Formal Proof and Analysis of an Incremental Cycle Detection Algorithm

Anonymous

Abstract. We exploit Separation Logic with Time Credits to verify the correctness and worst-case amortized asymptotic complexity of a state-of-the-art incremental cycle detection algorithm.

1 Introduction

A good algorithm must be correct. Yet, to err is human: algorithm designers and algorithm implementors sometimes make mistakes. 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 an implementation, a specification, and a machine-checked proof that the algorithm always produces a result that the specification permits.

A growing number of verified algorithms appear in the literature. To cite just a very few examples, in the area of graph algorithms, Lammich and Neumann [27,26] verify a generic depth-first search algorithm which, among other applications, can be used to detect a cycle in a directed graph; Lammich [25], Pottier [37], and Chen *et al.* [13,12] verify various algorithms for finding the strongly connected components of a directed graph. A verified algorithm can serve as a building block in the construction of larger verified software: for instance, Esparza *et al.* [15] use a cycle detection algorithm as a component in a verified LTL model-checker.

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 existence of these algorithms is their alleged efficiency, not only their correctness, but also their complexity, should arguably be verified.

In this paper, we verify the correctness and worst-case amortized asymptotic complexity of an incremental cycle detection algorithm that has been recently published by Bender *et al.* [7, §2]. With this algorithm, the complexity of building a directed graph of n vertices and m edges, while incrementally ensuring that no edge insertion creates a cycle, is $O(m \cdot \min(m^{1/2}, n^{2/3}))$. Although the implementation of this algorithm is relatively straightforward, its design is quite subtle, and it is far from obvious, by inspection of the code, that the advertised complexity bound is respected.

In fact, we propose and verify a slightly enhanced version of Bender *et al.*'s algorithm. To handle the insertion of a new edge, the original algorithm depends on a runtime parameter, which limits the extent of a certain backward search. This parameter influences only the algorithm's complexity, not its correctness. Bender *et al.* show that setting it to $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm allows achieving the advertised complexity. Yet, this means that, in order to run the algorithm, one must anticipate the *final* values of m and n. This seems awkward, as one often 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 2; it is explained later on (§4). The modified algorithm has the same complexity as the original algorithm and can be used online.

We implement our algorithm in OCaml [29]. The code appears in Figure 3. (We omit several auxiliary functions that deal with low-level data representation; see §5.) Beyond its algorithmic interest and beyond the challenge that its verification represents, a motivation for verifying this code is that we have deployed a generalized version of it in the kernel of the Coq proof assistant [43], where it is used to check the satisfiability of universe constraints [39, §2]. There, reliability and high performance are crucial. Switching from a previous naïve algorithm to this algorithm yields a dramatic improvement in the overall performance of Coq's proof checker: for instance, the total time required to check the Mathematical Components library drops from 25 to 18 minutes [3]. Thus, we view the present work as an important step towards verifying Coq's universe inference system.

We verify our OCaml code using CFML [9,10], an implementation inside Coq of Separation Logic [38] with Time Credits [6,11,19,20,33]. This program logic makes it possible to simultaneously verify the correctness and the complexity of an algorithm, and allows the complexity argument to depend on properties whose validity is established as part of the correctness argument. We provide additional background on Separation Logic with Time Credits in Section 2.

Following traditional practice in the algorithms literature [24,42], we focus on asymptotic complexity, as opposed to physical worst-case execution time. While a bound on physical execution time would be of interest in real-time applications, it would be difficult to establish, and fragile, as it would be highly dependent on the compiler, the runtime system, and the hardware. In contrast, an asymptotic complexity bound allows reasoning at the level of source code, and is robust: a minor change in the implementation, which does not alter the algorithm's asymptotic complexity, requires no change in the specification.

In summary, our contribution is as follows. We consider a state-of-the-art incremental cycle detection algorithm, which we slightly improve and implement in OCaml. We then exploit Separation Logic with Time Credits to simultaneously verify the functional correctness and worst-case amortized asymptotic complexity of this implementation. Our proofs are checked in Coq [4]. Their size is about 5Kloc and they represent roughly a 6 man-month effort. We believe that this approach can be used to verify the correctness and complexity of many other nontrivial data structures and algorithms.

2 Separation Logic with Time Credits

Hoare Logic [21] allows verifying the correctness of an imperative algorithm by using *assertions* to describe the state of the program. Separation Logic [38] improves modularity by employing assertions that describe only a fragment of the state and at the same time assert the unique ownership of this fragment. In general, a Separation Logic assertion claims the ownership of certain *resources*, and (at the same time) describes the current state of these resources. A heap fragment is an example of a resource.

Separation Logic with Time Credits [6] is a simple extension of Separation Logic in which "a permission to perform one computation step" is also a resource, known as a *credit*. The assertion \$1 represents the unique ownership of one credit. CFML enforces the rule that every function call consumes one credit: thus, the "computation steps" that are counted are the function calls, and nothing else. Under mild assumptions [11], the number of function calls performed by the source code is an adequate measure of the asymptotic complexity of the compiled code.

