A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification

Armaël Guéneau, Arthur Charguéraud, François Pottier

A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification

Armaël Guéneau[1], Arthur Charguéraud[1,2], and François Pottier[1⋆]

[1] Inria
[2] Université de Strasbourg, CNRS, ICube UMR 7357

Abstract. We present a framework for simultaneously verifying the functional correctness and the worst-case asymptotic time complexity of higher-order imperative programs. We build on top of Separation Logic with Time Credits, embedded in an interactive proof assistant. We formalize the O notation, which is key to enabling modular specifications and proofs. We cover the subtleties of the multivariate case, where the complexity of a program fragment depends on multiple parameters. We propose a way of integrating complexity bounds into specifications, present lemmas and tactics that support a natural reasoning style, and illustrate their use with a collection of examples.

1 Introduction

A program or program component whose functional correctness has been verified might nevertheless still contain complexity bugs: that is, its performance, in some scenarios, could be much poorer than expected.

Indeed, many program verification tools only guarantee partial correctness, that is, do not even guarantee termination, so a verified program could run forever. Some program verification tools do enforce termination, but usually do not allow establishing an explicit complexity bound. Tools for automatic complexity inference can produce complexity bounds, but usually have limited expressive power.

In practice, many complexity bugs are revealed by testing. Some have also been detected during ordinary program verification, as shown by Filliâtre and Letouzey [14], who find a violation of the balancing invariant in a widely-distributed implementation of binary search trees. Nevertheless, none of these techniques can guarantee, with a high degree of assurance, the absence of complexity bugs in software.

To illustrate the issue, consider the binary search implementation in Figure 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

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

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

Fig. 1. A flawed binary search. This code is provably correct and terminating, yet exhibits linear (instead of logarithmic) time complexity for some input parameters.

might even 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: a search for a value that is stored in the second half of the range $[i, j)$ takes linear time. It would be embarrassing if such faulty code was deployed, as it would aggravate benevolent users and possibly allow malicious users to mount denial-of-service attacks.

As illustrated above, complexity bugs can affect execution time, but could also concern space (including heap space, stack space, and disk space) or other resources, such as the network, energy, and so on. In this paper, for simplicity, we focus on execution time only. That said, much of our work is independent of which resource is considered. We expect that our techniques could be adapted to verify asymptotic bounds on the use of other non-renewable resources, such as the network.

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 this model is very remote from physical running time, it is independent of the compiler, operating system, and hardware [18,24] and still allows establishing asymptotic time complexity bounds, and therefore, detecting complexity bugs—situations where a program is asymptotically slower than it should be.

In prior work [11], the second and third authors 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 Separation Logic with Time Credits, a simple extension of Separation Logic [23] where the assertion \$1 represents a permission to perform one step of computation, and is consumed when exercised. The assertion $\$n$ is a separating conjunction of n such time credits. Separation Logic with Time Credits is implemented in the second author's interactive verification framework, CFML [9,10], which is embedded in the Coq proof assistant.

Using CFML, the second and third authors verify the correctness and time complexity of an OCaml implementation of the Union-Find data structure [11]. 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 $f \in O(\alpha(n))$, calling *find* requires and consumes $\$f(n)$. This is the purpose of this paper.

We argue that the use of asymptotic bounds, such as $O(\alpha(n))$, is necessary for (verified or unverified) 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 *find* costs $2\alpha(n) + 4$ is undesirable: if a tiny modification of the Union-Find module changes this cost to $2\alpha(n) + 5$, then all direct and indirect clients of the Union-Find module must be updated, which is intolerable. Furthermore, sometimes, the constant factors are unknown anyway. Applying the Master Theorem [12] to a recurrence equation only yields an order of growth, not a concrete bound. Finally, for most practical purposes, no critical information is lost when concrete bounds such as $2\alpha(n) + 4$ are replaced with asymptotic bounds such as $O(\alpha(n))$. Indeed, the number of computation steps that take place at the source level is related to physical time only up to a hardware- and compiler-dependent constant factor. The use of asymptotic complexity in the analysis of algorithms, initially advocated by Hopcroft and by Tarjan, has been widely successful and is nowadays standard practice.

One must be aware of several limitations of our approach. First, it is not a worst-case execution time (WCET) analysis: it does not yield bounds on actual physical execution time. Second, it is not fully automated. We place emphasis on expressiveness, as opposed to automation. Our vision is that verifying the functional correctness *and* time complexity of a program, at the same time, should not involve much more effort than verifying correctness alone. Third, we control only the growth of the cost as the parameters grow large. A loop that counts up from 0 to 2^{60} has complexity $O(1)$, even though it typically won't terminate in a lifetime. Although this is admittedly a potential problem, traditional program verification falls prey to analogous pitfalls: for instance, a program that attempts to allocate and initialize an array of size (say) 2^{48} can be proved correct, even though, on contemporary desktop hardware, it will typically fail by lack of memory. We believe that there is value in our approach in spite of these limitations.

Reasoning and working with asymptotic complexity bounds is not as simple as one might hope. As demonstrated by several examples in §2, 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.

A key issue 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* is content with $\$f(n)$, 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 *gradually* construct a cost expression as the proof progresses. In the case of a nonrecursive function, such as *insertionsort*, the cost expression, once fully synthesized, yields the desired witness. In the case of a recursive function, such as *mergesort*, the cost expression yields the body of a recurrence equation, whose solution is the desired witness.

We make the following contributions:

1. 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 [6], that is, a quantifier $\mathbb{U}a.P$. (Eberl [13] does so as well.) We propose a library of lemmas and tactics that can prove nonnegativeness, monotonicity, and domination assertions (§3).
2. 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 (§4). 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.
3. We propose a methodology, supported by a collection of Coq tactics, to prove such specifications (§5). Our tactics, which heavily rely on Coq metavariables, 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.
4. We present several classic examples of complexity analyses (§6), 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 $O(\alpha(n))$.

