §5.1.4], but the second issue remains.

A promising idea can be found in the paper of Affeldt et al. [ACR18]. One can prove the following equivalent definition of the domination relation:

$$f \preceq_A g \;\Leftrightarrow\; \mathbb{U}_{\mathbb{Z}} c.\, \mathbb{U}_A x.\, |f(x)| \leq c\,|g(x)|.$$

Compared to the standard definition (Definition 5.1.1), this alternative definition replaces an existential quantification on c with an "ultimately" quantification. This makes it possible to elaborate the multiplicative constant by leveraging tooling developed for reasoning on filters. In other words, this makes it possible to prove a domination relation by only reasoning on filters, retaining a higher level of abstraction.

Expression of cost functions Most of the lemmas of the library relate (cost) functions. On a practical note, in a formal setting specifically, there are two options for expressing these cost functions.

The first option is to write these functions as lambda expressions. For instance, Lemma 5.3.4 ("Multiplication"), which in math notation is stated as $f_1 \preceq_A g_1 \wedge f_2 \preceq_A g_2 \implies f_1 \cdot f_2 \preceq_A g_1 \cdot g_2$, would correspond to the following Coq statement (`dominated` stands for "$\preceq$"):

```
Lemma dominated_mul A f1 f2 g1 g2 :
  dominated A f1 g1 →
  dominated A f2 g2 →
  dominated A (fun x ⇒ f1 x * f2 x) (fun x ⇒ g1 x * g2 x).
```

Note that standard multiplication on $\mathbb{Z}$ is used in the conclusion of the lemma; such a lemma can be applied directly assuming that cost functions in the goal are also expressed as lambda-functions.

The second option is to lift the usual operations on $\mathbb{Z}$ to pointwise operations on $A \to \mathbb{Z}$. In that setting, a Coq statement for Lemma 5.3.4 is as follows, where "$*\hat{}$" denotes the lifted multiplication:

```
Lemma dominated_mul A f1 f2 g1 g2 :
  dominated A f1 g1 →
  dominated A f2 g2 →
  dominated A (f1 *^ f2) (g1 *^ g2).
```

In our library, we use the first option, which seems simpler because it does not require writing boilerplate in order to lift the usual operators at the level of functions. However,

this approach also has some drawbacks, and it would be interesting to investigate the second option further, and see if it allows for increased proof automation.

More precisely, we would like to reason by rewriting on goals involving domination relations. The first option requires support for rewriting under binders, which is in the setting of Coq tricky to setup and use. We tried to setup Coq's "setoid rewriting" mechanism, but were not able to get rewriting "under O" to work reliably on non-trivial examples. Furthermore, one would ideally want to rewrite not only with respect to equality, but also "less than" and "ultimately less than"...

Automated proofs of domination properties The Coq library provides automated tactics that try to prove domination properties and associated side-conditions, by using the lemmas of the library. These tactics are also extensible: an external user of the library should be able to extend the automation to work on their user-defined functions, provided they provide adequate lemmas.

Our current implementation relies on the `eauto` tactic and hint bases. It includes a few tricks, for instance to allow the automation to make some progress on the goal even if it cannot prove it completely. The tactics work relatively well on basic examples, but would probably need further development for them to be robust and usable on larger goals.

Handling multivariate cost functions Additional technical issues arise with respect to multivariate cost functions (perhaps unsurprisingly, at this point). Because our definitions are generic with respect to the filter, multivariate cost functions are implemented as functions from a filter whose domain is a product of types (e.g., $\mathbb{Z}^2$). This means that cost functions have to be written as un-curried functions.

To our knowledge, the most natural way of writing uncurried functions in Coq is:

```
(fun '(m, n) ⇒ m² + m + n)
```

which is in fact syntactic sugar for:

```
(fun p ⇒ let '(m, n) := p in m² + m + n)
```

This allows for fairly natural syntax. However, the fact that the expression of cost functions performs an extra pattern matching on the pair argument ("`let '(m, n):= p in ...`") prevents applying any filter-agnostic lemma from the library (either manually or through automation). Indeed, the conclusion of lemmas from the library (e.g. Lemma 5.3.1) is typically of the form:

```
Lemma dominated_sum_distr A f g h :
  ... →
  dominated A (fun x ⇒ f x + g x) h.
```

whereas a corresponding goal with functions of multiple variables would be (without the syntactic sugar):

```
dominated A (fun p ⇒ let '(m, n) := p in m² + n) h
```

Applying the lemma `dominated_sum_distr` on the goal above requires solving the following unification problem:

$$(\texttt{fun x} \Rightarrow \texttt{?f x} + \texttt{?g x}) \equiv (\texttt{fun p} \Rightarrow \texttt{let '(m,n):= p in m}^2\texttt{+n})\,.$$

Because Coq's unification algorithm does not work "modulo pattern-matching", Coq is not able to solve the unification problem, and the lemma cannot be applied directly.

One possibly promising solution is to define the product type we use in filters to use Primitive Projections. As a consequence, writing (`fun '(m,n)⇒ m²+ m + n`) gets desugared into (`fun p ⇒ (fst p)²+ (snd p)`)—which does not use pattern-matching and unifies with the conclusion of filter-agnostic lemmas. As of now, this unfortunately does not constitute a fully workable solution: it leads to terms that get quickly too verbose to be readable [Guéa], and seems to hit some limitations of the unification algorithm [Guéb].

Alternatively, it is possible to derive by hand versions of the lemmas that are specialized for functions of arity two (and three, four, etc). For instance, from `dominated_sum_distr` one can derive a specialized lemma `dominated_sum_distr2` that directly applies on functions of arity 2:

```
Lemma dominated_sum_distr (A: filteredType) f g h :
  dominated A f h →
  dominated A g h →
  dominated A (fun x ⇒ f x + g x) h.

Lemma dominated_sum_distr2 (X Y: Type) (M: FilterMixin (X * Y)) f g h :
  dominated (FilterType (X * Y) M) f h →
  dominated (FilterType (X * Y) M) g h →
  dominated (FilterType (X * Y) M) (fun '(x,y) ⇒ f (x,y) + g (x,y)) h.
```

In the statement of `dominated_sum_distr2`, "`FilterType (X * Y) M`" denotes any filter on the domain `X * Y` (`M` provides a proof for the additional properties that a filter must satisfy). This option is obviously not very satisfying: for each lemma of the library, it requires to manually produce derived lemmas for each arity that one might want to use.

In the current implementation of the library, we implement a slightly more general solution, at the price of even more boilerplate. In this approach, for each lemma of the library, we derive one *arity-generic* lemma, which works for any filter which domain corresponds to an iterated product over a list of types. To state and prove such lemmas, we rely on a small library of combinators that define the iterated product, function type, abstraction, function application, and so on, over a list of types [Gué18b, src/Generic.v]. Some of these combinators, along with their types, are listed below. They are typically defined by induction over the list of type ("`domain`").

```
(* Rtuple [X;Y;Z] is (Z * Y * X) *)
Rtuple : list Type → Type

(* Rarrow [X;Y;Z] T is (Z → Y → X → T) *)
Rarrow : list Type → Type → Type

(* @Uncurry [X;Y;Z] T (fun z y x ⇒ ...) is (fun '(z,y,x) ⇒ ...) *)
Uncurry : ∀{domain : list Type} {range : Type},
          (Rarrow domain range) → Rtuple domain → range

(* @Fun' [X;Y;Z] T (fun p ⇒ ... p ...) is (fun '(x,y,z) ⇒ ... (x,y,z) ...) *)
Fun' : ∀{domain: list Type} {range: Type},
       (Rtuple domain → range) → (Rtuple domain) → range
```

```
(* @App [X;Y;Z] T (fun z y x → ...) (z,y,x) is (...) *)
App : ∀{domain : list Type} {range : Type},
      (Rarrow domain range) → Rtuple domain → range
```

Using these combinators, one can state and prove a variant of `dominated_sum_distr`, which is similar in spirit to `dominated_sum_distr2`, but works for any arity.

```
Lemma dominated_sum_distr_nary (domain: list Type) (M: FilterMixin (Rtuple domain))
  (f g h : Rarrow domain Z) :
  dominated (FilterType (Rtuple domain) M) (Uncurry f) (Uncurry h) →
  dominated (FilterType (Rtuple domain) M) (Uncurry g) (Uncurry h) →
  dominated (FilterType (Rtuple domain) M)
    (Fun' (fun p ⇒ Z.add (App f p) (App g p))) (Uncurry h).
```

In particular, the lemma obtained by specializing `dominated_sum_distr_nary` to a list `domain` of length two is the same as `dominated_sum_distr2` (after unfolding the expression of the combinators). Given a concrete goal involving uncurried cost functions, the `domain` can be determined, and lemmas such as `dominated_sum_distr_nary` can be applied directly to the goal (because the combinators such as `Uncurry` or `App` have been carefully defined and used so that they unfold to the usual hand-written expressions of cost-functions).

This is a fairly elaborate workaround, which requires some amount of boilerplate, while not being completely satisfactory (one still needs to use a custom tactic to apply the lemmas, which is required to first infer the `domain` argument from the goal). It is unclear whether it would be a good solution in the long run.

Chapter 6

Program Logic for Asymptotic Complexity Reasoning

"Reasoning about the complexity of a program is a two-step process". Let us recall the conclusion from Chapter 4: in order to establish an asymptotic complexity specification, the *first step* is to reason about the program itself, matching it against a *concrete cost function*. Then, as a *second step*, one reasons on the cost function, establishing an asymptotic bound for it. In Chapter 5, we present reasoning principles for establishing and composing asymptotic bounds on cost functions. These principles form the key tools we have to take care of the second step.

There remains to explain how one would address the first step. A first naive 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 do perform anyway in CFML for the purpose of functional correctness.

In this chapter, we present a set of reasoning principles and proof techniques designed for establishing the concrete cost expression of programs, minimizing the amount of low-level cost-related manipulations from the user. For that purpose, we introduce Separation Logic with Possibly Negative Time Credits (§6.3), which reduces the number of side conditions related to Time Credits (compared to ordinary Separation Logic with Time Credits), and enables more elegant specifications and loop invariants. We describe techniques for synthesizing the cost expression of straight-line (nonrecursive) code (§6.4), while allowing the user to interactively guide the synthesis process, and for instance bound the synthesized cost using any functional invariant that may be available during the proof. Finally, we introduce a mechanism which allows carrying out the proof *assuming the existence* of some abstract function or constants—allowing to interactively collect side-conditions that are deferred (§6.5). This mechanism is crucial for the analysis of recursive programs—and has other uses as well, for instance when reasoning about an amount of credits "up to some multiplicative constant".

6.1 Example: Binary Search, Revisited

Let us recall the binary search program previously introduced as part of the background chapter (§2, the code for binary search is reproduced here in Figure 6.1). In the background chapter, we demonstrated how a proof of complexity for this program can be carried out using only the basic techniques for reasoning on Time Credits. This required coming up

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

Figure 6.1: A binary search program.

with a concrete cost function beforehand, and manually splitting quantities of credits to pay for each individual operation.

Let us now explain how a proof of `bsearch` can be carried out using our framework. Following the discussion of Chapter 4, we state—then prove—an *asymptotic* complexity specification for `bsearch`. Informally, `bsearch` runs in $O(\log(j-i))$. One might worry that $\log(j-i)$ is not defined for $j \leq i$. However, O is an asymptotic notion: we only need to consider "big enough" values of $j-i$, and in particular, nonnegative ones. In practice, in Coq—a logic of total functions—log is defined to be zero on negative inputs, but we never make use of this fact in our proofs. Formally, the specification exposes only an asymptotic bound for the cost function, and abstracts over its concrete expression. It is of the following form:

$$\begin{array}{l}\exists(\mathit{cost} : \mathbb{Z} \to \mathbb{Z}).\\ \quad \text{nonnegative } \mathit{cost} \wedge \text{monotonic } \mathit{cost} \wedge \mathit{cost} \preceq_{\mathbb{Z}} \log \wedge\\ \quad \forall a\ \mathit{xs}\ v\ i\ j.\ 0 \leq i \leq |\mathit{xs}| \wedge 0 \leq j \leq |\mathit{xs}| \implies\\ \qquad \{\$\mathit{cost}(j-i) \star a \rightsquigarrow \mathsf{Array}\ \mathit{xs}\}\ \texttt{bsearch}\ a\ v\ i\ j\ \{\lambda k.\ a \rightsquigarrow \mathsf{Array}\ \mathit{xs}\}.\end{array}$$

In Section 6.2, we discuss how such specifications can be stated in actual Coq syntax, in order to make working with this style of specifications most convenient.

The specification above starts with an existential quantification over the cost function. We must therefore exhibit a concrete cost function *cost* such that $\mathit{cost}(j-i)$ credits justify the call `bsearch a v i j` and *cost* is asymptotically dominated by log. Because `bsearch` is a relatively small program, it is still doable to manually come up with an appropriate cost function for it. One valid guess is $\lambda n.\ 3\log n + 4$. Another valid guess, expressed as a recursive function, would be written in informal math notation as:

$$f(n) \triangleq \begin{cases} 1 \quad \text{if } n \leq 0 \\ 3 + \max(f(\frac{n}{2}),\ f(n - \frac{n}{2} - 1)) \quad \text{otherwise} \end{cases}$$

Formally, one could express it using a fixed point combinator Fix [Cha10b], as $\mathrm{Fix}(\lambda f\ n.\ \mathsf{If}\ n \leq 0\ \mathsf{then}\ 1\ \mathsf{else}\ 3 + \max(f(\frac{n}{2}),\ f(n - \frac{n}{2} - 1)))$. Yet another witness, obtained from the previous one via an approximation step, is $\mathrm{Fix}(\lambda f\ n.\ \mathsf{If}\ n \leq 0\ \mathsf{then}\ 1\ \mathsf{else}\ 3 + f(\frac{n}{2}))$. As the reader can see, there is in fact a spectrum of valid witnesses, ranging from verbose, low-level to compact, high-level mathematical expressions. Also, it should be evident

that, as the code grows larger, it can become very difficult to guess a valid concrete cost function.

To provide as much machine assistance as possible, our system mechanically synthesizes a low-level cost expression for a piece of OCaml code. The syntactic structure of the synthesized cost expression is the same as the syntactic structure of the code. Coming up with a compact, high-level expression of the cost requires human insight: it is deferred to a separate step.

For instance, because `bsearch` is a recursive program, we do not attempt to synthesize a closed cost function right away. Instead, we first introduce an abstract name *cost* for our cost function, about which we can interactively collect proof obligations. Technically speaking, this reasoning is made possible by our Coq library for deferring constraints, that we present later in Section 6.5.

Then, we can mechanically synthesize a cost expression for the body of `bsearch`. The synthesized cost expression can depend on the *cost* function introduced previously, in order to express the cost of recursive calls. This synthesis process—described in Section 6.4—is performed transparently, at the same time as the user constructs a proof of the code in Separation Logic. Furthermore, we take advantage of the fact that we are using an interactive proof assistant: we allow the user to guide the synthesis process. There are two main situations where the synthesis process requires guidance:

- Controlling how a local variable should be eliminated. In `bsearch`, the cost of the recursive calls depend on the local variable `k`, whereas the overall cost expression can only depend on $j - i$. Here, it is enough for the user to substitute `k` with its definition, and perform some local rewriting using simple mathematical lemmas. In full generality, eliminating a local variable may require approximating the locally-inferred cost, possibly using monotonicity properties of auxiliary cost functions (as described in Section 4.2.1 with the "avoidance problem").

- Deciding how the cost of a conditional construct should be approximated. The implementation of `bsearch` contains three conditionals. The condition of the first conditional (`if j <= i then ...`) is relevant for the complexity analysis: the corresponding cost expression should be of the form "$\mathsf{If}\ j - i \leq 0\ \mathsf{then} \ldots$". The conditions of the second and third conditionals (`if v = a.(k) then ...` and `if v < a.(k) then ...`) are however *not* relevant to the complexity analysis. The corresponding cost expressions should be of the form "$\max(\ldots, \ldots)$".

As the result of stepping through the code, the synthesis process yields the following proof obligation:

$$\forall n.\ cost(n) \geq 1 + \mathsf{If}\ n \leq 0\ \mathsf{then}\ 0\ \mathsf{else}\ (1 + \max(0, 1 + \max(cost(\tfrac{n}{2}), cost(n - \tfrac{n}{2} - 1)))).$$

That is, the overall cost of `bsearch`—which we assumed to be $cost(n)$—must be greater than the cost expression synthesized for its body, which appears on the right hand side of the equation. This proof obligation gives us a (low-level) recursive equation that characterizes valid cost functions for `bsearch`.

From this point on, the user no longer needs to reason about the implementation of `bsearch`. What remains is to prove that that there exists an actual cost function

that satisfies the given proof obligation, and that such a cost function is asymptotically dominated by log.

A good first step is to simplify the low-level cost inequality that we obtained. For instance, assuming that *cost* is monotonic—a side-condition that we can defer—we can simplify $\max(cost(\frac{n}{2}), cost(n - \frac{n}{2} - 1))$ into $cost(\frac{n}{2})$. Going a bit further, the previous low-level inequality can be simplified into the following set of simpler proof obligations:

$$\begin{array}{ll} & \text{monotonic } cost \\ \wedge & \forall n.\ cost(n) \geq 0 \\ \wedge & \forall n.\ n \leq 0 \implies cost(n) \geq 1 \\ \wedge & \forall n.\ n \geq 1 \implies cost(n) \geq cost(\frac{n}{2}) + 3. \end{array}$$

One way to complete the proof is to remark that these recurrence equations for *cost* are an instance of "Divide and Conquer" recurrences, for which the Master Theorem (by Bentley, Haken and Saxe [BHS80], later popularized by Cormen *et al.* [CLRS09, Theorem 4.1]) applies. In a formalized setting, Eberl's formalization of the Akra-Bazzi theorem [Lei96, AB98, Ebe17] (a generalization of the Master Theorem) should be able to automatically discharge the proof obligations above, as well as establishing the logarithmic bound. Unfortunately, at present, for this theorem to be usable in our setting, we would need to port Eberl's proofs and automation from Isabelle/HOL to Coq.

Alternatively, we can use once again the deferring mechanism of Section 6.5 to implement Cormen *et al.*'s substitution method [CLRS09, §4], a low-tech, simpler proof technique. The idea is to guess a parameterized *shape* for the solution; substitute this shape into the goal; gather a set of constraints that the parameters must satisfy for the goal to hold; finally, show that these constraints are indeed satisfiable. On this example, we propose that the solution should be of the form "$\lambda n.$ If $0 < n$ then $a \log n + b$ else c", where a, b and c are parameters, about which we wish to gradually accumulate a set of constraints.

Injecting this solution into the last recurrence inequality yields:

$$\begin{array}{l} n \geq 1 \implies \\ \text{If } 0 < n \text{ then } (a \log n + b) \text{ else } c \ \geq \ \big(\text{If } 0 < \frac{n}{2} \text{ then } \big(a \log \frac{n}{2} + b\big) \text{ else } c\big) + 3. \end{array}$$

Consequently, by distinguishing the case $n = 1$:

$$\begin{array}{l} n = 1 \implies b \geq c + 3 \\ n \geq 2 \implies a \log n + b \geq a \log \frac{n}{2} + b + 3. \end{array}$$

Using the properties of log, these inequalities simplify to $b \geq c + 3 \ \wedge \ a \geq 3$.

The same process can be applied to all of the proof obligations involving *cost*, eventually producing the following subgoal, which requires proving the existence of actual constants a, b and c that satisfy the side-conditions:

$$\exists a\, b\, c.\ c \geq 1 \ \wedge \ b \geq 0 \ \wedge \ a \geq 0 \ \wedge \ b \geq c + 3 \ \wedge \ a \geq 3.$$

Such a goal can be discharged using an automated decision procedure (§6.6). Then, proving that the function $\lambda n.$ If $0 < n$ then $a \log n + b$ else c is dominated by log is straightforward, thus completing the proof.

```
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 6.2: Definition of `specO`.

In summary, we are able to set up our proofs in a style that closely follows the traditional paper style. During a first phase, as we analyze the code, we synthesize a cost function and (if the code is recursive) a recurrence equation. During a second phase, we guess the shape of a solution, and, as we analyze the recurrence equation, we synthesize a constraint on the parameters of the shape. During a last phase, we check that this constraint is satisfiable. Compared to an approach where one would guess a concrete initial amount of credits, and substract from it as the proof goes, the strength of the present approach is that it allows for more robust proofs, that scale to larger programs. The user does not need to guess concrete cost expressions, that would be closely tied to the implementation at a given point; instead, they just need to guess the high-level shape of the solution. Additionally, the fact that a proof is separated in well-distinguished phases enables special purpose automation to be used in each phase. The synthesis process is automated and does not add overhead to the proof of functional correctness. Then, to reason about the cost function itself, the substitution method is only one of the tactics that we have developed—more are described in Section 6.6.

6.2 Specifications with Asymptotic Complexity Claims

Program specifications in the style that we argue for in Chapter 4 and use in the previous section share a common structure. They abstract over the cost function, while exporting some additional properties (such as nonnegativity and monotonicity) that are required to enable modular reasoning (§4.2).

In mathematical notation, these specifications are of the form:

$$\begin{aligned}&\exists(\mathit{cost} : A \to \mathbb{Z}).\\ &\quad \text{nonnegative } \mathit{cost} \ \wedge\ \text{monotonic } \mathit{cost} \ \wedge\ \mathit{cost} \preceq_A \mathit{bound} \ \wedge\\ &\quad \forall a \ldots.\{\$\mathit{cost}(a) \star \ldots\} \ \ldots\ \{\lambda r.\ \ldots\}\end{aligned}$$

In Coq, we define a (dependent) 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, which we name `specO`, is defined in Figure 6.2. The definition relies on implicit projections, allowing us to write A both for a filtered type and for its carrier type. The predicate "`dominated A f g`" corresponds to the relation $f \preceq_A g$. The first field asserts the existence of a function `cost` of A to $\mathbb{Z}$, where A is a user-specified filtered

```
Notation "'specZ' [ X '\in_O' f ] E" :=
  (specO Z_filterType Z.le (fun X ⇒ E) f)
    (X ident, f at level 90, E at level 0).

Notation "'spec1' [ X ] E" :=
  (specO unit_filterType eq (fun X ⇒ E) (fun tt ⇒ 1))
    (X ident, E at level 0).
```

Figure 6.3: Definition of the `specZ` and `spec1` notations. The brackets bind the name (`X`) of the concrete cost function in the specification (denoted by `E`). In the `specZ` notation, `f` denotes the asymptotic bound. In the case of `spec1`, `X` denotes a constant cost function, and the bound is set to $\lambda().1$.

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`. Finally, the last two fields export the fact that `cost` is non-negative and monotonic, as motivated in Sections 4.2.1 and 4.2.2.

Using this definition, the specification of `bsearch` introduced in the previous section is stated in concrete Coq syntax as follows:

```
Theorem bsearch_spec:
  specO Z_filterType Z.le Z.log2 (fun cost ⇒
    ∀a xs v i j,
      0 ≤ i ≤ length xs ∧ 0 ≤ j ≤ length xs →
      app bsearch [a v i j]
        PRE ($ (cost (j − i)) ⋆ a ⇝ Array xs)
        POST (fun (k:int) ⇒ a ⇝ Array xs)).
```

The key elements of this specification are `Z_filterType`, which is $\mathbb{Z}$, equipped with its standard filter; the asymptotic bound `Z.log2`, which means that the time complexity of `bsearch` is $O(\log n)$; and the Separation Logic triple, which describes the behavior of `bsearch`, and refers to the concrete cost function `cost`.

One might argue that this specification could still be made more concise and more readable, perhaps by exploiting type classes or canonical structures to let Coq infer which filter should be used, and by exploiting Coq notations to add a layer of syntactic sugar. For now, in the specific case of a specification expressed over the standard filter on $\mathbb{Z}$, we introduce a notation that allows to write the specification as "`specZ [cost \in_O Z.log2] ...`". In the case of specifications with a constant time function, where we use the universal filter on `unit`, we introduce the notation "`spec1 [cost] ...`". Their definition as Coq notations is given in Figure 6.3.

Using the first notation, the specification for `bsearch` can be written as below:

```
Theorem bsearch_spec:
  specZ [cost \in_O Z.log2]
    (∀ a xs v i j,
      0 ≤ i ≤ length xs ∧ 0 ≤ j ≤ length xs →
      app bsearch [a v i j]
        PRE ($ (cost (j − i)) ⋆ a ⇝ Array xs)
        POST (fun (k:int) ⇒ a ⇝ Array xs)).
```

One important technical point is that `specO` acts as a strong existential, whose witnesses can be referred to via projections corresponding to the field names. For instance, the concrete cost function associated with `bsearch` can be referred to as `cost bsearch_spec`. This is useful when reasoning about a call to `bsearch`: at a call site of the form `bsearch a v i j`, the number of required credits is simply `cost bsearch_spec (j−i)`.

The fact that these projections exist does not necessarily jeopardize abstraction. Provided that we make sure to terminate the proof of a specification with `Qed` (and not `Defined`), Coq enforces that the concrete expression of the cost function cannot be recovered using the `cost` projection.

Requirements for useful specifications with O

The main limitation of `specO` is that it is monolithic: it is not designed to allow exporting additional properties about the cost function (other than it being nonnegative and monotonic). One could add these properties as part of the `cost_spec` field, but it would defeat the automation which expects `cost_spec` to be a Separation Logic triple. Alternatively, one could define a new record, containing the same fields as `specO` plus some new fields for the additional properties, but would not benefit from the automation associated with `specO`. One could think of alternative setups, possibly using type classes or the module system, but those seem to involve a fair amount of boilerplate. The Derive command [The19a] is also worth mentioning, but does not completely fits our goal: the definitions that it generates are necessarily transparent (as if terminated with `Defined`), whereas we want them to be abstract (as if terminated with `Qed`).

Before attempting to find a solution that addresses these limitations, let us state explicitly the features that we expect from such a solution. From a high-level perspective, to be usable in our setting, a way of expressing asymptotic complexity specifications should satisfy the following properties:

- When proving a specification, it must be possible to synthesize the expression of the cost function *during* the proof of the Separation Logic triple. This is possible with `specO` because it gathers all the proof obligations in a single sub-goal, but would not be possible if we were to write the cost function and the specification as a separate top-level definition and theorem.
- After proving the specification, the system must abstract away the definition of the cost function. This is the case with `specO` as long as the proof is terminated with `Qed`.
- When using the specification, one needs to be able to refer to its separate components (the cost function, the Separation Logic specification, ...) using convenient names. This is relatively easy with `specO` thanks to record projections.
- It should be possible to extend specification to export additional properties if needed. This is not possible with `specO`.

An hypothetical Coq extension

Rather than try to satisfy the requirements above using some verbose encoding, it might be possible to implement a new Coq construct that would allow for a smooth integration of

```
Section Bsearch_proof.

  (* Hypothetical syntax. *)
  Delayed Definition bsearch_cost : Z → Z.

  (* At this point, the definition of bsearch_cost is simply
     a "hole": bsearch_cost := ?f. *)

  Theorem bsearch_spec : ∀a xs v i j,
    0 ≤ i ≤ length xs ∧ 0 ≤ j ≤ length xs →
    app bsearch [a v i j]
      PRE ($ (bsearch_cost (j − i)) ⋆ a ⇝ Array xs)
      POST (fun (k:int) ⇒ a ⇝ Array xs).
  Proof. ... Qed.

  (* After this point, a cost expression for bsearch_cost
     has been synthesized. *)

  Theorem bsearch_cost_dominated :
    dominated Z_filterType bsearch_cost Z.log2.
  Proof. ... Qed.

End Bsearch_proof.
(* The definition of bsearch_cost is now abstract. *)
```

Figure 6.4: Proof excerpt using the hypothetical `Delayed Definition` construct.

the workflow described here. One proposal is the introduction of a "`Delayed Definition`" construct, which would be used as illustrated in Figure 6.4. When used in a section, this new construct would introduce a top-level definition whose body is not yet determined. Then, the Separation Logic specification would be stated as a standard top-level theorem. Proving this theorem would—as a side effect—instantiate the body of the delayed definition, using the same synthesis mechanism that is used to construct the cost function of a `specO`. Finally, when exiting the section, the system would check that the delayed definition has been completely instantiated, and would make it abstract. There remains to see whether such a feature can reasonably be implemented in Coq (or as a plugin), but it seems to make sense from a user experience point of view.

6.3 Separation Logic with Possibly Negative Time Credits

One contribution of our work is the introduction of Separation Logic with *Possibly Negative* Time Credits, a variant of Separation Logic with Time Credits which provides more convenient principles for reasoning with Time Credits.

6.3.1 Motivation

Time Credits in $\mathbb{N}$ lead to redundant side conditions In the original presentations of Separation Logic with Time Credits [Atk11, CP17b, GCP18, HN18], credits are counted in $\mathbb{N}$. This seems natural because $\$n$ is interpreted as a permission to take n steps of computation, and a number of execution steps is never a negative value. In this setting, time credits satisfy the following properties:

$$\begin{array}{rlcll} & \$0 & \equiv & [\,] & (\textit{zero credit is nothing}) \\ \forall n \in \mathbb{N}. & \$n & \vdash & \mathrm{GC} & (\textit{credits are affine}) \\ \forall m\, n \in \mathbb{N}. & \$(m+n) & \equiv & \$m \star \$n & (\textit{credits are additive}) \end{array}$$

The first law expresses that an amount of zero credit is equivalent to nothing at all. As per the second law, it is always sound to discard any amount of credits. Recall that $[\,]$ characterizes the empty heap, and GC any heap (Figure 2.3) that the user might want to discard (using rule DISCARD-POST). The third law is used when one wishes to spend a subset of the credits at hand. For instance, if one owns $\$(m+1)$ credits and wishes to pay for a single `pay()` operation (which costs $\$1$), one can split this amount of credits into $\$m \star \$1$, which enables paying for $\$1$—while keeping the remainder of credits $\$m$ to pay for the rest of the program.

Yet, in practice, the law that is most often needed is a slightly different formulation. Indeed, if n credits are at hand and if one wishes to step over an operation whose cost is m, a more appropriate law is the following subtraction law, which does not require rewriting n as a sum of the form "$\ldots + m$" beforehand.

$$\forall m \leq n.\ \$n \equiv \$(n-m) \star \$m \quad (\textit{subtraction law})$$

The subtraction law holds only under the side condition $m \leq n$: in Coq, a logic of total functions, $n - m$ for $n < m$ is defined to be 0. This side condition gives rise to a proof obligation, and these proof obligations tend to accumulate.

To illustrate this fact, let us first consider a situation where n credits are initially at hand, and one wishes to pay for a single operation `pay m` which cost is m. Thereafter, we write between comments "$(*\ \$(\ldots)\ *)$" the number of credits available at a given point of a program, and write "$\rightsquigarrow P$" to denote that a reasoning step produces a side condition P.

$$\begin{array}{ll} (*\ \$n\ *) & \\ \texttt{pay m} & \rightsquigarrow m \leq n \quad \text{(proof obligation)} \\ (*\ \$(n-m)\ *) & \end{array}$$

That is, if n credits are initially available, then we have to prove the side condition $m \leq n$, and get back the amount of credits $\$(n-m)$. Now, if one wishes to step over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$, then k proof obligations arise: $m_1 \leq n$, then $m_2 \leq n - m_1$, and so on.

$$\begin{array}{ll} (*\ \$n\ *) & \\ \texttt{pay m\_1} & \rightsquigarrow m_1 \leq n \\ (*\ \$(n-m_1)\ *) & \\ \texttt{pay m\_2} & \rightsquigarrow m_2 \leq n - m_1 \\ (*\ \$(n-m_1-m_2)\ *) & \\ \ldots & \rightsquigarrow \ldots \\ (*\ \$(n-m_1-m_2-\ldots-m_{k-1})\ *) & \\ \texttt{pay m\_k} & \rightsquigarrow m_k \leq n - m_1 - m_2 - \ldots - m_{k-1} \\ (*\ \$(n-m_1-m_2-\ldots-m_{k-1}-m_k)\ *) & \end{array}$$

In fact, these proof obligations are redundant: the last one alone implies all of 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.

The issue of extraneous side conditions is actually more fundamental than simply a proof engineering issue. In the proof of Bender *et al.*'s algorithm (Section 8), we have encountered a more complex situation where, instead of looking at a straight-line sequence of k 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 this situation, proving that the very last proof obligation implies all previous obligations may be possible in principle, but requires a nontrivial strengthening of the loop invariant, which we would rather avoid, if at all possible!

Time Credits in $\mathbb{Z}$ yield smoother reasoning rules To avoid the accumulation of redundant proof obligations, we introduce a variant of Separation Logic where Time Credits are counted in $\mathbb{Z}$. Its basic laws are as follows:

$$\begin{array}{rrcll} & \$0 & \equiv & [\,] & (\textit{zero credit is nothing}) \\ \forall n \in \mathbb{Z}. & \$n \star [n \geq 0] & \Vdash & \mathrm{GC} & (\textit{nonnegative credits are affine}) \\ \forall m\, n \in \mathbb{Z}. & \$(m+n) & \equiv & \$m \star \$n & (\textit{credits are additive}) \end{array}$$

Quite remarkably, in the third law, there is no side condition. In particular, this law implies $\$0 \equiv \$n \star \$(-n)$, which creates positive credit out of thin air, but creates negative credit at the same time. In this setting, a negative amount of credits must

be understood not as a an impossibility, but as a debt that must be paid off at a later point. As put by Tarjan [Tar85], "we can allow borrowing of credits, as long as any debt incurred is eventually paid off". In the second law, the side condition $n \geq 0$ guarantees that a debt cannot be forgotten. Without this requirement, the logic would be unsound, as the second and third laws together would imply $\$0 \Vdash \$1 \star \mathrm{GC}$. Indeed, using the second law, one could artificially create a credit and a debt by exploiting the entailment $\$0 \Vdash \$1 \star \$(-1)$, then throw away $\$(-1)$, ending up with $\$1$, that is, a time credit created out of thin air.

$$\$n \equiv \$(n-m) \star \$m \quad (\textit{subtraction law})$$

In this setting, the subtraction law holds for every $m, n \in \mathbb{Z}$ without any side condition.

Sequences of operations Thanks to the unrestricted subtraction law, 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.

$(*\ \$n\ *)$	
`pay m_1`	$\rightsquigarrow$ no proof obligation
$(*\ \$(n-m_1)\ *)$	
`pay m_2`	$\rightsquigarrow$ no proof obligation
$(*\ \$(n-m_1-m_2)\ *)$	
$\ldots$	$\rightsquigarrow \ldots$
$(*\ \$(n-m_1-m_2-\ldots-m_{k-1})\ *)$	
`pay m_k`	$\rightsquigarrow$ no proof obligation
$(*\ \$(n-m_1-m_2-\ldots-m_{k-1}-m_k)\ *)$	
	$\rightsquigarrow n-m_1-m_2-\ldots-m_{k-1}-m_k \geq 0$
$(*\ []\ *)$	

At the end of the sequence, $n - m_1 - m_2 - \ldots - m_k$ credits remain, which the user typically wishes to discard. Indeed, if we wish to bound the execution time of a complete program execution, then we cannot terminate by leaving debts (see Definition 6.3.1 later on). Discarding the remaining amount of credits is done by applying the second law, giving rise to just one proof obligation: $n - m_1 - m_2 - \ldots - m_k \geq 0$. 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.

Loops Interestingly, Time Credits in $\mathbb{Z}$ also help with reasoning on loops (and recursion), especially when the number of loop iterations is not easily expressible as a function of the input parameters of the program. Let us consider the following recursive function `walk`, which computes the index of the first zero-valued cell of an array of integers, if any. This function is implemented by iterating over the array, and terminating if it reaches the end of the array, *or if it encounters an array cell whose content is equal to zero*. This means that the time complexity of `walk` also depends on the content of the array, not only its length.

```
let rec walk (a: int array) (i: int): int =
  if i < Array.length a && a.(i) <> 0 then
```

```
    walk a (i+1)
  else
    i+1
```

One might be tempted to over-approximate the complexity of `walk`, and state that it is linear in the length of the input array `a`. This is correct, but too coarse: consider the following program `full_walk`, which calls `walk` successively in order to fully traverse the array.