Credits do not exist at runtime; they appear only in assertions, such as pre- and postconditions, loop invariants, and data structure invariants. For instance, the Separation Logic triple:

$$\forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\, G \ast \$(3\,|\mathrm{edges}\ G| + 5) \,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\, G \,\}$$

can be read as follows. If initially g is a runtime representation of the graph G and if $3m + 5$ credits are at hand, where m is the number of edges of G, then the function call $\mathit{dfs}(g)$ executes safely and terminates; after this call, g remains a valid representation of G, and no credits remain.

In the *dfs* example, assuming that the assertion $\mathrm{IsGraph}\, g\, G$ is credit-free (which means, roughly, that this assertion definitely does not own any credits), the precondition guarantees the availability of $3m + 5$ credits (and no more), and no credits remain in the postcondition. So, this triple guarantees that the execution of $\mathit{dfs}(g)$ involves at most $3m + 5$ computation steps. Later on in this paper (§6), we define $\mathrm{IsGraph}\, g\, G$ in such a way that it is *not* credit-free: its definition involves a nonnegative number of credits. If that were the case in the above example, then $3m + 5$ would have to be interpreted as an amortized bound. Amortization is discussed in greater depth in the next section (§3).

Admittedly, $3m + 5$ is too low-level a bound: it would be preferable to state that the cost of $\mathit{dfs}(g)$ is $O(m)$, a more abstract and more robust specification. Following Guéneau *et al.* [19], this can be expressed in the following style:

$$\begin{aligned}\exists(f : \mathbb{Z} \to \mathbb{Z}).\quad & \mathrm{nonnegative}\ f\ \wedge\ \mathrm{monotonic}\ f\ \wedge\ f \preceq_{\mathbb{Z}} \lambda m.m \\ & \wedge\ \forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\, G \ast \$\, f(|\mathrm{edges}\ G|) \,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\, G \,\}\end{aligned}$$

The concrete function $\lambda m.(3m{+}5)$ is no longer visible; it has been abstracted away under the name f. The specification states that f is nonnegative ($\forall m.\ f(m) \geq 0$), monotonic ($\forall m m'.\ m \leq m' \Rightarrow f(m) \leq f(m')$), and dominated by the function $\lambda m.m$, which means that f grows linearly.

The soundness of Separation Logic with Time Credits stems from the fact that a credit cannot be spent twice. Technically, credits obey three laws. The first law asserts that zero credit is equivalent to no resource: $\$0 \equiv \mathit{true}$. The second law asserts that credits are additive: $\$(m+n) \equiv \$m * \$n$. In this law, m and n range over $\mathbb{Z}$. There is no requirement that m and n be nonnegative, so, out of thin air, one can create a credit and a debt: the entailment $\$0 \vdash \$1 * \$(-1)$ holds. The third law asserts that a nonnegative amount of credits can be discarded: $\forall n \geq 0.\ \$n \vdash \mathit{true}$. The side condition $n \geq 0$ guarantees that a debt cannot be forgotten. (Allowing negative credits to appear is a contribution of this paper; see §A.) The soundness metatheorem for Separation Logic with Time Credits guarantees that, for every valid Hoare triple, the following inequality holds:

$$\text{credits in precondition} \;\geq\; \text{steps taken} \;+\; \text{credits in postcondition}.$$

This metatheorem is proved by Charguéraud and Pottier [11, §3] for a Separation Logic with nonnegative credits. We adapt their proof to a setting where negative credits can exist (§A).

3 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 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 two variables, namely IsGraph 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 [36]. 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 $\exists L.\ \$\phi(G, L) * \ldots$ (§6), where ϕ is a suitable potential function [41]. ϕ is parameterized by the graph G and by a map L of vertices to levels. Intuitively,

InitGraph
$\exists k.\ \{\$k\}\ \texttt{init\_graph()}\ \{\lambda g.\mathrm{IsGraph}\ g\ \varnothing\}$

DisposeGraph
$\forall g\, G.\ \mathrm{IsGraph}\ g\ G \Vdash \mathit{true}$

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

AddEdge
$\forall g\, G\, v\, w.$
$\text{let } m, n := |\text{edges } G|, |\text{vertices } G| \text{ in}$
$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$
$\left\{ \begin{array}{l} \mathrm{IsGraph}\ g\ G\ * \\ \$(\psi\,(m+1, n) - \psi\,(m, n)) \end{array} \right\}$
$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$
$\left\{ \begin{array}{l} \lambda res.\ \text{match } res \text{ with} \\ \quad \mid \mathrm{Ok} \Rightarrow \mathrm{IsGraph}\ g\ (G + (v, w)) \\ \quad \mid \mathrm{Cycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}$

Acyclicity
$\forall g\, G.\ \mathrm{IsGraph}\ g\ G \Vdash$
$\quad \mathrm{IsGraph}\ g\ G\ *\ [\forall x.\ x \not\longrightarrow_G^+ x]$

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

Fig. 1. Specification of an incremental cycle detection algorithm.