Our code can be found online in the form of two standalone Coq libraries and a self-contained archive [16].

2 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. Yet, these assertions are too informal: they do not have sufficiently precise meaning, and can be easily abused to produce flawed paper proofs.

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

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

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

- **Case** $n \leq 0$: `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, $O(1)$.

Fig. 2. A flawed proof that `waste`(n) costs $O(1)$, when its actual cost is $O(n)$.

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

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

Flawed proof:

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

Fig. 3. A flawed proof that `f`(n) costs $O(1)$, when its actual cost is $O(n)$.

A striking example appears in Figure 2, which shows how one might "prove" that a recursive function has complexity $O(1)$, whereas its actual cost is $O(n)$. The flawed proof exploits the (valid) relation $O(1) + O(1) = O(1)$, which means that a sequence of two constant-time code fragments is itself a constant-time code fragment. The flaw lies in the fact that the O notation hides an existential quantification, which is inadvertently swapped with the universal quantification over the parameter n. Indeed, the claim is that "there exists a constant c such that, for every n, `waste`(n) runs in at most c computation steps". However, the proposed proof by induction establishes a much weaker result, to wit: "for every n, there exists a constant c such that `waste`(n) runs in at most c steps". This result is certainly true, yet does not entail the claim.

An example of a different nature appears in Figure 3. There, the auxiliary function `g` takes two integer arguments n and m and involves two nested loops, over the intervals $[1, n]$ and $[1, m]$. Its asymptotic complexity is $O(n+nm)$, which, *under the hypothesis that m is large enough*, can be simplified to $O(nm)$. The reasoning, thus far, is correct. The flaw lies in our attempt to substitute 0 for m in the bound $O(nm)$. Because this bound is valid only for sufficiently large m, it does not make sense to substitute a specific value for m. In other words, from the fact that "`g`(n, m) costs $O(nm)$ when n and m are sufficiently large", one *cannot*

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

```
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
```

Flawed proof:

- The body of the outer loop (lines 3-4) 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 $\mathtt{h}(m, n)$ is the sum of the costs of the iterations of the outer loop:

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

Fig. 4. A flawed proof that $\mathtt{h}(m, n)$ costs $O(nm^2)$, when its actual cost is $O(2^n + nm^2)$.

deduce anything about the cost of $\mathtt{g}(n, 0)$. To repair this proof, one must take a step back and prove that $\mathtt{g}(n, m)$ has asymptotic complexity $O(n + nm)$ *for sufficiently large n and for every m.* This fact *can* be instantiated with $m = 0$, allowing one to correctly conclude that $\mathtt{g}(n, 0)$ costs $O(n)$. We come back to this example in §3.3.

One last example of tempting yet invalid reasoning appears in Figure 4. We borrow it from Howell [19]. 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 8) 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 (§4.5) why cost functions should and can be monotonic.

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. In this paper, we do assign O specifications to functions, because we believe that this style is elegant, modular and scalable. However, during the analysis of a function body, we abandon the O notation. We first synthesize a cost expression for the function body, then check that this expression is indeed dominated by the asymptotic bound that appears in the specification.

3 Formalizing the O Notation

3.1 Domination

In many textbooks, the fact that f is bounded above by g asymptotically, up to constant factor, is written "$f = O(g)$" or "$f \in O(g)$". However, the former notation is quite inappropriate, as it is clear that "$f = O(g)$" cannot be literally understood as an equality. Indeed, if it truly were an equality, then, by symmetry and transitivity, $f_1 = O(g)$ and $f_2 = O(g)$ would imply $f_1 = f_2$. The latter notation makes much better sense: $O(g)$ is then understood as a set of functions. This approach has in fact been used in formalizations of the O notation [3]. Yet, in this paper, we prefer to think directly in terms of a *domination* preorder between functions. Thus, instead of "$f \in O(g)$", we write $f \preceq g$.

Although the O notation is often defined in the literature only in the special case of functions whose domain is $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$, we must define domination in the general case of functions whose domain is an arbitrary type A. By later instantiating A with a product type, such as $\mathbb{Z}^k$, we get a definition of domination that covers the multivariate case. Thus, let us fix a type A, and let f and g inhabit the function type $A \to \mathbb{Z}$.[3]

Fixing the type A, it turns out, is not quite enough. In addition, the type A must be equipped with a *filter* [6]. 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".[4] 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 (§3.2) and sum up what has been explained so far:

Definition 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:*

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

[3] At this time, we require the codomain of f and g to be $\mathbb{Z}$. Following Avigad and Donnelly [3], we could allow it to be an arbitrary nondegenerate ordered ring. We have not yet needed this generalization.

[4] 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.

3.2 Filters

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 [6] states that a filter is "a set of subsets of A", it is crucial to note that $\mathcal{P}(\mathcal{P}(A))$ is the type of a quantifier in higher-order logic.

Definition 2 (Filter). *A* filter [6] *on a type A is an object $\mathbb{U}$ of type $\mathcal{P}(\mathcal{P}(A))$ that enjoys the following four properties, where $\mathbb{U}x.P$ is short for $\mathbb{U}\ (\lambda x.P)$:*

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

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

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

3.3 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.

Definition 3 (Universal filter). *Let T be a nonempty type. Then $\lambda Q.\forall x.Q(x)$ 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 4 (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 1 with the filtered type $\mathbb{Z}$, we recover the classic definition of domination between functions of $\mathbb{Z}$ to $\mathbb{Z}$:

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

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

Definition 5 (Product filter). *Let A_1 and A_2 be filtered types. Then*

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

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

To understand this definition, it is useful to consider the special case where A_1 and A_2 are both $\mathbb{Z}$. Then, for $i \in \{1, 2\}$, the formula $\mathbb{U}_{A_i}\, x_i.\ Q_i$ means that the predicate Q_i contains an infinite interval of the form $[a_i, \infty)$. Thus, the formula $\forall x_1, x_2.\ Q_1(x_1) \wedge Q_2(x_2) \Rightarrow Q(x_1, x_2)$ requires the predicate Q to contain the infinite rectangle $[a_1, \infty) \times [a_2, \infty)$. Thus, a predicate Q on $\mathbb{Z}^2$ is "ultimately true" w.r.t. to the product filter if and only if it is "true on some infinite rectangle". In Bourbaki's terminology [6, Chapter 1, §6.7], the infinite rectangles form a *basis* of the product filter.