```
let full_walk (a: int array) =
  let j = ref 0 in
  while !j < Array.length a do
    j := walk a !j
  done
```

The `full_walk` function terminates in time linear in the length of `a`. However, if we over-approximate the cost of `walk` as being linear in the length the whole array, then this would lead us to conclude that `full_walk` has quadratic complexity. (The worst case is for an array full of zeros.)

One solution out of this problem is to refine the specification of `walk` to reason in the logic over the contents of the array. One could express that `walk` runs in time linear in *the length of the longest sequence of non-zero values of* `a` *starting from* `i`. However, this significantly complicates the specification.

Time Credits in $\mathbb{Z}$ allow for a much more elegant solution. Instead of specifying the cost of the function as a positive amount of credits in the precondition, one can instead express it as a *negative* amount of credits in the postcondition—which can additionally depend on the return value of the function. In some sense, the program specification *produces a debt*. One can therefore prove the following specification for `walk`:

$$\begin{array}{l}\forall a\, i\, A.\\ 0 \le i < |A| \implies\\ \{a \rightsquigarrow \mathsf{Array}\, A\}\ \texttt{walk}\ a\ i\ \ \{\lambda j.\ a \rightsquigarrow \mathsf{Array}\, A \star \$(i-j) \star [i < j \le |A|]\}.\end{array}$$

In this specification, a denotes an array, A the logical contents of the array associated to a, and i the starting index. As a client of this specification, calling `walk` has no upfront cost, but produces a debt ($i-j$ is negative) corresponding to the cost of exploring the elements between index i and j. Such a specification makes it possible to prove that `full_walk` has linear complexity. Let us note j_0, j_1, …, j_k the intermediate indices successively returned by `walk`. Then, the cost of all calls to `walk` is $(0-j_0)+(j_0-j_1)+\ldots+(j_k-|A|)$. We obtain a telescopic sum, which simplifies to $-|A|$. Provided that `full_walk` requires in its precondition an amount of credits greater than $|A|$, then this debt can be paid off. This establishes that `full_walk` has a linear time complexity. (Formally, given an initial amount of credits $\$|A|$, we can prove that a valid loop invariant is $\exists p.\ j \hookrightarrow p \star \$(|A|-p) \star [0 \le p \le |A|]$.)

It is worth noting that `walk` also satisfies the following alternative specification:

$$\begin{array}{l}\forall a\, i\, A.\\ 0 \le i < |A| \implies\\ \{a \rightsquigarrow \mathsf{Array}\, A \star \$(|A|-i)\}\ \texttt{walk}\ a\ i\ \ \{\lambda j.\ a \rightsquigarrow \mathsf{Array}\, A \star \$(|A|-j) \star [i < j \le |A|]\}.\end{array}$$

This specification asks upfront for enough credits to walk through the whole array, then gives back in the postcondition an amount of credits corresponding to elements that have not been explored. Interestingly, this second specification is trivially equivalent to the first one: using the fact that $\$(|A| - j)$ is equivalent to $\$(|A| - i) \star \$(i - j)$ and the FRAME rule to frame out $\$(|A| - i)$, one recovers the first specification. Generally speaking, in presence of time credits in $\mathbb{Z}$, specifications are "invariant by translation": one can always add or subtract a given amount of credits to both the pre and post condition.

As a more technical remark, note that this second specification does not contain any negative amount of credits in its statement: both $|A| - i$ and $|A| - j$ are known to be nonnegative. However, for it to be *usable* in full generality, time credits in $\mathbb{Z}$ are still required. Indeed, even if a client of the specification knows that a given call to `walk` will stop before the end of the array, they are required to pay first for the entire traversal. This may require the client to temporarily "go in debt" by paying for more than what they have, only to get reimbursed afterwards. This is only possible if the client of the specification uses time credits in $\mathbb{Z}$; otherwise, the specification can only be used in the cases where the client would have enough credits to pay for the complete traversal.

In summary, switching from $\mathbb{N}$ to $\mathbb{Z}$ greatly reduces the number of proof obligations that appear about credits, and lead to simpler loop invariants and specifications (one further example is the interruptible iterator of Section 7.6, that is used in our proof of Bender *et al.*'s algorithm (§8)).

6.3.2 Model and Reasoning Rules

From a metatheoretical prespective, the main challenge for defining Separation Logic with Time Credits in $\mathbb{Z}$ comes from the fact that negative time credits are not an affine resource. (Recall that we say that a resource is affine if it entails GC.) This means that we have to work in a logic where not every assertion is affine, contrarily to other logics such as Iris [JKJ+18]. A points-to assertion, which describes a heap-allocated object, remains affine; the assertion $\$n$ is affine if and only if n is nonnegative; an abstract assertion may or may not be affine, depending on its definition.

We have adapted CFML so as to support this distinction. Fortunately, the changes are minimal, thanks to the fact that the Separation Logic implemented by CFML is linear. To adapt CFML to possibly negative time credits, it is sufficient to modify the definition of GC.

Soundness of Separation Logic with Time Credits in $\mathbb{Z}$ Let us detail the (minimal) set of changes required to go from the Separation Logic with Time Credits in $\mathbb{N}$ (as described in Section 2.2) to our Separation Logic with Time Credits in $\mathbb{Z}$.

First, we revisit the definition of the type Heap of heap fragments: instead of defining Heap as $\mathsf{Memory} \times \mathbb{N}$, we allow time credits to live in $\mathbb{Z}$.

$$\mathsf{Heap} \triangleq \mathsf{Memory} \times \mathbb{Z}$$

The definition of the assertion $\$n$ and the basic laws it satisfies are preserved as is:

$$\begin{array}{rrcll} \forall n \in \mathbb{Z}. & \$n & \triangleq & \lambda(m,c).\ m = \varnothing \wedge c = n & \\ \forall m\, n \in \mathbb{Z}. & \$(n+m) & \equiv & \$n \star \$m & \text{(unchanged)} \\ & \$0 & \equiv & [\,] & \end{array}$$

Before going to the definition of triples, we need to revisit the definition of the heap predicate GC, which is used in the definition of triples to represent any heap fragment that we may want to discard. With Time Credits in $\mathbb{N}$, the predicate GC is defined as $\lambda h.\ \mathsf{True}$ (allowing any part of the heap to be discarded). We refine this definition to only allow discarding heap fragments that contain a nonnegative amount of credits.

$$\mathrm{GC} \triangleq \lambda(m,c).\ c \geq 0$$

In the updated logic, we do not have $H \Vdash \mathrm{GC}$ for every H unlike previously. Instead we have a few rules of the form $H \Vdash \mathrm{GC}$ for specific Hs (positive credits, points-to assertions, conjunctions, existential quantifications, etc.).

$$[P] \Vdash \mathrm{GC} \qquad l \hookrightarrow v \Vdash \mathrm{GC} \qquad n \geq 0 \implies \$n \Vdash \mathrm{GC}$$
$$H_1 \Vdash \mathrm{GC} \wedge H_1 \Vdash \mathrm{GC} \implies (H_1 \star H_2) \Vdash \mathrm{GC}$$
$$(\forall x.\ H \Vdash \mathrm{GC}) \implies \exists x.\ H \Vdash \mathrm{GC}$$

No further changes are required from this point. The rest of the rules of the logic, the definition of triples and the statement of the soundness theorem can be ported without modification.

Definition 6.3.1 (Triples in Separation Logic with Time Credits in $\mathbb{Z}$) *A triple $\{H\}\ t\ \{Q\}$ is short for the following proposition:*

$$\forall m\, c\, H'.\ (H \star H')\ (m,c) \implies \exists n\, v\, m'\, c'.\ \begin{cases} t/m \Downarrow^n v/m' \\ (Q\ v \star H' \star \mathrm{GC})\ (m', c') \\ c = n + c' \end{cases}$$

Note than even though the definition of triples above is syntactically the same as with Time Credits in $\mathbb{N}$, there is a new subtlety with how it should be interpreted. When using Time Credits in $\mathbb{N}$, it is always the case that the initial amount of credits c provided in the precondition is an *upper bound* for the cost of running the program. Indeed, for c, n and c' in $\mathbb{N}$, the equality $c = n + c'$ entails that c is greater than n. When using Time Credits in $\mathbb{Z}$, it is not necessarily the case. In the equality $c = n + c'$, it is now possible for c or c' to be negative, if the precondition or the postcondition assert the ownership of a negative amount of credits. (Note that n is still in $\mathbb{N}$, because it denotes a number of steps in the instrumented cost semantics.) In practice, it is more natural for specifications to require a positive amount of credits in the precondition and zero credit in the postcondition. Nevertheless, the previous example of `walk` (§6.3.1) illustrates how in some situations it can be useful to have a negative amount of credits in the postcondition: in some sense, the program "returns a debt", that must be disposed of by its caller.

Independently of this subtlety, this definition of triples implies the expected guarantee for a full program execution:

Corollary 6.3.2 (Triple Interpretation for a Full Program Execution) *Given an initial number of credits c and a pure postcondition Q, the following holds:*

$$\{\$c\}\ t\ \{[Q]\} \implies \exists n\, v\, m.\ t/\varnothing \Downarrow^n v/m \ \wedge\ Q\ v \ \wedge\ c \geq n$$

That is, if $\{\$c\}\ t\ \{[Q]\}$ holds, then t terminates safely in an most c computation steps, and reduces to a value that satisfies the postcondition Q.

The updated soundness theorem reads as follows, similarly to the soundness theorem for Time Credits in $\mathbb{N}$.

Theorem 6.3.3 (Soundness of Separation Logic with Time Credits in $\mathbb{Z}$) *Each of the reasoning rules of Separation Logic with Time Credits in $\mathbb{N}$ (Figure 2.4 where rule* APP *is swapped with* APP-PAY*) is valid with respect to the interpretation of triples given by Definition 6.3.1.*

We have successfully updated our pre-existing Coq proof of this result [CP17b]; an updated proof is available online [Cha19a].

The definition of triples above makes it clear that they are "invariant by translation", as first illustrated with `walk`. Namely, the following equivalence holds for any n, m, H, t, Q:

$$\forall p.\ \{H \star \$n\}\ t\ \{Q \star \$m\} \iff \{H \star \$(n+p)\}\ t\ \{Q \star \$(m+p)\}.$$

In practice, it seems convenient in most cases to state specifications with either a positive amount of credits in the precondition and no credits in the postcondition; or no credits in the precondition and a negative amount of credits in the postcondition (this allows the cost to depend on the return value, as with `walk`).

6.3.3 Perspectives on Partial Correctness

In this section, we informally discuss how Time Credits (either in $\mathbb{N}$ or in $\mathbb{Z}$) interact with other features of the program logic, and in particular, whether it is a logic of partial or total correctness.

In the previous section, we present Separation Logic with Time Credits in $\mathbb{Z}$ as an extension of a logic of *total correctness*. Additionally, the fact that the base logic we consider does not require every assertion to be affine makes our life significantly easier (we only have to amend the definition of the GC predicate to accommodate for possibly negative time credits).

We now describe the properties of several variants of this setup, by considering: (1) whether Time Credits are in $\mathbb{N}$ or in $\mathbb{Z}$; (2) whether an unrestricted recursion rule for recursive functions (such as the REC rule shown below) is present or not; (3) whether every assertion is affine or if the logic allows some assertions to be linear. For each combination of these features, we describe whether the resulting logic is a logic of total correctness, partial correctness, or unsound.

The REC rule that appears below corresponds to the standard unrestricted rule for reasoning on recursive functions. It asserts that, to prove a specification about a recursive function, it is enough to prove the body of the function by assuming the same specification for recursive calls. (In the rule, z stands for an auxiliary variable that might be bound in H or Q.) The presence of such a rule generally means that the resulting logic is a

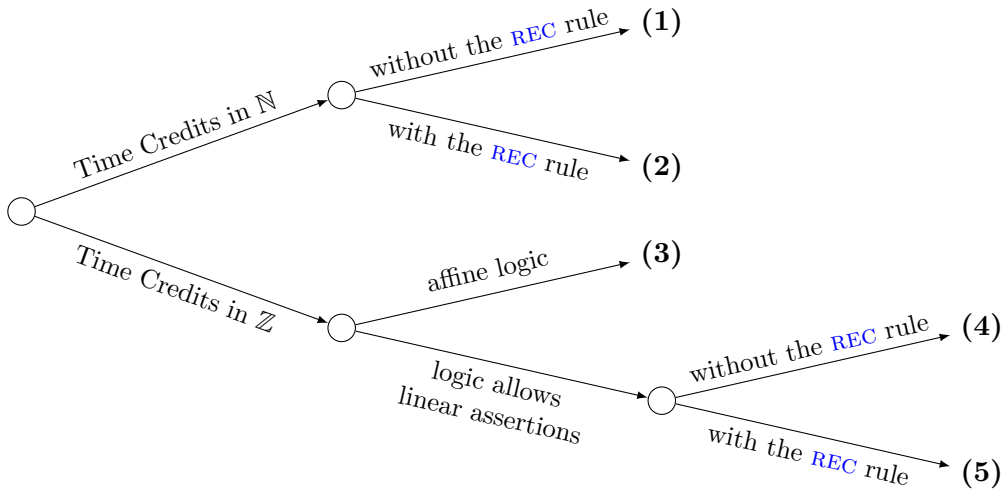

Figure 6.5: Flavors of Separation Logic with Time Credits

logic of partial correctness; however, we will see that this is not necessarily the case in presence of time credits. Without this rule, proving a specification for a recursive function requires reasoning by induction in the ambient logic. This is what we do in the rest of this dissertation (our ambient logic being the logic of Coq).

$$\frac{\text{REC}\quad \forall f\, x\, z.\ (\forall x\, z.\ \{H\}\ f\ x\ \{Q\}) \implies \{H\}\ t\ \{Q\}}{\forall z.\ \{H\}\ (\mu f.\lambda x.t)\ x\ \{Q\}}$$

Figure 6.5 lists the different flavors of Separation Logic with Time Credits that we consider. Notice that in the case where Time Credits are in $\mathbb{N}$, it does not in fact matter whether the logic requires all assertions to be affine or not (because in that case credits are always affine).

Variant (1) (Time Credits in $\mathbb{N}$, no REC rule) corresponds to the Separation Logic introduced previously in Section 2.2. It is a logic of *total correctness*.

Variant (2) (Time Credits in $\mathbb{N}$, with the REC rule) corresponds to the extension of Iris with Time Credits [MJP19]. (Technically speaking, in this setting, the REC rule is a consequence of Löb rule and the definition of `wp`.) Iris is a logic of partial correctness. Nevertheless, the extension of Iris with Time Credits in $\mathbb{N}$ entails *total correctness* [MJP19, Theorem 1]. In other words, if $\{\$n\}\ e\ \{\mathsf{True}\}$ holds, then e is safe and terminates in at most n steps. Intuitively, because Mével *et al.* model running out of time credits as a crash, establishing safety also ensures termination.

Interestingly, one could consider a weaker logic of partial correctness, based on Time Credits in $\mathbb{N}$, the REC rule, and the additional rule DIVERGE-WITH-MORE-CREDITS shown below. In that setting, if $\{\$n\}\ e\ \{\mathsf{True}\}$ holds, then e is safe and either terminates in at most n steps *or diverges*. However, it is unclear whether this weaker variant would be of practical interest.

$$\frac{\text{DIVERGE-WITH-MORE-CREDITS}\quad \{\$(n+1)\}\ e\ \{\mathsf{False}\}}{\{\$n\}\ e\ \{\mathsf{False}\}}$$

Variant (3) (Time Credits in $\mathbb{Z}$, a logic where every assertion is affine) is *unsound*. In that setting, one can create credits out of thin air: $\$0 \vdash \$1$ holds. Indeed, the following sequence of entailments holds, by using the splitting rule of credits then discarding $\$(-1)$: $\$0 \vdash \$(1 + -1) \vdash \$1 \star \$(-1) \vdash \$1$. The amount of time credits that appear in pre or post conditions therefore provide no guarantee with respect to the program execution. Consequently, implementing Time Credits in $\mathbb{Z}$ in Iris (which is an affine logic) would be a challenge. One could perhaps consider working in a layered logic, built on top of the Iris base logic, in the style of Iron [BGKB19].

Variant (4) (Time Credits in $\mathbb{Z}$, linear negative credits, no REC rule) corresponds to the logic presented in the previous section (§6.3). It is a logic of *total correctness*.

Variant (5) (Time Credits in $\mathbb{Z}$, linear negative credits, with the REC rule) corresponds to a logic of *partial correctness*. If $\{\$n\}\ e\ \{\mathsf{True}\}$ holds, then e is safe and either terminates in at most n steps or diverges. Using the REC rule, one can prove that the looping function $\mathsf{loop} \triangleq \mu f.\lambda().f\ ()$ satisfies the specification $\forall n.\ \{\$n\}\ (\mathsf{loop}\ ())\ \{\mathsf{False}\}$. With that caveat in mind, we believe that this variant does constitute a somewhat reasonable proof framework, as it is usually well understood that partial correctness logics do provide weaker guarantees for programs that may not terminate.

6.4 Proof Techniques for Credit Inference

The example proof that we present in Section 6.1 relies on a *cost synthesis* mechanism, able to (semi-)automatically synthesize a syntax-directed cost expression for a straight-line piece of code. There, it is used to elaborate a cost expression for the body of `bsearch`.

In this section, we describe several approaches that we investigated to implement such a synthesis mechanism. From a design perspective, we are interested in approaches that are *mostly* automated, simple-minded and predictable. We want the synthesis mechanism to only do the obvious, without relying on heuristics, and without "loss of generality". Since we are using an interactive proof assistant, all reasoning steps that require external insight should be left to the user. As such, our synthesis mechanisms are only semi-automated. As previously highlighted with the `bsearch` example (§6.1), there are two main situations where the synthesis process may require user input: when eliminating a local variable, and when inferring the cost of a conditional. Another reasoning step that typically requires human insight is to perform approximations, in order to simplify the cost expression. However, this can generally be done in a second step, and does not need to happen during the synthesis process.

The first approach (§6.4.2) is a minimalistic one, where we use the subtraction rule of Time Credits in $\mathbb{Z}$ to push through the proof an "available amount of credits", subtracting from it as we go. In the second approach (§6.4.3), we infer cost expressions by using Coq metavariables to represent amounts of credits that are yet to be synthesized. Finally, the third approach (§6.4.4) is a refinement of the first one, where the user accumulates a debt through the proof; the amount of the debt effectively corresponds to the inferred cost of the various operations. These approaches are described roughly in historical order of our "discovery"—this should give some insight on the iterative thought process behind them.

Before diving into the technical details, let us equip ourselves with a few simple example programs, which we will use to evaluate the different synthesis mechanisms.

$$\{\$1\}\ \texttt{pay}\ ()\ \{\lambda().\ [\,]\} \qquad \forall n.\ \{\$1 \star [n \geq 0]\}\ \texttt{rand}\ n\ \{\lambda m.\ 0 \leq m \leq n\}$$

$$\begin{array}{l}\forall n.\ \{\$\mathit{cost\_loop}(n)\}\ \texttt{loop}\ n\ \{\lambda().\ [\,]\}\\ \wedge\ \text{nonnegative}\ \mathit{cost\_loop}\\ \wedge\ \text{monotonic}\ \mathit{cost\_loop}\\ \wedge\ \mathit{cost\_loop} \preceq_{\mathbb{Z}} \lambda n.\ n\end{array}$$

Figure 6.6: Separation Logic specifications for the `pay`, `rand` and `loop` auxiliary operations.

For convenience purposes, in these programs we will assume the existence of three base operations, `pay`, `rand` and `loop`, for which a Separation Logic specification is given in Figure 6.6. The `pay` operation has already been mentioned previously (§2.2.1): its role is to consume a single time credit. Usually, a call to `pay` is automatically inserted at the beginning of every function and loop; here, for clarity, all calls to `pay` have been made explicit. The `rand` operation models the case of a specification which does not fully specify its return result (this is common for real world program specifications). Here, the specification only publishes a bound on the return value of `rand`: the output value is comprised between zero and the input value. Finally, the `loop` operation is more interesting complexity-wise: it consumes a linear number of credits. Following the style of specifications advertised in §4, we keep its cost function abstract, and only publish an asymptotic bound, as well as the fact that the cost function is nonnegative and monotonic.

The example programs themselves appear in Figure 6.7, along with their specification. The specifications we give here anticipate on the synthesis process, and indicate a concrete amount of credits in the precondition. Then, we expect the cost synthesis mechanism to elaborate a cost expression for which we can easily check that it matches the precondition.

The `pay3` example illustrates that the cost of a sequence of operations is simply the sum of the cost of the operations. Typically, we expect the synthesis process to synthesize the cost expression $1 + 1 + 1$.

The `let_bound` example illustrates the situation where a `let`-binding introduces a local name, on which the cost expression depends locally. In first approach, the cost of a `let` expression "`let x = e1 in e2`" is the same as the cost of a sequence "`e1; e2`": it is the sum of the cost `e1` and the cost of `e2`. Here, the cost of calling `loop m` is $\mathit{cost\_loop}(m)$ (per the specification of `loop`), so the synthesized cost expression for `let_bound` would be $1 + 1 + \mathit{cost\_loop}(m) + 1$. However, m is a local name; the final cost expression must only depend on the parameter n. We could systematically eliminate the dependency by generating a "max" over all possible values of m, and let the user perform an approximation or simplification (if desired) later on (see the discussion in Section 4.2.1). We choose instead to explicitly require the user to eliminate the dependency, because we feel that it is a simpler option in our setting. At some point or another, the synthesis mechanism must therefore give to the user the possibility of eliminating m in the cost expression. Here, the situation is similar to the example involving `List.filter` in Section 4.2.1: thanks to the fact that *cost_loop* is monotonic, we can eliminate m by bounding $\mathit{cost\_loop}(m)$ with $\mathit{cost\_loop}(n)$. The synthesis mechanism must therefore allow

```
let pay3 () =
  pay (); pay (); pay ()
```

$\{\$3\}$ `pay3 ()` $\{\lambda().\ []\}$

```
let let_bound n =
  pay ();
  let m = rand n in
  loop m;
  pay ()
```

$\forall n.$
$\{\$(cost\_loop(n) + 3)\}$
`let_bound` n
$\{\lambda().\ []\}$

```
let if_rand n cond =
  pay ();
  let a = rand n in
  let b = rand n in
  if cond
    then loop a
    else loop b
```

$\forall n\ cond.$
$\{\$(cost\_loop(n) + 3)\}$
`if_rand` $n\ cond$
$\{\lambda().\ []\}$