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 1. Yet, the fact that $\mathrm{IsGraph}\ g\ G$ can involve credits means that the user must read AddVertex and AddEdge as amortized specifications [41]: the actual cost of 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 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 build a graph of n vertices and m vertices. 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 it is permitted to drop the assertion $\mathrm{IsGraph}\ g\ G$, that is, 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 (§2).

AddVertex states that `add_vertex` requires a valid data structure, described by the assertion $\mathrm{IsGraph}\ g\ G$, and returns a valid data structure, described by $\mathrm{IsGraph}\ g\ (G + v)$. (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.) 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. (The proposition $w \longrightarrow_G^* v$ appears within square brackets, which convert an ordinary proposition to a Separation Logic assertion.) 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 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 $\{\$\psi(m, n)\}\ \ldots\ \{true\}$. Because Separation Logic with Time Credits is sound, this implies that the actual worst-case cost of this sequence of operations is $\psi(m, n)$. This confirms our earlier informal claim that $\psi(m, n)$ represents the cost of building a graph of 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 valid state, the graph G remains acyclic.

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, the relation $\preceq_{\mathbb{Z}\times\mathbb{Z}}$ is a domination relation between functions of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ [19]. Our complexity bound thus matches the one published by Bender *et al.* [7].

4 Overview and Informal Analysis of the Algorithm

We provide pseudocode for Bender *et al.*'s algorithm [7, §2] and for our improved algorithm in Figure 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.

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.

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
    - $\rightarrow$ in Bender et al.'s algorithm, F is a constant Δ
    - $\rightarrow$ 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

Fig. 2. Pseudocode for Bender *et al.*'s algorithm and for our improved algorithm.

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 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 further on) 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 explained earlier (§1), though, 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 $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

Informal analysis. We now present an informal complexity analysis of Bender *et al.*'s original algorithm. In this algorithm, the parameter F is fixed: it remains constant throughout the execution of the algorithm. Under this hypothesis, the following invariant holds: for every level k except the highest level, there exist at least F horizontal edges at level k (edges whose endpoints are both at level k). A proof is given in Appendix B.

From this invariant, one can derive two upper bounds on the number of levels. Let K denote the number of nonterminal levels. 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, which therefore must have at least $\sqrt{F}$ vertices. In other words, 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 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 cost of a forward search is linear in the number of edges whose source vertex is promoted to a higher level. 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 $O(m \cdot (F + \min(m/F, n/\sqrt{F}))$. A mathematical analysis (§B) shows that setting F to Δ, where Δ is defined as $\min(m^{1/2}, n^{2/3})$, leads to the asymptotic bound $O(m \cdot \min(m^{1/2}, n^{2/3}))$. 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. 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.

5 Implementation

Our OCaml code, shown in Figure 3, relies on auxiliary operations whose implementation belongs in a lower layer. We do not prescribe how they should be implemented and what data structures 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.

For brevity, we do not give the specifications of these auxiliary operations. Instead, we list them and briefly describe what they are supposed to do. Each of them is required to have constant time complexity.

To update the graph, the algorithm requires the ability to create new vertices (`create_vertex`) and to create new edges (`add_edge`). To avoid creating duplicate edges, it must be able to test the equality of two vertices (`vertex_eq`).

The backward search requires the ability to efficiently enumerate the horizontal incoming edges of a vertex (`get_incoming`). The collection of horizontal incoming edges of a vertex y is updated during a forward search. It is reset when the level of y is increased (`clear_incoming`). An edge is added to it when a horizontal edge from x to y is traversed (`add_incoming`). The backward search also requires the ability to generate a fresh mark (`new_mark`), to mark a vertex (`set_mark`), and to test whether a vertex is marked (`is_marked`). These marks are consulted also during the forward search.

The forward search requires the ability to efficiently enumerate the outgoing edges of a vertex (`get_outgoing`). It also reads and updates the level of certain vertices (`get_level`, `set_level`).

Several choices arise in the implementation of graph search. First, the frontier 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 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 (§4), reaches zero.