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

We stress that there are several filters on $\mathbb{Z}$, including the universal filter and the order filter, and therefore several filters on $\mathbb{Z}^k$. Therefore, it does not make sense to use the O notation without specifying which filter one considers. Consider again the function $\mathtt{g}(n, m)$ in Figure 3 (§2). One can prove that $\mathtt{g}(n, m)$ has complexity $O(nm+n)$ with respect to the standard filter on $\mathbb{Z}^2$. With respect to *this filter*, this complexity bound is equivalent to $O(mn)$, as the functions $\lambda(m, n).mn + n$ and $\lambda(m, n).mn$ dominate each other. Unfortunately, this *does not allow* deducing anything about the complexity of $\mathtt{g}(n, 0)$, since the bound $O(mn)$ holds only when n and m grow large. An alternate approach is to prove that $\mathtt{g}(n, m)$ has complexity $O(nm + n)$ with respect to a stronger filter, namely the product of the standard filter on $\mathbb{Z}$ and the universal filter on $\mathbb{Z}$. With respect to *that filter*, the functions $\lambda(m, n).mn + n$ and $\lambda(m, n).mn$ are *not* equivalent. This bound *does allow* instantiating m with 0 and deducing that $\mathtt{g}(n, 0)$ has complexity $O(n)$.

3.4 Properties of Domination

Many properties of the domination relation can be established with respect to an arbitrary filtered type A. Here are two example lemmas; there are many more. As before, f and g range over $A \to \mathbb{Z}$. The operators $f + g$, $\max(f, g)$ and $f.g$ denote pointwise sum, maximum, and product, respectively.

Lemma 6 (Sum and Max Are Alike). *Assume f and g are ultimately nonnegative, that is, $\mathbb{U}_A\, x.\ f(x) \geq 0$ and $\mathbb{U}_A\, x.\ g(x) \geq 0$ hold. Then, we have $\max(f, g) \preceq_A f + g$ and $f + g \preceq_A \max(f, g)$.*

Lemma 7 (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 7 corresponds to Howell's Property 5 [19]. Whereas Howell states this property on $\mathbb{N}^k$, our lemma is polymorphic in the type A. As noted by Howell, this lemma is useful when the cost of a loop body is independent of the loop index. In the case where the cost of the i-th iteration may depend on the loop index i, the following, more complex lemma is typically used instead:

Lemma 8 (Summation). *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

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

Lemma 8 uses the product filter on $A \times \mathbb{Z}$ in its hypothesis and conclusion. It corresponds to Howell's property 2 [19]. The variable i represents the loop index, while the variable a collectively represents all other variables in scope, so the type A is usually instantiated with a tuple type (an example appears in §6).

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 (examples appear in §6). Due to space limitations, we refer the reader to the Coq library for details [16].

3.5 Tactics

Our formalization of filters and domination forms a stand-alone Coq library [16]. In addition to many lemmas about these notions, the library proposes automated tactics that can prove nonnegativeness, monotonicity, and domination goals. These tactics currently support functions built out of variables, constants, sums and maxima, products, powers, logarithms. Extending their coverage is ongoing work. This library is not tied to our application to the complexity analysis of programs. It could have other applications in mathematics.

4 Specifications with Asymptotic Complexity Claims

In this section, we first present our existing approach to verified time complexity analysis. This approach, proposed by the second and third authors [11], does not use the O notation: instead, it involves explicit cost functions. We then discuss how to extend this approach with support for asymptotic complexity claims. We find that, even once domination (§3) is well-understood, there remain nontrivial questions as to the style in which program specifications should be written. We propose one style which works well on small examples and which we believe should scale well to larger ones.

4.1 CFML With Time Credits For Cost Analysis

CFML [9,10] is a system that supports the interactive verification of OCaml programs, using higher-order Separation Logic, inside Coq. It is composed of a trusted standalone tool and a Coq library. The CFML tool transforms a piece of OCaml code into a *characteristic formula*, a Coq formula that describes the semantics of the code. The characteristic formula is then exploited, inside Coq, to state that the code satisfies a certain specification (a Separation Logic triple) and to interactively prove this statement. The CFML library provides a set of Coq tactics that implement the reasoning rules of Separation Logic.

In prior work [11], the second and third authors extend CFML with time credits [2,22] and use it to simultaneously verify the functional correctness and the (amortized) time complexity of OCaml code. To illustrate the style in which they write specifications, consider a function that computes the length of a list:

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

About this function, one can prove the following statement:

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

This is a Separation Logic triple $\{H\}\,(t)\,\{Q\}$. The postcondition $\lambda y.\,[\,y = |l|\,]$ asserts that the call `length` l returns the length of the list l.[5] The precondition $\$(|l|+1)$ asserts that this call requires $|l|+1$ credits. This triple is proved in a variant of Separation Logic where every function call and every loop iteration consumes one credit. Thus, the above specification guarantees that the execution of `length` l involves no more than $|l|+1$ function calls or loop iterations. Our previous paper [11, Def. 2] gives a precise definition of the meaning of triples.

As argued in prior work [11, §2.7], bounding the number of function calls and loop iterations is equivalent, up to a constant factor, to bounding the number of reduction steps of the program. Assuming that the OCaml compiler is

[5] The square brackets denote a pure Separation Logic assertion. $|l|$ denotes the length of the Coq list l. CFML transparently reflects OCaml integers as Coq relative integers and OCaml lists as Coq lists.

complexity-preserving, this is equivalent, up to a constant factor, to bounding the number of instructions executed by the compiled code. Finally, assuming that the machine executes one instruction in bounded time, this is equivalent, up to a constant factor, to bounding the execution time of the compiled code. Thus, the above specification guarantees that `length` runs in linear time.