```
let rec myloop n =
  pay ();
  if n <= 0
    then ()
    else myloop (n-1)
```

$\forall n.$
$\{\$(1 + \max(0, n))\}$
`myloop` n
$\{\lambda().\ []\}$

Figure 6.7: Example programs and their specifications

the user to bound the inferred cost expression using inequalities available in the proof context.

The `if_rand` example shows the case of a conditional where the condition is not relevant to the complexity analysis. In that case, the synthesized expression should express the cost of the conditional as a max, and be of the form $1{+}1{+}1{+}\max(cost\_loop(a), cost\_loop(b))$ which can be bounded by $3 + cost\_loop(n)$.

Finally, `myloop`, although not a straight-line program, shows an instance where expressing the cost of a conditional as a max would be harmful, as it loses information. Indeed, it would lead to the following recurrence equation to be synthesized (where *cost* is the unknown of the equation): $\forall n.\ cost(n) \geq 1 + max(0, cost(n-1))$. This recurrence equation is unsatisfiable. Instead, the conditional in the program should be reflected as a conditional in the synthesized cost expression. In that case, we obtain the following recurrence equation for *cost*: $\forall n.\ cost(n) \geq 1 + (\mathsf{If}\ n \leq 0\ \mathsf{then}\ 0\ \mathsf{else}\ cost(n-1))$, which does have a solution. These last two examples illustrate that the user needs to be able to control how conditional expressions are treated by the synthesis process.

Using these examples, we next present three approaches for cost synthesis.

6.4.1 No Inference: Manual Splitting of Credits

As a baseline, let us first briefly illustrate the case of fully manual proofs, without using any inference mechanism. In that case, the proof starts from a concrete amount of credits, that the user must split successively in order to pay for the individual operations. Note

that what we describe here works similarly for both Time Credits in $\mathbb{N}$ and Time Credits in $\mathbb{Z}$.

Paying for a sequence of operations Let us illustrate this process on the `pay3` example. Starting with the specification in Figure 6.7, and after replacing `pay3` with its definition, we obtain the following initial goal:

$$\{\$3\}\ \texttt{pay(); pay(); pay()}\ \{\lambda().\ []\}.$$

Then, we step through the first call to `pay()`. The specification for `pay()` requires that we provide one time credit. Consequently, we get the following two sub-goals:

$$\$3 \Vdash \$1 \star ?F \quad \wedge \quad \{?F\}\ \texttt{pay(); pay()}\ \{\lambda().\ []\}.$$

In the first subgoal, we are required to prove that the resources that we have ($3) entail the conjunction of what we must pay ($1) and "the rest" ($?F$), which we get to pay for the rest of the proof. The term $?F$ denotes a Coq metavariable—a temporarily un-instantiated hole in the proof—which has been introduced by CFML: it represents the framed heap predicate around the call to `pay`. In order to solve the first subgoal, we split $3 into $2 $\star$ $1, using the fact that credits are additive.

$$\$2 \star \$1 \Vdash \$1 \star ?F$$

The $1 can now be cancelled out on both sides of the entailment; finally, solving the subgoal is done by instantiating $?F$ with the leftover resources ($2), which get propagated as the precondition for the rest of the proof.

$$\{\$2\}\ \texttt{pay(); pay()}\ \{\lambda().\ []\}$$

The rest of the proof is similar. Once all calls to `pay()` have been stepped through, the final entailment is $0 $\Vdash$ [] $\star$ GC, which trivially holds (the GC predicate accounts for the possibility of discarding some leftover resources, but is not useful here).

Bounding the cost of an operation Let us now consider the `let_bound` example. Again, starting from the specification in Figure 6.7, after unfolding the definition of `let_bound`, the initial goal is as follows:

$$\{\$\mathit{cost\_loop}(n) + 3\}\ \texttt{pay(); let m = rand n in loop m; pay()}\ \{\lambda().\ []\}.$$

Stepping through the calls to `pay` and `rand` consumes two time credits, in a similar fashion as in the proof of `pay3`. We get to the following goal for the call to `loop`. At this point, thanks to the Separation Logic rule for `let`, we have a variable m in the proof context, along with the fact that it is bounded by n.

$$0 \le m \le n \implies \{\$(\mathit{cost\_loop}(n) + 1)\}\ \texttt{loop}\ m\texttt{; pay()}\ \{\lambda().\ []\}$$

Applying the specification for `loop` yields the following sub-goals:

$$0 \le m \le n \implies \begin{array}{ll} & \$(\mathit{cost\_loop}(n) + 1) \Vdash \$\mathit{cost\_loop}(m) \star ?F \\ \wedge & \{?F\}\ \texttt{pay()}\ \{\lambda().\ []\}. \end{array}$$

$$\textsc{SubCredits}\ \frac{\$(n-m) \star H \ \Vdash\ Q}{\$n \star H \ \Vdash\ \$m \star Q} \qquad \textsc{GcCredits}\ \frac{n \geq 0}{\$n \ \Vdash\ \mathrm{GC}}$$

Figure 6.8: Reasoning principles on credits when using the Subtraction Rule

In order to solve the first subgoal, we need to split $\$(\mathit{cost\_loop}(n)+1)$ to extract $\$\mathit{cost\_loop}(m)$. Thanks to the fact that *cost_loop* is monotonic, we can use the splitting rule twice to rewrite the first subgoal as:

$$\$(\mathit{cost\_loop}(n) - \mathit{cost\_loop}(m)) \star \$1 \star \$\mathit{cost\_loop}(m) \ \Vdash\ \$\mathit{cost\_loop}(m) \star ?F.$$

This allows paying for $\mathit{cost\_loop}(m)$; then, the metavariable $?F$ can be instantiated with the remaining credits $\$(\mathit{cost\_loop}(n) - \mathit{cost\_loop}(m)) \star \$1$. We are left with a goal for the final call to `pay`:

$$\{\$(\mathit{cost\_loop}(n) - \mathit{cost\_loop}(m)) \star \$1\}\ \texttt{pay()}\ \{\lambda().\ [\,]\}.$$

The proof can be completed; with the leftover credits being simply discarded.

At this point, it should be evident that *manually* following this reasoning style is very tedious and unpractical. The process of successively splitting the initial amount of credits can be automated (see for instance the work of Zhan and Haslbeck [ZH18], who integrate with the auto2 prover [Zha18]). Nevertheless, this workflow fundamentally requires that the user provide upfront a concrete cost expression for the whole program, in a form that is syntactically close to the syntax of the program, so that the automation (which operates in a syntax directed manner) can succeed.

6.4.2 Using the Subtraction Rule

The motivation for the developments that we present next is to get rid of the requirement of the user guessing concrete cost functions.

A first attempt is to use the Subtraction Rule, already mentioned in Section 6.3.1. The idea is that we can pay for the cost of an operation by simply subtracting the cost from the amount of credits that we already have, instead of manually splitting credits. We are effectively pushing downwards an "available amount of credits" and subtracting from it as we go. So, in a sense, we are in "checking mode" (in the sense of bidirectional type checking), but at the same time, we are synthesizing a negated cost.

Historically, we first investigated this approach using Time Credits in $\mathbb{N}$. As explained earlier (§6.3.1), the approach is not practical in that setting, because of the numerous redundant side-conditions that it produces. When using Time Credits in $\mathbb{Z}$, however, this simple minded approach works relatively well. It is very lightweight: the framework only needs to extend the way heap entailments are handled, by implementing the two reasoning principles, displayed in Figure 6.8, and explained next.

Paying for a sequence of operations Assuming Time Credits in $\mathbb{Z}$, let us consider the `pay3` example. Just like previously, stepping through the first call to `pay()` yields the following two sub-goals:

$$\$3 \Vdash \$1 \star ?F \quad \wedge \quad \{?F\}\ \texttt{pay(); pay()}\ \{\lambda().\ []\}.$$

Applying the rule SubCredits on the first sub-goal yields:

$$\$(3-1) \Vdash ?F.$$

The first sub-goal can now be easily discharged by instantiating $?F$ with $\$(3-1)$. The proof continues with the following goal:

$$\{\$(3-1)\}\ \texttt{pay(); pay()}\ \{\lambda().\ []\}.$$

Eventually, after similarly stepping through the remaining calls to `pay()`, we get the following final proof obligation:

$$\$(3-1-1-1) \Vdash [] \star \mathrm{GC}.$$

After simplification, the rule GcCredits can be applied, producing the following arithmetic goal, which is here easily discharged.

$$3-1-1-1 \geq 0$$

Remark that at no point during the proof the reasoning rules relied on the expression of the initial amount of credits ($3). The synthesized cost expression takes the form of a series of costs that are subtracted from it; only in the very last goal one has to reason about the initial amount of credits, in order to show that it is indeed sufficient to pay for the inferred cost.

Bounding the cost of an operation By applying the same mechanism on the `let_bound` example (using SubCredits to pay for operations and using GcCredits at the very end), we obtain the following final subgoal:

$$0 \leq m \leq n \implies$$
$$(\mathit{cost\_loop}(n)+3)-1-1-\mathit{cost\_loop}(m)-1 \geq 0.$$

As before, this subgoal can be discharged using the fact that *cost_loop* is monotonic. More importantly, this example illustrates that in the present setup, it is possible to wait until the very end before eliminating local variables in the cost expression. In other words, this "checking mode" approach removes the need to eliminate the local variable by introducing a max or by bounding it; instead, the local variable is universally bound in the proof obligation that ultimately results.

On the one hand, this is quite convenient: it means that the synthesis process does not interfere with the verification process, and the user only needs to worry about local variables at the very end, after stepping through the whole program. On the other hand, this means that eliminating local variables has to be performed on the final cost expression, which might be quite large for real world programs. Instead, it might be

preferable to eliminate local variables in cost expressions immediately after applying the Separation Logic reasoning rule for `let`, that is, at the same time as reasoning locally about the corresponding part of the program.

We will see how the approach described next in Section 6.4.3 behaves differently in that regard, and how we might get the best of both worlds with the approach presented in Section 6.4.4.

Paying for a conditional expression In the case of a program involving a conditional expression—`if_rand`, for example—applying the standard Separation Logic rule for conditionals (Rule IF) splits the proof into two sub-goals, one for each branch. Consequently, in the present setup, instead of synthesizing an "if" or a "max", we synthesize a conjunction of two proof obligations.

Let us consider `if_rand`. After stepping through the initial calls to `pay` and `rand`, the goal is:

$$
\begin{array}{l}
0 \le a \le n \ \wedge \ 0 \le b \le n \implies \\
\{\$((\mathit{cost\_loop}(n) + 3) - 1 - 1 - 1)\} \\
\quad \texttt{if cond then loop}\ a\ \texttt{else loop}\ b \\
\{\lambda().\ []\}.
\end{array}
$$

After applying the standard Separation Logic rule for `if`, we obtain the two subgoals that follow.

$$
\begin{array}{l}
0 \le a \le n \ \wedge \ 0 \le b \le n \ \wedge \ \mathit{cond} = \mathsf{true} \implies \\
\{\$((\mathit{cost\_loop}(n) + 3) - 1 - 1 - 1)\}\ \texttt{loop}\ a\ \{\lambda().\ []\} \\
\quad \wedge \\
0 \le a \le n \ \wedge \ 0 \le b \le n \ \wedge \ \mathit{cond} = \mathsf{false} \implies \\
\{\$((\mathit{cost\_loop}(n) + 3) - 1 - 1 - 1)\}\ \texttt{loop}\ b\ \{\lambda().\ []\}.
\end{array}
$$

By stepping through the rest of the program in each subgoal, we obtain the following two proof obligations:

$$
\begin{array}{l}
0 \le a \le n \ \wedge \ 0 \le b \le n \ \wedge \ \mathit{cond} = \mathsf{true} \implies \\
(\mathit{cost\_loop}(n) + 3) - 1 - 1 - 1 - \mathit{cost\_loop}(a) \ge 0 \\
\quad \wedge \\
0 \le a \le n \ \wedge \ 0 \le b \le n \ \wedge \ \mathit{cond} = \mathsf{false} \implies \\
(\mathit{cost\_loop}(n) + 3) - 1 - 1 - 1 - \mathit{cost\_loop}(b) \ge 0.
\end{array}
$$

In other words, the cost expression synthesized for the first branch is $1 + 1 + 1 + \mathit{cost\_loop}(a)$, and the one synthesized for the second branch is $1 + 1 + 1 + \mathit{cost\_loop}(b)$.

Again, by monotonicity of $\mathit{cost\_loop}$, the previous proof obligations can be simplified into the following simple one, which can now be easily discharged.

$$(\mathit{cost\_loop}(n) + 3) - 1 - 1 - 1 - \mathit{cost\_loop}(n) \ge 0$$

Note that we discard the information from the context, in particular the value of the condition: in this case, it is not needed for analyzing the final cost expression.

For completeness, let us illustrate what happens when a conditional *is* relevant for the complexity analysis, with the `myloop` example. Under the assumption that the cost

of `myloop n` is $cost(n)$, similar application of the reasoning rules yields the following two subgoals:

$$\begin{array}{rrl} & n \leq 0 & \implies cost(n) - 1 \geq 0 \\ \wedge & n > 0 & \implies cost(n) - 1 - cost(n-1) \geq 0. \end{array}$$

In that case, information from the context must not be discarded. Including the assumptions on n, the resulting proof obligations define the recurrence equations for *cost* that one would expect, and which can be solved as described previously (§6.1).

Summary Assuming that one can use Separation Logic with Time Credits in $\mathbb{Z}$, we believe that the approach presented here, based on the Subtraction Rule, is the "least effort, best value" solution for dealing with functions involving just a few lines of code. It can be implemented easily, by simply adding support for the two rules of Figure 6.4.2, and already goes a long way.

There are a few pain points from a usability perspective, though. First, during verification, the precondition of the goal is polluted by the—possibly large—cost expression being inferred. The final inequalities that are produced involve subtractions, while it would be more natural to present an inequality between the initial amount of credits and the inferred cost. Finally, eliminating local parameters occurs at the very end, where one has to tackle the entire synthesized cost expression, which again can be quite large and verbose in practice. In summary, there are issues for scaling up the approach.

In Section 6.4.4, we present an approach ("Debt Collection") which is technically very similar, but with usability improvements which address the issues above, inspired from what we present next in Section 6.4.3.

6.4.3 Using Coq Evars as Placeholders

In this section, we describe the approach that we have used and experimented with the most. Notably, it is used through most of the proofs of our main case study, described in Section 8. It is a more heavyweight approach—compared to the one presented earlier—and in particular requires instrumenting the Separation Logic reasoning rules of the framework. Interestingly, this approach is compatible with both Time Credits in $\mathbb{N}$ and in $\mathbb{Z}$, which makes it applicable in situations where extending the logic with integer credits would not be an option.

The key idea of the present approach is to use Coq "evars" in order to represent cost expressions that are yet to be synthesized. Coq evars [The19b, §2.11]—that we also sometimes call metavariables, or holes—are *placeholder* terms, which are introduced at a given point during a proof, and are instantiated at a later point.

We use specialized variants of the Separation Logic reasoning rules, whose premises and conclusions take the form $\{\$\, n \star H\}\ (e)\ \{Q\}$. Furthermore, to mark placeholder cost expressions explicitly, and make them easily recognizable by tactics, we wrap them as $\langle c \rangle$. Here, $\langle \cdot \rangle$ is simply defined as the identity:

$$\langle c \rangle \triangleq c.$$

Because we wish to synthesize a cost expression, Coq tactics maintain the following invariant: whenever the goal is $\{\$\, \langle c \rangle \star H\}\ (e)\ \{Q\}$, the cost c is *uninstantiated*: it is represented in Coq as an evar.

$$\textsc{WeakenCost}\ \frac{\{\$\,\langle c_2\rangle \star H\}\ (e)\ \{Q\} \qquad c_2 \le c_1}{\{\$\,c_1 \star H\}\ (e)\ \{Q\}} \qquad \textsc{Pay}\ \frac{H \ \Vdash\ Q()}{\{\$\,\langle 1\rangle \star H\}\ (\mathrm{pay}())\ \{Q\}}$$

$$\textsc{Seq}\ \frac{\{\$\,\langle c_1\rangle \star H\}\ (e_1)\ \{Q'\} \qquad \{\$\,\langle c_2\rangle \star Q'()\}\ (e_2)\ \{Q\}}{\{\$\,\langle c_1 + c_2\rangle \star H\}\ (e_1; e_2)\ \{Q\}}$$

$$\textsc{Let}\ \frac{\{\$\,\langle c_1\rangle \star H\}\ (e_1)\ \{Q'\} \qquad \forall x.\ \{\$\,\langle c_2\rangle \star Q'(x)\}\ (e_2)\ \{Q\}}{\{\$\,\langle c_1 + c_2\rangle \star H\}\ (\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2)\ \{Q\}}$$

$$\textsc{IfIf}\ \frac{\begin{array}{c} b = \mathrm{true} \Rightarrow \{\$\,\langle c_1\rangle \star H\}\ (e_1)\ \{Q\} \\ b = \mathrm{false} \Rightarrow \{\$\,\langle c_2\rangle \star H\}\ (e_2)\ \{Q\} \end{array}}{\{\$\,\langle \mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2\rangle \star H\}\ (\mathrm{if}\ b\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2)\ \{Q\}}$$

$$\textsc{IfMax}\ \frac{\begin{array}{c} b = \mathrm{true} \Rightarrow \{\$\,\langle c_1\rangle \star H\}\ (e_1)\ \{Q\} \\ b = \mathrm{false} \Rightarrow \{\$\,\langle c_2\rangle \star H\}\ (e_2)\ \{Q\} \end{array}}{\{\$\,\langle \max(c_1, c_2)\rangle \star H\}\ (\mathrm{if}\ b\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2)\ \{Q\}}$$

$$\textsc{For}\ \frac{\forall i.\ a \le i < b\ \Rightarrow\ \{\$\,\langle c(i)\rangle \star I(i)\}\ (e)\ \{I(i+1)\} \qquad H\ \Vdash\ I(a) \star Q}{\{\$\,\langle \Sigma_{a \le i < b}\ c(i)\rangle \star H\}\ (\mathrm{for}\ i = a\ \mathrm{to}\ b-1\ \mathrm{do}\ e\ \mathrm{done})\ \{I(b) \star Q\}}$$

Figure 6.9: The reasoning rules of Separation Logic, specialized for cost synthesis using Coq evars. In the conclusion of the rules, the expression between angle brackets $\langle\cdot\rangle$ must be read as *instantiating* an evar.

Custom Separation Logic rules As such, our synthesis rules work with triples of the form $\{\$\,\langle c\rangle \star H\}\ (e)\ \{Q\}$. They are shown in Figure 6.9. The uninstantiated cost metavariable appearing in a goal is instantiated when the goal is proved by applying one of the reasoning rules. Such an application produces new subgoals, whose preconditions contain new metavariables. As this process is repeated, a cost expression is incrementally constructed.

The rule WeakenCost is a special case of the consequence rule of Separation Logic. It is typically used once at the root of the proof: even though the initial goal $\{\$\,c_1 \star H\}\ (e)\ \{Q\}$ may not satisfy our invariant, because it lacks a $\langle\cdot\rangle$ wrapper and because c_1 is typically not a metavariable, WeakenCost gives rise to a subgoal $\{\$\,\langle c_2\rangle \star H\}\ (e)\ \{Q\}$ that satisfies it. Indeed, when this rule is applied, a fresh metavariable c_2 is generated. WeakenCost can also be explicitly applied by the user when desired. For instance, it can be used if the user anticipates that they want to approximate the cost expression that will be synthesized at a given point.

$$\textsc{InstRHS}\ \frac{H \Vdash Q}{\$\langle c\rangle \star H \Vdash \$c \star Q} \qquad \textsc{InstZero}\ \frac{H \Vdash Q}{\$\langle 0\rangle \star H \Vdash Q}$$

$$\textsc{PayRHS}\ \frac{\$\langle c\rangle \star H \Vdash Q}{\$\langle c+p\rangle \star H \Vdash \$p \star Q} \qquad \textsc{Compare}\ \frac{c+\phi \geq p \qquad H \Vdash Q \star \mathrm{GC}}{\$\langle c\rangle \star \$\phi \star H \Vdash \$p \star Q \star \mathrm{GC}}$$

Figure 6.10: Specialized reasoning rules for heap entailments involving cost evars. In the conclusion of the rules, the expression between angle brackets $\langle\cdot\rangle$ must be read as *instantiating* an evar. These rules are not syntax directed: it is ultimately up to the user to apply them appropriately.

The Pay rule handles the `pay()` instruction. This instruction costs one credit: applying this rule therefore instantiates the cost evar with 1.

The Seq rule is a special case of the Let rule. It states that the cost of a sequence is the sum of the costs of its subexpressions. When this rule is applied to a goal of the form $\{\$\,\langle c\rangle \star H\}\ (e)\ \{Q\}$, where c is a metavariable, two new metavariables c_1 and c_2 are introduced, and c is instantiated with $c_1 + c_2$.

The Let rule is similar to Seq, but involves an additional subtlety: the cost c_2 must not refer to the local variable x. This is enforced by Coq: any attempt to instantiate the metavariable c_2 with an expression where x occurs fails. Indeed, the initial cost metavariable was introduced in a context where the variable x does not appear. In such a situation, it is up to the user to rewrite or bound the cost expression synthesized for e_2 so as to avoid this dependency.

The IfIf rule states that the cost of an OCaml conditional expression is a mathematical conditional expression. The alternative rule IfMax can be used instead to approximate the cost of the conditional as a max, which can beneficial since it leads to a simpler cost expression. We let the user decide which rule to apply when reasoning about a conditional expression.

The For rule states that the cost of a `for` loop is the sum, over all values of the index i, of the cost of the i-th iteration of the body. There is in principle no need for a primitive treatment of loops. The cost of a loop can be expressed as part of the loop invariant, using the standard reasoning rule. Loops can also be encoded in terms of higher-order recursive functions, and our program logic can express the specifications of these combinators. Nevertheless, in practice, primitive support for loops is convenient.

One might wonder what happens at the leaves of the proof, i.e. at calls to auxiliary functions (for which we already proved a specification) and when returning values. In these cases, we can use the standard Separation Logic reasoning rules, which we recall below (calling a function translates into using the consequence rule with this function's specification).

$$\textsc{Val}\ \frac{H \Vdash Q(v)}{\{H\}\ (v)\ \{Q\}} \qquad \textsc{Conseq}\ \frac{H \Vdash H' \qquad \{H'\}\ (f\ x)\ \{Q'\} \qquad \forall y.\ Q'(y) \Vdash Q(y)}{\{H\}\ (f\ x)\ \{Q\}}$$

However, there is a catch. This means that we now have to deal with heap entailments of the form $\$\langle c\rangle \star H \Vdash Q$, where c is a metavariable, and Q possibly contains time credits. We introduce specialized reasoning rules on heap entailments for solving this kind of goals. They appear in Figure 6.10.

In most cases, it is sufficient to automatically apply the rule InstRHS to instantiate c with the credits that appear on the right-hand side, or InstZero if there are none. More subtle situations require the user to interactively use the additional rules PayRHS and Compare. The rule PayRHS allows one to successively pay for some (but typically, not every) quantity of credits that appear in the right-hand side. Then, Compare can be used to pay for credits that appear in the right-hand side by bounding them by credits that appear on the left-hand side. These might include credits from other sources (noted ϕ in the rule), such as a potential function. Compare is used typically when the credits to pay p refer to local variables, which prevent instantiating c with p. Applying the rule produces a mathematical inequality, which can be simplified into a bound which does not refer to the local variables, and can be then used to instantiate c.

This concludes our exposition of the reasoning rules of Figures 6.9 and 6.10; let us now come back to the example programs of Figure 6.7.

Paying for a sequence of operations Let us start with the `pay3` program. Just as before, the initial goal after unfolding the definition of `pay3` is the following:

$$\{\$3\}\ \texttt{pay(); pay(); pay()}\ \{\lambda().\ []\}.$$

We start the proof by applying WeakenCost, in order to introduce a cost metavariable $?c$, along a second sub-goal relating it to the initial amount of credits.

$$\{\$\langle ?c\rangle\}\ \texttt{pay(); pay(); pay()}\ \{\lambda().\ []\} \quad \wedge \quad ?c \leq 3$$

Then, we can start applying the rules of the logic as usual, following the syntax of the program. After applying the Seq rule, we get two sub-goals, with two fresh cost evars $?c_1$ and $?c_2$ denoting the costs of the left and right branch, respectively. Notice how $?c$ has been automatically refined into $?c_1 + ?c_2$ in the last sub-goal by the application of the rule.

$$\{\$\langle ?c_1\rangle\}\ \texttt{pay()}\ \{\lambda().\ []\} \quad \wedge \quad \{\$\langle ?c_2\rangle\}\ \texttt{pay(); pay()}\ \{\lambda().\ []\} \quad \wedge \quad ?c_1 + ?c_2 \leq 3$$

The first sub-goal can be discharged by applying the Pay rule, which instantiates $?c_1$ with 1 as a side effect.

$$\{\$\langle ?c_2\rangle\}\ \texttt{pay(); pay()}\ \{\lambda().\ []\} \quad \wedge \quad 1 + ?c_2 \leq 3$$

A further application of the Seq and Pay rules instantiates $?c_2$ with $1 + ?c_3$ (with $?c_3$ a fresh evar), yielding the sub-goals:

$$\{\$\langle ?c_3\rangle\}\ \texttt{pay()}\ \{\lambda().\ []\} \quad \wedge \quad 1 + 1 + ?c_3 \leq 3$$

An application of Pay discharges the first sub-goal and instantiates $?c_3$; we are left with the last sub-goal, where the left hand-side now holds the fully synthesized cost expression.

$$1 + 1 + 1 \leq 3$$

Bounding the cost of an operation Let us now illustrate what happens in the case of cost expressions depending on local variables, with the `let_bound` example. Similarly to the previous example, we start the proof by applying WeakenCost then stepping through the call to `pay()`. This leads to the following goal:

$$\{\$\langle ?c\rangle\}\ \texttt{let m = rand n in loop m; pay()}\ \{\lambda().\ []\}\ \wedge\ 1 + ?c \leq \mathit{cost\_loop}(n) + 3.$$

We proceed by applying the Let rule. Similarly to the Seq rule, this produces two sub-goals with two fresh cost metavariables $?c_1$ and $?c_2$.

$$\begin{array}{l}\{\$\langle ?c_1\rangle\}\ \texttt{rand n}\ \{?Q\}\\ \quad\wedge\\ \forall m.\ \{\$\langle ?c_2\rangle \star ?Q(m)\}\ \texttt{loop}\ m\texttt{; pay()}\ \{\lambda().\ []\}\\ \quad\wedge\\ 1 + ?c_1 + ?c_2 \leq \mathit{cost\_loop}(n) + 3\end{array}$$

One subtlety lurks there. Syntactically, $?c_2$ appears under the binder "$\forall m. \ldots$", so it seems like it could depend on m. However, $?c_2$ has been used to instantiate $?c$, which was introduced in an outer context and cannot depend on m. As per Coq scoping rules, it is now forbidden for $?c_2$ to depend on m. Discharging the first goal poses no particular problem; as per `rand`'s specification, $?Q$ is instantiated with $\lambda m.\ [0 \leq m \leq n]$. After extracting the bound on m ($0 \leq m \leq n$) into the proof context (extract-prop) in the second subgoal, and applying the sequence rule, we get the following goals:

$$\begin{array}{l}0 \leq m \leq n \implies \{\$\langle ?c_2\rangle\}\ \texttt{loop}\ m\ \{?R\}\\ \quad\wedge\\ \{\$\langle ?c_3\rangle \star ?R()\}\ \texttt{pay()}\ \{\lambda().\ []\}\\ \quad\wedge\\ 1 + 1 + ?c_2 + ?c_3 \leq \mathit{cost\_loop}(n) + 3.\end{array}$$

We can reason on the first sub-goal as usual, by applying the Conseq rule with the specification of `loop`. Doing so does not completely discharge the sub-goal: we are left with a heap entailment to prove manually, in order to instantiate the evar $?c_2$. Indeed, one cannot simply instantiate $?c_2$ with $\mathit{cost\_loop}(m)$: $?c_2$ is not allowed to depend on m.

$$\begin{array}{l}0 \leq m \leq n \implies \$\langle ?c_2\rangle \Vdash \$\mathit{cost\_loop}(m) \star \mathrm{GC}\\ \quad\wedge\\ \{\$\langle ?c_3\rangle \star []\}\ \texttt{pay()}\ \{\lambda().\ []\}\\ \quad\wedge\\ 1 + 1 + ?c_2 + ?c_3 \leq \mathit{cost\_loop}(n) + 3\end{array}$$

To discharge the first sub-goal, we apply the Compare rule. It yields the following inequality (along with a trivial heap entailment which can be discharged immediately):

$$?c_2 \geq \mathit{cost\_loop}(m).$$

This gives to the user the opportunity of bounding the synthesized cost expression (on the right-hand side) with another expression, which must not depend on m. Here, by transitivity of $\geq$ and because *cost_loop* is monotonic, one can turn the goal into:

$$?c_2 \geq cost\_loop(n).$$

Then, one can finally discharge the goal by reflexivity of $\geq$, instantiating $?c2$ with $cost\_loop(n)$. Zooming out, after handling the last call to `pay()` in the second subgoal—instantiating $?c_3$ with 1— the remaining proof obligation corresponds to the inequality between the synthesized cost, and the initial amount of credits.

$$1 + 1 + cost\_loop(n) + 1 \leq cost\_loop(n) + 3$$

Inferring the cost of a conditional expression Inferring the cost of conditional expressions comes with no particular difficulty. On the `if_rand` example, exploiting the IfMax rule yields the following final goal:

$$1 + 1 + 1 + \max(cost\_loop(n), cost\_loop(n)) \leq cost\_loop(n) + 3.$$

On the `myloop` example, one must use the IfIf rule instead. Under the assumption that $cost(n)$ describes the cost of `myloop n`, application of the rules yields:

$$1 + (\mathsf{If}\ n \leq 0\ \mathsf{then}\ 0\ \mathsf{else}\ cost(n-1)) \leq cost(n).$$

Interestingly, these two examples illustrate that even in the presence of conditionals, the result of cost synthesis is always a single expression. This is unlike the previous approach, where several cost expressions would be synthesized; one for each branch in the program.

Summary The approach that we present here is, at this point, the one that we have used and tested the most [GCP18]. It is also nicer than the approach of the previous section on several aspects. First, when working on the bulk of a proof, the main goal is not polluted by the cost expression that has been (partially) synthesized. Instead, at every moment of the proof, the pre-condition only contains an evar $\langle ?c \rangle$; the global cost expression only appears in a separate sub-goal. This separate sub-goal can be left on the side until the very end of the proof. Additionally, whenever manual intervention is required to eliminate a local variable, subsequent reasoning only needs to occur on the relevant local cost expression. This is particularly helpful when verifying larger programs, where the synthesized cost expression for the whole program can be very verbose. Finally, the present approach works even in the standard setting where Time Credits live in $\mathbb{N}$.

There are drawbacks, however. From an implementation perspective, this approach is fairly heavyweight—it requires extending most of the standard Separation Logic reasoning rules, as well as the rules for handling heap entailments. From a usability perspective, the strict syntactic discipline imposed (for good reasons) by Coq on the context of evars means that they can get slightly tricky to handle in practice.

$$\$[\![n]\!] \triangleq \$(-n) \quad \text{(displayed as "DEBT", hiding } n\text{)}$$

$$\textsc{PayAsDebt}\ \frac{\$n \star \$(-m) \star H \Vdash Q}{\$n \star H \Vdash \$m \star Q} \qquad \textsc{ApproxDebt}\ \frac{n \le m \qquad \{\$(-m) \star H\}\ (e)\ \{Q\}}{\{\$(-n) \star H\}\ (e)\ \{Q\}} \qquad \textsc{DebtIntro}\ \frac{\{\$[\![0]\!] \star H\}\ (e)\ \{Q\}}{\{H\}\ (e)\ \{Q\}}$$

$$\textsc{CollectDebt}\ \frac{\$[\![n+m]\!] \star H \Vdash Q}{\$[\![n]\!] \star \$(-m) \star H \Vdash Q} \qquad \textsc{RepayDebt}\ \frac{n \ge m}{\$n \star \$[\![m]\!] \Vdash \mathrm{GC}}$$

Figure 6.11: Reasoning rules on debts

6.4.4 Collecting Debts

The approach that we describe in this section is an attempt at getting the "best of both worlds" from the approaches described in 6.4.2 and 6.4.3. What we present next is possibly the best approach overall. However, at this time, it has not been exercised as much as the approach of the previous section, especially on larger examples—therefore our assessment should be taken with a grain of salt.

In this approach, we require Time Credits to be in $\mathbb{Z}$. The setup is in essence quite similar to our previous use of the subtraction rule (§6.4.2), but with a few twists: inspired by the advantages of the approach using evars (§6.4.3), we implement usability improvements that try to get the best of the two approaches.

From a high-level perspective, the idea of the present approach is as follows: when we have to pay for an operation, we can instead accumulate a *debt*. At the end of the proof, the user has to justify that they can pay off their debts. In fact, the amount of the debts corresponds to the cost of the program, which is synthesized in the process. This is very similar to subtracting the cost of operations from the initial amount of credits, but enables a nicer workflow. In particular, debts and credits can be represented as separate resources in the goal, and the concrete expression of the successive debts that have been collected up to a point can be hidden using a Coq notation, so as not to visually clutter the goal.

A debt of n credits corresponds to the ownership of $-n$ credits, written $\$(-n)$. The reasoning rules on debts appear in Figure 6.11. Paying for an operation is done using the PayAsDebt rule, which turns the cost of the operation into a debt. This rule is reminiscent of the SubCredits rule of Section 6.4.2. Then, the cost of a single operations can be bounded using rule ApproxDebt (typically to eliminate a local variable)—by bounding the corresponding debt.

In the case of large programs, accumulating many debts in the pre-condition would make the goal visually cluttered. We could decide to systematically hide all debts, but that would prevent the interactive use of ApproxDebt. Instead, we introduce an auxiliary definition $\$[\![n]\!]$, which is an alias for $\$(-n)$, but is additionally equipped with a Coq notation which displays it as simply "DEBT". In other words, it hides the amount of the debt it represents to the user. We use this auxiliary definition to hold the *synthesized cost*

of the program. (In the previous section, the user handles goals containing $\$\langle c \rangle$, where c is a Coq evar. In this section, Coq goals contain $\$[\![n]\!]$, where n denotes a concrete cost expression, not an evar, but that we hide so as not to clutter the goal.)

DebtIntro is used to introduce an initial $\$[\![0]\!]$ resource (holding a debt of zero). Then, at any point during the proof, the goal is expected to only contain one instance of $\$[\![\cdot]\!]$, which collects individual debts which are "not interesting to look at". To hide an individual debt $\$(-m)$, the rule CollectDebt is used. At the end of the proof, the rule RepayDebt pays off the debts collected. (This last rule is reminiscent of the GcCredits rule of Section 6.4.2.)

It would not be practical to manually apply CollectDebt after each function call. Instead, we can automate the application of the rule, *following the same policy than as evars*. More precisely, we automatically apply CollectDebt only if the collected debt $\$(-m)$ does not depend on local variables. Otherwise, the debt is left apparent in the goal, which indicates to the user that they should do something about it (e.g. using ApproxDebt). From a technical perspective, this policy can be implemented by having DebtIntro introduce a marker in the proof context; then, every name that has been introduced after this marker is considered to be a local variable.

Let us now review, as before, how the approach behaves on our favorite toy examples.

Paying for a sequence of operations The initial goal for the verification of `pay3` is as before:

$$\{\$3\}\ \texttt{pay(); pay(); pay()}\ \{\lambda().\ [\,]\}.$$

We start the proof by applying DebtIntro, so as to setup the goal for later application of CollectDebt.

$$\{\$3 \star \$[\![0]\!]\}\ \texttt{pay(); pay(); pay()}\ \{\lambda().\ [\,]\}$$

By applying the sequence rule and the specification of `pay`, we get the following intermediate goal, where we are required to pay for one credit:

$$\$3 \star \$[\![0]\!] \Vdash \$1 \star ?F \quad \wedge \quad \{?F\}\ \texttt{pay(); pay()}\ \{\lambda().\ [\,]\}.$$

The heap entailment is automatically discharged using the rules PayAsDebt and CollectDebt. (In practice, one would directly go from the goal before to the next goal in a single tactic invocation, without this intermediate step.)

$$\{\$3 \star \$[\![1]\!]\}\ \texttt{pay(); pay()}\ \{\lambda().\ [\,]\}$$

After repeating this process, we get to prove that the collected debts and the initial amount of credits indeed cancel out:

$$\$3 \star \$[\![3]\!] \Vdash \mathrm{GC}$$

By applying the RepayDebt rule, this goal becomes $3 \leq 3$, which is easily discharged.

Bounding the cost of an operation On the `let_bound` example, after an initial invocation to DebtIntro and stepping through the first call to `pay()`, we get the following goal:

$$\{\$(\mathit{cost\_loop}(n)+3) \star \$[\![1]\!]\}\ \texttt{let m = rand n in loop m; pay()}\ \{\lambda().\ []\}.$$

Using the standard rule for `let`, handling the call to `rand` is automatic and increments the debt:

$$0 \le m \le n \implies \{\$(\mathit{cost\_loop}(n)+3) \star \$[\![2]\!]\}\ \texttt{loop}\ m\texttt{; pay()}\ \{\lambda().\ []\}.$$

Then, unlike in the evar-based approach, handling the call to `loop` is automatic as well. However, as per the policy described earlier, the corresponding debt is not integrated as part of $\$[\![2]\!]$.

$$\begin{array}{l} 0 \le m \le n \implies \\ \{\$(\mathit{cost\_loop}(n)+3) \star \$[\![2]\!] \star \$(-\mathit{cost\_loop}(m))\}\ \texttt{pay()}\ \{\lambda().\ []\} \end{array}$$

This is a sign for the user to apply the ApproxDebt rule, bounding $\mathit{cost\_loop}(m)$ with $\mathit{cost\_loop}(n)$. Because this results in a bound which does not depend on the local variable m anymore, the system automatically applies CollectDebt.

$$\begin{array}{l} 0 \le m \le n \implies \\ \{\$(\mathit{cost\_loop}(n)+3) \star \$[\![2+\mathit{cost\_loop}(n)]\!]\}\ \texttt{pay()}\ \{\lambda().\ []\} \end{array}$$

After processing the last call to `pay()`, we obtain:

$$\$(\mathit{cost\_loop}(n)+3) \star \$[\![2+\mathit{cost\_loop}(n)+1]\!] \Vdash \text{GC}.$$

As before, it now suffices to apply the RepayDebt rule, in order to obtain the following (trivial) inequality, between the initial amount of credits on the left, and the synthesized expression on the right:

$$\mathit{cost\_loop}(n)+3 \ge 2+\mathit{cost\_loop}(n)+1.$$

Inferring the cost of a conditional expression Conditional expressions are handled similarly as in the approach based on the subtraction rule (§6.4.2). For instance, on the `myloop` program, the conditional in the program results in two separate debt-related sub-goals:

$$\begin{array}{rl} & n \le 0 \implies \$(\mathit{cost}(n)) \star \$[\![1]\!] \Vdash \text{GC} \\ \wedge & n > 0 \implies \$(\mathit{cost}(n)) \star \$[\![1+\mathit{cost}(n-1)]\!] \Vdash \text{GC}. \end{array}$$

6.4.5 Summary

We have presented three different approaches for semi-interactively elaborating a cost expression for straight-line code. We summarize below the key principle each approach relies on, and the shape of the goals that the user has to handle.

1. **Subtraction rule (§6.4.2).** In this approach, program operations are payed by subtracting their cost from the amount of credits currently available. The cost expression synthesized for the overall program is the sum of all these subtracted costs. The user therefore handles goals that mention a single amount of credits, itself expressed as an iterated subtraction between the initial amount of credits and the cost of the operations that have been previously analyzed:

$$\{\$(\text{initial\_credits} - \text{cost\_a} - \text{cost\_b}) \star \ldots\} \ \ldots \ \{\ldots\} \quad \wedge$$
$$\{\$(\text{initial\_credits} - \text{cost\_c}) \star \ldots\} \ \ldots \ \{\ldots\}$$

   At the end of the proof, one must establish that each amount of credits is greater than zero, i.e. here $\text{cost\_a} + \text{cost\_b} + \ldots \leq \text{initial\_credits}$ and $\text{cost\_c} + \ldots \leq \text{initial\_credits}$.

2. **Coq evars (§6.4.3).** This approach relies on Coq metavariables (evars) to represent a cost expression that is yet to be synthesized. Syntax-directed reasoning rules on programs are instrumented so as to progressively instantiate these placeholders while stepping through the program. The user handles goals that each only contain a placeholder, identified by angle brackets $\langle\cdot\rangle$. An additional goal keeps track of the fact that the initial amount of credits available must be greater than the overall cost being synthesized.

$$\{\$\langle ?c_1\rangle \star \ldots\} \ \ldots \ \{\ldots\} \quad \wedge$$
$$\{\$\langle ?c_2\rangle \star \ldots\} \ \ldots \ \{\ldots\} \quad \wedge$$
$$\max(\text{cost\_a} + \text{cost\_b} + ?c_1,\ \text{cost\_c} + ?c_2) \leq \text{initial\_credits}$$

   While approach 1 and 3 synthesize one cost expression for each branch of the program, this approach synthesizes a cost expression for the whole program, possibly using max and If.

3. **Collecting debts (§6.4.4).** This approach is similar in essence to the first approach, but collects the costs under a dedicated notation $\$[\![n]\!]$, representing a debt, and defined as $\$(-n)$.

$$\{\$\text{initial\_credits} \star \$[\![\text{cost\_a} + \text{cost\_b}]\!] \star \ldots\} \ \ldots \ \{\ldots\} \quad \wedge$$
$$\{\$\text{initial\_credits} \star \$[\![\text{cost\_c}]\!] \star \ldots\} \ \ldots \ \{\ldots\}$$

   Additionally, to avoid cluttering the goal, the expression of debts is hidden from the user: in Coq goals, $\$[\![n]\!]$ is simply displayed as "DEBT". At the end of the proof, one must establish that debts can be paid off using available credits, i.e. here $\text{cost\_a} + \text{cost\_b} + \ldots \leq \text{initial\_credits}$ and $\text{cost\_c} + \ldots \leq \text{initial\_credits}$.

6.5 Synthesizing and Solving Recurrence Equations

There now remains to explain how to deal with recursive functions. Suppose $S(f)$ is the program specification that we wish to establish, where f stands for an as-yet-unknown cost function. For instance, in the case of the earlier `bsearch` example (§6.1), $S(f)$ would

be defined as follows:

$$\begin{array}{l} S(f) \triangleq \\ \quad \text{nonnegative } f \wedge \text{monotonic } f \wedge \\ \quad \forall a\, xs\, v\, i\, j.\ 0 \le i \le |xs| \wedge 0 \le j \le |xs| \implies \\ \quad\quad \{\$f(j-i) \star a \rightsquigarrow \mathsf{Array}\, xs\}\ \texttt{bsearch}\ a\ v\ i\ j\ \{\lambda k.\ a \rightsquigarrow \mathsf{Array}\, xs\}. \end{array}$$

Following common informal practice, we would like to establish f in two steps. First, from the code, derive a "recurrence equation" $E(f)$, which in fact is usually not an equation, but a constraint (or a conjunction of constraints) bearing on f. Second, prove that this recurrence equation admits a solution that is dominated by the desired asymptotic cost function g. This approach can be formally viewed as an application of the following tautology:

$$\forall E.\ (\forall f.\ E(f) \to S(f)) \ \to\ (\exists f.\ E(f) \wedge f \preceq g) \ \to\ (\exists f.\ S(f) \wedge f \preceq g).$$

The conclusion $S(f) \wedge f \preceq g$ states that the code is correct and has asymptotic cost g. In Coq, applying this tautology gives rise to a new metavariable E, as the recurrence equation is initially unknown, and two subgoals.

During the proof of the first subgoal, $\forall f.E(f) \to S(f)$, the cost function f is abstract (universally quantified), but we are allowed to assume $E(f)$, where E is initially a metavariable. So, should the need arise to prove that f satisfies a certain property, this can be done just by instantiating E.

In the example of the OCaml function `bsearch` (§6.1), we prove $S(f)$ by induction over n $(= j - i)$, under the hypothesis $n \ge 0$. Thus, we assume that the cost of the recursive call `bsearch k` is $f(n')$ (for $0 \le n' < n$), and must prove that the cost of `bsearch n` is $f(n)$. We synthesize a cost expression for the body of `bsearch` as explained earlier (§6.4.3) and find that our proof is complete, provided we are able to prove the following inequation:

$$1 + \mathsf{If}\ n \le 0\ \mathsf{then}\ 0\ \mathsf{else}\ 1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) \ \le\ f(n).$$

We achieve that simply by instantiating E as follows:

$$E := \lambda f.\ \forall n.\ 1 + \mathsf{If}\ n \le 0\ \mathsf{then}\ 0\ \mathsf{else}\ 1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) \ \le\ f(n).$$

This is our "recurrence equation"—in fact, a universally quantified, conditional inequation. We are done with the first subgoal.

We then turn to the second subgoal, $\exists f.E(f) \wedge f \preceq g$. The metavariable E is now instantiated. The goal is to solve the recurrence and analyze the asymptotic growth of the chosen solution. As we have seen, we can proceed by guessing that the shape of the solution f is of the form $\lambda n.\ \mathsf{If}\ 0 < n\ \mathsf{then}\ a \log n + b\ \mathsf{else}\ c$, where a, b and c are parameters about which we will accumulate a set of constraints. From a formal point of view, this amounts to applying the following tautology:

$$\begin{array}{ll} \forall P.\ \forall C. & (\forall a\, b\, c.\ C(a,b,c) \to P(\lambda n.\ \mathsf{If}\ 0 < n\ \mathsf{then}\ a \log n + b\ \mathsf{else}\ c)) \ \to \\ & (\exists a\, b\, c.\ C(a,b,c)) \ \to \\ & \exists f.P(f). \end{array}$$

This application again yields two subgoals. During the proof of the first subgoal, C is a metavariable and can be instantiated as desired (possibly in several steps), allowing us to gather a conjunction of constraints bearing on a, b and c. In our example, proving the first subgoal leads to instantiating C as follows:

$$C := \lambda a\, b\, c.\ c \geq 1 \ \wedge\ b \geq 0 \ \wedge\ a \geq 0 \ \wedge\ b \geq c + 3 \ \wedge\ a \geq 3.$$

There remains to check the second subgoal, that is, $\exists abc.\ C(a, b, c)$. This can be done using a dedicated decision procedure; more on that in the next section (§6.6).

Procrastination, a Coq library for collecting constraints and deferring their proof

In practice, we do not build and apply tautologies manually as above. Instead, we created a small Coq library dubbed `procrastination` [Gué18c], which implements support for this style of reasoning. Under the hood, the library automates the generation of tautologies in the style that was presented earlier. From a user perspective, it provides facilities for introducing new parameters, gradually gathering constraints of these parameters, and eventually checking that these constraints are satisfiable.

Figure 6.12 shows a Coq proof excerpt illustrating the use of the library, in order to prove recurrence equations using the substitution method. The `procrastination` library provides three main tactics:

- "`begin defer assuming a b c...`": this tactic initiates the "deferring" process. It adds new abstract names `a`, `b`, `c`, ... (more can be asked for) to the context, along with an hypothesis `g` whose type is an evar, and can be used to collect side-conditions about `a`, `b` and `c`. The `Group` construct which appears in `g` is in fact defined as the identity, and acts as a marker for the tactics that follow.

  Internally, the `begin defer` tactic generates and applies a lemma of the form:

$$\begin{array}{l}\forall (A\, B \ldots : \mathsf{Type})\ (G : A \to B \to \ldots \to \mathsf{Prop})\ (P : \mathsf{Prop}).\\ \quad (\forall a\, b \ldots .\ (G\ a\ b \ldots) \to P) \to\\ \quad (\exists a\, b \ldots .\ G\ a\ b \ldots) \to\\ \quad P.\end{array}$$

- "`defer`": this tactic "defers" the current subgoal. It solves the subgoal, refining the type of `g` as a side effect, effectively "storing" the goal in the set of side-conditions to prove later.

- "`end defer`": this terminates the deferring process, and retrieves the collected conditions, which the user must then prove.

Consequently, the library enforces a pattern where the proof is separated in two phases. We find this proof style to be quite effective in practice: in the first phase, high-level tactics can be used to reduce the interesting parts of the proof into low-level requirements (that typically involve low-level accounting on credits). These assumptions can be deferred immediately, and are collected into a single subgoal. Then, in the second

```
Goal ∃(f: nat →nat),
  1 ≤ f 0 ∧
  monotonic f ∧
  ∀n, 0 < n →3 + f (n/2) ≤ f n.
Proof.
  begin defer assuming a b c. (1)
  exists (fun n ⇒ if zerop n then c else a ∗ log2 n + b).
  repeat split.
  { simpl. (2) defer. (3) }
  { ... (* c ≤ b *) (4) defer. (5) ... }
  { intros. ... (* 3 + c ≤ b *) defer. ... (* 3 ≤ a *) defer.}
  end defer.
  (6) eomega.
Qed.
```

(1)

```
a: ?x
b: ?x0
c: ?x1
g: Group (?G a b c)
```

```
∃f: nat →nat,
  1 ≤ f 0 ∧monotonic f ∧∀n, 0 < n →3 + f (n/2) ≤ f n

subgoal 2 is:
 end defer
```

(2)

```
a, b, c: nat
g: Group (?G a b c)
```

```
1 ≤ c
```

(3)

```
a, b, c: nat
g: Group (1 ≤ c ∧?Goal0)
```

```
Subgoal solved
```

(4)

```
a, b, c: nat
g: Group (1 ≤ c ∧?Goal0)
```

```
c ≤ b
```

(5)

```
a, b, c: nat
g: Group (1 ≤ c ∧c ≤ b ∧?Goal1)
```

```
Subgoal solved
```

```
∃a b c, 1 ≤ c ∧c ≤ b ∧3 + c ≤ b ∧3 ≤ a
```

(6)

Figure 6.12: Proof excerpt illustrating the tactics from the `procrastination` library.

phase, these low-level side-conditions can be discharged all at once, using an appropriate procedure.

For more technical details about `procrastination` and the tactics it provides, we refer the reader to the manual of the library [Gué18a].

6.6 More on Automation

We end this chapter by describing the technical artifacts that were useful in order to put our approach into practice.

Automation for Separation Logic rules When implementing the various Separation Logic rules presented in Section 6.4, one would like to apply them "modulo associativity and commutativity of $\star$", and to take into account simplifications such as $\$0 = []$. Since we use a shallow embedding of Separation Logic, we cannot directly write a Coq function to traverse the formulae that appear in Coq goals. Instead, we draw inspiration from Iris [KTB17], and use logic programming using type classes [SO08, ZH17] to implement the required machinery at the meta-level.

For example, the rule SubCredits can be implemented as the following lemma:

```
Class GetCredits (H H': hprop) (c: Z) :=
  H = H' ⋆ $c.