The main public entry point of the algorithm is `add_edge_or_detect_cycle`, whose specification was presented in Figure 1. The other two public functions, `init_graph` and `add_vertex`, are trivial; they are not shown.

```
let rec visit_backward g target mark fuel stack =
  match stack with
  | [] -> VisitBackwardCompleted
  | x :: stack ->
    let (stack, fuel), interrupted = interruptible_fold (fun y (stack, fuel) ->
      if fuel = 0 then Break (stack, -1)
      else if vertex_eq y target then Break (stack, fuel)
      else if is_marked g y mark then Continue (stack, fuel - 1)
      else (set_mark g y mark; Continue (y :: stack, fuel - 1))
    ) (get_incoming g x) (stack, fuel) in
    if interrupted
    then if fuel = -1 then VisitBackwardInterrupted else VisitBackwardCyclic
    else 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 stack, interrupted = interruptible_fold (fun y stack ->
      if is_marked g y mark then Break stack
      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
    if interrupted then ForwardCyclic
    else 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; 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 ()
```

Fig. 3. OCaml implementation of the verified incremental cycle detection algorithm.

6 Data Structure Invariants

As explained earlier (§3), the specification of the algorithm refers to two variables, IsGraph and ψ, which must be regarded as abstract by a client. Figure 4 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.

Low-level data structure. The conjunct IsRawGraph g G L M I asserts that there is a data structure at address g in memory, claims the unique ownership of this data structure, and summarizes the information that is recorded in this structure. The parameters G, L, M, 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; and I is a map of vertices to sets of vertices, describing horizontal incoming edges. The parameters L, M and I are existentially quantified in the definition of IsGraph, indicating that they are internal data whose existence is not exposed to the user.

Functional invariant. The second conjunct, [Inv G L I], is a pure proposition that relates the graph G with the maps L and I. Its definition appears next in Figure 4. 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. 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 (§4). It asserts that "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. The vertices in the set E may disobey this property, which is temporarily broken during a forward search.

Potential. The last conjunct in the definition of IsGraph is $\$\phi(G, L)$. This is a *potential* [41], 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. $\phi(G, L)$ 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. "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.

$$\text{IsGraph}\, g\, G \;:=\; \exists L\, M\, I.\ \text{IsRawGraph}\, g\, G\, L\, M\, I \;*\; [\text{Inv}\ G\ L\ I] \;*\; \$\phi(G, L)$$

$$\text{Inv}\ G\ L\ I \;:=\; \text{InvExcept}\ \emptyset\ G\ L\ I$$

$$\text{InvExcept}\ E\ G\ L\ I \;:=\; \left\{ \begin{array}{ll} \textit{acyclicity:} & \forall x.\quad x \not\longrightarrow_G^+ x \\ \textit{positive levels:} & \forall x.\quad L(x) \geq 1 \\ \textit{pseudo-topological numbering:} & \forall x\, y.\ x \longrightarrow_G y \implies L(x) \leq L(y) \\ \textit{horizontal incoming edges:} & \forall x\, y.\ x \in I(y) \iff x \longrightarrow_G y \ \wedge\ L(x) = L(y) \\ \textit{replete levels:} & \forall x.\quad x \in E \ \vee\ \text{enough\_edges\_below}\ G\ L\ x \end{array} \right.$$

$$\text{enough\_edges\_below}\ G\ L\ x \;:=\; |\text{coacc\_edges\_at\_level}\ G\ L\ k\ x| \geq k \ \text{ where } k = L(x) - 1$$

$$\text{coacc\_edges\_at\_level}\ G\ L\ k\ x \;:=\; \{\, (y, z) \mid y \longrightarrow_G z \longrightarrow_G^* x \ \wedge\ L(y) = L(z) = k \,\}$$

$$\left.\begin{array}{ll} \phi(G, L) & := C \cdot (\text{net}\ G\ L) \\ \text{net}\ G\ L & := \text{received}\ m\ n - \text{spent}\ G\ L \end{array}\right\} \ \text{where } m = |\text{edges}\ G| \text{ and } n = |\text{vertices}\ G|$$

$$\text{spent}\ G\ L \;:=\; \sum_{(u,v)\,\in\,\text{edges}\ G} L(u)$$

$$\text{received}\ m\ n \;:=\; m \cdot (\text{max\_level}\ m\ n + 1)$$

$$\text{max\_level}\ m\ n \;:=\; \min(\lceil (2m)^{1/2} \rceil, \lfloor (\tfrac{3}{2} n)^{2/3} \rfloor) + 1$$

$$\psi(m, n) \;:=\; C' \cdot (\text{received}\ m\ n + m + n)$$

Fig. 4. Definitions of IsGraph and ψ, with auxiliary definitions.

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 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 4, that Complexity is satisfied, that is, $\psi(m, n)$ is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

7 Specifications for the Algorithm's Main Functions

We cannot present the proof of the algorithm step by step, but attempt to convey its general flavor and provide some technical details. We give the specifications of the main two functions, `backward_search` and `forward_search`. This spells out exactly what each search achieves and how its cost is accounted for.

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 1). The top part of Figure 5 shows the same specification, where IsGraph (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.