Instead of understanding Separation Logic with Time Credits as a variant of Separation Logic, one can equivalently view it as standard Separation Logic, applied to an instrumented program, where a `pay()` instruction has been inserted at the beginning of every function body and loop body. The proof of the program is carried out under the axiom $\{\$1\}\,(\texttt{pay()})\,\{\lambda\_.\top\}$, which imposes the consumption of one time credit at every `pay()` instruction. This instruction has no runtime effect: it is just a way of marking where credits must be consumed.

For example, the OCaml function `length` is instrumented as follows:

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

Executing "`length` l" involves executing `pay()` exactly $|l| + 1$ times. For this reason, a valid specification of this instrumented code in ordinary Separation Logic must require at least $|l| + 1$ credits in its precondition.

4.2 A Modularity Challenge

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

After initially publishing a specification that requires $\$(|l| + 1)$, one could of course still switch to the more efficient implementation and update the published specification so as to require $\$(|l| + 2)$ instead of $\$(|l| + 1)$. However, that would in turn require updating the specification and proof of every (direct and indirect) client of the list library, which is intolerable.

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

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

This is a better specification, in the sense that it is more modular. The naïve implementation of `length` shown earlier and the efficient implementation in

OCaml's standard library both satisfy this specification, so one is free to choose one or the other, without any impact on the clients of the list library. In fact, any reasonable implementation of `length` should have linear time complexity and therefore should satisfy this specification.

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

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

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

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

In general, it is up to the user to choose what the parameters of the cost analysis should be, what these parameters represent, and which filter on these parameters should be used. The example of the Bellman-Ford algorithm (§6) illustrates this.

4.3 A Record For Specifications

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

```
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; }.
```

Fig. 5. Definition of `specO`.

This type, `specO`, is defined in Figure 5. The first three fields in this record type correspond to what has been explained so far. The first field asserts the existence of a function `cost` of A to $\mathbb{Z}$, where A is a user-specified filtered type. The second field asserts that a certain property `P cost` is satisfied; it is typically a Separation Logic triple whose precondition refers to `cost`. The third field asserts that `cost` is dominated by the user-specified function `bound`. The need for the last two fields is explained further on (§4.4, §4.5).

Using this definition, our proposed specification of `length` (§4.2) is stated in concrete Coq syntax as follows:

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

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

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

In the next subsections, we explain why, in the definition of `specO`, we require the concrete cost function to be nonnegative and monotonic. These are design decisions; although these properties may not be strictly necessary, we find that enforcing them greatly simplifies things in practice.

4.4 Why Cost Functions Must Be Nonnegative

There are several common occasions where one is faced with the obligation of proving that a cost expression is nonnegative. These proof obligations arise from several sources.

One source is the Separation Logic axiom for splitting credits, whose statement is $\$(m+n) = \$m \star \$n$, subject to the side conditions $m \geq 0$ and $n \geq 0$. Without these side conditions, out of \$0, one would be able to create $\$1 \star \$(-1)$. Because our logic is affine, one could then discard $\$(-1)$, keeping just \$1. In short, an unrestricted splitting axiom would allow creating credits out of thin air.[6] Another source of proof obligations is the Summation lemma (Lemma 8), which requires the functions at hand to be (ultimately) nonnegative.

[6] Another approach would be to define $\$n$ only for $n \in \mathbb{N}$, in which case an unrestricted axiom would be sound. However, as we use $\mathbb{Z}$ everywhere, that would be inconvenient. A more promising idea is to view $\$n$ as linear (as opposed to affine) when n is negative. Then, $\$(-1)$ cannot be discarded, so unrestricted splitting is sound.

Now, suppose one is faced with the obligation of proving that the expression `cost length_spec |xs|` is nonnegative. Because `length_spec` is an existential package (a `specO` record), this is impossible, unless this information has been recorded up front within the record. This is the reason why the field `cost_nonneg` in Figure 5 is needed.

For simplicity, we require cost functions to be nonnegative everywhere, as opposed to within a certain domain. This requirement is stronger than necessary, but simplifies things, and can easily be met in practice by wrapping cost functions within "$\max(0, -)$". Our Coq tactics automatically insert "$\max(0, -)$" wrappers where necessary, making this issue mostly transparent to the user. In the following, for brevity, we write c^+ for $\max(0, c)$, where $c \in \mathbb{Z}$.

4.5 Why Cost Functions Must Be Monotonic

One key reason why cost functions should be monotonic has to do with the "avoidance problem". When the cost of a code fragment depends on a local variable x, can this cost be reformulated (and possibly approximated) in such a way that the dependency is removed? Indeed, a cost expression that makes sense outside the scope of x is ultimately required.

The problematic cost expression is typically 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. Furthermore, an upper bound on $|x|$ is typically available. 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.

To illustrate the problem, 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 =
  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 `length_spec` and `filter_spec` 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 functions `cost length_spec` and `cost filter_spec`. Also, at the mathematical level, let us write $l\downarrow$ 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 a `specO` statement about `count_pos`.