Lemma sub_credits: H H' Q Q' c1 c2,
  GetCredits H H' c1 →
  GetCredits Q Q' c2 →
  $(c1 − c2) ⋆ H' ⊫ Q' →
  H ⊫ Q.
```

Instances of the type class `GetCredits H H' c` should be read as clauses of a logic program with input `H` (the Separation Logic formula to traverse) and outputs `H'` and `c` (the resulting formula after extracting the credits, and the extracted credits, respectively). For example, the three instances below allow exploring under the $\star$ connective, and recognizing formulae containing credits.

```
Class Star (H1 H2 H3: hprop) :=
  H1 ⋆ H2 = H3.
Class Add (a b c: Z) :=
  a + b = c.

Instance GetCredits_star: ∀H1 H1' H2 H2' H1H2' c1 c2 c1c2,
  GetCredits H1 H1' c1 →
  GetCredits H2 H2' c2 →
  Star H1' H2' H1H2' →
  Add c1 c2 c1c2 →
  GetCredits (H1 ⋆ H2) H1H2' c1c2.

Instance GetCredits_credits: ∀c,
  GetCredits ($ c) [] c.

Instance GetCredits_default: ∀H,
```

```
GetCredits H H 0 | 100.
```

The auxiliary type classes `Star` and `Add` are meta-level versions of the "$\star$" and "$+$" connectives, which additionally perform simplifications. For instance, we define instances for `Star` which eliminate superfluous empty heaps:

```
Instance Star_emp_l: ∀H, Star [] H H.
Instance Star_emp_r: ∀H, Star h [] H.
Instance Star_default: ∀H1 H2, Star H1 H2 (H1 ⋆ H2) | 2.
```

All in all, we found Coq type classes to be a powerful mechanism, in which tactics can be defined modularly (using reusable components such as `Star` or `GetCredits`), and are easily extensible (adding support for a new connective is achieved by simply defining new instances).

Extracting side-conditions from synthesized costs A key strength of our verification approach is the separation of the proof into several phases. In a first phase, a cost expression is synthesized, then, in a second phase, it is reduced into simpler side-conditions on a few constants, which can be eventually discharged automatically. In this second phase, it is important that the proof steps that simplify the (possibly verbose) cost function down to side-conditions do not rely too much on the specific expression of the cost function: this would make the proof very brittle.

Throughout our case studies, we rely on mainly three key tools to simplify cost expressions. First, we add "setoid rewrite" instances to extend the `rewrite` tactic with support for rewriting with arithmetic inequalities—this comes in handy in many situations. Second, we make use of the simplification procedure from the PolTac library [Thé05], which behaves better than e.g. `ring_simplify`. Finally, our main tool for simplifying cost expressions is a custom tactic called `cancel`. As an example, let us consider the following inequality, which is a simplified version of a goal that appears during the verification of Bender *et al.*'s graph algorithm (§8).

$$\begin{aligned}&1 + \mathit{cost\_aux1}() + n \cdot k + \max(\mathit{cost\_aux2}(), (a-k)\cdot n + b\cdot(m-1) + c - p)\\ &\leq a\cdot n + b\cdot m + c\end{aligned}$$

In this goal, n and m are local variables that need to be eliminated, a, b and c are parameters about which we need to produce side-conditions, and the remaining terms can be seen as (abstract) constants. Additionally, we assume both n and m to be nonnegative.

Taking advantage of the fact that n and m are nonnegative, we can eliminate n and m by comparing their factors on both sides of the inequality. In other words, we can use the following lemma:

$$\forall n\, a\, a'\, b\, b'.\ n \geq 0 \implies a \leq a' \wedge b \leq b' \implies a\cdot n + b \leq a'\cdot n + b'.$$

The `cancel` tactic automates the application of the lemma. To simplify the complicated goal above, one first calls `cancel n`. This produces the following sub-goals:

$$\begin{aligned}&k + \max(0, a-k) \leq a\\ \wedge\ \ &1 + \mathit{cost\_aux1}() + \max(\mathit{cost\_aux2}(), b\cdot(m-1) + c - p) \leq b\cdot m + c.\end{aligned}$$

Notice that n has been eliminated, and that `cancel` is able to work "under max". Now, we can call `cancel m` on the second subgoal.

$$\begin{array}{ll} & k + \max(0, a - k) \leq a \\ \wedge & \max(0, b) \leq b \\ \wedge & 1 + \mathit{cost\_aux1}\,() + \max(\mathit{cost\_aux2}\,(), c - b - p) \leq c \end{array}$$

As a result, we get inequalities about a, b and c from which n and m have been eliminated. These are the side-conditions that we can collect for later.

Technically speaking, `cancel` is implemented using logic programming with type classes, similarly to the Separation Logic tactics described previously. It corresponds to the following Coq lemma:

```
Class FactorBy (f: Z) (a b c : Z) :=
  MkFactorBy : a = b * f + c.
Class FactorByLe (f: Z) (a b c : Z) :=
  MkFactorByLe : 0 ≤ f → a ≤ b * f + c.

Lemma cancel_factors f : ∀e1 e2 a1 b1 a2 b2,
  FactorByLe f e1 a1 b1 →
  FactorBy f e2 a2 b2 →
  0 ≤ f →
  a1 ≤ a2 →
  b1 ≤ b2 →
  e1 ≤ e2.
```

The auxiliary classes `FactorBy` and `FactorByLe` are used to extract factors of the provided expression `f`, and are equipped with instances that allow them to traverse synthesized cost expressions, which typically involve sum and max.

Decision procedures for existentially quantified arithmetic goals After the synthesized cost expression has been simplified down to simpler side-conditions, the very last goal is to prove that there indeed exist parameters satisfying the side-conditions (§6.5). For instance, with the binary search example of Section 6.1, the final goal is the following:

$$\exists a\, b\, c.\ c \geq 1 \ \wedge\ b \geq 0 \ \wedge\ a \geq 0 \ \wedge\ b \geq c + 3 \ \wedge\ a \geq 3.$$

More generally, the goals that we have encountered in practice are existentially quantified arithmetic goals, *which can involve abstract constants*. The implementation of `bsearch` does not call auxiliary functions; however, in the general case, the inequalities can involve the cost of auxiliary functions, which are abstracted (§6.2) and are not exposed as numerical constants. For instance, our verification of Bender *et al.*'s algorithm involve final goals of the following form, where $\mathit{cost\_aux1}\,()$, $\mathit{cost\_aux2}\,()$, etc. represent the (constant) cost of auxiliary functions:

$$\exists a\, b.\quad \begin{array}{l} 1 + \mathit{cost\_aux1}\,() \leq b \ \wedge\ a \leq b \ \wedge\ 0 \leq a \ \wedge\ 1 \leq a + b \wedge \\ 1 + \mathit{cost\_aux2}\,() + \mathit{cost\_aux3}\,() + b \leq a + b \ \wedge\ 1 + \mathit{cost\_aux4}\,() \leq a + b. \end{array}$$

We are therefore looking for an automated procedure that would be able to handle existentially quantified arithmetic goals with abstract constants. (Alternatively: arithmetic

goals under a $\forall \ldots \exists \ldots$ quantification.) As of now, we focused on the case of abstract *additive* constants (multiplicative constants must be explicit numbers). Below are the three ideas that we have investigated, or would like to investigate. There might be more.

(1) Cooper's quantifier elimination procedure Cooper [Coo72] describes a quantifier elimination procedure for Presburger arithmetic. Such a procedure takes as input an arithmetic formula, and produces an equivalent *quantifier-free* formula. One could therefore eliminate the existential quantifiers from the arithmetic goals above, and then solve the resulting quantifier free goal using the `lia` tactic, which is part of Coq's standard library.

Following the detailed description of the algorithm from the *Handbook of Practical Logic and Automated Reasoning* by Harrison [Har09], we have implemented in Coq a verified version of Cooper's algorithm. We also provide a reflexive tactic which makes it easy to apply the algorithm to actual Coq goals. The development is available as a standalone Coq library [GP18].

Unfortunately, we found out that formulae produced by Cooper's procedure often contain many spurious disjunctions (we could find equivalent formulae with a lot less disjunctions), which `lia` does not process well (in practice, it would loop and eventually time out), and are more verbose than one would like.

(2) Fourier-Motzkin elimination and the Omega test The Fourier-Motzkin procedure can be used to eliminate quantifiers on linear constraints over real variables. However, we are interested here in integer variables. The Omega test is an algorithm that can be seen as a variant of Fourier-Motzkin which works with integer variables. A good description of both algorithms can be found in the book *Decision Procedures: An Algorithmic Point of View* by Kroening and Strichman [KS08].

The Omega test is primarily described as a decision procedure, and in that case requires both multiplicative and additive constants to be explicit numbers. However, it seems that in fact, one could implement the *dark shadow* phase of the Omega test and obtain a quantifier elimination procedure that works even in the presence of abstract additive constants. Such a procedure is *incomplete* (the quantifier free formula might not be provable even if the input formula is), but we hope that it would work well in practice. We still have to put this idea in practice, by implementing a verified version of the Omega test's dark shadow phase.

At the moment, we implement a simpler version of Fourier-Motzkin which is restricted to the case where all multiplicative coefficients are 1 or -1. (It is complete, in that case.) Although quite limited, such a procedure is already useful in many of our proofs. It appears in our proof as the `elia` Coq tactic.

(3) SMT solvers The SMTCoq library [AFG+11] integrates certificate-producing SAT and SMT solvers with Coq. A promising idea—but which we did not have time to investigate at that point—would be to try to make use of this library, either directly on our existentially quantified goals, or on the output of our quantifier elimination procedure based on Cooper's algorithm.

Chapter 7

Example Gallery

7.1 Binary Search

We prove that binary search has time complexity $O(\log n)$, where $n = j - i$ denotes the width of the search interval $[i, j)$ [Gué18b, examples/proofs/Dichotomy_proof.v]. Binary search serves as a running example in Chapter 6 to illustrate reasoning techniques on time credits. Below, we recall the implementation, and only reproduce the key steps of the proof.

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

As outlined earlier (§6.1), we use the `procrastination` library (§6.5) in combination with credits inference (§6.4) to synthesize the following recurrence equations on the cost function *cost*:

$$\begin{array}{ll} & \text{monotonic } \mathit{cost} \\ \wedge & \forall n.\ \mathit{cost}(n) \geq 0 \\ \wedge & \forall n.\ n \leq 0 \implies \mathit{cost}(n) \geq 1 \\ \wedge & \forall n.\ n \geq 1 \implies \mathit{cost}(n) \geq \mathit{cost}(\tfrac{n}{2}) + 3. \end{array}$$

(In fact, the equations above are obtained after some manual simplification and cleanup). We apply the substitution method and search for a solution of the form $\lambda n.\ \mathit{if}\ n \leq 0\ \mathit{then}\ c\ \mathit{else}\ a \log n + b$, which is dominated by $\lambda n.\ \log n$. Guessing $\lambda n.\ a \log n + b$ does not lead to a solution: as the definition in Coq of $\log n$ is extended to be 0 when $n \leq 0$, this would make the last equation equivalent to $b \geq b + 3$ in the case $n = 1$.

We use `procrastination` again to assume the existence of some constants a, b and c. After substituting this shape into the above constraints and simplifying them, we find that they boil down to:

$$c \geq 1 \wedge b \geq 0 \wedge a \geq 0 \wedge b \geq c + 3 \wedge a \geq 3$$

Finally, we automatically find suitable values for a, b and c, by applying our `elia` tactic (§6.6, "Fourier-Motzkin elimination").

7.2 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 (the corresponding Coq proof appears in [Gué18b, examples/proofs/Dependent_nested_proof.v]).

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

Intuitively, the set of pairs (i, j) enumerated by the nested loops covers a triangle. (In comparison, indices of non-dependent loops cover a square.)

By applying the Separation Logic reasoning rules instrumented for credit synthesis (§6.4.3, Figure 6.9), the cost function $\lambda n.\ \sum_{i=1}^{n}(1 + \sum_{j=1}^{i} 1)$ is synthesized for this code. There remains to prove that it is dominated by $\lambda n.n^2$. There are at least two ways to proceed.

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 ultimately monotonic and positive, then $\sum_{i=1}^{n} f(i)$ is dominated by $n.f(n)$. This corresponds to Lemma 5.3.9 ("Coarse Summation, in the single parameter case"). 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 the full power of the Summation lemma (Lemma 5.3.8), is often applicable. The next example illustrates a situation where the Summation lemma is required.

7.3 A Loop Whose Body has Exponential Cost

In the following simple example, the loop body is just a function call (corresponding Coq proof in [Gué18b, examples/proofs/Exponential_proof.v]).

```
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, `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 by applying the rules of Figure 6.9. Bounding the sum $\sum_{i=0}^{n}(1 + \texttt{cost b}\ (i))$ by $n(1 + \texttt{cost b}\ (n))$ (as in §7.2) would be valid, but would lead to a strictly coarser bound of $O(n2^n)$. In order to obtain a tight bound, the Summation lemma seems required.

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, in the single paramater case, are met (Lemma 5.3.8):

- $\mathbb{U}_{\mathbb{Z}} i.\ 1 + \texttt{cost b}\ (i) > 0$ holds, because we work with nonnegative cost functions everywhere; as such, `cost b` is known to be nonnegative;

```
let bellman_ford inf source edges nb_nodes =
  let d = Array.make nb_nodes inf in
  d.(source) <- 0;
  for i = 0 to nb_nodes - 2 do
    for j = 0 to Array.length edges - 1 do
      let (v1, v2, w) = edges.(j) in
      d.(v2) <- min d.(v2) (d.(v1) + w)
    done
  done;
  d
```

Figure 7.1: OCaml implementation of Bellman-Ford's algorithm. The graph is represented by its adjacency matrix, described by the "`edges`" parameter, of type `(int * int * int) array`.

- $\mathbb{U}_{\mathbb{Z}} i.\ 2^i > 0$ holds (for any i greater or equal than 0).

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 $\lambda n.\ \sum_{i=0}^{n} 2^i = \lambda n.\ 2^{n+1} - 1 \preceq_{\mathbb{Z}} \lambda n.\ 2^n$.

7.4 Bellman-Ford's 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 [Gué18b, examples/proofs/Bellman_ford_proof.v]. The algorithm (in its most naïve version) involves an outer loop that is repeated $n - 1$ times and an inner loop that iterates over all m edges. The code appears in Figure 7.1; n corresponds there to `nb_nodes` and m to `Array.length 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 $m \leq n^2$, 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 the composition lemma (Lemma 5.3.16), where the bridge function p is $\lambda n.\ (n^2, n)$. In practice, we publish both specifications and let clients use whichever one is more convenient.

Note that in this example, the asymptotic complexity of the code is independent from its functional correctness (and indeed, for simplicity, we only establish a complexity bound in our Coq proof).

```
type ('acc, 'break) step =
  | Continue of 'acc
  | Break of 'break

let rec interruptible_fold
    (f: 'a -> 'acc -> ('acc, 'break) step)
    (l: 'a list) (acc: 'acc):
  ('acc, 'break) step
=
  match l with
  | [] -> Continue acc
  | x :: xs ->
    let res = f x acc in
    match res with
    | Continue acc -> interruptible_fold f xs acc
    | Break _ -> res
```

Figure 7.2: OCaml implementation of the `interruptible_fold` iterator.

7.5 Union-Find

Charguéraud and Pottier [CP17b] use Separation Logic with Time Credits to verify the correctness and time complexity of a Union-Find implementation. For instance, they prove that the (amortized) concrete cost of `find` is $2\alpha(n) + 4$, where n is the number of elements in the data structure. Note that contrarily to the previous example (§7.4), establishing this complexity bound does require establishing the correctness of the code. With a few lines of proof, we derive a specification where the cost of `find` is expressed under the form $O(\alpha(n))$ [Gué18b, examples/proofs/UF.v]:

```
specZ [cost \in_O alpha]
  (∀ D R V x, x \in D → app (UnionFind_ml.find x)
    PRE (UF D R V ⋆ $(cost (card D)))
    POST (fun y ⇒ UF D R V ⋆ [ R x = y ])).
```

(The specification above is using the `specZ` notation introduced in Section 6.2). 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.

7.6 Higher-Order Interruptible Iterator

We verify a higher-order iteration function on lists, named `interruptible_fold` [GJCP19, proofs/IFold_proof.v]. Its OCaml code appears in Figure 7.2. `interruptible_fold` is similar to the `List.fold_left` function from the OCaml standard library. It iterates a client function `f` on successive elements of the input list `l`, while maintining an accumulator (of type `'acc`). Yet, unlike `List.fold_left`, which walks through `l` completely,

```
spec1 [cost]
 (∀ A AccT BreakT Inv (l: list A) (f:func),
   (∀ x xs (acc:AccT), suffix (x :: xs) l →
      app f [x acc]
        PRE (Inv (x :: xs) (Continue acc))
        POST (fun (ret: step_ AccT BreakT) ⇒
          Inv xs ret ⋆ $cost tt))
   →
   ∀acc, app interruptible_fold [f l acc]
     PRE (Inv l (Continue acc) ⋆ $cost tt)
     POST (fun ret ⇒∃xs, Inv xs ret ⋆
       [match ret with Continue _ ⇒ xs = nil
                     | Break _ ⇒ suffix xs l end ]))
```

Figure 7.3: Specification for `interruptible_fold` where the client pays as it walk through the list.

the `interruptible_fold` function allows the client to stop the iteration. At each step, the client function `f` decides to either continue the iteration (by returning `Continue`), or to stop (by returning `Break`).

We make use of `interruptible_fold` in our implementation (§8.4) of the incremental cycle detection algorithm presented in Chapter 8. There, the ability to interrupt the iteration is crucial: using `List.fold_left` instead of `interruptible_fold` would lead to an incorrect asymptotic complexity. This is reminiscent of the previous `walk` example (§6.3.1): both examples correspond to an iteration which may be interrupted for a reason that can be hard to specify upfront (in the case of `walk`, the cell of an array containing zero, here, the client function returning `Break`).

The specification that we establish appears in Figure 7.3. It is itself higher-order in some sense: a user of this specification is required to provide a specification for the client function `f`; in return, they get a specification for `interruptible_fold f`. With respect to time complexity, the specification corresponds to a "pay as you go" pricing scheme:

> *"Calling `interruptible_fold` has an initial constant cost. Then, the client `f` has to pay a constant amount for each new element of the list that it visits."*

In Figure 7.3, the function `cost` quantified by the `spec1` notation (§6.2) denotes a constant amount of credits, "`cost tt`" (`cost` has type `unit → Z`, and `tt` denotes the constructor of `unit`). `cost tt` appears in the precondition of `interruptible_fold`: this corresponds to the initial cost of calling the function. It also appears if the postcondition of `f`: the client function is required to *return* credits in order to pay for the next iteration. One can wonder how the client might be able to do so. The specification for `f` is allowed to maintain some invariant `Inv`, which can depend on the unexplored suffix of the list, as well as the current accumulator. "`Inv xs r`" is a Separation Logic predicate: it can describe not only functional invariants, but also the ownership of some data structure as well as some amount of time credits. This invariant can depend on both the suffix `xs` of elements that are yet to be explored, and the current accumulator `r`. It is then up to the user of the specification to instantiate `Inv` with enough time credits so as to be able to pay for exploring the list.

```
spec1 [cost]
 (∀ A AccT BreakT Inv (l: list A) (f:func),
   (∀ x xs (acc:AccT), suffix (x :: xs) l →
      app f [x acc]
        PRE (Inv (x :: xs) (Continue acc))
        POST (fun (ret: step_ AccT BreakT) ⇒ Inv xs ret))
   →
   ∀acc, app interruptible_fold [f l acc]
     PRE (Inv l (Continue acc) ⋆ $(length l ∗ cost tt) ⋆ $cost tt)
     POST (fun ret ⇒∃xs,
       Inv xs ret ⋆ $(length xs ∗ cost tt) ⋆
       [match ret with Continue _ ⇒ xs = nil
                     | Break _ ⇒ suffix xs l end ])).
```

Figure 7.4: Alternative specification for `interruptible_fold` where the client pays upfront for iterating on the whole list.

From a client's perspective, one easy way of satisfying this requirement is to provide initially (as part of `Inv`) enough credits to be able to iterate through the whole list. If the iteration is stopped prematurely, then one gets back the unspent credits. In fact, one can prove an alternative specification for `interruptible_fold`, which corresponds this time to a "fixed-rate" pricing scheme:

"When calling ***`interruptible_fold`*** *the user pays upfront for iterating through the whole list. After the iteration, the user gets back credits for the elements of the list that have not been explored."*

The specification appears in Figure 7.4. In this setting, a user of the specification pays upfront the cost of iterating through the entire list (`$ (length l ∗ cost tt)`). Consequently, the client function `f` is not required to produce credits anymore. Finally, if the traversal is interrupted, then the user has paid too much: they get back the amount of unspent credits (`$ (length xs ∗ cost tt)`, where `xs` is the suffix of the input list that has not been explored).

Finally, one can prove a third specification, similar to the first specification of `walk`, which corresponds to a "pay-per-use" pricing scheme:

"Calling ***`interruptible_fold`*** *and iterating through the list is (seemingly) free. After the iteration, the user obtains a debt which corresponds to the cost of the iteration."*

The corresponding Coq specification appears in Figure 7.5. In that setting, the user obtains a debt after the iteration, linear in the number of elements that have been explored.

The proof of all three specifications is straightforward, using the techniques described earlier with the binary search example (§7.1). This example illustrates that we are able to state specifications—including in a higher-order setting—that strike a right balance between precision and abstraction. The specifications that we presented for `interruptible_fold` allow for a precise accounting of the cost of a traversal, while abstracting over the exact cost of an iteration step (thanks to the existential quantification on `cost`).

```
spec1 [cost]
 (∀ A AccT BreakT Inv (l: list A) (f:func),
   (∀ x xs (acc:AccT), suffix (x :: xs) l →
      app f [x acc]
        PRE (Inv (x :: xs) (Continue acc))
        POST (fun (ret: step_ AccT BreakT) ⇒ Inv xs ret))
   →
   ∀acc, app interruptible_fold [f l acc]
     PRE (Inv l (Continue acc))
     POST (fun ret ⇒∃xs,
       Inv xs ret ⋆ $((length xs − length l − 1) ∗ cost tt) ⋆
       [match ret with Continue _ ⇒ xs = nil
                     | Break _ ⇒ suffix xs l end ])).
```

Figure 7.5: Alternative specification for `interruptible_fold` where the client iterates "for free" then obtains a debt corresponding to the cost of the iteration.

7.7 Binary Random Access Lists

As a medium-sized case study, we verify an implementation of binary random access lists, a purely functional data structure due to Okasaki [Oka99]. We present below our proof of functional correctness and complexity, which involves some interesting reasoning on O.

7.7.1 Implementation

A "binary random access list" is a functional data structure that stores a sequence of elements, and provides logarithmic-time list operations (`cons` adds an element at the head of the list, `head` and `tail` remove the head of the list) as well as logarithmic-time random access operations (`lookup` and `update` read and modify the element at a given position, respectively). That is, one can add or remove an element at the head of the list, but also modify or query the i^{th} element of the structure. These five operations run in worst-case $O(\log(n))$ steps, where n is the number of elements stored in the structure.

In short, a binary random access list is a list which stores in its i^{th} cell either nothing, or a complete binary tree of depth i. As an example, Figure 7.7 represents a binary random access list storing the sequence $1, 2, 3, 4, 5$.

Let us walk through our OCaml implementation. The complete code appears in Figure 7.6.

Type definitions and implicit invariants

A binary random access list is a list of binary trees: we first define the OCaml type for trees.

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

Notice that only the leaves store elements. Nodes contain an integer corresponding to the number of elements stored in the leaves of the tree. Now, a binary random access list is a list of either a tree, either nothing. We consequently define an type (isomorphic to `tree option`) dubbed `digit`.

```
exception Empty

type 'a tree = Leaf of 'a | Node of int * 'a tree * 'a tree
type 'a digit = Zero | One of 'a tree
type 'a rlist = 'a digit list

let empty : 'a rlist = []
let is_empty = function [] -> true | _ -> false

let size = function
  | Leaf x -> 1
  | Node (w, _, _) -> w

let link t1 t2 =
  Node (size t1 + size t2, t1, t2)

let rec cons_tree t = function
  | [] -> [One t]
  | Zero :: ts -> One t :: ts
  | One t' :: ts -> Zero :: cons_tree (link t t') ts

let rec uncons_tree = function
  | [] -> raise Empty
  | [One t] -> (t, [])
  | One t :: ts -> (t, Zero :: ts)
  | Zero :: ts -> match uncons_tree ts with
                  | Node (_, t1, t2), ts' -> (t1, One t2 :: ts')
                  | _ -> assert false

let cons x ts =
  cons_tree (Leaf x) ts

let head ts =
  match uncons_tree ts with
  | (Leaf x, _) -> x
  | _ -> assert false

let tail ts =
  let (_,ts') = uncons_tree ts in ts'

let rec lookup_tree i = function
  | Leaf x -> if i = 0 then x else raise (Invalid_argument "lookup")
  | Node (w, t1, t2) -> if i < w/2
                        then lookup_tree i t1
                        else lookup_tree (i - w/2) t2

let rec update_tree i y = function
  | Leaf x -> if i = 0 then Leaf y else raise (Invalid_argument "update")
  | Node (w, t1, t2) -> if i < w/2
                        then Node (w, update_tree i y t1, t2)
                        else Node (w, t1, update_tree (i - w/2) y t2)

let rec lookup i = function
  | [] -> raise (Invalid_argument "lookup")
  | Zero :: ts -> lookup i ts
  | One t :: ts -> if i < size t
                   then lookup_tree i t
                   else lookup (i - size t) ts

let rec update i y = function
  | [] -> raise (Invalid_argument "update")
  | Zero :: ts -> Zero :: update i y ts
  | One t :: ts -> if i < size t
                   then One (update_tree i y t) :: ts
                   else One t :: update (i - size t) y ts
```

Figure 7.6: OCaml implementation of the binary random access list structure and operations.

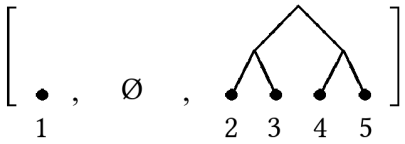

Figure 7.7: A binary random access list storing the sequence $1, 2, 3, 4, 5$.

```
type 'a digit = Zero | One of 'a tree
```

The name `digit` comes from the similarity between a binary random access list and a list of bits, representing an integer—adding an element at the head being similar to incrementing the integer, etc. We detail this intuition further in the next subsection. Finally, we define the type for the whole structure: the binary random access list.

```
type 'a rlist = 'a digit list
```

In fact, the type definition `rlist` represents both complete binary random access lists and binary random access list *segments*. The difference between the two lies in invariants of the structure, which are not made explicit by the OCaml types:

- Trees (of type `'a tree`) are complete trees—a tree of depth d must always have 2^d leaves;
- A binary random access list *segment* (of type `'a rlist`) contains trees of increasing depth starting at some initial depth p. If the i^{th} cell (indexing from 0) contains a tree (i.e. `One t`), then this tree has depth $p + i$;
- A *complete* binary random access list (also of type `'a rlist`) is a list segment which starting depth p is equal to 0: its first tree, if present, must be a leaf.

Recursive functions that work on the type `rlist` typically operate on list segments; then, at the toplevel, they get applied to a complete list.

Binary random access lists look like integers in binary

As mentioned before, a binary random access list is in fact quite similar to an integer represented in binary, e.g. as a list of bits.

Actually, if one erases the trees from the implementation (`type 'a digit = Zero | One of 'a tree` becomes `type digit = Zero | One`), one obtains an implementation of integers, represented as a list of bits (least significant bit at the head of the list); the `cons` operation being incrementing, and `uncons` decrementing.

Incrementing an integer consists in looking at the least significant bit; if it's a zero, turning it into a one; if it's a one, turning it into zero, and recursively continuing to increment, starting with the next bit. The `cons_tree` operation on binary random access lists works in a similar fashion. Instead of adding 1, we add a tree `t` (more precisely, a digit `One t`): if the first element of the list is `Zero`, we turn it into `One t`. If it's a `One t'`, we turn it into `Zero`, and recursively continue, but with a new tree which combines the elements of `t` and the elements of `t'`, using the auxiliary `link` operation. The `cons` operations simply corresponds to applying `cons_tree` to a tree that contains a single element.

The `uncons_tree` operation follows the same idea: it is similar to decrementing, except that we also have to invert the `link` operation to obtain trees of smaller depth.

Calling `uncons_tree` on a list segment of starting depth p always returns the pair of a tree of depth p, and a list segment of starting depth $p + 1$ (or fails if the list is empty). On a complete list (as the `head` function assumes) where the starting depth is zero, this entails that the tree returned by `uncons_tree` is necessarily a leaf.

Random access

The structure invariants make it easy to implement random access lookup and update. The idea is to walk through the list; faced to a `Node (w, l, r)` tree, we know how much elements it contains: it is exactly `w`. Knowing the index i of the element we want to visit, we can compare it to `w` to know whether we should explore this tree (if $i < w$), or continue walking the list. This gives the implementation of `lookup`.

If we have to walk through the tree (`lookup_tree`), we can also do this without performing an exhaustive exploration: by comparing the index to half the size of the tree, we can decide whether we should explore the left or right subtree.

The `update` and `update_tree` functions work in a similar fashion.

7.7.2 Functional Invariants

CFML automatically generates Coq inductive definitions that correspond to the OCaml definitions of `tree`, `digit` and `rlist`. We manually define three Coq predicates `btree`, `inv` and `Rlist`, that make explicit the data structure invariants associated to a complete tree, a list segment, and a complete list, respectively. Their definition appear in Figure 7.8. We note "`a`" the type (in Coq) of elements stored in the structure (in the Coq formalization, it is quantified as a section variable, i.e. a parameter of the whole development).

The Coq proposition `btree n t L` asserts that the tree `t` of type `tree a` is a complete tree of depth `n` which contains the sequence of elements in the list `L`. The proposition `inv p ts L` means that `ts` is a list segment (of type `rlist a`) composed of complete trees of increasing depth, starting at depth `p`. `L` is the sequence of elements represented by `ts`. Finally, the `Rlist` predicate corresponds to a complete list, that is, a list segment with initial depth equal to zero. The proposition `Rlist s L` asserts that `s` is a complete random access list structure (of type `rlist a`) containing the elements described by the Coq list `L` (of type `list a`).

Provided that these invariants hold, one can deduce two additional properties, given by lemmas `btree_length` and `inv_ts_bound_log` (Figure 7.8). In particular, `inv_ts_bound_log` shows that length of the "spine" of a binary random access list (i.e. the length of the underlying list of trees) is logarithmic in the number of elements stored in the structure. These lemmas will be key in our verification of the asymptotic complexity.

7.7.3 Complexity Analysis

Let us walk through the complexity analysis of two representative functions of the library: `cons_tree` and `lookup`.

```
Inductive btree : int → tree a → list a → Prop :=
  | btree_leaf : ∀x,
      btree 0 (Leaf x) (x::nil)
  | btree_node : ∀p p' n t1 t2 L1 L2 L',
      btree p t1 L1 → btree p t2 L2 →
      p' = p+1 →
      n = 2^p' →
      L' = L1 ++ L2 →
      btree p' (Node n t1 t2) L'.

Inductive inv : int → rlist a → list a → Prop :=
  | inv_nil : ∀p,
      0 ≤ p →
      inv p nil nil
  | inv_cons : ∀p (t: tree a) ts d L L' T,
      inv (p+1) ts L →
      L' ≠nil →
      0 ≤ p →
      (match d with
       | Zero ⇒L = L'
       | One t ⇒btree p t T ∧L' = T ++ L
       end) →
      inv p (d :: ts) L'.

Definition Rlist (s: rlist a) (L: list a) :=
  inv 0 s L.

Lemma btree_length : ∀t p L,
  btree p t L → length L = 2^p.
Lemma inv_ts_bound_log : ∀ts p L,
  inv p ts L → length ts ≤ Z.log2 (2 * (length L) + 1).
```

Figure 7.8: Binary random access list: functional invariants.

```
Lemma cons_tree_spec_aux :
  specZ [cost \in_O (fun n ⇒ n)]
    (∀ a (t: tree a) (ts: rlist a) p T L,
     btree p t T → inv p ts L →
     app cons_tree [t ts]
       PRE ($ cost (length ts))
       POST (fun ts' ⇒ [inv p ts' (T ++ L)])).

Lemma cons_tree_spec :
  specZ [cost \in_O Z.log2]
    (∀ a (t: tree a) (ts: rlist a) p T L,
     btree p t T → inv p ts L →
     app cons_tree [t ts]
       PRE ($ cost (length L))
       POST (fun ts' ⇒ [inv p ts' (T ++ L)])).
```

Figure 7.9: Auxiliary and main specification for `cons_tree`.

`cons_tree`: a proof sketch

A call to "`cons_tree t ts`" adds the tree `t` to the list segment `ts`. It may recursively walk through the list `ts`, calling the auxiliary `link` function. Because `link` runs in constant time, `cons_tree` performs $O(|ts|)$ operations. Moreover, we showed earlier that $|ts|$ is logarithmic in $|L|$, the number of elements contained in `ts`. Therefore, `cons_tree` runs in $O(\log(|L|))$ (ultimately, we wish to express the complexity of operations on the structure depending on the number of elements $|L|$).

Our formal proof follows this two-step informal reasoning. First, as an auxiliary specification, we prove that `cons_tree` has linear complexity depending on $|ts|$, by reasoning by induction on `ts`. Then, we use the `inv_ts_bound_log` lemma to prove that `cons_tree` has logarithmic complexity depending on the number of elements stored in the structure. Both auxiliary and main specifications appear in Figure 7.9.

We establish the auxiliary specification `cons_tree_spec_aux` by directly picking a cost function of the form "$\lambda n.\ a \cdot n + b$", for some a and b (introduced using our `procrastination` library §6.5). Carrying on the proof (by induction on `ts`) yields the following side-conditions on a and b:

$$1 \leq b \ \wedge\ 0 \leq a \ \wedge\ 1 \leq a + b \ \wedge\ 1 + \texttt{cost link\_spec}\ () + b \leq a + b$$

Appropriate values for a and b are then automatically chosen by our `elia` tactic (§6.6).

The key step of the proof of the main specification `cons_tree_spec` is to show that $\lambda n.$ `cost cons_tree_spec_aux` $(\log(2 \cdot n + 1)))$ is an adequate cost function. Recall that "`cost cons_tree_spec_aux`" denotes the cost function associated to the auxiliary specification, as per the definition of `specO` (§6.2).

From the auxiliary specification we know that calling `cons_tree t ts` has cost `cost cons_tree_spec_aux` $|ts|$. Additionally, lemma `inv_ts_bound_log` allows us to bound the length of `ts`: $|ts| \leq \log(2 \cdot |L| + 1)$. Finally, `cost cons_tree_spec_aux` is monotonic (from the definition of `specO`). Therefore, one can bound the cost of `cons_tree` by `cost cons_tree_spec_aux` $(\log(2 \cdot |L| + 1))$, which only depends on $|L|$.

```
Lemma lookup_spec_aux :
  specO product_positive_order ZZle
    (fun cost ⇒
     ∀(a: Type)  {Inhab a} i (ts: rlist a) p L,
     inv p ts L →
     0 ≤ i < length L →
     app lookup [i ts]
       PRE ($ cost (p, length ts))
       POST (fun x ⇒ [x = L[i]]))
    (fun '(p,s) ⇒ p + s).

Lemma lookup_spec :
  specZ [cost \in_O Z.log2]
    (∀ (a: Type)  {Inhab a} i (ts: rlist a) L,
     Rlist ts L →
     0 ≤ i < length L →
     app lookup [i ts]
       PRE ($ cost (length L))
       POST (fun x ⇒ [x = L[i]])).
```

Figure 7.10: Auxiliary and main specification for `lookup`.