$\forall g\ G\ L\ M\ I\ v\ w.$ let $m := |\text{edges } G|$ and $n := |\text{vertices } G|$ in
$v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies$

$$\left\{\begin{array}{ll} \text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ *\ \$\phi(G, L)\ * & (01)\\ \$(C' \cdot (\text{received}\ (m+1)\ n - \text{received}\ m\ n + 1)) & (02)\end{array}\right\}$$

(`add_edge_or_detect_cycle` $g\ v\ w$)

$$\left\{\begin{array}{l} \lambda res.\ \text{match } res \text{ with}\\ \quad |\ \text{Ok} \Rightarrow\ \text{let } G' := G + (v, w) \text{ in } \exists L'\ M'\ I'.\\ \qquad\quad \text{IsRawGraph}\ g\ G'\ L'\ M'\ I'\ *\ [\text{Inv}\ G'\ L'\ I']\ *\ \$\phi(G', L')\\ \quad |\ \text{Cycle} \Rightarrow [w \longrightarrow_G^* v]\end{array}\right\}$$

$\forall \mathit{fuel}\ g\ G\ L\ M\ I\ v\ w.$
$\mathit{fuel} \geq 0\ \wedge\ v, w \in \text{vertices } G\ \wedge\ v \neq w\ \wedge\ L(w) \leq L(v) \implies$

$$\left\{\begin{array}{ll} \text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (03)\\ \$(A \cdot \mathit{fuel} + B) & (04)\end{array}\right\}$$

(`backward_search` $g\ v\ w\ \mathit{fuel}$)

$$\left\{\begin{array}{ll} \lambda res.\ \exists M'. & \\ \quad \text{IsRawGraph}\ g\ G\ L\ M'\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (05)\\ \quad [\text{match } res \text{ with} & \\ \quad |\ \text{BackwardCyclic} \Rightarrow w \longrightarrow_G^* v & (06)\\ \quad |\ \text{BackwardAcyclic} \Rightarrow L(v) = L(w)\ \wedge\ w \not\longrightarrow_G^* v & (07)\\ \quad |\ \text{BackwardForward}(k, \mathit{mark}) \Rightarrow & \\ \qquad M'\ v = \mathit{mark}\ \wedge\ M'\ w \neq \mathit{mark}\ \wedge & (08)\\ \qquad (\forall x.\ M'\ x = \mathit{mark} \implies L(x) = L(v)\ \wedge\ x \longrightarrow_G^* v)\ \wedge & (09)\\ \qquad (\quad (k = L(v)\ \wedge\ L(w) < L(v)\ \wedge & (10)\\ \qquad\quad \forall x.\ L(x) = L(v)\ \wedge\ x \longrightarrow_G^* v \implies M'\ x = \mathit{mark}) & (11)\\ \qquad\ \vee\ (k = L(v) + 1\ \wedge\ \mathit{fuel} \leq |\text{coacc\_edges\_at\_level}\ G\ L\ (L(v))\ v|))] & (12)\end{array}\right\}$$

$\forall g\ G\ L\ M\ I\ w\ k\ \mathit{mark}.$
$w \in \text{vertices } G\ \wedge\ L(w) < k\ \wedge\ M\ w \neq \mathit{mark} \implies$

$$\left\{\begin{array}{ll} \text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (13)\\ \$(B' + \phi(G, L)) & (14)\end{array}\right\}$$

(`forward_search` $g\ w\ k\ \mathit{mark}$)

$$\left\{\begin{array}{ll} \lambda res.\ \exists L'\ I'. & \\ \quad \text{IsRawGraph}\ g\ G\ L'\ M\ I'\ * & (15)\\ \quad [L'(w) = k\ \wedge\ (\forall x.\ L'(x) = L(x)\ \vee\ w \longrightarrow_G^* x)]\ * & (16)\\ \quad \text{match } res \text{ with} & \\ \quad |\ \text{ForwardCyclic} \Rightarrow [\exists x.\ M\ x = \mathit{mark}\ \wedge\ w \longrightarrow_G^* x] & (17)\\ \quad |\ \text{ForwardCompleted} \Rightarrow & \\ \qquad \$\phi(G, L')\ * & (18)\\ \qquad [(\forall x\ y.\ L(x) < k\ \wedge\ w \longrightarrow_G^* x \longrightarrow_G y \implies M\ y \neq \mathit{mark})\ \wedge & (19)\\ \qquad\ \text{InvExcept}\ \{x \mid w \longrightarrow_G^* x\ \wedge\ L'(x) = k\}\ G\ L'\ I'] & (20)\end{array}\right\}$$

Fig. 5. Specifications for edge creation and its main two auxiliary functions.

The auxiliary function `backward_search` is parameterized with 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 known-but-irrelevant constants A and B. Indeed, the cost of a backward search is linear in the number of edges that are processed, therefore linear in *fuel*.

This function returns BackwardCyclic, BackwardAcyclic, or a value of the form BackwardForward(k, mark). The postcondition asserts that, in either of these cases, the graph remains valid (05), and the only state change is that some marks have been updated: M changes to M'.