What might this upper bound be? That is, which functions *cost* of $\mathbb{Z}$ to $\mathbb{Z}$ are such that (A) $1 + f(|l|) + g(|l'|) \leq cost(|l|)$ can be proved (in the scope of the local variable l') and (B) $cost \preceq_{\mathbb{Z}} \lambda n.\, n$ holds? Three potential answers come to mind:

1. Within the scope of l', the equality $l' = l{\downarrow}$ is available, as it follows from the postcondition of `filter`. Thus, within this scope, $1 + f(|l|) + g(|l'|)$ is provably equal to $let\ l' = l{\downarrow}\ in\ 1 + f(|l|) + g(|l'|)$, that is, $1 + f(|l|) + g(|l{\downarrow}|)$. 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, $|l{\downarrow}|$, 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. Within the scope of l', the inequality $|l'| \leq |l|$ is available, as it follows from $l' = l{\downarrow}$. Thus, inequality (A) can be proved, provided we take:

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

   Furthermore, for this definition of *cost*, the domination assertion (B) holds as well. The proof relies on the fact the functions g and $\hat{g}$, where $\hat{g}$ is $\lambda n.\ \max_{0 \leq n' \leq n}\ g(n')$ [19], dominate each other. 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:

$$cost = \lambda n.\, 1 + f(n) + g(n)$$

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, which is short for `cost length_spec`, be monotonic. This explains why we build a monotonicity condition in the definition of `specO` (Figure 5, last line). Another motivation for doing so is the fact that some lemmas (such as Lemma 8, which allows reasoning about the asymptotic cost of an inner loop) also have monotonicity hypotheses.

The reader may be worried that, 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. Fortunately, in the common case of functions of $\mathbb{Z}$ to $\mathbb{Z}$, the "running maximum" function $\hat{f}$ can always be used in place of f: indeed, it is monotonic and has the same asymptotic behavior as f. Thus, we see that both approaches 2 and 3 above involve running maxima in some places, but their use seems less frequent with approach 3.

5 Interactive Proofs of Asymptotic Complexity Claims

To prove a specification lemma, such as `length_spec` (§4.3) or `loop_spec` (§4.4), one must construct a `specO` record. By definition of `specO` (Figure 5), this means that one must exhibit a concrete cost function *cost* and prove a number of properties of this function, including the fact that, when supplied with $(*cost* ...)$, the code runs correctly (`cost_spec`) and the fact that *cost* is dominated by the desired asymptotic bound (`cost_dominated`).

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

To illustrate the problem, consider the recursive function `p`, whose integer argument `n` is expected to satisfy $n \geq 0$. For the sake of this example, `p` calls an auxiliary function `g`, which we assume runs in constant time.

```
let rec p n =
  if n <= 1 then () else begin g(); p(n-1) end
```

Suppose we wish to establish that `p` runs in linear time. As argued at the beginning of the paper (§2, Figure 2), it does not make sense to attempt a proof by induction on n that "`p n` runs in time $O(n)$". Instead, in a formal framework, we must exhibit a concrete cost function *cost* such that $cost(n)$ credits justify the call `p n` and *cost* grows linearly, that is, $cost \preceq_{\mathbb{Z}} \lambda n.\, n$.

Let us assume that a specification lemma `g_spec` for the function `g` has been established already, so the number of credits required by a call to `g` is `cost g_spec ()`. In the following, we write G as a shorthand for this constant.

Because this example is very simple, it is reasonably easy to manually come up with an appropriate cost function for `p`. One valid guess is $\lambda n.\ 1+\Sigma_{i=2}^{n}(1+G)$. Another valid guess, obtained via a simplification step, is $\lambda n.\ 1+(1+G)(n-1)^{+}$. Another witness, obtained via an approximation step, is $\lambda n.\ 1 + (1 + G)n^{+}$. 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.

This gives rise to two questions. Among the valid cost functions, which one is preferable? Which ones can be systematically constructed, without guessing?

Among the valid cost functions, there is a tradeoff. At one extreme, a low-level cost function has exactly the same syntactic structure as the code, so it is easy to prove that it is an upper bound for the actual cost of the code, but a lot of work may be involved in proving that it is dominated by the desired asymptotic bound. At the other extreme, a high-level cost function can be essentially identical to the desired asymptotic bound, up to explicit multiplicative and additive constants, so the desired domination assertion is trivial, but a lot of accounting work may be involved in proving that this function represents enough credits to execute the code. Thus, by choosing a cost function, we shift some of the burden of the proof from one subgoal to another. From this point of view, no cost function seems inherently preferable to another.

From the point of view of systematic construction, however, the answer is more clear-cut. It seems fairly clear that it is possible to systematically build a cost function whose syntactic structure is the same as the syntactic structure of the code. This idea goes at least as far back as Wegbreit's work [26]. Coming up with a compact, high-level expression of the cost, on the other hand, seems to require human insight.

To provide as much machine assistance as possible, our system mechanically synthesizes a low-level cost expression for a piece of OCaml code. This is done 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. For instance, the user controls how a local variable should be eliminated, how the cost of a conditional construct should be approximated (i.e., by a conditional or by a maximum), and how recurrence equations should be solved. In the following, we present this semi-interactive synthesis process. We first consider straight-line (nonrecursive) code (§5.1), then recursive functions (§5.2).

5.1 Synthesizing Cost Expressions For Straight-Line Code

The CFML library provides the user with interactive tactics that implement the reasoning rules of Separation Logic. We set things up in such a way that, as these rules are applied, a cost expression is automatically synthesized.

To this end, we use specialized variants of the reasoning rules, whose premises and conclusions take the form $\{\$\, n \star H\}\,(e)\,\{Q\}$. Furthermore, to simplify the nonnegativeness side conditions that must be proved while reasoning, we make all cost expressions obviously nonnegative by wrapping them in $\max(0, -)$. Recall that c^+ stands for $\max(0, c)$, where $c \in \mathbb{Z}$. Our reasoning rules work with triples of the form $\{\$\, c^+ \star H\}\,(e)\,\{Q\}$. They are shown in Figure 6.