The proof of the asymptotic bound for such a cost function follows immediately from the auxiliary specification, using the composition lemma on O (Lemma 5.3.16).

`lookup`: a proof sketch involving multivariate O

The `lookup` function operates on binary random access list segments, whose size depends on two parameters: p, the height of the first tree, and $|ts|$, the length of the list (of trees).

An informal analysis of the complexity of `lookup` goes as follows. The `lookup` function first walks through the list of trees: this costs $O(|ts|)$. Then, it calls `lookup_tree` to explore one of the trees; `lookup_tree` is linear in the height of the tree. Trees that appear in the list segment have height greater than p, and smaller than $p + |ts|$. Therefore, the overall complexity of `lookup` is $O(|ts| + p + |ts|)$, i.e. $O(p + |ts|)$. Finally, when `lookup` is called on a complete list segment, we know that p is equal to zero. In that case, the complexity of `lookup` is therefore $O(|ts|)$, and thus is also $O(\log(|L|))$ (through a similar reasoning than with `cost_tree`).

Our Coq proof follows a similar two-step process, by establishing an auxiliary specification which is then used to prove the main specification. Both specifications appear in Figure 7.10. First, we establish (by induction) a multivariate asymptotic bound for of "`lookup` i ts" of the form $O(\lambda(p, s).\ p + s)$, where ts is a list segment, s corresponds to $|ts|$ and p to the height of the first tree of the list segment. Then, we deduce that, if ts is a complete binary random access list, then the cost of "`lookup` i ts" is bounded by $O(\lambda n.\ \log(n))$ where n corresponds to the number of elements $|L|$.

For this last reasoning step to hold, the auxiliary multivariate bound must hold even in the case where p is equal to zero (as discussed previously in Section 5.4). We therefore establish the multivariate bound with respect to the nonstandard filter $(\top \mid \lambda n.\ n \geq 0) \times \mathbb{Z}$

(named `product_positive_order` in the Coq statement of the specification). This filter requires the second parameter to be "large enough", and the first parameter to be "any nonnegative integer" (and in particular, allows it to be zero).

Let us give the main steps of the proof of the auxiliary specification `lookup_spec_aux`. Reasoning by induction on `ts`, we infer the following recurrence equation for the cost function *cost* (which has type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$):

$$\begin{array}{l} \forall p\, s.\ p \geq 0 \ \wedge\ s \geq 0 \implies \\ 1 + \max \left(\begin{array}{l} \mathit{cost}(p+1, s), \\ \texttt{cost size\_spec}\ () + \max \left(\begin{array}{l} \texttt{cost lookup\_tree\_spec}\ p, \\ \texttt{cost size\_spec}\ () + \mathit{cost}(p+1, s) \end{array} \right) \end{array} \right) \\ \leq \mathit{cost}(p, s+1). \end{array}$$

Then, we use the substitution method, and guess a solution of the following form, for some constants a, b, c and d:

$$\mathit{cost}(p, s) \ \triangleq\ a \cdot p + b \cdot s + c \cdot \texttt{cost lookup\_tree\_spec}\ (p + s) + d.$$

Injecting this shape into the recurrence equation yields the following constraints on a, b, c and d, which are easily proved specifiable using our `elia` tactic (§6.6).

$$\begin{array}{ll} & 1 \leq c \ \wedge\ 0 \leq a \ \wedge\ 0 \leq b \ \wedge\ 1 + (a + d) \leq b + d \\ \wedge & 1 + \texttt{cost size\_spec}\ () + \texttt{cost size\_spec}\ () + a + d \leq b + d \\ \wedge & 1 + \texttt{cost size\_spec}\ () \leq b + d \end{array}$$

Then, one needs to show that *cost* satisfies the expected asymptotic bound, that is:

$$\mathit{cost} \ \preceq_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ \lambda(p, s).\ p + s$$

By repeated application of Lemmas 5.3.1, 5.3.4 and 5.3.5 (that work on any filter), we turn this goal into several domination sub-goals about subterms of the expression of *cost*, and get rid of the multiplicative constants a, b and c.

$$\begin{array}{ll} & \lambda(p, s).\ p \ \preceq_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ \lambda(p, s).\ p + s \\ \wedge & \lambda(p, s).\ s \ \preceq_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ \lambda(p, s).\ p + s \\ \wedge & \lambda(p, s).\ \texttt{cost lookup\_tree\_spec}\ (p + s) \ \preceq_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ \lambda(p, s).\ p + s \\ \wedge & \lambda(p, s).\ d \ \preceq_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ \lambda(p, s).\ p + s \end{array}$$

The first two obligations can be discharged by noticing that $\mathbb{U}_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ (p, s).\ p \leq p + s$ holds, by unfolding the definition of the filter. Respectively, $\mathbb{U}_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}\ (p, s).\ s \leq p + s$ holds. The third obligation can be proved by a direct application of the Composition Lemma (Lemma 5.3.16), using the fact that $\texttt{cost lookup\_tree\_spec} \preceq_{\mathbb{Z}} \lambda n.\ n$. To solve the last obligation, we first apply Lemma 5.3.6, which then requires proving that the function $\lambda(p, s).\ p + s$ "grows large", i.e. $(\lambda(p, s).\ p + s) \nearrow^{\mathbb{Z}}_{(\top | \lambda n.\, n \geq 0) \times \mathbb{Z}}$. This can be proved directly by unfolding the definitions.

We prove the main specification `lookup_spec` as a weakening of the auxiliary specification. We show that an appropriate cost function for the main specification is:

$$\lambda n.\ \texttt{cost lookup\_spec\_aux}\ (0, \log(2 \cdot n + 1)).$$

Indeed, we can show that this cost function bounds the cost advertised by the auxiliary specification. If `Rlist` ts L holds, then using lemma `inv_ts_bound_log` and the monotonicity of cost functions, we obtain that `cost lookup_spec_aux` $(0, |ts|)$ is bounded by `cost lookup_spec_aux` $(0, \log(2 \cdot |L| + 1))$. Finally, by using the Composition Lemma, we show that such a cost function is dominated by $\lambda n.\ \log n$. (As a side condition, we have to prove that $(\lambda\_.\ 0) \nearrow_{\mathbb{Z}}^{(\top \mid \lambda n.\ n \geq 0)}$, which holds directly after unfolding the definitions.)

Chapter 8

Case Study: A State-of-the-art Incremental Cycle Detection Algorithm

The previous chapter focuses on small examples that illustrate specific programming patterns. In this chapter, we present a more challenging case study, a major contribution of this thesis. We verify the correctness and worst-case amortized asymptotic complexity of an incremental cycle detection algorithm (and data structure) due to Bender, Fineman, Gilbert, and Tarjan [BFGT16, §2] (2016). With this data structure, the complexity of building a directed graph of n vertices and m edges, while incrementally checking that no edge insertion creates a cycle, is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Although its implementation is relatively straightforward (only 200 lines of OCaml), its design is subtle, and it is far from obvious, by inspection of the code, that the advertised complexity bound is respected.

As a second contribution, on the algorithmic side, we simplify and enhance 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 (i.e., know in advance) 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 propose a modified algorithm, where the extent of the backward search is limited by a value that depends only on the *current* state. The pseudocode for both algorithms appears in Figure 8.2; it is explained later on (§8.2). The modified algorithm has the same complexity as the original algorithm and is a genuine online algorithm. It is the one that we verify.

Our verification effort has had some practical impact already. For instance, the Dune build system [Str18] needs an incremental cycle detection algorithm in order to reject circular build dependencies as soon as possible. For this purpose, the authors of Dune developed an implementation of Bender *et al.*'s original algorithm, which we recently replaced with our improved and verified algorithm [Gué19]. Our contribution increases the trustworthiness of Dune's code base, without sacrificing its efficiency: in fact, our measurements suggest that our code can be as much as 7 times faster than the original code in a real-world scenario. As another potential application area, it is worth mentioning that Jacques-Henri Jourdan has deployed an as-yet-unverified incremental cycle detection algorithm in the kernel of the Coq proof assistant [The19b], where it

is used to check the satisfiability of universe constraints [ST14, §2]. At the time, this yielded a dramatic improvement in the overall performance of Coq's proof checker: the total time required to check the Mathematical Components library dropped from 25 to 18 minutes [Jou15]. The algorithm deployed inside Coq is more general than the verified algorithm considered in this paper, as it also maintains strong components, as in Section 4 of Bender *et al.*'s paper [BFGT16]. Nevertheless, we view the present work as one step towards verifying Coq's universe inference system.

Our code and proofs are available online [GJCP19]. Our methodology is modular: at the end of the day, the verified data structure is equipped with a succinct specification (Figure 8.1) which is intended to serve as the sole reference when verifying a client of the algorithm. We believe that this case study illustrates the great power and versatility of our approach, and we claim that this approach is generally applicable to many other nontrivial data structures and algorithms.

8.1 Specification of the Algorithm

The interface for an incremental cycle detection algorithm 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.

Figure 8.1 shows a formal specification for an incremental cycle detection algorithm. It consists of six statements. InitGraph, AddVertex, and AddEdge are Separation Logic triples: they assign pre- and postconditions to the three public operations. DisposeGraph and Acyclicity are Separation Logic entailments. The last statement, Complexity, provides a complexity bound. It is the only statement that is specific to the algorithm discussed in this paper. Indeed, the first five statements form a generic specification, which any incremental cycle detection algorithm could satisfy.

The six statements in the specification share three variables, namely IsGraph, IsNewVertex, and ψ. These variables are implicitly existentially quantified in front of the specification: a user of the algorithm must treat them as abstract.

The predicate IsGraph is an *abstract representation predicate*, a standard notion in Separation Logic [PB05]. It is parameterized with a memory location g and with a mathematical graph G. The assertion $\mathrm{IsGraph}\, g\, G$ means that a well-formed data structure, which represents the mathematical graph G, exists at address g in memory. At the same time, this assertion denotes the unique ownership of this data structure.

Because this is Separation Logic with Time Credits, the assertion $\mathrm{IsGraph}\, g\, G$ can also represent the ownership of a certain number of credits. For example, for the specific algorithm considered in this paper, we later define $\mathrm{IsGraph}\, g\, G$ as $\exists L.\ \$\phi(G, L) \star \ldots$ (§8.5), where ϕ is a suitable potential function [Tar85]. ϕ is parameterized by the graph G and by a map L of vertices to integer levels. Intuitively, this means that $\phi(G, L)$ credits are stored in the data structure. These details are hidden from the user: ϕ does not appear in Figure 8.1. Yet, the fact that $\mathrm{IsGraph}\, g\, G$ can involve credits means that the user must read AddVertex and AddEdge as amortized specifications [Tar85]: the actual cost of

InitGraph
$\exists k.\ \{\$k\}\ \texttt{init\_graph()}\ \{\lambda g.\ \mathrm{IsGraph}\, g\, \varnothing\}$

DisposeGraph
$\forall g\, G.\ \mathrm{IsGraph}\, g\, G \ \Vdash\ \mathrm{GC}$

AddVertex
$\forall g\, G\, v.$
$\mathrm{let}\ m, n := |\mathrm{edges}\ G|\,, |\mathrm{vertices}\ G|\ \mathrm{in}$
$v \notin \mathrm{vertices}\ G \implies$
$\{\ \mathrm{IsGraph}\, g\, G \star \mathrm{IsNewVertex}\, v \star \$(\psi\,(m, n+1) - \psi\,(m, n))\ \}$
$(\texttt{add\_vertex}\ g\ v)$
$\{\ \lambda().\ \mathrm{IsGraph}\, g\, (G + v)\ \}$

AddEdge
$\forall g\, G\, v\, w.$
$\mathrm{let}\ m, n := |\mathrm{edges}\ G|\,, |\mathrm{vertices}\ G|\ \mathrm{in}$
$v, w \in \mathrm{vertices}\ G \wedge (v, w) \notin \mathrm{edges}\ G \implies$
$\{\ \mathrm{IsGraph}\, g\, G \star \$(\psi\,(m+1, n) - \psi\,(m, n))\ \}$
$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$
$\left\{\ \lambda res.\ \mathrm{match}\ res\ \mathrm{with} \mid \mathrm{Ok} \Rightarrow \mathrm{IsGraph}\, g\, (G + (v, w)) \mid \mathrm{Cycle} \Rightarrow [w \longrightarrow_G^* v]\ \right\}$

Acyclicity
$\forall g\, G.\ \mathrm{IsGraph}\, g\, G \ \Vdash\ \mathrm{IsGraph}\, g\, G \star [\forall x.\ x \not\longrightarrow_G^+ x]$

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

Figure 8.1: Specification of an incremental cycle detection algorithm.

a single `add_vertex` or `add_edge_or_detect_cycle` operation is not directly related to the number of credits that explicitly appear in the precondition of this operation.

The predicate IsNewVertex is also an abstract representation predicate. The assertion IsNewVertex v asserts that a well-formed isolated vertex exists at address v in memory.

The function ψ has type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. In short, $\psi(m, n)$ 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. This informal claim is explained later on in this section.

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 $O(1)$.

DisposeGraph states that the assertion IsGraph g G is affine: that is, it is permitted to forget about the existence of a valid graph data structure. By publishing this statement, we guarantee that we are not hiding a debt inside the abstract predicate IsGraph. Indeed, to prove that DisposeGraph holds, we must verify that the potential $\phi(G, L)$ is nonnegative (§6.3). In a programming langage without garbage collection, this statement would be replaced with a Separation Logic triple for an explicit `dispose_graph` operation.

AddVertex states that `add_vertex` requires a valid data structure, described by the assertion IsGraph g G, a valid isolated vertex v, described by IsNewVertex v, and returns a valid data structure, described by IsGraph g $(G + v)$. (We write $G + v$ for 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.) Inserting the vertex into the graph structure consumes the assertion IsNewVertex v: ownership of the vertex has been merged within the updated IsGraph assertion. 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`.

Similarly, AddEdge states that the cost of `add_edge_or_detect_cycle` is $\psi(m + 1, n) - \psi(m, n)$. This operation returns either Ok, in which case the graph has been successfully extended with a new edge from v to w, or Cycle, in which case this new edge cannot be added, because there already is a path in G from w to v. In the latter case, the data structure is invalidated: the assertion IsGraph g G is not returned. Thus, in that case, no further operations on the graph are allowed.

By combining the first four statements in Figure 8.1, 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 `add_edge_or_detect_cycle`, satisfies the specification $\{\$(k + \psi(m, n))\}$ ... $\{\text{GC}\}$, where k is the cost of `init_graph`. Indeed, the cumulated cost of the calls to `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 $\psi(m, n)$.

By soundness of Separation Logic with Time Credits, the triple $\{\$(k+\psi(m, n))\}$ ... $\{\text{GC}\}$ implies that the actual worst-case cost of the sequence of operations is $k + \psi(m, n)$. This confirms our earlier informal claim that $\psi(m, n)$ represents the cost of creating n vertices and m edges.

Acyclicity states that, from the Separation Logic assertion IsGraph g G, the user can deduce that G is acyclic. In other words, as long as the data structure remains in a

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 F edges have been traversed, interrupt the backward search
    → in Bender et al.'s algorithm, F is a constant Δ
    → in our algorithm, F is $L(v)$
- 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 $L(x) \geq L(w)$, 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 8.2: Pseudocode for Bender *et al.*'s algorithm and for our improved algorithm.

valid state, the graph G remains acyclic. In particular, this statement can be exploited to prove that, if `add_edge_or_detect_cycle` returns `Ok`, then the new edge indeed does not create a cycle.

Although the exact definition of ψ is not exposed, Complexity provides an asymptotic bound: $\psi(m, n) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Technically, we are using the domination relation $\preceq$ along with the product filter on $\mathbb{Z} \times \mathbb{Z}$. Our complexity bound thus matches the one published by Bender *et al.* [BFGT16].

8.2 Overview of the Algorithm

We provide pseudocode for Bender *et al.*'s algorithm [BFGT16, §2] and for our improved algorithm in Figure 8.2. The only difference between the two algorithms is the manner in which a certain internal parameter, named F, is set. The value of F influences the complexity of the algorithm, not its correctness.

Interactive version. We developed an interactive user interface for the algorithm, for pedagogical purposes. The interface allows the user to interactively construct a graph and visualize the changes made by the algorithm on the level of nodes, as well as which nodes are explored during the backward and forward search.

`https://agueneau.gitlabpages.inria.fr/incremental-cycles/webapp/`

We encourage the reader to experiment using this interactive version of the algorithm while reading the textual explanations below; we find that it helps greatly in getting a grasp on the key ideas behind the algorithm.

Overview. 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.

One key feature of Bender *et al.*'s algorithm is that a positive integer level $L(v)$ is associated with every vertex v, and the following invariant is maintained: L forms a pseudo-topological numbering. That is, "no edge goes down": if there is an edge from v to w, then $L(v) \leq L(w)$ 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 feature of Bender *et al.*'s 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 of 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 a predetermined number of edges, denoted by the letter F in Figure 8.2.

A third key characteristic of Bender *et al.*'s 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 of 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.

The last key aspect of Bender *et al.*'s algorithm is the choice of F. On the one hand, as F increases, backward searches become more expensive, as each backward search processes up to F edges. On the other hand, as F decreases, forward searches become more expensive. Indeed, a smaller value of F leads to the creation of a larger number of levels, and (as explained later) the total cost of the forward searches is proportional to the number of levels.

Bender *et al.* set F to a constant Δ, defined as $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm, where m and n are upper bounds on the *final* numbers of edges and vertices in the graph. As mentioned earlier in the introduction of the chapter, it seems preferable to set F to a value that does not depend on such upper bounds, as they may not be known ahead of time. In our modified algorithm, F stands for $L(v)$, where v is the source of the edge that is being inserted. This value depends only on the *current*

state of the data structure, so our algorithm is truly online. We prove that it has the same complexity as Bender *et al.*'s original algorithm, namely $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

8.3 Informal Complexity Analysis

We now present an informal complexity analysis of Bender *et al.*'s original algorithm. It is written under the assumption that the parameter F is fixed: it remains constant throughout the execution of the algorithm. These results are not new; they are a more detailed reconstruction of the proof that appears in Bender *et al.*'s article.

The proof sketch for Lemma 8.3.1 below has a temporal aspect: it refers to the state of the data structure at various points in time. Our formal proofs, on the other hand, are carried out in Separation Logic, which implies that they are based purely on assertions that describe the current state at each program point.

Let us say that an edge is at level k when both of its endpoints are at level k. In Bender *et al.*'s algorithm, the following key invariant holds:

Lemma 8.3.1 (Bound on the number of levels) *For every level k except the highest level, there exist at least F edges at level k.*

Proof We consider a vertex v at level $k+1$, and show that there exist F edges at level k. Because there is a vertex v at level $k+1$, there must exist a vertex v' at level $k+1$ that has no predecessor at this level. The backward search that promoted v' to this level must have traversed F edges that were at level k at that time. Thus, it suffices to show that, at the present time, these edges are still at level k. By way of contradiction, suppose that the target vertex v'' of one of these edges is promoted to some level k' that is greater than k. (If the source vertex of this edge is promoted, then its target vertex is promoted as well.) Because v'' is an ancestor of v', the vertex v' is necessarily also promoted to level k' during the same forward search. But v' is now at level $k+1$, and the level of a vertex never decreases, so k' must be equal to $k+1$. There follows that v' has an ancestor v'' at level $k+1$, contradicting the assumption that v' has no predecessor at its level.

From this invariant, one can establish the complexity of the algorithm, parameterized by F:

Lemma 8.3.2 (Asymptotic complexity (parametric)) *The cost of a sequence of n vertex creation and m edge creation is $O(m \cdot (F + \min(m/F, n/\sqrt{F})))$.*

Proof From the invariant of Lemma 8.3.1, one can derive two upper bounds on the number of levels. Let K denote the number of nonterminal levels (i.e., the number of levels containing graph nodes, excluding the highest level). First, the invariant implies $m \geq KF$, therefore $K \leq m/F$. Furthermore, for each nonterminal level k, the vertices at level k form a subgraph with at least F edges. A graph (without duplicate edges) with n vertices has at most n^2 edges. Equivalently, a graph with m edges has at least $\sqrt{m}$ vertices. Therefore, at every nonterminal level, there are at least $\sqrt{F}$ vertices. This implies $n \geq K\sqrt{F}$, therefore $K \leq n/\sqrt{F}$.

Let us now estimate the algorithm's complexity. Consider a sequence of n vertex creation and m edge creation operations. The cost of one backward search is $O(F)$, as it traverses at most F edges. Because each edge insertion triggers one such search, the total cost of the backward searches is $O(mF)$. The forward search traverses an edge if and only if this edge's source vertex is promoted to a higher level. Therefore, the cost of a forward search is linear in the number of edges whose source vertex is thus promoted. Because there are m edges and because a vertex can be promoted at most K times, the total cost of the forward searches is $O(mK)$. In summary, the cost of this sequence of operations is $O(mF + mK)$.

By combining this result with the two bounds on K obtained above, one finds that the complexity of the algorithm is $O(m \cdot (F + \min(m/F, n/\sqrt{F}))$.

Let Δ stand for $\min(m^{1/2}, n^{2/3})$. Then, setting F to Δ yields the desired asymptotic complexity bound:

Lemma 8.3.3 (Asymptotic complexity) *Suppose F is Δ. Then, a sequence of n vertex creation and m edge creation operations costs $O(m \cdot \min(m^{1/2}, n^{2/3}))$.*

Proof We have established in Lemma 8.3.2 that the algorithm has time complexity:

$$O(m \cdot (F + \min(m/F,\ F + n/\sqrt{F}))$$

Our goal is to establish that, when F is Δ, this bound is equivalent to $O(m\Delta)$. To that end, it suffices to show that $\min(m/\Delta, \Delta + n/\sqrt{\Delta})$ is $O(\Delta)$. Let V stand for $\min(m\Delta^{-1}, n\Delta^{-1/2}))$, and let us show that V is $O(\Delta)$. Recall that Δ is defined as $\min(m^{1/2}, n^{2/3})$. We distinguish two cases.

First, assume $m^{1/2} \leq n^{2/3}$. Then, $\Delta = m^{1/2}$ and $V = \min(m^{1/2}, nm^{-1/4})$. The left-hand side of this minimum is smaller, because $m^{1/2} \leq nm^{-1/4}$ iff $m^{3/4} \leq n$ iff $m^{1/2} \leq n^{2/3}$. Thus, $V = m^{1/2} = \Delta$.

Second, assume $m^{1/2} \geq n^{2/3}$. Then, $\Delta = n^{2/3}$ and $V = \min(mn^{-2/3}, n^{2/3})$. The right-hand side of this minimum is smaller, because $mn^{-2/3} \geq n^{2/3}$ iff $m \geq n^{4/3}$ iff $m^{1/2} \geq n^{2/3}$. Thus, $V = n^{2/3} = \Delta$.

The above proof sketch may appear to "work by magic". How can one guess an appropriate setting of F? The following discussion provides some insight.

Lemma 8.3.4 (Selecting an optimal value of F) *Setting the parameter F to Δ leads to the best asymptotic complexity bound for Bender* et al.*'s algorithm.*

Proof Let $f(m, n, F)$ denote the quantity $F + \min(m/F,\ F + n/\sqrt{F})$ which appears in the asymptotic time complexity of Bender *et al.*'s algorithm (§8.3). We are seeking an optimal setting for the constant F, expressed in terms of the final values of m and n. Thus, F is technically a function of m and n. When F is presented as a function, the function f may be defined as follows:

$$\begin{aligned} f_1(m, n, F) &= F(m, n) + m \cdot F^{-1}(m, n) \\ f_2(m, n, F) &= F(m, n) + n \cdot F^{-1/2}(m, n) \\ f(m, n, F) &= \min(f_1(m, n, F), f_2(m, n, F)) \end{aligned}$$

Our goal is to find a function $F(m,n)$ such that $\lambda(m,n).\, f(m,n,F(m,n))$ is a function minimal with respect to the domination relation between functions over $\mathbb{Z}\times\mathbb{Z}$.

For fixed values of m and n, the value of $F(m,n)$ is a constant. Consider the function $g_1(\delta)=\delta+m\cdot\delta^{-1}$ and the function $g_2(\delta)=\delta+n\cdot\delta^{-1/2}$. They are defined in such a way that $f_1(m,n,F)=g_1(F(m,n))$, and $f_2(m,n,F)=g_2(F(m,n))$, and $f(m,n,F)=\min(g_1(F(m,n)),g_2(F(m,n)))$, for the values of m and n considered.

The functions g_1 and g_2 are convex, and thus each of them admits a unique minimum. Let δ_1 be the argument that minimizes g_1 and δ_2 the argument that minimizes g_2. The value of $F(m,n)$ that we are seeking for is the value that minimizes the expression $\min(g_1(F(m,n)),g_2(F(m,n)))$, so it either δ_1 or δ_2, depending on the comparison between $g_1(\delta_1)$ and $g_2(\delta_2)$.

The values of δ_1 and δ_2 are the input values that cancel the derivatives of g_1 and g_2 (derivatives with respect to δ). On the one hand, the derivative of g_1 is $1-m\delta^{-2}$. This value is zero when $\delta^2=m$. Thus, δ_1 is $\Theta(m^{1/2})$. On the other hand, the derivative of g_2 is $1-\frac{1}{2}n\delta^{-3/2}$. This value is zero when $\delta^{3/2}=\frac{1}{2}n$. Thus, δ_2 is $\Theta(n^{2/3})$.

Let us evaluate the two functions g_1 and g_2 at their minimum. First, $g_1(\delta_1)$ is $\Theta(m^{1/2}+mm^{-1/2})$, thus is $\Theta(m^{1/2})$. Second, $g_2(\delta_2)$ is $\Theta(n^{2/3}+nn^{-1/3})$, thus is $\Theta(n^{2/3})$.

As explained earlier, for the values of m and n considered, the minimum value of $f(m,n,F)$ is equal to either $g_1(\delta_1)$ or $g_2(\delta_2)$. Thus, this minimum value is $\Theta(\min(m^{1/2},n^{2/3}))$. To reach this minimum value, $F(m,n)$ should be defined as: $\Theta(\text{if } m^{1/2}\leq n^{2/3} \text{ then } \delta_1 \text{ else } \delta_2)$. Interestingly, this expression can be reformulated as $\Theta(\text{if } m^{1/2}\leq n^{2/3} \text{ then } m^{1/2} \text{ else } n^{2/3})$, which simplifies to: $\Theta(\min(m^{1/2},n^{2/3}))$.

In conclusion, setting $F(m,n)$ to $\Theta(\min(m^{1/2},n^{2/3}))$ leads to the minimal asymptotic value of $f(m,n,F)$, hence to the minimal value of $m\cdot f(m,n,F)$, which captures the time complexity of Bender *et al.*'s algorithm for the final values m, n, and for the choice of the parameter F, where F itself depends on m and n.

This completes our informal analysis of Bender *et al.*'s original algorithm.

In our modified algorithm, in contrast, F is not a constant. Instead, each edge insertion operation has its own value of F: indeed, we let F stand for $L(v)$, where v is the source vertex of the edge that is being inserted. We are able to establish the following invariant: for every level k except the highest level, there exist at least k horizontal edges at level k. Thus, we establish that there are enough edges, in a certain sense, at every level. This subsequently allows us to establish a bound on the number of levels: we prove that $L(v)$ is bounded by a quantity that is asymptotically equivalent to Δ (Lemma 8.5.1).

At a high level, rather than ensuring to have Δ edges at each level, we ensure that the first level contains at least one edge, the second level contains at least two edges, and so on... and if the last level is Δ, then the prior-to-last level contains at least $\Delta-1$ edges. Thus, levels contain in average approximately $\Delta/2$ edges. The arguments presented above remain essentially the same, only the details are slightly more complex because of the need to manipulate the sum of the number of edges at every level.

8.4 Implementation

8.4.1 Underlying Graph Interface

Our implementation of the algorithm relies on auxiliary operations whose implementation belongs in a lower layer. We do not prescribe how they should be implemented and what data structures (adjacency lists, hash tables, etc.) they should rely upon; instead, we provide a specification for each of them, and prove that our algorithm is correct, regardless of which implementation choices are made. We provide and verify one concrete implementation, so as to guarantee that our requirements can be met [GJCP19, src/raw_graph.ml, proofs/RawGraph.v].

This lower layer is responsible for implementing a graph data structure, which additionally provides storage for extra information specific to the algorithm. This data structure is represented by the abstract Separation Logic predicate IsRawGraph. In the specifications that form the underlying graph interface, the conjunct IsRawGraph $g\ G\ L\ M\ V\ I$ asserts that there is a data structure at address g in memory and claims the unique ownership of this data structure; the parameters $g\ G\ L\ M\ V\ I$ together form a logical model of this data structure. G is a mathematical graph; L is a map of vertices to integer levels; M is a map of vertices to integer marks; V is a set of valid marks; and I is a map of vertices to sets of vertices, describing horizontal incoming edges. Note that at this level, we require no specific invariant to hold about L, M or I: the data structure is simply responsible for associating some data with vertices of the graph.

We now describe the specifications that form the data structure interface. There are many functions, but they all enjoy straightforward specifications (and implementation). Recall as well that we do provide one possible implementation that we prove correct against this interface, so the reader can trust that this set of specifications can indeed be implemented. Each function is required to have constant time complexity.

Figure 8.3 shows the specifications for the subset of operations that operate on the whole "raw graph" structure, as well as standard graph operations related to the logical graph G. A new structure can be freshly allocated using the `init_graph` function; the associated logical graph is the empty graph $\varnothing$ (InitRawGraph), levels are initially all set to one, incoming sets are all empty. Nothing particular can be assumed about the initial marks and set of valid marks. Symmetrically, an IsRawGraph structure can be freely discarded, as per rule DisposeRawGraph. To update the graph, the algorithm requires the ability to create new vertices and new edges. The `new_vertex` function allocates a fresh vertex (NewVertex). As per the low-level specification RawAddVertex, inserting the vertex into the graph structure simply consumes the assertion IsNewVertex x. Similarly, the function `add_edge` is used to insert an edge into the graph, provided that is not already present (RawAddEdge). To avoid creating duplicate edges, the algorithm must be able to test the equality of two vertices, using function `vertex_eq` (VertexEq). Finally, the `get_outgoing` function can be used to traverse the graph: given a vertex x, it returns the list ys of successors of x in the graph; that is, the list of vertices y such that there is an edge from x to y. Its specification (GetOutgoing) relies on the auxiliary predicate "noduplicates" that asserts that a list does not contain duplicates, and the "to_set" function which turns a list into the set of its elements.

InitRawGraph
$\exists k.\ \{\$k\}\ \texttt{init\_graph}\ ()\ \{\lambda g.\ \exists M\ V.\ \text{IsRawGraph}\ g\ \varnothing\ (\lambda\_.\ 1)\ M\ V\ (\lambda\_.\ \emptyset)\}$

DisposeRawGraph
$\forall g\ G\ L\ M\ V\ I.\ \text{IsRawGraph}\ g\ G\ L\ M\ V\ I\ \Vdash\ \text{GC}$

NewVertex
$\exists k.\ \{\$k\}\ \texttt{new\_vertex}\ ()\ \{\lambda x.\ \text{IsNewVertex}\ x\}$

RawAddVertex
$\exists k.\ \forall g\ G\ L\ M\ V\ I\ x.$
$\{\ \$k \star \text{IsRawGraph}\ g\ G\ L\ M\ V\ I \star \text{IsNewVertex}\ x\ \}$
$(\texttt{add\_vertex}\ g\ x)$
$\{\ \lambda().\ \text{IsRawGraph}\ g\ (G + x)\ L\ M\ V\ I\ \}$

RawAddEdge
$\exists k.\ \forall g\ G\ L\ M\ V\ I\ x\ y.$
$x \in \text{vertices}\ G\ \wedge\ y \in \text{vertices}\ G\ \wedge\ (x, y) \notin \text{edges}\ \implies$
$\{\ \$k \star \text{IsRawGraph}\ g\ G\ L\ M\ V\ I\ \}$
$(\texttt{add\_edge}\ g\ x\ y)$
$\{\ \lambda().\ \text{IsRawGraph}\ g\ (G + (x, y))\ L\ M\ V\ I\ \}$

VertexEq
$\exists k.\ \forall x\ y.\ \{\$k\}\ \texttt{vertex\_eq}\ x\ y\ \{\lambda b.\ [b = \mathit{true}\ \iff\ x = y]\}$

GetOutgoing
$\exists k.\ \forall g\ G\ L\ M\ V\ I\ x.\ x \in \text{vertices}\ G\ \implies$
$\{\ \$k \star \text{IsRawGraph}\ g\ G\ L\ M\ V\ I\ \}$
$(\texttt{get\_outgoing}\ g\ x)$
$\left\{\begin{array}{l} \lambda ys.\ \text{IsRawGraph}\ g\ G\ L\ M\ V\ I \star \\ \quad [\text{noduplicates}(ys)\ \wedge\ \text{to\_set}(ys) = \{y \mid x \longrightarrow_G y\}] \end{array}\right\}$

Figure 8.3: Underlying graph interface: common graph operations

GetLevel
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x.\ x \in \text{vertices}\ G \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
(`get_level` $g\ x$)
$\{\ \lambda l.\ \text{IsRawGraph}\, g\, G\, L\, M\, V\, I \star [l = L(x)]\ \}$

SetLevel
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x\, l.\ x \in \text{vertices}\ G \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
(`set_level` $g\ x\ l$)
$\{\ \lambda().\ \text{IsRawGraph}\, g\, G\, (L[x := l])\, M\, V\, I\ \}$

Figure 8.4: Underlying graph interface: operations on levels

Figure 8.4 shows the specifications for the operations related to levels. To each vertex in the graph is associated an integer level, as described by the parameter L, a map from vertices to integers. One can obtain the current level of a node using the `get_level` function (GetLevel), and modify the level stored for a node by using the `set_level` function (SetLevel).

Figure 8.5 displays specifications for the operations related to the sets of "incoming edges". The parameter I associates a set of vertices $I(x)$ to each vertex x. (The fact that this represents incoming edges is only enforced at an upper level.) The functions of the interface allow reading this set using `get_incoming` (GetIncoming), resetting it to the empty set using `clear_incoming` (ClearIncoming), and inserting a vertex using `add_incoming` (AddIncoming).

Finally, Figure 8.6 shows specifications related to marks. Marks are useful when traversing the graph, in order to denote vertices that have already been visited. The IsRawGraph structure provides support for associating an integer mark to each vertex, and generating fresh marks. This is perhaps a bit surprising: one might expect instead each vertex to be either marked or not, with functions to mark and un-mark a given vertex. Instead, we avoid the need to un-mark vertices by using a slightly more involved interface, where we have several integer marks. When starting a new traversal, one generates a new mark, that is then known to be distinct from all marks used previously.[1] At the logical level, the parameter M associates each vertex to its current mark, and the parameter V denotes the set of *valid marks*, that is, marks that we know have been created previously. The `new_mark` function generates a new mark. The mark is added to the set of valid marks, and we get in the postcondition that this new mark is distinct from every other mark stored in the graph (NewMark). One can update the current mark of a vertex—provided that it is a valid mark—using `set_mark` (SetMark). Finally, one

[1] In our setting, we model program integers as being unbounded: it is always possible to generate a fresh mark. In presence of bounded integers, one could carry a "proof of work" argument using Time Receipts [MJP19]: running out of marks would require running the algorithm for an unrealistic amount of time. Additionally, in practice one would also run out of memory before exhausting all marks, since the algorithm allocates memory (for inserting a new edge) after allocating a new mark.