If the return value 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 the latter case, no forward search is needed.

The postcondition ends with a description of the most complex case, where the return value is BackwardForward(k, mark). In this case, a forward search is required. The integer k indicates the level to which the vertex w and its descendants should be promoted during the 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, in this case, asserts that the vertex v is marked, whereas w is not (08), since it has not been reached. Moreover, every marked vertex lies at the same level as v and is an ancestor of v (09). 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 auxiliary 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 auxiliary 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 marked vertex: 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 4) 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 $g\ G'\ L'\ M'\ I'$ and Inv $G'\ L'\ I'$, so `add_edge_or_detect_cycle` satisfies its postcondition, under the form shown in Figure 5.

8 Related Work

Neither interactive program verification nor Separation Logic with Time Credits are new (§1). Outside the realm of Separation Logic, several researchers present machine-checked complexity analyses, carried out in interactive proof assistants. Van der Weegen and McKinna [45] study the average-case complexity of Quicksort, represented in Coq as a monadic program. The monad is used to introduce both nondeterminism and comparison-counting. Danielsson [14] implements a *Thunk* monad in Agda and uses it to reason about the amortized complexity of data structures that involve delayed computation and memoization. McCarthy *et al.* [31] present a monad that allows the time complexity of a Coq computation to be expressed in its type. Nipkow [34] proposes machine-checked amortized complexity analyses of several data structures in Isabelle/HOL. The code is manually transformed into a cost function.

Several mostly-automated program verification systems can verify complexity bounds. Madhavan *et al.* [30] present such a system, which can deal with programs that involve memoization, and is able to infer some of the constants that appear in user-supplied complexity bounds. Srikanth *et al.* [40] propose an automated verifier for user-supplied complexity bounds that involve polynomials, exponentials, and logarithms. When a bound is not met, a counter-example can be produced.

There is also a vast body of work on fully-automated inference of complexity bounds, beginning with Wegbreit [46] and continuing with more recent papers and tools [18,17,2,16,22]. Carbonneaux *et al.*'s analysis produces certificates whose validity can be checked by Coq [8]. It is possible in principle to express these certificates as derivations in Separation Logic with Time Credits. This opens the door to provably-safe combinations of automated and interactive tools.

Finally, there is a rich literature on static and dynamic analyses that aim at detecting performance anomalies [35,32,23,28,44].

9 Conclusion

In this paper, we have used a powerful program logic to simultaneously verify the correctness and complexity of an actual implementation of a state-of-the-art incremental cycle detection algorithm. Although neither interactive program verification nor Separation Logic with Time Credits are new, there are still relatively few examples of applying this simultaneous-verification approach to nontrivial algorithms or data structures. We hope we have demonstrated that this approach is indeed viable, and can be applied to a wide range of algorithms, including ones that involve mutable state, dynamic memory allocation, higher-order functions, and amortization.

As a technical contribution, whereas all previous works use credits in $\mathbb{N}$, we use credits in $\mathbb{Z}$ and allow negative credits to exist temporarily. We explain in an appendix (§A) why this is safe and convenient.

Following Guéneau *et al.* [19], our public specification exposes an asymptotic complexity bound: no literal constants appear in it. We remark, however, that it is often difficult to use something that resembles the O notation in specifications and proofs. Indeed, in its simplest form, a use of this notation in a mathematical statement $S[O(g)]$ can be understood as an occurrence of a variable f that is existentially quantified at the beginning of the statement: $\exists f.\ (f \preceq g) \wedge S[f]$. An example of such a statement was given earlier (§2). Here, f denotes an unknown function, which is dominated by the function g. The definition of the domination relation $\preceq$ involves further quantifiers [19]. In the analysis of a complex algorithm or data structure, however, it is often the case that an existential quantifier must be hoisted very high, so that its scope encompasses not just a single statement, but possibly a group of definitions, statements, and proofs. The present paper shows several instances of this phenomenon. In the public specification (Figure 1), the cost function ψ must be existentially quantified at the outermost level. In the definition of the data structure invariant (Figure 4) and in the proofs that involve this invariant, several constants appear, such as C and C', which must be defined beforehand. Thus, even if one could formally use $S[O(g)]$ as syntactic sugar for $\exists f.\ (f \preceq g) \wedge S[f]$, we fear that one might not be able to use this sugar very often, because a lot of mathematical work is carried out under the existential quantifier, in a context where f must be explicitly referred to by name. That said, novel ways of understanding the O notation may permit further progress; Affeldt *et al.* [1] make interesting steps in such a direction.