Because we wish to *synthesize* a cost expression, our Coq tactics maintain the following invariant: whenever the goal is $\{\$\, c^+ \star H\}\,(e)\,\{Q\}$, the cost c is *uninstantiated*, that is, it is represented in Coq by a metavariable, a placeholder. This metavariable 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.

$$\textsc{WeakenCost}\quad \frac{\{\$\, c_2^+ \star H\}\,(e)\,\{Q\} \qquad c_2^+ \le c_1}{\{\$\, c_1 \star H\}\,(e)\,\{Q\}} \qquad \textsc{Seq}\quad \frac{\{\$\, c_1^+ \star H\}\,(e_1)\,\{Q'\} \qquad \{\$\, c_2^+ \star Q'()\}\,(e_2)\,\{Q\}}{\{\$\,(c_1^+ + c_2^+)^+ \star H\}\,(e_1; e_2)\,\{Q\}}$$

$$\textsc{Let}\quad \frac{\{\$\, c_1^+ \star H\}\,(e_1)\,\{Q'\} \qquad \forall x.\ \{\$\, c_2^+ \star Q'(x)\}\,(e_2)\,\{Q\}}{\{\$\,(c_1^+ + c_2^+)^+ \star H\}\,(\text{let } x = e_1 \text{ in } e_2)\,\{Q\}} \qquad \textsc{Val}\quad \frac{H \Vdash Q(v)}{\{\$\, 0^+ \star H\}\,(v)\,\{Q\}}$$

$$\textsc{If}\quad \frac{\begin{array}{c} b = \text{true} \Rightarrow \{\$\, c_1^+ \star H\}\,(e_1)\,\{Q\} \\ b = \text{false} \Rightarrow \{\$\, c_2^+ \star H\}\,(e_2)\,\{Q\} \end{array}}{\{\$\,(\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2)^+ \star H\}\,(\text{if } b \text{ then } e_1 \text{ else } e_2)\,\{Q\}} \qquad \textsc{Pay}\quad \frac{H \Vdash Q()}{\{\$\, 1^+ \star H\}\,(\text{pay}())\,\{Q\}}$$

$$\textsc{For}\quad \frac{\forall i.\ a \le i < b \ \Rightarrow\ \{\$\, c(i)^+ \star I(i)\}\,(e)\,\{I(i+1)\} \qquad H \Vdash I(a) \star Q}{\{\$\,(\Sigma_{a \le i < b}\ c(i)^+)^+ \star H\}\,(\text{for } i = a \text{ to } b - 1 \text{ do } e \text{ done})\,\{I(b) \star Q\}}$$

Fig. 6. The reasoning rules of Separation Logic, specialized for cost synthesis.

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 $-^+$ wrapper and because c_1 is not necessarily a metavariable, WeakenCost gives rise to a subgoal $\{\$\, c_2^+ \star H\}\,(e)\,\{Q\}$ 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. It is typically used just before leaving the scope of a local variable x to approximate a cost expression c_2^+ that depends on x with an expression c_1 that does not refer to x.

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 $\{\$\, c^+ \star H\}\,(e)\,\{Q\}$, 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. Naturally, Coq enforces this condition: any attempt to instantiate the metavariable c_2 with an expression where x occurs fails. In such a situation, it is up to the user to use WeakenCost so as to avoid this dependency. The example of `count_pos` (§4.5) illustrates this issue.

The Val rule handles values, which in our model have zero cost. The symbol $\Vdash$ denotes entailment between Separation Logic assertions.

The If rule states that the cost of an OCaml conditional expression is a mathematical conditional expression. Although this may seem obvious, one subtlety lurks here. Using WeakenCost, the cost expression *if* b *then* c_1 *else* c_2 can be approximated by $\max(c_1, c_2)$. Such an approximation can be beneficial, as it leads to a simpler cost expression, or harmful, as it causes a loss of information. In particular, when carried out in the body of a recursive function, it can

lead to an unsatisfiable recurrence equation. We let the user decide whether this approximation should be performed.

The Pay rule handles the `pay()` instruction, which is inserted by the CFML tool at the beginning of every function and loop body (§4.1). This instruction costs one credit.

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. In practice, it is typically used in conjunction with WeakenCost, which allows the user to simplify and approximate the iterated sum $\Sigma_{a \leq i < b}\ c(i)^+$. In particular, if the synthesized cost $c(i)$ happens to not depend on i, or can be approximated so as to not depend on i, then this iterated sum can be expressed under the form $c(b-a)^+$. A variant of the For rule, not shown, covers this common case. There is in principle no need for a primitive treatment of loops, as loops can 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.

This concludes our exposition of the reasoning rules of Figure 6. Coming back to the example of the OCaml function `p` (§5), under the assumption that the cost of the recursive call `p(n-1)` is $f(n-1)$, we are able, by repeated application of the reasoning rules, to automatically find that the cost of the OCaml expression:

```
if n <= 1 then () else begin g(); p(n-1) end
```

is: $1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1))$. The initial 1 accounts for the implicit `pay()`. This may seem obvious, and it is. The point is that this cost expression is automatically constructed: its synthesis adds no overhead to an interactive proof of functional correctness of the function `p`.

5.2 Synthesizing and Solving Recurrence Equations

There now remains to explain how to deal with recursive functions. Suppose $S(f)$ is the Separation Logic triple that we wish to establish, where f stands for an as-yet-unknown cost function. Following common informal practice, we would like to do this 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 `p` (§5), we prove $S(f)$ by induction over n, under the hypothesis $n \geq 0$. Thus, we assume that the cost of the recursive call `p(n-1)` is $f(n-1)$, and must prove that the cost of `p n` is $f(n)$. We synthesize the cost of `p n` as explained earlier (§5.1) and find that this cost is $1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1))$. We apply WeakenCost and find that our proof is complete, provided we are able to prove the following inequation:

$$1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1)) \ \leq\ f(n)$$

We achieve that simply by instantiating E as follows:

$$E := \lambda f.\, \forall n.\, n \geq 0 \ \rightarrow\ 1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1)) \ \leq\ f(n)$$

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. There are at least three approaches to solving such a recurrence.

First, one can guess a closed form that satisfies the recurrence. For example, the function $f := \lambda n.\, 1 + (1+G)n^+$ satisfies $E(f)$ above. But, as argued earlier, guessing is in general difficult and tedious.

Second, one can invoke Cormen *et al.*'s Master Theorem [12] or the more general Akra–Bazzi theorem [21,1]. Unfortunately, at present, these theorems are not available in Coq, although an Isabelle/HOL formalization exists [13].