GetIncoming
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x.\ x \in \text{vertices}\ G \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
(`get_incoming` $g\ x$)
$\{\ \lambda xs.\ \text{IsRawGraph}\, g\, G\, L\, M\, V\, I \star [\text{noduplicates}(xs) \wedge \text{to\_set}(xs) = I(x)]\}$

ClearIncoming
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x.\ x \in \text{vertices}\ G \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
(`clear_incoming` $g\ x$)
$\{\ \lambda xs.\ \text{IsRawGraph}\, g\, G\, L\, M\, V\, (I[x := \emptyset])\ \}$

AddIncoming
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x\, y.$
$x \in \text{vertices}\ G \wedge y \in \text{vertices}\ G \wedge y \notin I(x) \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
(`add_incoming` $g\ x\ y$)
$\{\ \lambda().\ \text{IsRawGraph}\, g\, G\, L\, M\, V\, (I[x := I(x) \cup \{y\}])\ \}$

Figure 8.5: Underlying graph interface: operations on sets of incoming edges

can test whether a vertex is marked with a given mark, using the `is_marked` function (IsMarked).

As a technical remark, note that GetOutgoing and GetIncoming require both `get_outgoing` and `get_incoming` to return a list, in constant time. This is a strong requirement. In practice, the algorithm only needs to be able to iterate on the contents of the list: returning an iterator (e.g., implemented as a lazy list) would be enough. We leave this improvement for future work.

8.4.2 Algorithm

Our OCaml code appears in Figure 8.7. It relies on the functions of the underlying graph interface (§8.4.1), as well as the `interruptible_fold` function introduced earlier in the Examples chapter (§7.6).

The main public entry point of the algorithm is `add_edge_or_detect_cycle`, whose specification was presented in Figure 8.1. The other two public functions, `init_graph` and `add_vertex`, come from the "raw graph" interface: Figure 8.1 equips them with high-level specifications, established on top of their lower level specifications InitRawGraph and RawAddVertex.

The backward search is composed of the functions `visit_backward` (which implements the graph traversal itself) and `backward_search` (which is a wrapper on top of `visit_backward`). The backward search uses `get_incoming` to efficiently enumerate the

NewMark
$\exists k.\ \forall g\, G\, L\, M\, V\, I.$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
$(\texttt{new\_mark}\ g)$
$\{\ \lambda m.\ \text{IsRawGraph}\, g\, G\, L\, M\, (V \cup \{m\})\, I \star [\forall n.\ M(n) \neq m]\}$

SetMark
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x\, m.\ x \in \text{vertices}\ G \wedge m \in V \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
$(\texttt{set\_mark}\ g\ x\ m)$
$\{\ \lambda().\ \text{IsRawGraph}\, g\, G\, L\, (M[x := m])\, V\, I\ \}$

IsMarked
$\exists k.\ \forall g\, G\, L\, M\, V\, I\, x\, m.\ x \in \text{vertices}\ G \implies$
$\{\ \$k \star \text{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \}$
$(\texttt{is\_marked}\ g\ x\ m)$
$\{\ \lambda b.\ \text{IsRawGraph}\, g\, G\, L\, M\, V\, I \star [b = \mathit{true} \iff M(x) = m]\}$

Figure 8.6: Raw graph interface: operations on marks

horizontal incoming edges of a vertex. It also relies on marks to test whether a vertex has already been visited. These marks are consulted also during the forward search.

The forward search consists of the functions `visit_forward` and `forward_search`. It uses `get_outgoing` to efficiently enumerate the outgoing edges of a vertex. The collection of horizontal incoming edges of a vertex y is also updated during a forward search. It is reset when the level of y is increased (`clear_incoming`), and an edge is added to it when a horizontal edge from x to y is traversed (`add_incoming`). The forward search does not need marks. The manner in which it increases levels and traverses only vertices of lower levels is sufficient to ensure that a vertex is processed at most once. However, the forward search does check for marks left by the previous backward search. If it finds such a mark, then a cycle has been detected. Technically, reading these marks during the forward search is optional: to detect a cycle, it would be sufficient to check whether the vertex v is reached. This is a tradeoff. On the one hand, reading these marks can help detect cycles sooner; on the other hand, these read operations have a cost.

Several choices arise in the implementation of graph search. First, a graph search maintains a frontier—a set of nodes that have been discovered but not yet visited—that can be either implicit, if the search is formulated as a recursive function, or represented as an explicit data structure. We choose the latter approach, as it lends itself better to the implementation of an interruptible search. Second, one must choose between an imperative style, where the frontier is represented as a mutable data structure and the code is structured in terms of "while" loops and "break" and "continue" instructions, and a functional style, where the frontier is an immutable data structure and the code is organized in terms of tail-recursive functions or higher-order loop combinators. Because

```
let rec visit_backward g target mark fuel stack =
  match stack with
  | [] -> VisitBackwardCompleted
  | x :: stack ->
    let res = interruptible_fold (fun y (stack, fuel) ->
      if fuel = 0 then Break true
      else if is_marked g y mark then Continue (stack, fuel - 1)
      else if vertex_eq y target then Break false
      else (set_mark g y mark; Continue (y :: stack, fuel - 1))
    ) (get_incoming g x) (stack, fuel) in
    match res with
    | Break timeout ->
      if timeout then VisitBackwardInterrupted else VisitBackwardCyclic
    | Continue (stack, fuel) ->
      visit_backward g target mark fuel stack

let backward_search g v w fuel =
  let mark = new_mark g in
  let v_level = get_level g v in
  set_mark g v mark;
  match visit_backward g w mark fuel [v] with
  | VisitBackwardCyclic -> BackwardCyclic
  | VisitBackwardInterrupted -> BackwardForward (v_level + 1, mark)
  | VisitBackwardCompleted -> if get_level g w = v_level
                              then BackwardAcyclic
                              else BackwardForward (v_level, mark)

let rec visit_forward g new_level mark stack =
  match stack with
  | [] -> ForwardCompleted
  | x :: stack ->
    let res = interruptible_fold (fun y stack ->
      if is_marked g y mark then Break ()
      else
        let y_level = get_level g y in
        if y_level < new_level then begin
          set_level g y new_level;
          clear_incoming g y;
          add_incoming g y x;
          Continue (y :: stack)
        end else if y_level = new_level then begin
          add_incoming g y x;
          Continue stack
        end else Continue stack
    ) (get_outgoing g x) stack in
    match res with
    | Break () -> ForwardCyclic
    | Continue stack -> visit_forward g new_level mark stack

let forward_search g w new_w_level mark =
  clear_incoming g w;
  set_level g w new_w_level;
  visit_forward g new_w_level mark [w]

let add_edge_or_detect_cycle (g : graph) (v : vertex) (w : vertex) =
  let succeed () =
    add_edge g v w;
    if get_level g v = get_level g w then
      add_incoming g w v;
    Ok
  in
  if vertex_eq v w then Cycle
  else if get_level g w > get_level g v then succeed ()
  else match backward_search g v w (get_level g v) with
    | BackwardCyclic -> Cycle
    | BackwardAcyclic -> succeed ()
    | BackwardForward (new_level, mark) ->
        match forward_search g w new_level mark with
        | ForwardCyclic -> Cycle
        | ForwardCompleted -> succeed ()
```

Figure 8.7: OCaml implementation of the verified incremental cycle detection algorithm.

OCaml does not have "break" and "continue", we choose the latter style.

The function `visit_backward`, for instance, can be thought of as two nested loops. The outer loop is encoded via a tail call to `visit_backward` itself. This loop runs until the stack is exhausted or the inner loop is interrupted. The inner loop is implemented via the loop combinator `interruptible_fold`, a functional-style encoding of a "for" loop whose body may choose between interrupting the loop (`Break`) and continuing (`Continue`). This inner loop iterates over the horizontal incoming edges of the vertex x. It is interrupted when a cycle is detected or when the variable `fuel`, whose initial value corresponds to F (§8.2), reaches zero. The function `visit_forward` has similar structure.

8.5 Data Structure Invariants

As explained earlier (§8.1), the specification of the algorithm refers to two variables, IsGraph and ψ, which must be regarded as abstract by a client. Figure 8.8 gives their formal definitions. The assertion IsGraph g G captures both the invariants required for functional correctness and those required for the complexity analysis. It is a conjunction of three conjuncts, which we describe in turn.

8.5.1 Low-level Data Structure

The conjunct IsRawGraph g G L M V I asserts the unique ownership of the underlying graph data structure, described in Section 8.4.1. As explained earlier (§8.4.1), the choice of a low-level data structure and the implementation of the required operations on this data structure are left to the client. The client is therefore also in charge of providing a suitable definition of the predicate IsRawGraph and of verifying that the low-level operations meet their specifications, which involve IsRawGraph. We do provide one possible implementation; this guarantees that our requirements can be met.

8.5.2 Functional Invariant

The second conjunct, [Inv G L I], is a pure proposition that relates the graph G with the maps L and I.

Indeed, the assertion IsRawGraph g G L M V I alone implies no connection between G, L, V, and I. The low-level graph data structure can perfectly well carry arbitrary integer levels and arbitrary sets of incoming vertices. Separating the description of the data structure (IsRawGraph) and the functional invariant (Inv) allows us to temporarily break the invariant while keeping access to the data structure. For instance, during a forward search, the invariant is temporarily broken while levels and sets of incoming edges are being updated. Still, the algorithm is allowed to operate on the data structure, because the assertion IsRawGraph g G L M V I remains available, for certain values of G, L, M, V, I.

The definition of Inv G L I appears next in Figure 8.8. Anticipating on the fact that we sometimes need a relaxed invariant, we actually define a more general predicate InvExcept E G L I, where E is a set of "exceptions", that is, a set of vertices where certain properties are allowed *not* to hold. In particular, InvExcept appears in the

$$\text{IsGraph}\, g\, G := \exists L\, M\, V\, I.\ \text{IsRawGraph}\, g\, G\, L\, M\, V\, I \star [\text{Inv}\, G\, L\, I] \star \$\phi(G, L)$$

$$\text{Inv}\, G\, L\, I := \text{InvExcept}\, \emptyset\, G\, L\, I$$

$$\text{InvExcept}\, E\, G\, L\, I := \begin{cases} \textit{acyclicity:} & \forall x.\quad x \not\longrightarrow_G^+ x \\ \textit{positive levels:} & \forall x.\quad L(x) \geq 1 \\ \textit{pseudo-topological numbering:} & \forall x\, y.\ x \longrightarrow_G y \implies L(x) \leq L(y) \\ \textit{horizontal incoming edges:} & \forall x\, y.\ x \in I(y) \iff x \longrightarrow_G y \ \wedge\ L(x) = L(y) \\ \textit{replete levels:} & \forall x.\quad x \in E \vee \text{enoughEdgesBelow}\, G\, L\, x \end{cases}$$

$$\text{enoughEdgesBelow}\, G\, L\, x := |\text{coaccEdgesAtLevel}\, G\, L\, k\, x| \geq k \quad \text{where } k = L(x) - 1$$

$$\text{coaccEdgesAtLevel}\, G\, L\, k\, x := \{\, (y, z) \mid y \longrightarrow_G z \longrightarrow_G^* x \ \wedge\ L(y) = L(z) = k \,\}$$

$$\left.\begin{array}{lcl} \phi(G, L) & := & C \cdot (\text{net}\, G\, L) \\ \text{net}\, G\, L & := & \text{received}\, m\, n - \text{spent}\, G\, L \end{array}\right\} \text{where } m = |\text{edges}\, G| \text{ and } n = |\text{vertices}\, G|$$

$$\text{spent}\, G\, L := \sum_{(u,v)\, \in\, \text{edges}\, G} L(u)$$

$$\text{received}\, m\, n := m \cdot (\text{max\_level}\, m\, n + 1)$$

$$\text{max\_level}\, m\, n := \min(\lceil (2m)^{1/2} \rceil, \lfloor (\tfrac{3}{2} n)^{2/3} \rfloor) + 1$$

$$\psi(m, n) := C' \cdot (\text{received}\, m\, n + m + n)$$

Figure 8.8: Definitions of IsGraph and ψ, with auxiliary definitions.

specification of `forward_search` (Figure 8.11). Instantiating E with the empty set $\emptyset$ yields Inv $G\ L\ I$.

The proposition InvExcept $E\ G\ L\ I$ is a conjunction of five properties. The first four capture functional correctness invariants: the graph G is acyclic, every vertex has positive level, L forms a pseudo-topological numbering of G, and the sets of horizontal incoming edges represented by I are accurate with respect to G and L. The last property plays a crucial role in the complexity analysis (§8.3). It asserts that "levels are replete", that is, "every vertex has enough coaccessible edges at the previous level": for every vertex x at level $k + 1$, there must be at least k horizontal edges at level k from which x is accessible. This implies that there are at least k edges at every level k except the highest level. This fact was mentioned in our informal overview of the algorithm (§8.3). The vertices in the set E may disobey this property, which is temporarily broken during a forward search.

8.5.3 Potential

The last conjunct in the definition of IsGraph is $\$\phi(G, L)$. This is a *potential* [Tar85], a number of credits that have been received from the user (through calls to `add_vertex` and `add_edge_or_detect_cycle`) and not yet spent. Instead, these credits have been saved and are now stored inside the data structure, so to speak. As usual in an amortized complexity analysis, the role of the potential is to help pay for costly operations in the future. $\phi(G, L)$ is defined as $C \cdot (\text{net}\ G\ L)$. The constant C is derived from the code; its

exact value is in principle known, but irrelevant, so we refer to it only by name. The quantity "net G L" is defined as the difference between "received m n", an amount that has been received, and "spent G L", an amount that has been spent[2]. "net G L" can also be understood as a sum over all edges of a per-edge amount, which for each edge (u, v) is "max_level m $n - L(u)$". This is a difference between "max_level m n", which one can prove is an upper bound on the current level of every vertex, and $L(u)$, the current level of the vertex u. This difference can be intuitively understood as the number of times the edge (u, v) might be traversed in the future by a forward search, due to a promotion of its source vertex u.

A key fact is that "max_level m n", a function of m and n, is indeed an upper bound on the level of a vertex after m edge additions and n vertex additions.

Lemma 8.5.1 (max_level **is a bound on levels)**

$$\text{Inv } G\ L\ I \implies \forall x \in \text{vertices } G.\ L(x) \leq \text{max\_level } |\text{edges } G|\ |\text{vertices } G|$$

Proof Lemma `Inv_max_level` of [GJCP19, proofs/GraphCostInvariants.v].

Corollary 8.5.2 (net **holds at least one potential per edge.)**

$$\text{Inv } G\ L\ I \implies |\text{edges } G| \leq \text{net } G\ L$$

Proof Lemma `Inv_net_lowerbound` of [GJCP19, proofs/GraphCostInvariants.v].

From Lemma 8.5.1, it follows that the per-edge amount "max_level m $n - L(u)$" is nonnegative. Therefore, the global potential net G L (and $\phi(G, L)$) is nonnegative as well, as per Corollary 8.5.2. As mentioned earlier (§8.1), the latter property is required in the proof of DisposeGraph.

8.5.4 Advertised Cost

We have reviewed the three conjuncts that form IsGraph g G. There remains to define ψ, which also appears in the public specification (Figure 8.1). Recall that $\psi(m, n)$ denotes the number of credits that we request from the user during a sequence of m edge additions and n vertex additions. Up to another known-but-irrelevant constant factor C', it is defined as "$m + n +$ received m n", that is, a constant amount per operation plus a sufficient amount to justify that $\phi(m, n)$ credits are at hand, as claimed by the invariant IsGraph g G. It is easy to check, by inspection of the last few definitions in Figure 8.8, that Complexity is satisfied, that is, $\psi(m, n)$ is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

The public function `add_edge_or_detect_cycle` expects a graph g and two vertices v and w. Its public specification has been presented earlier (Figure 8.1). The top part of Figure 8.9 shows the same specification, where IsGraph, ϕ (01) and ψ (02) have been unfolded. This shows that we receive time credits from two different sources: the potential of the data structure, on the one hand, and the credits supplied by the user for this operation, on the other hand.

$$\forall g\ G\ L\ M\ V\ I\ v\ w.\ \text{let } m := |\text{edges } G| \text{ and } n := |\text{vertices } G| \text{ in}$$
$$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$$
$$\left\{ \begin{array}{ll} \text{IsRawGraph } g\ G\ L\ M\ V\ I \star [\text{Inv } G\ L\ I] \star \$(C \cdot \text{net } G\ L) \star & (01) \\ \$(C' \cdot (\text{received } (m+1)\ n - \text{received } m\ n + 1)) & (02) \end{array} \right\}$$
$$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$$
$$\left\{ \begin{array}{l} \lambda res.\ \text{match } res \text{ with} \\ \quad \mid \text{Ok} \Rightarrow \ \text{let } G' := G + (v, w) \text{ in } \exists L'\ M'\ V'\ I'. \\ \qquad\qquad \text{IsRawGraph } g\ G'\ L'\ M'\ V'\ I' \star [\text{Inv } G'\ L'\ I'] \star \$(C \cdot \text{net } G'\ L') \\ \quad \mid \text{Cycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}$$

Figure 8.9: Specification for edge creation, after unfolding of the representation predicate.

8.6 Specifications for the Algorithm's Main Functions

The specifications of the two search functions, `backward_search` and `forward_search`, appear in Figures 8.10 and 8.11 respectively. They capture the algorithm's key internal invariants and spell out exactly what each search achieves and how its cost is accounted for.

The function `backward_search` expects a nonnegative integer *fuel*, which represents the maximum number of edges that the backward search is allowed to process. In addition, it expects a graph g and two distinct vertices v and w which must satisfy $L(w) \leq L(v)$. (If that is not the case, an edge from v to w can be inserted immediately.) The graph must be in a valid state (03). The specification requires $A \cdot \mathit{fuel} + B$ credits to be provided (04), for some constants A and B. The definition of A and B is known during the proof of the specification, but irrelevant for clients of the specifications: A and B are therefore quantified existentially. The cost of a backward search is linear in the number of edges that are processed, therefore linear in *fuel*.

This function returns either BackwardCyclic, BackwardAcyclic, or a value of the form BackwardForward(k, *mark*). The first line in the postcondition (05) asserts that the graph remains valid and changes only in that some marks are updated: M changes to M'.

The remainder of the postcondition depends on the function's return value, *res*. If it is BackwardCyclic, then there exists a path in G from w to v (06). If it is BackwardAcyclic, then v and w are at the same level and there is no path from w to v (07). In this case, no forward search is needed. If it is of the form BackwardForward(k, *mark*), then a forward search is required.

In the latter case, the integer k is the level to which the vertex w and its descendants should be promoted during the subsequent forward search. The value *mark* is the mark that was used by this backward search; the subsequent forward search uses this mark to recognize vertices reached by the backward search. The postcondition asserts that the vertex v is marked, whereas w is not (08), since it has not been reached. Moreover,

[2]These amounts are not expressed in credits, but in packets of C credits.

$\exists A\, B.\ A \geq 0\ \wedge\ \forall \mathit{fuel}\, g\, G\, L\, M\, V\, I\, v\, w.$
$\mathit{fuel} \geq 0\ \wedge\ v, w \in \text{vertices}\ G\ \wedge\ v \neq w\ \wedge\ L(w) \leq L(v)\ \Longrightarrow$

$$\left\{\begin{array}{ll} \text{IsRawGraph}\, g\, G\, L\, M\, V\, I \star [\text{Inv}\, G\, L\, I] \star & (03) \\ \$(A \cdot \mathit{fuel} + B) & (04) \end{array}\right\}$$

(`backward_search` $g\ v\ w\ \mathit{fuel}$)

$$\left\{\begin{array}{ll} \lambda \mathit{res}. \exists M'\, V'. & \\ \quad \text{IsRawGraph}\, g\, G\, L\, M'\, V'\, I \star [\text{Inv}\, G\, L\, I] \star & (05) \\ \quad [\text{match}\ \mathit{res}\ \text{with} & \\ \quad \mid \text{BackwardCyclic} \Rightarrow w \longrightarrow_G^* v & (06) \\ \quad \mid \text{BackwardAcyclic} \Rightarrow L(v) = L(w)\ \wedge\ w \not\longrightarrow_G^* v & (07) \\ \quad \mid \text{BackwardForward}(k, \mathit{mark}) \Rightarrow & \\ \qquad M'\, v = \mathit{mark}\ \wedge\ M'\, w \neq \mathit{mark}\ \wedge & (08) \\ \qquad (\forall x.\, M'\, x = \mathit{mark}\ \Longrightarrow\ L(x) = L(v)\ \wedge\ x \longrightarrow_G^* v)\ \wedge & (09) \\ \qquad (\quad (k = L(v)\ \wedge\ L(w) < L(v)\ \wedge & (10) \\ \qquad\quad \forall x.\, L(x) = L(v)\ \wedge\ x \longrightarrow_G^* v\ \Longrightarrow\ M'\, x = \mathit{mark}) & (11) \\ \qquad \vee\ (k = L(v) + 1\ \wedge\ \mathit{fuel} \leq |\text{coaccEdgesAtLevel}\, G\, L\, (L(v))\, v|))] & (12) \end{array}\right\}$$

Figure 8.10: Specification for the main backward search function.

$\exists B'.\ B' \geq 0\ \wedge\ \forall g\, G\, L\, M\, V\, I\, w\, k\, \mathit{mark}.$
$w \in \text{vertices}\ G\ \wedge\ L(w) < k\ \wedge\ M\, w \neq \mathit{mark}\ \Longrightarrow$

$$\left\{\begin{array}{ll} \text{IsRawGraph}\, g\, G\, L\, M\, V\, I \star [\text{Inv}\, G\, L\, I] \star & (13) \\ \$(B' + \phi(G, L)) & (14) \end{array}\right\}$$

(`forward_search` $g\ w\ k\ \mathit{mark}$)

$$\left\{\begin{array}{ll} \lambda \mathit{res}. \exists L'\, I'. & \\ \quad \text{IsRawGraph}\, g\, G\, L'\, M\, V\, I' \star & (15) \\ \quad [L'(w) = k\ \wedge\ (\forall x.\, L'(x) = L(x)\ \vee\ w \longrightarrow_G^* x)] \star & (16) \\ \quad \text{match}\ \mathit{res}\ \text{with} & \\ \quad \mid \text{ForwardCyclic} \Rightarrow [\exists x.\, M\, x = \mathit{mark}\ \wedge\ w \longrightarrow_G^* x] & (17) \\ \quad \mid \text{ForwardCompleted} \Rightarrow & \\ \qquad \$\phi(G, L') \star & (18) \\ \qquad [(\forall x\, y.\, L(x) < k\ \wedge\ w \longrightarrow_G^* x \longrightarrow_G y\ \Longrightarrow\ M\, y \neq \mathit{mark})\ \wedge & (19) \\ \qquad\ \text{InvExcept}\ \{x \mid w \longrightarrow_G^* x\ \wedge\ L'(x) = k\}\ G\ L'\ I'] & (20) \end{array}\right\}$$

Figure 8.11: Specification for the main forward search functions.

every marked vertex lies at the same level as v and is an ancestor of v (09) (indeed, only vertices satisfying these two properties are discovered by the backward search). Finally, one of the following two cases holds. In the first case, w must be promoted to the level of v and currently lies below the level of v (10) and the backward search is complete, that is, every ancestor of v that lies at the level of v is marked (11). In the second case, w must be promoted to level $L(v) + 1$ and there exist at least *fuel* horizontal edges at the level of v from which v can be reached (12).

The function `forward_search` expects the graph g, the target vertex w, the level k to which w and its descendants should be promoted, and the mark *mark* used by the backward search. The vertex w must be at a level less than k and must be unmarked. The graph must be in a valid state (13). The forward search requires a constant amount of credits B'. Furthermore, it requires access to the potential $\$\phi(G, L)$, which is used to pay for edge processing costs.

This function returns either ForwardCyclic or ForwardCompleted. It affects the low-level graph data structure by updating certain levels and certain sets of horizontal incoming edges: L and I are changed to L' and I' (15). The vertex w is promoted to level k, and a vertex x can be promoted only if it is a descendant of w (16).

If the return value is ForwardCyclic, then, according to the postcondition, there exists a vertex x that is accessible from w and that has been marked by the backward search (17). This implies that there is a path from w through x to v. Thus, adding an edge from v to w would create a cycle. In this case, the data structure invariant is lost.

If the return value is ForwardCompleted, then, according to the postcondition, $\phi(G, L')$ credits are returned (18). This is precisely the potential of the data structure in its new state. Furthermore, two logical propositions hold. First (19), the forward search has not encountered a vertex that was marked during the backward search: for every edge (x, y) that is accessible from w, where x is at level less than k, the vertex y is unmarked. (This implies that there is no path from w to v.) Second (20), the invariant Inv $G\ L'\ I'$ is satisfied *except* for the fact that the property of "replete levels" (Figure 8.8) may be violated at descendants of w whose level is now k. Fortunately, this proposition (20), combined with a few other facts that are known to hold at the end of the forward search, implies Inv $G'\ L'\ I'$, where G' stands for $G + (v, w)$. In other words, at the end of the forward search, all levels and all sets of horizontal incoming edges are consistent with the mathematical graph G', where the edge (v, w) exists. Thus, after this edge is effectively created in memory by the call `add_edge g v w`, all is well: we have both IsRawGraph $g\ G'\ L'\ M'\ V'\ I'$ and Inv $G'\ L'\ I'$, so `add_edge_or_detect_cycle` satisfies its postcondition, under the form shown in Figure 8.9.

8.7 A Taste of the Formal Complexity Proof

In Section 8.3 we present an *informal* complexity analysis of the algorithm of Bender *et al.* In particular, the analysis reasons about a complete sequence of calls to the algorithm, and considers that the value of the initial fuel F is fixed to a constant. Our *formal* complexity analysis is carried out differently: it is based on inductive invariants (Figure 8.8), and corresponds to our variant of the algorithm in which the value of F is

$$
\begin{array}{l}
\exists A_0\, A_1\, B_0.\ A_0 \geq 0 \ \wedge\ A_1 \geq 0 \ \wedge \\
\forall g\, G\, L\, M\, V\, I\ \mathit{source}\ \mathit{target}\ \mathit{visited}\ \mathit{mark}\ \mathit{fuel}\ \mathit{stack}. \\
\left\{ \begin{array}{l} \$(A_0 \cdot \mathit{fuel}\ +\ A_1 \cdot (\mathit{fuel} + |\mathit{stack}|)\ +\ B_0)\ \star \\ \mathrm{IsRawGraph}\, g\, G\, L\, M\, V\, I\ \star \\ [\mathrm{INVB}\ G\, L\, M\, V\, I\ \mathit{source}\ \mathit{target}\ \mathit{visited}\ \mathit{mark}\ \mathit{stack}] \end{array} \right\} \\
(\texttt{visit\_backward}\ g\ \mathit{target}\ \mathit{mark}\ \mathit{fuel}\ \mathit{stack}) \\
\left\{ \begin{array}{l} \lambda \mathit{res}.\ \exists M'.\ \mathrm{IsRawGraph}\, g\, G\, L\, M'\, V\, I\ \star \\ \quad [\mathrm{POSTB}\ G\, L\, M'\, I\ \mathit{source}\ \mathit{target}\ \mathit{visited}\ \mathit{mark}\ \mathit{res}] \end{array} \right\}
\end{array}
\tag{21}
$$

Figure 8.12: Specification for the auxiliary search function `visit_backward`.
The definition of the pure predicates INVB and POSTB is omitted for conciseness.

not a constant but varies at each edge (or vertex) insertion, depending on the size of the graph at that point.

In this section, we present a proof sketch of the complexity analysis as it is actually performed in our mechanized proof. Note that our complexity proof relies on many functional invariants; however, for clarity of the presentation, we will omit most of the (often intricate) functional invariants that are not directly related to the complexity proof. Instead, we will simply mention when the complexity analysis relies on them, without explaining how they have been established (for that we refer the reader to the Coq proof scripts [GJCP19]).

Our formal analysis follows the structure of the implementation: we state and prove a specification involving time credits for each function of the implementation. There are five such functions: the toplevel function `add_edge_or_detect_cycle` (whose specification appears in Figure 8.9), the search functions `backward_search` (Figure 8.10) and `forward_search` (Figure 8.11) , and the auxiliary functions `visit_backward` and `visit_forward`, who actually implement the search by traversing the graph. Their specifications are introduced next.

The `visit_backward` function is responsible for the bulk of the backward search, the first phase of the algorithm. Its specification appears in Figure 8.12. A call to the function searches the graph g for the vertex *target*, within the same level as the source vertex *source*. The traversal is bounded by the *fuel* parameter (a nonnegative integer), which represents the maximum number of edges that the search is allowed to traverse. Vertices that are visited by the search are marked using *mark*. The *stack* parameter represents the "frontier" of the search: a list of vertices from which the search can be carried on.

The implementation of `visit_backward` (Figure 8.7) is a tail-recursive function. Each recursive call pops a vertex x from the stack and iterates over its neighbors, inserting them into the stack if they have not been visited already. Each time a neighbor vertex is visited, the fuel is decremented. If the target vertex is found, or if the fuel reaches zero, the search stops. Otherwise, `visit_backward` is called recursively with the remaining fuel and the updated stack.

The specification requires $A_0 \cdot \mathit{fuel} + A_1 \cdot (\mathit{fuel} + |\mathit{stack}|) + B_0$ credits to be provided (21), for some constants A_0, A_1 and B_0. First, the cost of the search is linear in the number of edges that are visited, i.e. linear in *fuel*. Additionally, the length of the stack $|\mathit{stack}|$ must be also taken into account. Indeed, it may happen that *stack* contains vertices with no outgoing edges. In that case, the implementation spends time removing these vertices from the stack, but does not decrement *fuel*, as no new edge is visited. To pay for time spent iterating on vertices of the stack that have no neighbors, we additionally require an amount of credits proportional to $\mathit{fuel} + |\mathit{stack}|$. Subtly enough, it would not be correct to require an amount of credits proportional to $|\mathit{stack}|$, as the length of the stack can increase. What makes $\mathit{fuel} + |\mathit{stack}|$ work is that every time a vertex is added to the stack, the fuel is decremented.

The proof of `visit_backward` goes as follows (focusing on complexity):

1. Using the mechanism described in Section 6.5 that relies on the `procrastination` library, we introduce two abstract integer constants K_{step} and K_{stop}, about which we will accumulate a set of constraints. Then, we introduce constants for A_0, A_1 and B_0. K_{step} and K_{stop} will be used to express the invariant of `interruptible_fold`: K_{step} represents the cost of one iteration of `interruptible_fold`, and K_{stop} the cost of stopping the iteration.

2. Let us consider the second branch of the toplevel match (line 4), where the stack argument is equal to $x :: \mathit{stack}$. We use the specification for `interruptible_fold` displayed in Figure 7.3, where the client has to include credits as part of an invariant (`Inv`), and use them to pay as it walks through the list. We instantiate the invariant `Inv` in `interruptible_fold`'s specification with an expression of the following form:

$$
\begin{array}{l}
\lambda ys\ \mathit{step}.\ \exists M'.\ \mathrm{IsRawGraph}\ g\ G\ L\ M'\ V\ I\ \star \\
\quad \text{match } \mathit{step} \text{ with} \\
\quad \mid \text{Break } b \Rightarrow [\ldots] \\
\quad \mid \text{Continue } (\mathit{stack}', \mathit{fuel}') \Rightarrow \\
\qquad \$(K_{step} \cdot \mathit{fuel}' + K_{stop})\ \star \\
\qquad [\, 0 \leq \mathit{fuel}' \leq \mathit{fuel} \wedge \\
\qquad\ \mathit{fuel}' + |\mathit{stack}'| \leq \mathit{fuel} + |\mathit{stack}| \ \wedge\ \ldots].
\end{array}
$$

   This is to be read as a loop invariant that indicates how many credits we have at hand at each iteration, namely $K_{step} \cdot \mathit{fuel}' + K_{stop}$. This quantity does not depend on the height of the stack, as fuel' is decremented at each loop iteration, and the loop is interrupted as soon as fuel' reaches zero, regardless of the state of the stack.

3. Stepping through the body of `interruptible_fold`'s client function yields some constraints on K_{step} and K_{stop} that we defer:

$$
\begin{array}{ll}
 & 1 + \texttt{cost interruptible\_fold}\ () \leq K_{stop} \\
\wedge & 1 + \texttt{cost is\_marked}\ () + \texttt{cost interruptible\_fold}\ () \leq K_{step} \\
\wedge & \ldots
\end{array}
$$

4. Let us now consider the cost of the whole function, focusing on the case where a recursive call is performed. Let us name fuel' and stack' the value of the final

fuel and the stack returned by `interruptible_fold` (in the `Continue` case). By stepping through the code of `visit_backward`, we synthesize the following cost for its body, which we must show is less than the initial number of credits (again, we focus on the branch of the program that performs the recursive call):

$$
\begin{array}{lll}
 & 1 & \text{(entering the function)} \\
+ & \texttt{cost get\_incoming}\ () & \text{(call to } \texttt{get\_incoming}\text{)} \\
+ & \texttt{cost interruptible\_fold}\ () & \text{(call to } \texttt{interruptible\_fold}\text{)} \\
+ & K_{step} \cdot \mathit{fuel} + K_{stop} & \text{(invariant, entering the loop)} \\
- & K_{step} \cdot \mathit{fuel}' + K_{stop} & \text{(invariant, leaving the loop)} \\
+ & A_0 \cdot \mathit{fuel}' + A_1 \cdot (\mathit{fuel}' + |\mathit{stack}'|) + B_0 & \text{(recursive call)} \\
\leq & & \\
 & A_0 \cdot \mathit{fuel} + A_1 \cdot (\mathit{fuel} + |x :: \mathit{stack}|) + B_0 & \text{(credits in the precondition)}
\end{array}
$$

Remark that the number of credits stored in the invariant contributes *negatively* to the overall cost when leaving the loop: indeed, these are credits that we *get back* from the specification of `interruptible_fold`.

5. In the cost inequality above, the values fuel' and stack' are local to the proof: we must eliminate them. The invariant provided at step (2) yields $\mathit{fuel}' \leq \mathit{fuel}$ and $\mathit{fuel}' + |\mathit{stack}'| \leq \mathit{fuel} + |\mathit{stack}|$. Using these two inequalities and by additionally assuming that $K_{step} \leq A_0$ holds, we have:

$$
\begin{array}{r}
(A_0 - K_{step}) \cdot \mathit{fuel}' + A_1 \cdot (\mathit{fuel}' + |\mathit{stack}'|) \\
\leq (A_0 - K_{step}) \cdot \mathit{fuel} \ + A_1 \cdot (\mathit{fuel} + |\mathit{stack}|).
\end{array}
$$

Consequently, fuel' and stack' can be eliminated from the inequality of step (4), which can be reduced to constraints about A_0, A_1, B_0, K_{step} and K_{stop} that we defer:

$$
\begin{array}{ll}
 & K_{step} \leq A_0 \ \wedge \ 1 \leq B_0 \\
\wedge & 1 + \texttt{cost get\_incoming}\ () + \texttt{cost interruptible\_fold}\ () + K_{stop} \leq B_0 \\
\wedge & 1 + \texttt{cost get\_incoming}\ () + \texttt{cost interruptible\_fold}\ () \leq A_1 \\
\wedge & \ldots
\end{array}
$$

6. Using our `elia` procedure (§6.6), we automatically find values for A_0, A_1, B_0, K_{step} and K_{stop} that satisfy the constraints collected at step (3) and (5). This completes the proof.

The `backward_search` function is a relatively short wrapper on top of `visit_backward`. The cost advertised in its specification (Figure 8.10) is of the form $A \cdot \mathit{fuel} + B$. Its complexity analysis can be carried out in a few lines, establishing suitable values for A and B (that depend in particular on A_0, A_1 and B_0 from `visit_backward`'s specification).