In future work, we would like to verify the algorithm that is used in the kernel of Coq to check the satisfiability of universe constraints. These are conjunctions of strict and non-strict ordering constraints, $x < y$ and $x \leq y$. This requires an incremental cycle detection algorithm that maintains strong components. Bender *et al.* [7, §5] present such an algorithm. It relies on a Union-Find data structure, whose correctness and complexity have been previously verified [11]. It is therefore tempting to re-use as much verified code as we can, without modification.

References

1. Affeldt, R., Cohen, C., Rouhling, D.: Formalization techniques for asymptotic reasoning in classical analysis. Journal of Formalized Reasoning **11**(1), 43–76 (2018)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012)
3. Anonymous: Coq pull request #XXX (Jul 2015), anonymized for review
4. Anonymous: Electronic appendix (Feb 2019), submitted via EasyChair
5. Anonymous: Electronic appendix (Feb 2019), submitted via EasyChair
6. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science **7**(2:17) (2011)
7. Bender, M.A., Fineman, J.T., Gilbert, S., Tarjan, R.E.: A new approach to incremental cycle detection and related problems. ACM Transactions on Algorithms **12**(2), 14:1–14:22 (2016)
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 (2017)
9. Charguéraud, A.: Characteristic formulae for the verification of imperative programs (2013), unpublished. `http://www.charglueraud.org/research/2013/cf/cf.pdf`
10. Charguéraud, A.: The CFML tool and library. `http://www.charglueraud.org/softs/cfml/` (2019)
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 (2017)
12. Chen, R., Cohen, C., Lévy, J.J., Merz, S., Théry, L.: Formal proofs of Tarjan's algorithm in Why3, Coq, and Isabelle (2018), manuscript
13. Chen, R., Lévy, J.: A semi-automatic proof of strong connectivity. In: Verified Software: Theories, Tools and Experiments. Lecture Notes in Computer Science, vol. 10712, pp. 49–65. Springer (2017)
14. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Principles of Programming Languages (POPL) (2008)
15. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 8044, pp. 463–478. Springer (2013)
16. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. Journal of Automated Reasoning **58**(1), 3–31 (2017)
17. Gulwani, S.: SPEED: symbolic complexity bound analysis. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 5643, pp. 51–62. Springer (2009)
18. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Principles of Programming Languages (POPL). pp. 127–139 (2009)
19. Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 10801, pp. 533–560. Springer (2018)

20. Haslbeck, M.P.L., Nipkow, T.: Hoare logics for time bounds: A study in meta theory. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 10805, pp. 155–171. Springer (2018)
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
22. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for OCaml. In: Principles of Programming Languages (POPL). pp. 359–373 (2017)
23. Holland, B., Santhanam, G.R., Awadhutkar, P., Kothari, S.: Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In: Source Code Analysis and Manipulation (SCAM). pp. 79–84 (2016)
24. Hopcroft, J.E.: Computer science: The emergence of a discipline. Communications of the ACM **30**(3), 198–202 (1987)
25. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Interactive Theorem Proving (ITP). Lecture Notes in Computer Science, vol. 8558, pp. 325–340. Springer (2014)
26. Lammich, P.: Refinement to Imperative/HOL. In: Interactive Theorem Proving (ITP). Lecture Notes in Computer Science, vol. 9236, pp. 253–269. Springer (2015)
27. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: Certified Programs and Proofs (CPP). pp. 137–146 (2015)
28. Lemieux, C., Padhye, R., Sen, K., Song, D.: PerfFuzz: Automatically generating pathological inputs. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 254–265 (2018)
29. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system: documentation and user's manual (2019)
30. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. In: Principles of Programming Languages (POPL). pp. 330–343 (2017)
31. McCarthy, J.A., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A Coq library for internal verification of running-times. In: Functional and Logic Programming. Lecture Notes in Computer Science, vol. 9613, pp. 144–162. Springer (2016)
32. Mudduluru, R., Ramanathan, M.K.: Efficient flow profiling for detecting performance bugs. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 413–424 (2016)
33. Mével, G., Jourdan, J.H., Pottier, F.: Time credits and time receipts in Iris (2019), to appear
34. Nipkow, T.: Amortized complexity verified. In: Interactive Theorem Proving (ITP). Lecture Notes in Computer Science, vol. 9236, pp. 310–324. Springer (2015)
35. Olivo, O., Dillig, I., Lin, C.: Static detection of asymptotic performance bugs in collection traversals. In: Programming Language Design and Implementation (PLDI). pp. 369–378 (2015)
36. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Principles of Programming Languages (POPL). pp. 247–258 (2005)
37. Pottier, F.: Depth-first search and strong connectivity in Coq. In: Journées Françaises des Langages Applicatifs (JFLA) (2015)
38. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)
39. Sozeau, M., Tabareau, N.: Universe polymorphism in Coq. In: Interactive Theorem Proving (ITP). Lecture Notes in Computer Science, vol. 8558, pp. 499–514. Springer (2014)
40. Srikanth, A., Sahin, B., Harris, W.R.: Complexity verification using guided theorem enumeration. In: Principles of Programming Languages (POPL). pp. 639–652 (2017)

41. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods **6**(2), 306–318 (1985)
42. Tarjan, R.E.: Algorithmic design. Communications of the ACM **30**(3), 204–212 (1987)
43. The Coq development team: The Coq Proof Assistant (2019)
44. Toffola, L.D., Pradel, M., Gross, T.R.: Synthesizing programs that expose performance bottlenecks. In: Code Generation and Optimization (CGO). pp. 314–326 (2018)
45. van der Weegen, E., McKinna, J.: A machine-checked proof of the average-case complexity of Quicksort in Coq. In: Types for Proofs and Programs. Lecture Notes in Computer Science, vol. 5497, pp. 256–271. Springer (2008)
46. Wegbreit, B.: Mechanical program analysis. Communications of the ACM **18**(9), 528–539 (1975)

A Metatheory of Negative Time Credits

In the original presentations of Separation Logic with Time Credits [6,11,19,20], credits are counted using natural numbers, that is, values in $\mathbb{N}$. In this setting, the law $\$(m+n) \equiv \$m * \$n$ holds for every $m, n \in \mathbb{N}$. Furthermore, credits are affine, that is, they can be discarded: the law $\$n \vdash \mathit{true}$ holds.

In this paper, however, we work in a variant of Separation Logic with Time Credits, where credits are counted using integers, that is, values in $\mathbb{Z}$. We set up the logic in such a way that the law $\$(m+n) \equiv \$m * \$n$ holds for every $m, n \in \mathbb{Z}$. This would be unsound if credits were affine. Indeed, from $\$0$, one could first derive $\$n * \$(-n)$, where n is an arbitrary positive number, then throw away $\$(-n)$, ending up with $\$n$, that is, with a positive amount of credits, created out of thin air. To ensure soundness, we posit that $\$n$ can be discarded only if $n \geq 0$ holds. In other words, a debt cannot be forgotten. As put by Tarjan [41], "we can allow borrowing of credits, as long as any debt incurred is eventually paid off".

We have verified in Coq the soundness of our variant of Separation Logic with Time Credits. The proof is available in an electronic appendix [5].

From a theoretical perspective, switching from $\mathbb{N}$ to $\mathbb{Z}$ may seem a relatively minor change. From a practical perspective, however, this change significantly simplifies specifications and proofs.

First, because OCaml programs manipulate signed machine integers, which we model directly as mathematical integers in $\mathbb{Z}$ (ignoring the risk of overflow, for the moment), this change reduces the number of conversions between $\mathbb{N}$ and $\mathbb{Z}$ that must be inserted in specifications and proofs.

Second, this change dramatically reduces the number of arithmetic side conditions that arise during verification. Indeed, when credits are in $\mathbb{N}$, although the law $\$(m+n) \equiv \$m * \$n$ has no side condition, the formulation that is most often needed in practice, namely $\$m \equiv \$(m-n) * \$n$, has the side condition $m - n \geq 0$. This reformulated law is typically used once at every function call. Consider a situation where m credits are available at a certain program point where a sequence of k function calls is about to take place. Suppose the first call requires n_1 credits, the second call requires n_2 credits, and so on. Then, the first

call gives rise to the proof obligation $m - n_1 \geq 0$; the second call gives rise to the proof obligation $m - n_1 - n_2 \geq 0$; and so on. Of course, the last proof obligation, namely $m - n_1 - n_2 - \ldots - n_k \geq 0$, entails the previous ones. However, in a proof assistant such as Coq, it is not easy to take advantage of this fact and present only the last proof obligation to the user. When credits are in $\mathbb{Z}$, on the other hand, this situation naturally gives rise to a single proof obligation: at the end of the sequence, the user must discard the credits that remain, and to this end, must prove that their number is nonnegative. The proof obligation is thus $m - n_1 - n_2 - \ldots - n_k \geq 0$.

In summary, in our experience, the switch from $\mathbb{N}$ to $\mathbb{Z}$ significantly improves the usability and the scalability of the verification framework.

B Asymptotic Analysis of Bender *et al.*'s Algorithm

This appendix offers informal proof sketches about Bender *et al.*'s algorithm. These results are not new. They are written under the assumption that the parameter F is a constant: its value does not change while the algorithm runs.

The proof sketch for Lemma 1 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 iff both of its endpoints are at level k. In Bender *et al.*'s algorithm, the following key invariant holds:

Lemma 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. □

Suppose we are interested in analyzing the cost of a sequence of n vertex creation and m edge creation operations, starting with an empty graph. Let Δ stand for $\min(m^{1/2}, n^{2/3})$. Then, setting F to Δ yields the desired asymptotic complexity bound:

Lemma 2 (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 §4 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 3 (Selecting an optimal value of Δ). *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 (§4). 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. □