The last approach is Cormen *et al.*'s substitution method [12, §4]. 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. In the above example, as we expect the code to have linear time complexity, we propose that the solution f should have the shape $\lambda n.(an^+ + b)$, where a and b are parameters, about which we wish to gradually accumulate a set of constraints. From a formal point of view, this amounts to applying the following tautology:

$$\forall P.\, \forall C.\quad (\forall ab.\, C(a,b) \rightarrow P(\lambda n.(an^+ + b))) \ \rightarrow\ (\exists ab.\, C(a,b)) \ \rightarrow\ \exists f.P(f)$$

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 and b. During the proof of the second subgoal, C is fixed and we must check that it is satisfiable. In our example, the first subgoal is:

$$E(\lambda n.(an^+ + b)) \quad \wedge \quad \lambda n.(an^+ + b) \ \preceq_{\mathbb{Z}}\ \lambda n.n$$

The second conjunct is trivial. The first conjunct simplifies to:

$$\forall n.\quad n \geq 0 \ \rightarrow\ 1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + a(n-1)^+ + b) \ \leq\ an^+ + b$$

By distinguishing the cases $n = 0$, $n = 1$, and $n > 1$, we find that this property holds provided we have $1 \leq b$ and $1 \leq a + b$ and $1 + G \leq a$. Thus, we prove this subgoal by instantiating C with $\lambda(a, b).(1 \leq b \wedge 1 \leq a + b \wedge 1 + G \leq a)$.

There remains to check the second subgoal, that is, $\exists ab.C(a, b)$. This is easy; we pick, for instance, $a := 1 + G$ and $b := 1$. This concludes our use of Cormen *et al.*'s substitution method.

In summary, by exploiting Coq's metavariables, 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. In practice, instead of explicitly building and applying tautologies as above, we use the first author's `procrastination` library [16], which provides facilities for introducing new parameters, gradually gathering constraints on these parameters, and eventually checking that these constraints are satisfiable.

6 Examples

Binary Search. We prove that binary search has time complexity $O(\log n)$, where $n = j - i$ denotes the width of the search interval $[i, j)$. The code is as in Figure 1, except that the flaw is fixed by replacing `i+1` with `k+1` on the last line. As outlined earlier (§5), we synthesize the following recurrence equation on the cost function f:

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

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

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

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

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

A higher-level approach is to exploit the fact that, if f is monotonic, then $\sum_{i=1}^{n} f(i)$ is less than $n.f(n)$. Applying this lemma twice, we find that the above

cost function is less than $\lambda n.\, \sum_{i=1}^{n}(1+i)$ which is less than $\lambda n.n(1+n)$ which is dominated by $\lambda n.n^2$. This simple-minded approach, which does not require the Summation lemma (Lemma 8), is often applicable. The next example illustrates a situation where the Summation lemma is required.

A Loop Whose Body Has Exponential Cost. In the following simple example, the loop body is just a function call:

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

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

For this loop, the cost function $\lambda n.\ \sum_{i=0}^{n}(1 + \texttt{cost b}\ (i))$ is automatically synthesized. We have an asymptotic bound for the cost of the loop body, namely: $\lambda i.\ 1 + \texttt{cost b}\ (i) \preceq_{\mathbb{Z}} \lambda i.\ 2^i$. The side conditions of the Summation lemma (Lemma 8) are met: in particular, the function $\lambda i.\ 1 + \texttt{cost b}\ (i)$ is monotonic. The lemma yields $\lambda n.\ \sum_{i=0}^{n}(1 + \texttt{cost b}\ (i)) \preceq_{\mathbb{Z}} \lambda n.\ \sum_{i=0}^{n} 2^i$. Finally, we have $\lambda n.\ \sum_{i=0}^{n} 2^i = \lambda n.\ 2^{n+1} - 1 \preceq_{\mathbb{Z}} \lambda n.\ 2^n$.

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

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

By exploiting the fact that a graph without duplicate edges must satisfy $m \leq n^2$, 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 a composition lemma (§3.4). In practice, we publish both specifications and let clients use whichever one is more convenient.

Union-Find. Charguéraud and Pottier [11] 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. With a few lines of proof, we derive a specification where the cost of `find` is expressed under the form $O(\alpha(n))$:

```
specO Z_filterType Z.le (fun n ⇒ alpha n) (fun cost ⇒
  ∀D R V x, x \in D → triple (UnionFind_ml.find x)
    PRE (UF D R V ⋆ $(cost (card D)))
    POST (fun y ⇒ UF D R V ⋆ [R x = y ])).
```

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

7 Related Work

Our work builds on top of Separation Logic [23] with Time Credits [2], which has been first implemented in a verification tool and exploited by the second and third authors [11]. We refer the reader to their paper for a survey of the related work in the general area of formal reasoning about program complexity, including approaches based on deductive program verification and approaches based on automatic complexity analysis. In this section, we restrict our attention to informal and formal treatments of the O notation.

The O notation and its siblings are documented in several textbooks [15,7,20]. Out of these, only Howell [19,20] draws attention to the subtleties of the multivariate case. He shows that one cannot take for granted that the properties of the O notation, which in the univariate case are well-known, remain valid in the multivariate case. He states several properties which, at first sight, seem natural and desirable, then proceeds to show that they are inconsistent, so no definition of the O notation can satisfy them all. He then proposes a candidate notion of domination between functions whose domain is $\mathbb{N}^k$. His notation, $f \in \hat{O}(g)$, is defined as the conjunction of $f \in O(g)$ and $\hat{f} \in O(\hat{g})$, where the function $\hat{f}$ is a "running maximum" of the function f, and is by construction monotonic. He shows that this notion satisfies all the desired properties, provided some of them are restricted by additional side conditions, such as monotonicity requirements.