The `visit_forward` function is responsible for the second phase of the algorithm, which consists in a forward search (while updating the levels of the nodes visited). Its specification appears in Figure 8.13. A call to `visit_forward` traverses the graph g, using

$$
\begin{array}{l}
\exists B_1\, C_0\,.\; B_1 \geq 0 \;\wedge\; C_0 \geq 0 \;\wedge \\
\forall g\, G\, L_0\, L\, M\, V\, I_0\, I\, \mathit{leftover}\, k\, \mathit{source}\, \mathit{mark}\, \mathit{stack}\, \mathit{visitedV}\, \mathit{visitedE}. \\
\left\{ \begin{array}{ll}
\$(B_1 \;+\; C_0 \cdot \text{stackPot}\; G\; \mathit{stack} \;+\; 2C_0 \cdot \text{net}\; G\; L \;+\; 2C_0 \cdot \mathit{leftover}) \star & (22) \\
\text{IsRawGraph}\, g\; G\, L\, M\, V\, I \star & \\
[\text{INVF}\; G\, L_0\, L\, M\, I_0\, I\, \mathit{leftover}\, k\, \mathit{source}\, \mathit{mark}\, \mathit{stack}\, \mathit{visitedV}\, \mathit{visitedE}] &
\end{array} \right\} \\
(\texttt{visit\_forward}\; g\; k\; \mathit{mark}\; \mathit{stack}) \\
\left\{ \begin{array}{ll}
\lambda \mathit{res}. \exists L'\, I'.\; \text{IsRawGraph}\, g\; G\, L'\, M\, V\, I' \star & \\
\quad [\text{POSTF}\; G\, L_0\, L'\, M\, I'\, k\, \mathit{source}\, \mathit{mark}\, \mathit{res}] \star & \\
\quad \text{match}\; \mathit{res}\; \text{with} & \\
\quad \mid \text{ForwardCyclic} \Rightarrow [\,] & (23) \\
\quad \mid \text{ForwardCompleted} \Rightarrow \$(2C_0 \cdot \text{net}\; G\; L') & (24)
\end{array} \right\}
\end{array}
$$

$$
\text{stackPot}\; G\; \mathit{stack} \;\triangleq\; \sum_{x \in \mathit{stack}} (2 \cdot |\{y \mid x \longrightarrow_G y\}| + 1)
$$

Figure 8.13: Specification for the auxiliary search function `visit_forward`.
The definition of the pure predicates INVF and POSTF is omitted for conciseness. The auxiliary stackPot predicate is used in the amount of credits in the precondition.

stack as the list of vertices (the "frontier") from which to continue the traversal. During the traversal, vertices that are below level k are raised to the level k (k corresponds to the `new_level` parameter in the implementation). The traversal stops on vertices that are already at level k or above. If the search encounters a vertex marked with *mark*, a cycle is reported.

The implementation of `visit_forward` (Figure 8.7) is structured similarly to the implementation of `visit_backward`: the function repeatedly pops a vertex x from the stack and iterates over its neighbors using `interruptible_fold`. For each neighbor vertex y, it is either: marked with *mark*, in which case the traversal stops and reports a cycle; at a level below k, in which case its level is set to k and the vertex is inserted into the stack; at level k or above, in which case it is not inserted into the stack (the traversal stops there and continues from the other vertices of the stack).

We previously mentioned that the overall complexity of the forward traversal is *amortized* constant time, using the potential function ϕ. Indeed, `visit_forward` runs in amortized constant time, relying on a somewhat more complex potential function. The expression of the potential appears in the precondition of the function (22): "$C_0 \cdot \text{stackPot}\; G\; \mathit{stack} + 2C_0 \cdot \text{net}\; G\; L + 2C_0 \cdot \mathit{leftover}$". This amount of credits must be provided initially and at every (tail-recursive) call: it is an invariant of the traversal.

There are three components to this potential. A first part of the potential is associated to the *stack* parameter ("$C_0 \cdot \text{stackPot}\; G\; \mathit{stack}$", where stackPot is defined in Figure 8.13). Each vertex of the stack carries an amount of credits proportional to the number of its neighbors. In other words, for each vertex of the stack, the potential stores enough credits to be able to iterate on all of its neighbors. The second part of

the potential ("$C_0 \cdot \text{net}\ G\ L$") corresponds closely to the potential function ϕ itself. For each edge in the graph, the quantity "net $G\ L$" (Figure 8.8) carries an amount of credits proportional to the difference between its current level and the maximum level. In other words, for each edge of the graph, the potential stores enough credits to be able to pay for raising this edge to an upper level. The third part of the potential ("$C_0 \cdot \mathit{leftover}$") consists in an amount of "leftover credits", that we keep around for technical reasons (more about that later (7)).

If `visit_forward` returns after detecting a cycle (23), no credits are returned: the potential that was initially available has been entirely consumed or disposed of. This corresponds to the fact that the graph is in a state where some of the functional invariants have not been restored. In the case where the traversal completes without detecting a cycle (24), we return the potential available at the end of the traversal. This is simply "$2C_0 \cdot \text{net}\ G\ L'$" (where L' denotes the levels of vertices in the final graph). After completing the traversal, *stack* is known to be empty, and its associated potential to be zero; and we can additionally dispose of the potential associated with the "leftover credits".

The proof of `visit_forward` goes as follows (focusing on complexity):

1. Using `procrastination` (§6.5), we introduce abstract integer constants B_1, C_0 and K about which we will accumulate a set of constraints. Constants B_1 and C_0 correspond to the ones quantified in the specification; K is used in the invariant of `interruptible_fold` and represents the cost of stopping the iteration.

2. Let us consider the second branch of the match, where the stack argument is equal to $x :: \mathit{stack}$ (line 31). In that case, the potential associated to the stack unfolds as follows:

$$C_0 \cdot \text{stackPot}\ G\ (x :: \mathit{stack}) = C_0 + 2C_0 \cdot |\{y \mid x \longrightarrow_G y\}| + C_0 \cdot \text{stackPot}\ G\ \mathit{stack}.$$

3. As with `visit_backward`, we use the specification of `interruptible_fold` displayed in Figure 7.3, instantiating the invariant `Inv` with an expression of the form:

$$\begin{aligned}
&\lambda \mathit{ys}\ \mathit{step}.\ \exists L'\ I'\ \mathit{leftover}.\ \text{IsRawGraph}\ g\ G\ L'\ M\ V\ I' \star \\
&\quad \$(2C_0 \cdot \text{net}\ G\ L' + 2C_0 \cdot \mathit{leftover} + C_0) \star \\
&\quad \text{match}\ \mathit{step}\ \text{with} \\
&\quad \mid \text{Continue}\ \mathit{stack}' \Rightarrow \\
&\qquad \$(2C_0 \cdot |\mathit{ys}| + C_0 \cdot \text{stackPot}\ G\ \mathit{stack}' + K) \star [\ldots] \\
&\quad \mid \text{Break}\ () \Rightarrow \exists \mathit{ys}'\ \mathit{stack}'. \\
&\qquad \$(2C_0 \cdot |\mathit{ys}'| + C_0 \cdot \text{stackPot}\ G\ \mathit{stack}') \star [\ldots]
\end{aligned}$$

This invariant threads through the loop the expression of the potential as it appears in the precondition of `visit_backward`, plus credits to pay for: the iteration itself ("$2C_0 \cdot |\mathit{ys}|$", where ys is the list of neighbors of x that are yet to be visited); stopping the iteration with `Break` (K); and the recursive call afterwards (C_0).

4. To show that this invariant is preserved, the key reasoning step is to show that it is preserved when updating the level of a vertex y (in the case where $L(y) < k$).

Before updating the level of the vertex, the invariant asserts the ownership of the following amount of credits:

$$2C_0 \cdot \text{net } G\ L' + C_0 \cdot \text{stackPot } G\ \mathit{stack} \\ + 2C_0 \cdot |y :: ys| + 2C_0 \cdot \mathit{leftover} + C_0 + K.$$

After updating the level of y, we must show that we can provide credits that correspond to the updated invariant:

$$2C_0 \cdot \text{net } G\ (L'[y := k]) + C_0 \cdot \text{stackPot } G\ (y :: \mathit{stack}) \\ + 2C_0 \cdot |ys| + 2C_0 \cdot \mathit{leftover} + C_0 + K.$$

Without detailing the precise accounting to go from this first expression to the second one, the key idea is as follows: incrementing the level of a node releases as many credits as the number of edges starting from that node (lemma `net_bump_level` below). This is exactly the amount we need to pay to preserve stackPot after inserting the node into the stack. In practice, we might increase the level of the node y by more than one: in that case, we get strictly more credits than required, and we can discard the extra credits (lemma `net_weaken_level` below).

`net_bump_level` [GJCP19, proofs/GraphCostInvariants.v]
$\text{net } G\ L = \text{net } G\ (L[x := L(x) + 1]) + |\{y \mid x \longrightarrow_G y\}|$

`net_weaken_level` [GJCP19, proofs/GraphCostInvariants.v]
$L(x) \leq k \implies \text{net } G\ (L[x := k]) \leq \text{net } G\ L$

5. By using the lemmas above and stepping through the body of `interruptible_fold`'s client function, we collect constraints about C_0 and K that we can defer:

$$\begin{array}{ll} & 1 + \texttt{cost is\_marked}\ () + \texttt{cost interruptible\_fold}\ () \leq K \\ \wedge & 1 + \texttt{cost is\_marked}\ () + \texttt{cost get\_level}\ () + \texttt{cost set\_parent}\ () + \\ & \texttt{cost set\_level}\ () + \texttt{cost clear\_incoming}\ () + \texttt{cost add\_incoming}\ () + \\ & \texttt{cost interruptible\_fold}\ () \leq C_0 \\ \wedge & \ldots \end{array}$$

6. In the case of a recursive call, stepping through the code yields side-conditions about B_1, of the form:

$$\begin{array}{ll} & B_1 \leq C_0 + K \\ \wedge & 1 + \texttt{cost get\_outgoing}\ () + \texttt{cost interruptible\_fold}\ () + K \leq B_1 \\ \wedge & \ldots \end{array}$$

We defer them, then automatically find suitable values for B_1, C_0 and K that satisfy the constraints, completing the proof.

7. In the case where the traversal terminates by returning ForwardCyclic (23), a non-trivial reasoning step is required. After the call to `interruptible_fold`, the following amount of credits is available:

$$2C_0 \cdot \text{net } G\ L' + 2C_0 \cdot \mathit{leftover} \\ + C_0 + 2C_0 \cdot |ys'| + C_0 \cdot \text{stackPot } G\ \mathit{stack}'.$$

To establish the post-condition of `visit_forward`, which is simply "[]", we must dispose of these credits. We therefore need to show that the quantity above is nonnegative—that is, it does not represent a debt.

The challenge here is with handling net $G\ L'$. Corollary 8.5.2 entails that net $G\ L'$ is nonnegative *assuming* that Inv $G\ L'\ I'$ holds. However, this assumption does not hold here; in fact, net $G\ L'$ might be negative.

The solution relies on the "leftover" credits, and is to thread the following additional invariant through the proof:

$$\text{(I)} \quad |\text{unvisitedE}\ G\ L\ k| \leq \text{net}\ G\ L + \mathit{leftover}$$

where "unvisitedE $G\ L\ k$" is the set of edges that have not been visited, i.e. whose level is below k.

$$\text{unvisitedE}\ G\ L\ k \triangleq \{(x, y) \mid x \longrightarrow_G y \wedge L(x) < k\}$$

One can show that (I) is indeed an invariant (using lemmas `net_bump_level` and `net_weaken_level`); additionally, it entails that "net $G\ L + \mathit{leftover}$" is nonnegative. This ensures that we can dispose of the amount of credits represented by "$2C_0 \cdot$ net $G\ L' + 2C_0 \cdot \mathit{leftover}$", even though "net $G\ L'$" might itself be negative.

The `forward_search` function is a wrapper on top of `visit_forward`. We explicitly define the constant C (which appears in the definition of ϕ, in Figure 8.8) to be $2C_0$, where C_0 comes from the specification of `visit_forward`. Then, establishing the expected specification for `forward_search` (Figure 8.11) from `visit_forward`'s specification is straightforward.

Finally, the `add_edge_or_detect_cycle` function ties everything together: it calls the two search functions (`backward_search` and `forward_search`), and finally adds the edge into the graph, if permitted. Its specification (after unfolding the IsGraph representation predicate) appears in Figure 8.9. One should consider the constants C and C'—that also appear in the definition of ϕ and ψ respectively (Figure 8.8)—to be quantified "at the top-level". In fact, the value of C has already been determined by the proof of `forward_search` (as being $2C_0$); the value of C' is determined by the proof of `add_edge_or_detect_cycle` (presented next).

Let us focus on the complexity analysis for the branch of the program where both the backward and forward search is run, and where the edge is finally inserted into the graph (lines 64 to 70).

1. By stepping through the code of `add_edge_or_detect_cycle`, we synthesize the following cost inequality, between the cost of its body (on the left-hand side) and the number of credits that appear in the pre and post-condition (on the right-hand side). Let us name v and w the vertices corresponding to the edge that we wish to insert into the graph; let us name G and L the initial mathematical graph and

levels, L' the updated levels after calling `forward_search`, and m, n the number of edges (resp vertices) of G.

$$
\begin{array}{lll}
 & 1 & \text{(entering the function)} \\
+ & \texttt{cost vertex\_eq}\ () & \text{(call to } \texttt{vertex\_eq}\text{)} \\
+ & 3 \cdot \texttt{cost get\_level}\ () & \text{(calls to } \texttt{get\_level}\text{)} \\
+ & A \cdot L(v) + B & \text{(call to } \texttt{backward\_search}\text{)} \\
+ & B' + \phi(G, L) - \phi(G, L') & \text{(call to } \texttt{forward\_search}\text{)} \\
+ & 1 + \texttt{cost add\_edge}\ () + \texttt{cost add\_incoming}\ () & \text{(call to } \texttt{succeed}\text{)} \\
\leq & & \\
 & C \cdot (\text{net}\ G\ L) & \text{(pre-condition (01))} \\
+ & C' \cdot (\text{received}\ (m+1)\ n - \text{received}\ m\ n + 1) & \text{(pre-condition (02))} \\
- & C \cdot (\text{net}\ (G + (v, w))\ L') & \text{(post-condition)}
\end{array}
$$

2. For conciseness, let us define $\mathcal{K}$ to be the sum of all constant costs appearing in the left-hand side above.

$$
\begin{array}{rcl}
\mathcal{K} & \triangleq & 1 + \texttt{cost vertex\_eq}\ () + 3 \cdot \texttt{cost get\_level}\ () + B + B' \\
 & & + 1 + \texttt{cost add\_edge}\ () + \texttt{cost add\_incoming}\ ()
\end{array}
$$

3. Recall from Figure 8.8 that $\phi(G, L)$ is defined as $C \cdot (\text{net}\ G\ L)$. By unfolding the definition of ϕ, using $\mathcal{K}$ and after simplification, the inequality becomes:

$$
\begin{array}{rl}
 & \mathcal{K} + A \cdot L(v) + C \cdot (\text{net}\ (G + (v, w))\ L' - \text{net}\ G\ L') \\
\leq & C' \cdot (\text{received}\ (m+1)\ n - \text{received}\ m\ n) + C'
\end{array}
$$

4. From the definition of net and received (Figure 8.8), one can prove the following lemma, that relates the variation of net to the variation of received when adding an edge. This is an immediate consequence of the definitions; intuitively, the lemma expresses that an edge $v \longrightarrow_G w$ carries an amount of potential equal to $L(v)$.

`delta_net_add_edge` [GJCP19, proofs/GraphCostInvariants.v]
let $m := |\text{edges}\ G|$ in
let $n := |\text{vertices}\ G|$ in
$v \not\longrightarrow_G w \implies$
$\text{received}\ (m+1)\ n - \text{received}\ m\ n = \text{net}\ (G + (v, w))\ L' - \text{net}\ G\ L' + L(v)$

5. After using the lemma `delta_net_add_edge`, the inequality of (3) becomes:

$$
\begin{array}{rl}
 & \mathcal{K} + A \cdot L(v) + C \cdot (\text{net}\ (G + (v, w))\ L' - \text{net}\ G\ L') \\
\leq & C' + C' \cdot L(v) + C' \cdot (\text{net}\ (G + (v, w))\ L' - \text{net}\ G\ L').
\end{array}
$$

6. At this stage, C' is a constant whose value is yet to be determined. In other words, we can pick a value for C' that makes the proof go through. It is enough to define C' to be $\max(\mathcal{K}, A, C)$ to prove the inequality above, thus completing the proof. Remark: in the formal proof, the value for C' is inferred using the combination of `procrastination` and `elia`, as for the previous functions.

8.8 Future work

A number of extensions to the algorithm and its specification are possible and would be interesting to pursue as future work.

A more general asymptotic complexity bound

The specification that we present in Figure 8.1 includes a multi-variate complexity bound (on ψ). This asymptotic bound is expressed using the product filter on $\mathbb{Z} \times \mathbb{Z}$ (Definition 5.2.10). As we have argued earlier (§5.4), such a bound is not as general as we might wish, as it is not usable to reason about the complexity of the algorithm where either the number of vertices or edges stays constant.

A relatively straightforward generalization of the current specification would be to establish an asymptotic bound for ψ with respect to the more general *norm* filter (Definition 5.2.12), generalizing the present specification.

Recovery strategy after detecting a cycle

The current specification for `add_edge_or_detect_cycle` (Figure 8.1) prevents reusing the graph after a cycle has been detected. Currently, if the function returns "Cycle", then the post-condition simply yields "$[w \longrightarrow_G^* v]$": the user loses ownership of the graph.

In a first approach, it is straightforward to strengthen the post-condition as follows. (In fact, this is a generalization that is already included in the Coq formalization.)

$$\exists L\, M\, V\, I.\, \mathrm{IsRawGraph}\, G\, L\, M\, V\, I \;\star\; [w \longrightarrow_G^* v].$$

This grants ownership over the underlying graph structure. Nevertheless, the internal functional invariants (Inv) and potential (ϕ) have been lost.

This is not only a specification issue. After detecting a cycle, one needs to effectively modify the graph to restore the node levels and incoming sets so as to restore the invariants.

In future work, one could implement an extension of the algorithm that provides an additional operation `restore`, that could be called after detecting a cycle in order to repair the graph and restore the ability to call `add_edge_or_detect_cycle`. It is likely that this operation would be expensive however: in first approach, its cost depends on the worst-case cost of the forward traversal, which is linear in the number of edges in the graph in pathological cases.

Interestingly, it is in fact possible to implement such an operation *without modifying the algorithm*. Indeed, since our implementation of the algorithm is parameterized by the implementation of the underlying "raw graph" structure, it is possible to instantiate it with an extended raw graph implementation that keeps track of the history of modifications. Then, the `restore` operation would simply consist in rolling back the modifications made when detecting the cycle.

Allow the raw graph to be a functional data structure

The current "raw graph" structure is assumed to implement an imperative graph structure, with constant time operations.

It would be interesting to consider whether this requirement can be relaxed. From a functional correctness point of view, the algorithm can work similarly with an immutable graph structure. Then, one could try to generalize the complexity analysis and the complexity bound to make it parametric in the cost of basic graph operations (e.g. they would be in logarithmic time instead of constant time in the case of an immutable data structure).

Extension to strong components

Section 4.1 of Bender *et al.*'s article [BFGT16] describes a non-trivial extension of the algorithm (described in Section 2 of their paper), where the extended algorithm is able to additionally maintain strong components. This extension is particularly interesting to us because it corresponds to the version of the algorithm that is actually used in the implementation of Coq. We plan to tackle it in future work.

Chapter 9

Related Work

9.1 Formalization of the O notation

We give a formal account of the O notation, as a Coq library. Our definition is generic with respect to the domain of the functions that are compared by the notation, providing support for single as well as multi-variate functions. We achieve this by parameterizing our definition by a filter, which characterizes what "big enough values" correspond to in the domain of functions considered. We discuss the choice of filters including the case of multivariate domains of the form $\mathbb{Z} \times \mathbb{Z}$.

A number of formalizations define the O notation in the specific case of functions of type $\mathbb{N} \to \mathbb{R}$, e.g. in Mizar [KRS99] or HOL4 [IHSA19]. We are interested in a more general notion of O, that accounts for functions of type e.g. $\mathbb{Z} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$, but also multivariate functions.

Avigad and Donnelly [AD04] give a somewhat more general definition of O in Isabelle/HOL. Their definition of O is as follows, without restriction on the domain of the function f (its codomain must be a nondegenerate ordered ring):

$$O(f) \triangleq \{h \mid \exists c.\, \forall x.\, |h(x)| \leq c \cdot |f(x)|\}\,.$$

However, this differs from the usual definition of O (as can be found e.g. in *Introduction to Algorithms* by Cormen et al. [CLRS09]): this definition requires the inequality to hold on all inputs, whereas the usual definition only requires that it holds for sufficiently large inputs. The two definitions are in fact almost equivalent if the inputs are natural numbers, but not if the inputs live in $\mathbb{Z}$ or $\mathbb{R}$. Avigad and Donnelly do mention that one can obtain the usual definition of O on top of their definition, by considering an "eventually" modality on sets, where "A eventually" (with A a set of functions) would be defined as:

$$A \text{ eventually} \triangleq \{f \mid \exists k.\, \exists g \in A.\, \forall x \geq k.\, (f(x) = g(x))\}\,.$$

Then, "$O(f)$ eventually" (using their definition of O) would correspond to the usual notion of O. The reasoning principles that they present only apply to O by itself however, without accounting for a possible "eventually" modality. We prefer to stick to a definition of O that directly generalizes the usual definition.

We are aware of two other formalizations of O that rely on filters in a manner similar as ours. Eberl [Ebe17] formalizes a library of Landau symbols (including O) in Isabelle/HOL in the context of his formalization of the Akra-Bazzi theorem, based on the library of filters by Hölzl *et al.* [HIH13]. Affeldt, Cohen and Rouhling [ACR18] formalize Landau notations and develop support for equational reasoning in Coq, as

part of an on-going effort to extend the Mathematical Components library with analysis. It is worth noting that the work of Affeldt *et al.*, as well as ours, takes inspiration in the Coquelicot library [BLM15], which uses filters for asymptotic reasoning (though it does not define the O notation). To our knowledge, the libraries of Eberl and Affeldt *et al.* were developed approximately at the same time as ours. The fact that all three formalizations (ours, Eberl's, and Affeldt *et al.*'s) adopt the same basic definitions (where the O notation is parameterized by a filter) weights in favor of the filter-based approach for formalizing the O notation and others Landau symbols.

It seems the question of what is a suitable notion of O for multivariate functions has been seldom discussed. In a technical report, Howell [How08] tackles the question in the setting of algorithm complexity. Note that this work is carried out informally, and has not been mechanized. Howell identifies several principles that seem desirable when analyzing the complexity of algorithms, and shows that these principles are not satisfied by the "natural" definition of multivariate O (which corresponds in our setting to the choice of the filter product on $\mathbb{N} \times \mathbb{N}$). Recall that our formal account of O, equipped with the filter product on $\mathbb{N} \times \mathbb{N}$ is written as "$\cdot \preceq_{\mathbb{N}\times\mathbb{N}} \cdot$" (§5.1). Howell proposes the following alternative definition of O for functions on $\mathbb{N} \times \mathbb{N} \to \mathbb{R}^+$, written "$\widehat{O}$":

$$f \in \widehat{O}(g) \triangleq f \preceq_{\mathbb{N}\times\mathbb{N}} g \wedge \widehat{f} \preceq_{\mathbb{N}\times\mathbb{N}} \widehat{g}$$

where given a function f, the function $\widehat{f}$ corresponds to an iterated maximum, and is defined as:

$$\widehat{f}(n_1, n_2) \triangleq \max\{f(i_1, i_2) \mid 0 \leq i_1 \leq n_1 \wedge 0 \leq i_2 \leq n_2\}.$$

Howell's $\widehat{O}$ is in general a stronger (more demanding) relation that $\preceq_{\mathbb{N}\times\mathbb{N}}$. However, in our setting, we require cost functions to be monotonic (for other reasons, see §4.2.1). Then, under the assumption that f and g are monotonic, f is equal to $\widehat{f}$ and g is equal to $\widehat{g}$, and therefore "$f \in \widehat{O}(g)$" is equivalent to "$f \preceq_{\mathbb{N}\times\mathbb{N}} g$". Working with product filters plus monotonicity side-conditions is thus equivalent to working with Howell's $\widehat{O}$. We show that Howell's reasoning principles hold in that setting (this includes in particular the summation lemmas in §5.3.1).

The issue of reusing a multivariate bound while fixing some parameters to constant values, which we discuss in Sections 4.1.3 and 5.4, is not addressed by Howell. In fact, Howell discusses an alternative definition of O, written $O_\exists$, defined as follows, for functions on $\mathbb{N} \times \mathbb{N} \to \mathbb{R}^+$:

$$f \in O_\exists(g) \triangleq \exists c\, N.\, \forall n_1, n_2.\, n_1 \geq N \vee n_2 \geq N \implies f(n_1, n_2) \leq c \cdot g(n_1, n_2)$$

This definition of O is in fact equivalent to our definition when using a norm filter, using the ∞-norm on $\mathbb{N}^2$. Howell discards this definition of O as it does not support some bounds that might seem intuitive. For instance, it does not always allow eliminating small terms by assuming that all parameters are big enough, e.g. $mn + 1 \notin O_\exists(mn)$. In our eyes, this is the price to pay to allow specializing multivariate bounds. It is not possible to have $mn + 1 \in O(mn)$, and at the same time allow m or n to be specialized in the bound (possibly to zero).

9.2 Approaches to Time Complexity Analysis

There exists a wide variety of techniques related to time complexity analysis (or resource analysis more generally speaking). Approaches range from fully automated cost inference to manual proofs in a program logic embedded in a proof assistant. In this work, we sit nearer to the second end of the spectrum: we use (and extend) an expressive program logic (Separation Logic with Time Credits) embedded into Coq, and focus on interactive verification relying on user-supplied complexity bounds and invariants. We are interested in the case where the complexity analysis depends on functional correctness invariants, and put the emphasis on the modularity of specifications that we devise (motivating the use of O), as well as the robustness of the proofs (using a limited form of cost inference).

In the rest of the section, we discuss approaches that relate to these goals in one way or another. This does not include approaches related to the computation of worst-case execution-time (WCET), as their motivation (deriving precise machine-dependent real-time bounds) is in essence opposed to ours (establishing asymptotic complexity bounds that abstract from the hardware). We refer to [WEE+08] for a survey on WCET. We also leave out of the picture approaches for automatic bound analysis based on abstract interpretation [BHHK10, Gul09, GMC09, SZV14, ZGSV11] or term rewriting systems [ADLM15, AM13, BEF+14, HM14]. These fully automated approaches are typically limited in the scope of programs they can handle (e.g. restricted to first-order integer-manipulating programs), the kind of bounds they can infer (e.g. restricted to polynomial bounds), and lack compositionality, which makes them remote from our goals.

9.2.1 A first intuition: ticking translation

In order to get some intuition on the problem at hand—analyzing both the correctness and complexity of a program—it is worth noting that there is a fairly simple solution.

Assume that we have available a framework for verifying the functional correctness of programs. To reason about the complexity of a program, it is enough to instrument this program with a global counter; at each step the instrumented program decrements the counter; if the counter reaches zero, then the instrumented program raises an error. Proving the functional correctness of the instrumented program means that we establish a bound on the running time of the initial program. We have reduced a proof of complexity to a standard proof of functional correctness.

This simple minded approach is "complete": there is no loss of precision when reasoning on the functional correctness of instrumented program with respect to the original problem. It is not completely satisfactory from a practical perspective, as the user needs to reason about an instrumented program without particular support from the framework, which is not very convenient.

Nevertheless, this initial simple idea can be traced back in a number of the more elaborate approaches that we present next. It is the essence of the construction of Separation Logic with Time Credits on top of Iris by Mével *et al.* [MJP19]. Using the expressive power of Iris, time credits are implemented as representing a share of the global counter; then, the rules of Iris plus Time Credits do allow the user to reason about a program instrumented through a "ticking translation". The type system of Crary and Weirich [CW00] refers to a "virtual clock"; the complexity of a function is described

as the variation of the virtual clock before and after its execution. This is similar to keeping track of a global counter (except that the counter would be decremented, while the clock is incremented as time passes). Finally, cost monads used either as a verification framework [MFN+16, vM08] or as an instrumentation technique to make explicit the cost semantics of a program [DPR13] can be seen as a monadic implementation of a counter or clock.

9.2.2 Separation Logic with Time Credits

The logical basis for our work is Separation Logic with Time Credits, first introduced by Atkey [Atk11]. This provides an expressive and versatile program logic, which supports higher-order functional and imperative programs, allows reasoning at the same time about functional correctness and complexity, and supports amortized resource analysis. Notably, the way the user can express the potential of data structures for amortized analysis is very flexible: since time credits are a first-class Separation Logic resource, they can be seamlessly threaded through Separation Logic predicates and invariants.

As detailed previously in the Background chapter (§2), Charguéraud and Pottier implement Separation Logic with Time Credits as an extension of the CFML tool [Cha13], and use it to verify the correctness and complexity of an OCaml implementation of the Union-Find data structure [CP17b]. Their specifications feature concrete cost functions, and the present work is a continuation of this line of work.

Recently, Zhan and Haslbeck [ZH18] introduce a similar framework for Isabelle/HOL, built on top of Imperative HOL [BKH+08, LM12], and also based on Separation Logic with Time Credits. They are able to integrate Eberl's formalization of Akra-Bazzi [Ebe17] to automatically establish O bounds for divide-and-conquer algorithm, and implement support in the auto2 prover [Zha18] for reasoning about Separation Logic with Time Credits and deriving asymptotic complexity bounds. Their specifications consequently publish asymptotic bounds using the O notation, in a style similar to ours. From a methodological perspective, the main difference with our approach is in the treatment of concrete cost functions. With the concern of proof robustness in mind, we try very hard to avoid writing down concrete costs in specifications, proofs and avoid manipulating them explicitly; for that purpose we rely on our `procrastination` library [Gué18c] which allows us to write "template costs" using abstract multiplicative constants, and on a limited form of cost inference as described in Section 6.4. As a result, typical proofs of programs follow an unusual structure in our setting, where forward deduction and inference are performed somewhat simultaneously. In contrast, the approach of Zhan and Haslbeck does requires the user to write explicitly the expression of concrete cost functions in their proofs (typically in a way that syntactically corresponds to the structure of the program). This makes proofs scripts more fragile with respect to changes in the program, but allows Separation Logic proofs to be much more automated. An interesting aspect of Zhan and Haslbeck's approach is that it makes it easier to separate functional correctness and complexity-related arguments from the proof of the program itself, as separate lemmas only referring to the explicit cost function. In our approach, the cost function is elaborated during the proof of the program itself, which prevents stating such lemmas separately.

In more recent work, Haslbeck and Lammich [HL19] build on top of the framework of Zhan and Haslbeck so as to provide support for proving program complexity following a refinement approach. In that setting, Separation Logic with Time Credits is one of the bottom layers in the refinement stack.

Mével, Jourdan and Pottier [MJP19] implement Time Credits on top of the powerful Iris Separation Logic [JKJ+18], as well as Time Receipts, a dual notion which can be used to establish running time lower-bounds and (as an application) reason about the absence of integer overflow. Using Time Credits in combination with the powerful features of the Iris logic (such as ghost state and invariants), the authors are able to verify a library of thunks, following the methodology proposed by Okasaki [Oka99] where each thunk is associated with a "debit" that keep tracks of the amount of credit that one must still pay before one is allowed to force the thunk. This analysis really requires a powerful Separation Logic on top of Time Credits: we would not be able to carry it in our setting, because CFML's Separation Logic does not support ghost state or invariants. It is worth noting that extending the logic of Mével *et al.* with possibly negative time credits (as we do in §6.3) is non trivial. In fact, Iris builds in the fact that every assertion is affine; this is incompatible with the existence of negative time credits (which cannot be discarded). One idea would be to reproduce the work of Mével *et al.* on top of Iron [BGKB19], a logic with support for linear resources, itself implemented on top of Iris.

9.2.3 Hoare-style program logics

Outside of Separation Logic with Time Credits, there is work on Hoare-style program logics that account for running time, pioneered by the work of Nielson [Nie84, Nie87, NN07].

A recent study by Haslbeck and Nipkow [HN18] examines the meta-theory of three Hoare logics for upper bounds on running times, and mechanize their meta-theory as well as verification condition generators, for a simple imperative language (IMP). The study compares small logics in the style of Nielson, Separation Logic with Time Credits, and the quantitative Hoare logic of Carbonneaux *et al.* [CHS15, CHRS17]. They establish that Nielson's logic and the quantitative Hoare logic are equivalent (under some assumptions). Additionally, they show that quantitative Hoare logic can be embedded into Separation Logic with Time Credits.

Let us say a bit more about the work of Carbonneaux *et al.* Their quantitative Hoare logic is the formal basis for applying the approach of automated amortized resource analysis (AARA, also at the basis of RAML that we describe after) to integer-manipulating C programs [CHS15, CHRS17, Car18]. Whereas in standard Hoare logic, a specification is expressed as a triple $\{P\}\ \mathtt{c}\ \{Q\}$ where P and Q are assertions on program states, the logic of Carbonneaux *et al.* handles triples of the form $\{\Phi\}\ \mathtt{c}\ \{\Phi'\}$, where Φ and Φ' are potential functions that map program states to non-negative numbers. Analyzing the cost of a program is done by applying the rules of the logic, which produce linear constraints that are ultimately solved using off-the-shelf LP solvers. As with other tools based on AARA, the method can in fact be used to infer polynomial multivariate cost functions. The idea is to describe the potential at each point in the program as a "template polynomial", a linear combination of a set of base polynomials, where multiplicative coefficients are variables that will be later instantiated by the LP solver, and where base polynomials depend on program variables (or difference between variables), using a clever

polynomial basis. Carbonneaux's tool also produces Coq proof certificates that witness the soundness of the analysis.

Since in principle it is possible to translate a derivation of Carbonneaux's quantitative Hoare logic into a derivation in Separation Logic with Time Credits, integrating the automated analysis of Carbonneaux with our framework seems like a promising idea. This would come with a number of challenges. First, the certificates of Carbonneaux *et al.* are expressed with respect to a relatively low-level representation of programs (their control flow graph), whereas we reason at the source level. Then, the "composition units" of the two approaches differ. We compose specifications that rely on abstract cost functions and asymptotic O bounds; Carbonneaux's analysis is compositional, but specifications compose as Hoare triples where polynomial potential functions contain uninstantiated coefficients along with a set of linear constraints on these coefficients. In other words, solving the linear constraints is a global operation that must occur at the very end of the analysis (and that may fail if the constraints are unsatisfiable). After constraint resolution, function specifications mention concrete cost functions (as polynomials with explicit coefficients). Finally, Carbonneaux's work infers complexity bounds that depend on program integers; we are interested in programs that are imperative but also manipulate data in an applicative style; for that, integrating with RAML would be perhaps more relevant.

In recent work, Radiček *et al.* [RBG+18] present two proof systems, for unary and relational cost analysis of higher-order purely functional programs, that allow establishing both functional correctness properties and a cost analysis, and the cost analysis to depend on the correctness invariants. Time consumption is made explicit by adding a cost monad to the language with an explicit "step" operation. In the unary setting, their logic uses the two following judgments, where Γ is a simply typed context, τ a simple type, e a pure expression and m a computation in the cost monad. Ψ is a set of assumptions, and ϕ an assertion about the return value (produced by e or m); it can be read as a post-condition. Only monadic computations have a cost: in the corresponding judgment, k gives a lower-bound and ℓ an upper bound on that cost.

$$\Gamma; \Psi \vdash e : \tau \mid \phi \qquad\qquad \Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi$$

The authors show that the amortized time analysis of RAML can be embedded in their system (in the unary case); they also postulate that it may be doable to embed Carbonneaux's logic as well, provided the monad is extended to include state as well as cost. Given that derivations in Radiček's logic enforce a syntactic distinction between correctness-related properties (ϕ) and costs (k and ℓ), it seems that they could be directly translated into Separation Logic with Time Credits (for upper bounds at least, lower bounds would require Time Receipts as well). The judgment $\Gamma; \Psi \vdash m \div \tau \mid 0 \mid \ell \mid \phi$ would be translated to a triple of the form $\Gamma \vdash \{[\Psi] \star \$\ell\}\ m\ \{\phi\}$. (A judgment with a non-zero lower bound would be translated similarly to a triple with additional Time Receipts in the postcondition.) Going the other direction (embedding some form of time credit based reasoning into their unary logic) seems less obvious.