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

Avigad and Donnelly [3] formalize the O notation in Isabelle/HOL. They consider functions of type $A \to B$, where A is arbitrary and B is an ordered ring. Their definition of "$f = O(g)$" requires $|f(x)| \leq c|g(x)|$ for every x, as opposed to "when x is large enough". Thus, they get away without equipping the type A with a filter. The price to pay is an overly restrictive notion of domination, except in the case where A is $\mathbb{N}$, where both $\forall x$ and $\mathbb{U}x$ yield the same notion of domination—this is Brassard and Bratley's "threshold rule" [7]. Avigad and Donnelly suggest defining "$f = O(g)$ eventually" as an abbreviation

for $\exists f', (f' = O(g) \wedge \mathbb{U}x.f(x) = f'(x))$. In our eyes, this is less elegant than parameterizing O with a filter in the first place.

Eberl [13] formalizes the Akra–Bazzi method [1,21], a generalization of the well-known Master Theorem [12], in Isabelle/HOL. He creates a library of Landau symbols specifically for this purpose. Although his paper does not mention filters, his library in fact relies on filters, whose definition appears in Isabelle's Complex library. Eberl's definition of the O symbol is identical to ours. That said, because he is concerned with functions of type $\mathbb{N} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$, he does not define product filters, and does not prove any lemmas about domination in the multivariate case. Eberl sets up a decision procedure for domination goals, like $x \in O(x^3)$, as well as a procedure that can simplify, say, $O(x^3+x^2)$ to $O(x^3)$.

TiML [25] is a functional programming language where types carry time complexity annotations. Its type-checker generates proof obligations that are discharged by an SMT solver. The core type system, whose metatheory is formalized in Coq, employs concrete cost functions. The TiML implementation allows associating a O specification with each toplevel function. An unverified component recognizes certain classes of recurrence equations and automatically applies the Master Theorem. For instance, *mergesort* is recognized to be $O(mn \log n)$, where n is the input size and m is the cost of a comparison. The meaning of the O notation in the multivariate case is not spelled out; in particular, which filter is meant is not specified.

Boldo *et al.* [4] use Coq to verify the correctness of a C program which implements a numerical scheme for the resolution of the one-dimensional acoustic wave equation. They define an ad hoc notion of "uniform O" for functions of type $\mathbb{R}^2 \to \mathbb{R}$, which we believe can in fact be viewed as an instance of our generic definition of domination, at an appropriate product filter. Subsequent work on the Coquelicot library for real analysis [5] includes general definitions of filters, limits, little-o and asymptotic equivalence. A few definitions and lemmas in Coquelicot are identical to ours, but the focus in Coquelicot is on various filters on $\mathbb{R}$, whereas we are more interested in filters on $\mathbb{Z}^k$.

The tools RAML [17] and Pastis [8] perform fully automated amortized time complexity analysis of OCaml programs. They can be understood in terms of Separation Logic with Time Credits, under the constraint that the number of credits that exist at each program point must be expressed as a polynomial over the variables in scope at this point. The a priori unknown coefficients of this polynomial are determined by an LP solver. Pastis produces a proof certificate that can be checked by Coq, so the trusted computing base of this approach is about the same as ours. RAML and Pastis offer much stronger automation than our approach, but have weaker expressive power. It would be very interesting to offer access to a Pastis-like automated system within our interactive system.

References

1. Akra, M.A., Bazzi, L.: On the solution of linear recurrence equations. Comp. Opt. and Appl. 10(2), 195–210 (1998)

2. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science 7(2:17) (2011)
3. Avigad, J., Donnelly, K.: Formalizing O notation in Isabelle/HOL. In: International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 3097, pp. 357–371. Springer (Jul 2004)
4. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. Journal of Automated Reasoning 50(4), 423–456 (Apr 2013)
5. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A user-friendly library of real analysis for Coq. Mathematics in Computer Science 9(1), 41–62 (Mar 2015)
6. Bourbaki, N.: General Topology, Chapters 1–4. Springer (1995)
7. Brassard, G., Bratley, P.: Fundamentals of algorithmics. Prentice Hall (1996)
8. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated resource analysis with Coq proof objects. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 10427, pp. 64–85. Springer (Jul 2017)
9. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: International Conference on Functional Programming (ICFP). pp. 418–430 (Sep 2011)
10. Charguéraud, A.: The CFML tool and library. `http://www.chargueraud.org/softs/cfml/` (2016)
11. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. Journal of Automated Reasoning (Sep 2017)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (Third Edition). MIT Press (2009)
13. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. Journal of Automated Reasoning 58(4), 483–508 (2017)
14. Filliâtre, J.C., Letouzey, P.: Functors for proofs and programs. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 2986, pp. 370–384. Springer (Mar 2004)
15. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete mathematics: a foundation for computer science. Addison-Wesley (1994)
16. Guéneau, A., Charguéraud, A., Pottier, F.: Electronic appendix (Jan 2018), `http://gallium.inria.fr/~agueneau/bigO/`
17. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for OCaml. In: Principles of Programming Languages (POPL). pp. 359–373 (Jan 2017)
18. Hopcroft, J.E.: Computer science: the emergence of a discipline. Communications of the ACM 30(3), 198–202 (1987)
19. Howell, R.R.: On asymptotic notation with multiple variables. Tech. Rep. 2007-4, Kansas State University (Jan 2008)
20. Howell, R.R.: Algorithms: A top-down approach (Jul 2012), draft.
21. Leighton, T.: Notes on better master theorems for divide-and-conquer recurrences (1996)
22. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Types in Language Design and Implementation (TLDI) (Jan 2011)
23. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)
24. Tarjan, R.E.: Algorithm design. Communications of the ACM 30(3), 204–212 (1987)

25. Wang, P., Wang, D., Chlipala, A.: TiML: A functional language for practical complexity analysis with invariants. Proceedings of the ACM on Programming Languages 1(OOPSLA), 79:1–79:26 (Oct 2017)
26. Wegbreit, B.: Mechanical program analysis. Communications of the ACM 18(9), 528–539 (Sep 1975)