Closer to traditional program verification techniques, Madhavan, Kulal and Kuncak [MKK17] present a verification framework for higher-order functional programs that use lazy evaluation and memoization. In their system, programs are annotated with assertions, invariants and post-conditions, in the form of contracts. Contracts resemble

Hoare logic triples, and are used to produce verification conditions that are checked statically, and that can include formulas about both correctness properties and the running time of the program. Their system is therefore able to establish complexity bounds for a number of data structures that rely on both laziness and functional invariants. Interestingly, in contracts specifying complexity bounds, users can use "template constants" written "?" that stand for constants to be inferred by the system. This is very similar to our use of `procrastination` as a way of introducing abstract constants on which to infer constraints. One difference is that in our case, these constants can depend on the abstract cost of auxiliary operations (as a specification abstracts over the concrete cost of a program), while in the work of Madhavan *et al.*, the concrete cost of a program is exposed to its caller, and template constants therefore correspond to purely numerical values.

9.2.4 Monadic shallow embeddings

Outside of full-fledged program logics, a number of approaches are instead based on monadic shallow embeddings in the setting of proof assistants. In that setting, programs are written as definitions in the logic of the proof assistant and represent monadic computations, for some form of cost monad which accounts for resource consumption. Such approaches are generally "semiformal", in the sense that programs have no fixed cost semantics in general; instead, time consumption is determined by explicit calls by the user to a time-consuming primitive of the monad.

These approaches provide a minimalist framework in which it is possible to carry out a machine-checked analysis of both the correctness and complexity of programs, by simply reusing the usual reasoning principles from the ambient logic of the proof assistant (although reasoning on monadic programs is not necessarily an easy task).

Danielsson [Dan08] presents a library for analyzing the amortized time complexity of purely functional programs in the presence of laziness (that is, delayed computation and memoization), embedded in the Agda proof assistant. His library formalizes reasoning on "thunks", following Okasaki's banker's method. The type of a thunk is the dependent type $\mathit{Thunk}\ n\ a$, where n is a natural number describing that the thunk can produce a value of type a after n computation steps. Then, computation steps are represented by a "tick" operation that is used to annotate the code of programs, and is of type $\mathit{Thunk}\ n\ a \to \mathit{Thunk}\ (n+1)\ a$. Danielsson shows that this framework can be used to verify data structures that make an essential use of laziness in order to ensure good performance. On the practical side, the framework comes with the usual caveats related to programming in a dependently-typed setting: the user may have to manually write equality proofs to rewrite arithmetic operations in the type of *Thunk*, in order to be able to typecheck its monadic programs. Functional correctness invariants have to be threaded as dependent types as well, in the return type (the second parameter) of *Thunk*. As mentioned previously, Mével *et al.* [MJP19] show that Danielsson's formal account of thunks can be reproduced in Separation Logic with Time Credits, given a powerful enough ambient logic (Iris in their case).

Van der Weegen and McKinna [vM08] study the average-case complexity of Quicksort, represented in Coq as a monadic program. Their quicksort program is in fact parameterized on the monad; they instantiate the monad to account for both nondeterminism and

comparison-counting. A "cost function" is obtained by threading a counter through the monad, and projecting the result of that counter out of the quicksort program. Although it is somewhat specific to the case-study at hand, and only analyzes a small program, we believe that this work is a nice example of a non-trivial complexity analysis carried out with a minimal amount of boilerplate or preliminary scaffolding.

McCarthy, Fetscher, New, Feltey and Findler [MFN+16] present a Coq library for the complexity analysis of functional programs, in the form of a generic cost monad. Similarly to Danielsson's approach, the cost semantics of a program is given by explicit calls to an abstract "step" operation provided by the monad. The authors verify the complexity of a number of challenging functional data structures, such as Braun trees or red-black trees. The complexity analysis for these structures typically depends on correctness invariants (e.g. on the size of the trees). One contribution of their work is to facilitate reasoning on programs and structures with additional functional invariants, by indexing the monad not by time, but by a predicate mentioning time, which can also include correctness properties (similar to the Dijkstra monad of Swamy *et al.* [SWS+13]). Following a methodology similar to the one of Zhan and Haslbeck (mentioned earlier), concrete cost functions are defined explicitly, following the structure of the program (e.g. defined recursively). The monadic program is shown to admit such a cost function, and as a last step, a O bound is established separately for the cost function. It would be relatively straightforward to reproduce the proofs of McCarthy *et al.* in our setting and in a similar style.

9.2.5 Extraction of recurrence relations

A related question is the one of, given a program, obtaining a recurrence relation that characterizes the cost of this program. Provided that one can obtain such a relation, reasoning about the cost of the program can then be done with respect to that relation directly. This idea is similar to our limited mechanism for "credit synthesis". In our setting, this mechanism allows synthesizing recurrence equations for the cost function, but in a semi-interactive fashion: user input is required to express the cost of local operations depending on the relevant parameters. To our knowledge, fully automated methods for extracting recurrence equations either focus on first-order imperative programs where the analysis does not depend on user-defined data [AAG+12, ABG12, FMH14] or purely functional programs with inductive types (see below).

The idea goes back at least as far as Wegbreit [Weg75], who describes a mechanism for inferring cost equations for simple first-order Lisp programs, and solving them in order to obtain closed-form expressions. Benzinger [Ben04] applies the approach of Wegbreit to the cost analysis of Nuprl terms; computing a closed form for the inferred equations is done by calling the Mathematica computer algebra system.

Forster and Kunze [FK19] give a formal account for the extraction of a subset of Coq terms to weak call-by-value lambda-calculus, including time bounds. The process is formalized in Coq, and their tactics automatically produce recurrence equations; obtaining a closed form requires manual work from the user. It should be possible to translate instances of Forster and Kunze's `computableTime` relation—that relates a Coq function to its extracted term and cost—to Separation Logic triples in our setting. Barring technical difficulties, this would allow integrating programs written as Coq functions with

the verification of programs written in OCaml that our framework currently supports. (The resulting setup would be reminiscent of the integration of CakeML's translator and characteristic formulae framework [GMKN17, §4].)

Nipkow and Brinkop [Nip15, NB19] show that, assuming that the complexity of a (functional, first-order) program is given as recurrence equations, then it is possible to carry out the verification of challenging amortized complexity analysis, in the case of functional data structures. The data structures they study include skew heaps, splay trees, splay heaps and pairing heaps, that they implement and verify in Isabelle/HOL. The authors do not mechanize the extraction of recurrence equations from the implementation of data structure operations; instead, recurrence equations are extracted by hand, but in a systematic way that is expected to be easy to automate. The focus in this work is on the mathematical analysis; Zhan and Haslbeck (presented in §9.2.2) show that the analysis of Nipkow and Brinkop can be easily integrated with their verification framework based on time-credits.

Regarding the extraction of recurrence equations for higher-order functional programs, Danner, Paykin and Royer [DPR13] and later Danner, Licata and Ramyaa [DLR15] give a formal account of the process for a simply-typed lambda-calculus with inductive datatypes. The extraction process consists in a first translation to a "cost monad", making costs explicit, then by abstracting values with their size. This yields a fairly generic framework, since the notion of size is ultimately up to the user (under some sanity conditions). Solving the recurrences is not discussed in these works. It would be interesting to consider whether one could prove a theorem relating the translation of Danner *et al.* to a triple in Separation Logic with Time Credits. It would be also worth investigating whether it could serve as a basis for an improved credit synthesis mechanism.

Vasconcelos and Hammond [VH04] describe an analysis inferring cost equations for higher-order functional programs based on a type-and-effect system using sized types. The analysis is fully automated, but restricted in the kind of equations it can infer. For instance, the cost of a conditional (in the program) is systematically expressed as the maximum of the cost of the two branches, which may be imprecise; in our credit synthesis mechanism, we also infer a maximum by default, but the user can also decide to use a conditional (in the cost expression) instead.

9.2.6 Type-system based approaches

A number of approaches for cost analysis rely on type systems. Earlier approaches include the work of Crary and Weirich [CW00], who express the cost of a function in its type as the variation of a "virtual clock". For instance, a function that takes an integer, returns an integer and does three computation steps would "start at time $n + 3$" and "finish at time n", and hence given the type $\forall n.\ (\mathsf{int}, n + 3) \to (\mathsf{int}, n)$—the second component of the pair representing the state of the virtual clock. This encoding only allows reasoning about worst-case complexity, not amortized complexity.

Pilkiewicz and Pottier [PP11] introduce the idea of Time Credits as linear capabilities, in the setting of a type-and-capability system. This is very similar to the encoding of Time Credits in Separation Logic. Their system supports reasoning about amortized complexity, including in the presence of thunks in the style of Okasaki and Danielsson's

approach (described earlier in §9.2.4). In that respect, it is more expressive that Atkey's Separation Logic with Time Credits (and ours); it is closer to Iris with time credits (§9.2.2).

The type system of Pilkiewicz and Pottier is very expressive, but not particularly geared towards practical verification of programs. Wang, Wang and Chlipala [WWC17] present an ML-like functional language, named TiML, that features a more practical but yet quite expressive type system. TiML is designed with practical complexity analysis is mind; the user can annotate data structures with size annotations (there is no built-in notion of size) and functions annotated accordingly to their cost. Some form of refinement types are also available, which makes it possible to express functional correctness invariants if required by the complexity analysis. Proof obligations generated by TiML's typechecker are discharged by an SMT solver. This allows expressing the complexity of a number of algorithms and data structures (such as red-black trees or Dijkstra's algorithm for shortest paths), with polynomial or poly-log multivariate bounds. Amortized complexity analysis can be carried out as well, provided on can define a potential function parameterized by the size of the corresponding data structure. Interestingly, TiML also includes some support for using O in specifications. More precisely, one can quantify existentially over a cost function at the type level, and specify a O bound over this cost function as a refinement type. (This is similar to our asymptotic complexity specifications in Separation Logic.) The automation is also able to apply the Master Theorem in order to solve a certain class of recurrence cost equations. The metatheory of the core type is formalized in Coq. However, it employs concrete cost functions; the meaning and composition rules of the O notation in the multivariate case is not spelled out. It is therefore unclear what the examples of Chapter 3 would become in that setting, for instance.

A significant line of work is devoted to the design of type systems as fully automated cost inference mechanisms. Jost *et al.* [JHLH10] pioneer a methodology corresponding to Tarjan's amortized resource analysis and expressed as a type-system, a line of work later refined by Hoffmann *et al.* [HH10, HAH12a], leading to the development of RAML [HAH12b, HDW17], a fully fledged automated resource analysis tool for a significant subset of the OCaml language. The work of Carbonneaux *et al.* (presented in §9.2.3) is also an application of this methodology, but as a program logic for C programs. In essence, the idea is to annotate types with an associated potential. The parameters of a potential typically depend on the associated type: for instance, the potential of a value belonging to an inductive datatype will depend on the number of nodes in the value. The annotated type of a function describes the potentials of both inputs and outputs. For automated inference to be possible, the set of possible potential functions is restricted to linear combinations of a set of well-chosen base polynomials. Type rules then produce linear constraints on the potentials coefficients, that can be discharged by an LP solver, effectively inferring suitable values for the coefficients. The fact that the type-system is affine is crucial for ensuring that the amortized analysis is meaningful: one must indeed ensure that the potential associated to a value is not duplicated, i.e. that potential-bearing values are used in a linear fashion. Similarly, in Separation Logic with Time Credits, the potential of a data-structure is described as an amount of time credits—an affine resource—and therefore cannot be duplicated. The automated analysis of Hoffmann *et al.* can infer multivariate polynomial bounds (while work on an exten-

sion to exponential bounds is ongoing [KH19]). The supported subset of OCaml was initially restricted to first-order functional programs with inductive types, but RAML has since then been extended with (restricted) support for higher-order polymorphic programs [HDW17] and imperative code using arrays and references [LH17].

Similarly to Carbonneaux's system, the composition unit is a potential-annotated type containing "template" coefficient along with a set of constraints; after constraint resolution, potentials are concrete functions with explicit coefficients—this departs from our asymptotic complexity specifications using O. It would be interesting to investigate whether the analysis of RAML could be made proof-producing (e.g. by producing Coq certificates like Carbonneaux's), and then relate it to Separation Logic with Time Credits. In principle it should be possible to translate a type derivation from RAML into a derivation in Separation Logic with Time Credits, where potentials annotating the types would be translated to potentials as amounts of time credits. This would allow calling RAML to automatically analyze the complexity of functions that can be then used in our framework as basic blocks to carry out a more subtle complexity analysis that would be outside the scope of RAML. It is not obvious whether the opposite direction would be possible however, that is, could we take an analysis carried out in Separation Logic with Time Credits and export it into RAML as an asymptotic complexity specification using a O bound?

Also in the setting of automated complexity analysis, but following a different approach, Avanzini and Dal Lago [ADL17] present a methodology for the automated complexity inference of higher-order functional programs, based on a type system expressing size analysis, sized-type inference and constraint solving. The approach of Avanzini *et al.* improves over RAML in the expressiveness and precision of its analysis on a number of examples involving higher-order code or closures. It is however restricted to worst-case time complexity, and does not currently account for amortized complexity.

9.2.7 Detection of performance anomalies

There is a rich literature on static and dynamic analyses that aim at detecting performance anomalies, and are therefore closer in spirit to complexity testing. This includes approaches based on a static analysis [ODL15], a mix of static and dynamic analysis [HSAK16] or a fully dynamic analysis [MR16]. A number of approaches related to fuzzing aim at synthesizing inputs that expose performance bugs or bottlenecks [TPG18, LPSS18, NKP18, PZKJ17, WCF+18]. Note that these approaches are incomplete by nature.

9.3 Complexity-preserving compilation

In our work, we consider an abstract cost model where we count β-reduction steps at the source level. Ideally, one would hope that an asymptotic complexity bound established at the source level yields the same asymptotic bound at the lowest level as well. A more detailed account of the question is given by Charguéraud and Pottier in [CP17b]. A part of the answer is that we expect our compiler to preserve asymptotic bounds.

The CerCo project [AR11, AAR12, AAB+14] has built compilers for C and ML where basic blocks of the source program are annotated with their cost, with respect to the

target cost model. These compilers are verified: the annotations are guaranteed to be correct upper bounds, and are compiled in a way that preserves asymptotic complexity.

In his PhD thesis, Wang [Wan18] equips TiML (introduced earlier in §9.2.6) with a compiler targetting a typed assembly language equipped with cost semantics. The compiler is proved to be type-preserving and complexity-preserving: well-typed TiML programs are compiled into well-typed assembly programs conforming to the same time bounds. A practical motivation for the work is to compile TiML code to smart contracts running on the Ethereum Virtual Machine bytecode language, establishing bounds on their resource usage ("gas").

9.4 Formal Verification of Graph Algorithms

Our improvement and verification of Bender *et al.*'s state-of-the-art algorithm for incremental cycle detection is a significant contribution, even without considering the verification of its complexity analysis.

Early work on the verification of garbage collection algorithms includes, in some form, the verification of a graph traversal. For example, Russinoff [Rus94] uses the Boyer-Moore theorem prover to verify Ben Ari's incremental garbage collector, which employs a two-color scheme. In more recent work, specifically focused on the verification of graph algorithms, Lammich [Lam14], Pottier [Pot15], and Chen *et al.* [CL17, CCL+18] verify various algorithms for finding the strongly connected components of a directed graph. In particular, Chen *et al.* [CCL+18] repeat a single proof using Why3, Coq and Isabelle. Abdulaziz, Mehlhorn and Nipkow [AMN19] report on the formal verification of Edmond's blossom shrinking algorithm for maximum matchings using Isabelle. None of these works include a verification of asymptotic complexity.

Dijkstra's short path algorithm has been used as a case study for complexity analysis tools such as RAML [HDW17] or TiML [WWC17]; in these cases, only the complexity bound is verified, not the functional correctness of the algorithm.

Haslbeck and Lammich [HL19] present the verification of both the functional correctness and time complexity of Kruskal's algorithm for computing a minimum-spanning-tree, and Edmonds-Karp algorithm for computing the maximum flow in a network, based on the previous work of Lammich and Sefidgar [LS16]. They use a framework built on top of a refinement methodology and Separation Logic with Time Credits, implemented in Isabelle/HOL. Along with our incremental cycle detection algorithm, these are the largest graph algorithm for which we are aware of a formal proof covering both their functional correctness and time complexity.

Finally, it is worth noting that we verify an algorithm that has been known for only a few years, whereas most of the related work on verified graph algorithms that we know of do study classical algorithms that have been known for a few decades.

Chapter 10

Conclusion

A program or algorithm should not only be correct, but also reliably fast. It is now standard practice in the algorithms literature to establish the asymptotic time complexity of an algorithm as a way to quantify its efficiency, in a way that is independent from the implementation (or hardware) details. In the formal methods community, a wide variety of approaches have been developed at this point dedicated to the verification of the functional correctness of programs. One might even say that formally verifying that a program is correct is progressively becoming a routine exercise.

However, formal approaches dedicated to the verification of both correctness and time complexity have been comparatively much less studied. The motivation is clear nevertheless: given an analysis of the correctness and complexity of an algorithm carried out informally on paper, it is desirable to be able to reproduce this analysis in a formal setting, with respect to a concrete implementation of the algorithm.

Establishing a time complexity bound for a program is an undecidable problem, as it requires proving that the program terminates. In practice, establishing functional correctness is often required as well, as it is common for the complexity analysis to depend on correctness invariants. In that setting, approaches based on a push-button automated analysis are doomed to be limited both in the range of programs they can analyze and the kind of complexity bounds they can infer.

In this thesis, we present a framework aimed at the verification of the correctness and complexity of concrete OCaml code, and that is able to express subtle complexity analyses. We lie at the more expressive and less automated end of the spectrum; our framework is embedded in the Coq proof assistant and proofs are carried out in a mostly interactive style, with the support of some special purpose automation that we develop. Additionally, our methodology should be able to scale up to larger software developments: we put a particular emphasis on the modularity and robustness of our specifications and proofs. Following the algorithms literature, we believe that a good specification of a program's complexity should abstract away from the details of the implementation and the machine it is running on; and crucially, a proof of such a specification should be also carried out in a manner that is robust in the face of minor changes in the implementation.

Key ingredients

Let us summarize what the key ingredients of our approach are.

An expressive program logic. We use Separation Logic with Time Credits, an expressive program logic which allows us to reason about the correctness of functional and imperative higher-order programs, and at the same time about their time complexity at an already abstract level (as β-reduction steps), and that handles amortized complexity

analysis, for instance in the style of Tarjan's potential method. Furthermore, we extend Separation Logic with Time Credits by allowing negative time credits. "Separation Logic with Possibly Negative Time Credits" inherits the qualities of standard Separation Logic with Time Credits, but presents more convenient reasoning rules in particular for automation performing cost inference, and makes it possible to state elegant specifications for programs that would be otherwise hard to specify, such as functions whose cost can easily be characterized depending on both their input and return values, but not the input value alone.

Program specifications with asymptotic complexity bounds. Separation Logic with Time Credits allows us to express the cost of a piece of code at the source level, as a number of β-reductions. This already abstracts from the details of the hardware. However, we argue that, as a specification, a number of β-reduction steps is itself not satisfactory: it is tied to the very details of the implementation, and therefore not abstract enough.

We propose a style of program specifications where the concrete cost of a program is made abstract, and only an asymptotic bound on the cost is published, using the O notation. We present and formalize sound composition principles for O in the setting of program verification (with rules for reasoning on e.g. for-loops), allowing modular and reusable specifications to be stated and composed, including in the case of complexity bounds depending on several parameters.

A verification framework for robust complexity proofs. When establishing a complexity specification, it is also desirable that the proof itself does not rely on a fragile accounting of individual time steps, as this would make the proof very tedious and brittle. One might think that a complexity proof could instead simply rely on a bottom-up composition of O bounds. However, this idea does not stand in the general case: a loop invariant can hardly be expressed as a O bound for instance, as O is only an asymptotic notion while an invariant needs to be precisely established and preserved through the body of the loop.

We present two key mechanisms that enable robust complexity proofs. First, a semi-automated cost inference mechanism, that synthesizes a cost expression for straight-line code while allowing user-provided invariants to guide the process. Second, a mechanism for introducing symbolic "template" constants and functions and collecting constraints about them. The combination of the two mechanisms allows cost functions and complexity invariants to be stated in an abstract way (e.g. with symbolic multiplicative constants), and implementation-specific cost expressions and recurrence equations to be automatically synthesized then solved.

Application to the verification of a state-of-the-art algorithm. We demonstrate the applicability of our methodology on a number of programs and data structures. Most notably, as our main case study, we verify the correctness and complexity of a state-of-the-art incremental cycle detection algorithm [BFGT16, §2], itself recently published (in 2016). In fact, we propose an improvement to the algorithm itself; this argues in favor of the usefulness of our methodology as a formal ground on which to study algorithm complexity.

Future work

Better support for local functions. As described in Section 3.4, analysis of code defining local functions is challenging in presence of O. Establishing the complexity of such programs is supported in our current framework; after all, our program logic is complete; and for example the implementation of our cycle detection algorithm relies on non trivial higher-order functions (e.g. interruptible fold) which are called by passing an anonymous function. However, reasoning on the cost of a local function is still somewhat inconvenient in the general case. Indeed, one typically needs to anticipate and existentially quantify on its cost function early enough in the proof. It is not completely clear what a satisfying solution would be. One idea would be a mechanism that allows quantifying late on the local function's cost, and automatically lift the quantification earlier in the proof.

Automation for solving cost recurrences in Coq. A Coq formalization (or just a formal statement) of the Akra-Bazzi theorem, as well as the corresponding automation—following Eberl's work in Isabelle [Ebe17]— would be very useful to have. Eberl's work on a verified solver for linear recurrences [Ebe19] would also be useful to replicate in Coq (albeit perhaps a bit less useful for the purpose of algorithms verification).

Resources other than time. We use time credits to keep track of running time, but the idea could be directly extended to keep track of resources that behave similarly, such as network usage or the number of disk accesses for instance. For that purpose, it might be enough to make the $\$(\cdot)$ predicate parametric with respect to a "resource type", and restrict the splitting and join rules on credits to credits of the same type.

Resources such as space or stack usage behave differently from running time; while running the program can only result in spending more time, space is recovered when data is freed, and stack space is recovered when returning from a function. It would be interesting to see a mechanism similar to time credits be applied to the accounting of these resources, but it is not clear how similar it would be with time credits. (With respect to stack space, related work includes [CHRS14] which presents a quantitative Hoare logic for stack space usage.)

Cache-oblivious algorithms. Real world program performance usually depends not only on the time complexity as we have considered it, but also on the interaction with CPU caches. Cache-oblivious algorithms are designed to take advantage of CPU caches without depending on the exact size of the caches (which is highly machine-specific). It would be interesting to consider whether our framework could be generalized to the more realistic—but still machine-independent—cache-oblivious model of computation, and could be used to verify the complexity of cache-oblivious algorithms.

More automation. Our approach is more interactive that automated, which makes it most suitable for the verification of intricate algorithms, but perhaps too heavyweight for simpler programs which are in the scope of automated approaches.

One possibility would be to integrate our framework with RAML, for instance. For programs in the scope of RAML, the automated analysis could be called in order to automatically infer a bound, which could be reused as part of a bigger interactive proof.

Another possibility would be to implement our verification methodology as part of a program verification framework able to leverage general purpose automation, such as Why3 [FP13] or Viper [MSS17] for instance.

Complexity of concurrent programs. Separation Logic has been widely studied as a program logic for reasoning on concurrent programs. It would make sense to consider an extension of Separation Logic with Time Credits to reason on the complexity of concurrent programs.

In fact, the implementation of time credits in Iris by Mével *et al.* [MJP19] can be seen as a first step in that direction: Iris is a concurrent Separation Logic, and their work defines time credits for HeapLang, a concurrent language. In principle, this allows using Time Credits to be used to reason on concurrent programs. However, a first limitation of the work of Mével *et al.* is that it would only allow to measure the *work* of a program (i.e. the total execution time across all threads), but not the *span*, an otherwise useful complexity measure (i.e. the maximum execution time of a thread, across all threads).

In order to account for the span of a concurrent program, it might be enough to consider an extension of Time Credits where credits are annotated with thread identifiers, and where each thread is required to pay credits labelled with its own thread identifier. This ensures that credits are not transferred from one thread to another. Then, the specification of `fork()` would allow giving as many time credits to the child as there are available in the parent.

Another major limitation originates from the presence of locks. For instance, a spinlock is typically implemented as a busy loop performing a compare-and-swap operation. Without additional assumptions on the scheduler, this is not guaranteed to terminate: the compare-and-swap operation might never succeed. In practice, this means that Time Credits require additional assumptions (e.g. about the fairness of the scheduler) to constitute a usable framework for reasoning on the complexity of concurrent programs.

In promising related work, Hoffmann, Marmar and Shao [HMS13] use time credits to establish the lock-freedom of several concurrent data structures. They use credits to express a protocol by which progress in one thread (e.g. a successful CAS operation) must transmit time credits to every other thread that has encountered a corresponding failure and therefore must retry. It would be interesting to consider whether this reasoning can be reproduced in the setting of Mével *et al.*'s work (as mentioned in their Discussion section).

Bibliography

[AAB+14] 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, 2014.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[AAR12] Nicholas Ayache, Roberto M. Amadio, and Yann Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In *Formal Methods for Industrial Critical Systems*, volume 7437 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2012.

[AB98] Mohamad A. Akra and Louay Bazzi. On the solution of linear recurrence equations. *Comp. Opt. and Appl.*, 10(2):195–210, 1998.

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[ABG12] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 405–421, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[ACR18] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *Journal of Formalized Reasoning*, 11(1):43–76, 2018.

[AD04] 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, 2004.

[ADL17] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP):43:1–43:29, August 2017.

[ADLM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 152–164, New York, NY, USA, 2015. ACM.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jouannaud, Jean-Pierre, Shao, and Zhong, editors, *CPP - Certified Programs and Proofs - First International Conference - 2011*, volume 7086 of *Lecture notes in computer science - LNCS*, pages 135–150, Kenting, Taiwan, December 2011. Springer.

[AM13] Martin Avanzini and Georg Moser. A combination framework for complexity. In *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, pages 55–70, 2013.

[AMN19] Mohammad Abdulaziz, Kurt Mehlhorn, and Tobias Nipkow. Trustworthy Graph Algorithms (Invited Talk). In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*, volume 138 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[AR11] Roberto Amadio and Yann Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In *Foundational and Practical Aspects of Resource Analysis*, volume 7177 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2011.

[Atk11] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2:17), 2011.

[BEF+14] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 140–155, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[Ben04] Ralph Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, June 2004.

[BFGT16] 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.

[BGKB19] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. Iron: Managing obligations in higher-order concurrent separation logic. *Proc. ACM Program. Lang.*, 3(POPL):65:1–65:30, January 2019.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: Algebraic bound computation for loops. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 103–118, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BHS80] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, September 1980.

[BKH+08] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.

[BLM15] 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, 2015.

[Bou95] Nicolas Bourbaki. *General Topology, Chapters 1–4*. Springer, 1995.

[Car18] Quentin Carbonneaux. *Modular and Certified Resource-Bound Analyses*. PhD thesis, 2018.

[CCL+18] Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. Formal proofs of Tarjan's algorithm in Why3, Coq, and Isabelle. Manuscript, 2018.

[Cha] Arthur Charguéraud. The `TLC` coq library. https://gitlab.inria.fr/charguer/tlc.

[Cha10a] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, 2010.

[Cha10b] Arthur Charguéraud. The optimal fixed point combinator. In *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2010.

[Cha13] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs, 2013. Unpublished. http://www.charguer aud.org/research/2013/cf/cf.pdf.

[Cha19a] Arthur Charguéraud. The `cfml2` framework, March 2019. https://gitlab.inria.fr/charguer/cfml2.

[Cha19b] Arthur Charguéraud. The CFML tool and library. http://www.charguer aud.org/softs/cfml/, 2019.

[CHRS14] 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, 2014.

[CHRS17] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated resource analysis with Coq proof objects. In *Computer Aided Verification (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 64–85. Springer, 2017.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478, New York, NY, USA, 2015. ACM.

[CL17] Ran Chen and Jean-Jacques Lévy. A semi-automatic proof of strong connectivity. In *Verified Software: Theories, Tools and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2017.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009.

[Coh12] Cyril Cohen. *Formalized algebraic numbers: construction and first-order theory*. PhD thesis, École Polytechnique, 2012.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence 7*, pages 91–99, 1972.

[CP17a] Arthur Charguéraud and François Pottier. Temporary read-only permissions for separation logic. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 260–286. Springer, April 2017.

[CP17b] 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*, 2017.

[cry] cryptanalysis.eu. Effective DoS attacks against web application platforms.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Principles of Programming Languages (POPL)*, pages 184–198, 2000.

[Dan08] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages (POPL)*, 2008.

[DLR15] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151, New York, NY, USA, 2015. ACM.

[DPR13] Norman Danner, Jennifer Paykin, and James S. Royer. A static cost analysis for a higher-order language. In *Programming Languages Meets Program Verification (PLPV)*, pages 25–34, 2013.

[Ebe17] Manuel Eberl. Proving divide and conquer complexities in Isabelle/HOL. *Journal of Automated Reasoning*, 58(4):483–508, 2017.

[Ebe19] Manuel Eberl. Verified solving and asymptotics of linear recurrences. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 27–37, New York, NY, USA, 2019. ACM.

[FK19] Yannick Forster and Fabian Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FL04] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2004.

[FMH14] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 275–295, Cham, 2014. Springer International Publishing.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018.

[GJCP19] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. The verified incremental cycle detection library, March 2019. https://gitlab.inria.fr/agueneau/incremental-cycles.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL)*, pages 127–139, 2009.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, 2017.

[GP18] Armaël Guéneau and François Pottier. The `minicooper` library, November 2018. https://github.com/Armael/minicooper.

[Guéa] Armaël Guéneau. Coq issue #6723. https://github.com/coq/coq/issues/6723.

[Guéb] Armaël Guéneau. Coq issue #6726. https://github.com/coq/coq/issues/6726.

[Gué18a] Armaël Guéneau. Manual of the `procrastination` library, January 2018.

[Gué18b] Armaël Guéneau. The `bigO` library, January 2018. https://gitlab.inria.fr/agueneau/coq-bigO.

[Gué18c] Armaël Guéneau. The `procrastination` library, January 2018. https://github.com/Armael/coq-procrastination.

[Gué19] Armaël Guéneau. Dune pull request #1955, March 2019.

[Gul09] Sumit Gulwani. SPEED: symbolic complexity bound analysis. In *Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009.

[HAH12a] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012.

[HAH12b] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

[Har09] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[HDW17] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*, pages 359–373, 2017.

[HH10] 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, 2010.

[HIH13] Johannes Hölzl, Fabian Immler, and Brian Huffman. Type classes and filters for mathematical analysis in isabelle/hol. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 279–294, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Principles of Programming Languages (POPL)*, pages 185–197, 2003.

[HL19] Maximilian P. L. Haslbeck and Peter Lammich. Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[HM14] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi*, pages 272–286, Cham, 2014. Springer International Publishing.

[HMS13] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Logic in Computer Science (LICS)*, pages 124–133, 2013.

[HN18] 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, 2018.

[Hof99] Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 464–, Washington, DC, USA, 1999. IEEE Computer Society.

[Hop87] John E. Hopcroft. Computer science: The emergence of a discipline. *Communications of the ACM*, 30(3):198–202, 1987.

[How08] Rodney R. Howell. On asymptotic notation with multiple variables. Technical Report 2007-4, Kansas State University, 2008.

[HSAK16] Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Source Code Analysis and Manipulation (SCAM)*, pages 79–84, 2016.

[IHSA19] Nadeem Iqbal, Osman Hasan, Umair Siddique, and Falah Awwd. Formalization of asymptotic notations in hol4. In *IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, 2019.

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages (POPL)*, pages 223–236, 2010.

[JKJ+18] 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, 2018.

[Jou15] Jacques-Henri Jourdan. Coq pull request #89, July 2015.

[KH19] David Kahn and Jan Hoffmann. Exponential automatic amortized resource analysis. Unpublished. https://www.cs.cmu.edu/~janh/assets/pdf/KahnH19.pdf, 2019.

[KRS99] Richard Krueger, Piotr Rudnicki, and Paul Shelley. Asymptotic notation. Part I: theory. In *Journal of Formalized Mathematics*, volume Volume 11, 1999.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures – An algorithmic point of view*. Springer, 2008.

[KTB17] Robert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*, 2017.

[Lam14] Peter Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *Interactive Theorem Proving (ITP)*, volume 8558 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2014.

[Lei96] Tom Leighton. Notes on better master theorems for divide-and-conquer recurrences, 1996.

[LH17] Benjamin Lichtman and Jan Hoffmann. Arrays and References in Resource Aware ML. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:20, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[LM12] Peter Lammich and Rene Meis. A separation logic framework for Imperative HOL. *Archive of Formal Proofs*, 2012.

[LPSS18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 254–265, 2018.

[LS16] Peter Lammich and S. Reza Sefidgar. Formalizing the edmonds-karp algorithm. *Archive of Formal Proofs*, August 2016. http://isa-afp.org/entries/EdmondsKarp_Maxflow.html, Formal proof development.

[MFN+16] 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, 2016.

[MJP19] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In Luis Caires, editor, *European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2019.

[MKK17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Principles of Programming Languages (POPL)*, pages 330–343, 2017.

[MR16] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient flow profiling for detecting performance bugs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 413–424, 2016.

[MSS17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, pages 104–125. 2017.

[NB19] Tobias Nipkow and Hauke Brinkop. Amortized complexity verified. *Journal of Automated Reasoning*, 62(3):367–391, Mar 2019.

[Nie84] H. Riis Nielson. *Hoare Logic's for Run-time Analysis of Programs*. PhD thesis, Edinburgh University, 1984.

[Nie87] Hanne Riis Nielson. A hoare-like proof system for analysing the computation time of programs. *Sci. Comput. Program.*, 9(2):107–136, October 1987.

[Nip15] Tobias Nipkow. Amortized complexity verified. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 2015.

[NKP18] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. Badger: Complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 322–332, 2018.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: an appetizer*. Springer, 2007.

[ODL15] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Programming Language Design and Implementation (PLDI)*, pages 369–378, 2015.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Principles of Programming Languages (POPL)*, pages 247–258, 2005.

[Pot15] François Pottier. Depth-first search and strong connectivity in Coq. In *Journées Françaises des Langages Applicatifs (JFLA)*, 2015.

[PP11] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*, 2011.

[PZKJ17] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, 2017.

[RBG+18] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.*, 2(POPL):36:1–36:32, December 2018.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[Rus94] David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6(4):359–390, 1994.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics (TPHOL)*, pages 278–293, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[ST14] 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, 2014.

[Str18] Jane Street. Dune: A composable build system, 2018.

[SWS+13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, New York, NY, USA, 2013. ACM.

[SZV14] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 745–761, Cham, 2014. Springer International Publishing.

[Tar85] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[Tar87] Robert Endre Tarjan. Algorithmic design. *Communications of the ACM*, 30(3):204–212, 1987.

[Thé05] Laurent Théry. Simplifying Polynomial Expressions in a Proof Assistant. Research Report RR-5614, INRIA, 2005.

[The19a] The Coq development team. *The Coq Manual, Miscellaneous extensions: Program Derivation*, 2019.

[The19b] The Coq development team. *The Coq Proof Assistant*, 2019.

[TPG18] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Code Generation and Optimization (CGO)*, pages 314–326, 2018.

[VH04] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, pages 86–101, Berlin, Heidelberg, 2004. Springer-Verlag.

[vM08] Eelis van der Weegen and James McKinna. A machine-checked proof of the average-case complexity of Quicksort in Coq. In *Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2008.

[Wan18] Peng Wang. *Type System for Resource Bounds with Type-Preserving Compilation*. PhD thesis, 2018.

[WCF+18] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pages 213–223, 2018.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[WWC17] Peng Wang, Di Wang, and Adam Chlipala. TiML: A functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):79:1–79:26, 2017.

[ZGSV11] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In Eran Yahav, editor, *Static Analysis*, pages 280–297, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[ZH17] Théo Zimmermann and Hugo Herbelin. Coq's prolog and application to defining semi-automatic tactics. In *Type Theory Based Tools (TTT)*, 2017.

[ZH18] Bohua Zhan and Maximilian P. L. Haslbeck. Verifying asymptotic time complexity of imperative programs in Isabelle. In *International Joint Conference on Automated Reasoning*, 2018.

[Zha18] Bohua Zhan. Efficient verification of imperative programs using auto2. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 23–40. Springer, 2018.