Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits

Arthur Charguéraud, François Pottier

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits

Arthur Charguéraud · **François Pottier**

Received: date / Accepted: date

Abstract Union-Find is a famous example of a simple data structure whose amortized asymptotic time complexity analysis is nontrivial. We present a Coq formalization of this analysis, following Alstrup *et al.*'s recent proof (2014). Moreover, we implement Union-Find as an OCaml library and formally endow it with a modular specification that offers a full functional correctness guarantee as well as an amortized complexity bound. In order to reason in Coq about imperative OCaml code, we use the CFML tool, which implements Separation Logic for a subset of OCaml, and which we extend with time credits. Although it was known in principle that amortized analysis can be explained in terms of time credits and that time credits can be viewed as resources in Separation Logic, we believe our work is the first practical demonstration of this approach. Finally, in order to explain the meta-theoretical foundations of our approach, we define a Separation Logic with time credits for an untyped call-by-value λ-calculus, and formally verify its soundness.

1 Introduction

The Union-Find data structure, also known as a disjoint set forest (Galler and Fischer, 1964; Cormen et al, 2009), is widely used in the areas of symbolic computation, where it serves as the foundation of every unification algorithm, and graph algorithms, such as Kruskal's minimum spanning tree algorithm. It maintains a collection of disjoint sets and keeps track in each set of a distinguished element, known as the representative of this set. It supports the following basic operations: `make` creates a new element, which forms a new singleton set; `find` maps an element to the representative of its set; `union` merges the sets associated with two elements; `eq` tests whether two elements belong to the same set.

Union-Find is among the simplest classic data structures, yet requires one of the most challenging complexity analyses. Tarjan (1975) and Tarjan and van Leeuwen (1984) prove

A. Charguéraud
Inria & LRI, Université Paris Sud, CNRS E-mail: Arthur.Chargueraud@inria.fr

F. Pottier
Inria E-mail: Francois.Pottier@inria.fr

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

that the worst-case time required by a sequence of m operations involving at most n elements is $O(m \cdot \alpha(n))$. The function α, an inverse of Ackermann's function, grows so slowly that $\alpha(n)$ does not exceed 5 in practice. This analysis is progressively simplified over the years (Kozen, 1992), resulting in a readable 2.5-page proof that appears in Tarjan's online course notes (1999). It also appears in the textbook *Introduction to Algorithms* (Cormen et al, 2009), where it occupies about 8 pages. In this proof, the parameter n must be a priori fixed, and represents an upper bound on the number of elements that can ever appear in the data structure. In more recent work, Kaplan *et al.* (2002) and Alstrup *et al.* (2014) establish (among other results) a more precise and more pleasant bound: each operation has worst-case amortized complexity $O(\alpha(n))$, where n is the number of elements in the data structure at the time this operation is performed. In fact, they prove even tighter results, stated in terms of the sizes of equivalence classes, as opposed to the size of the entire data structure. For our purposes, taking n to be the current number of elements in the data structure seems good enough, and allows exposing a specification that seems as simple as one might hope for.

In this paper, we make the following contributions:

1. We prove the correctness of an abstract (mathematical) presentation of the Union-Find data structure, described in terms of graphs and relations, and verify its time complexity, following Alstrup *et al.*'s potential-based analysis (2014). We check these proofs using the Coq proof assistant.
2. We extend Separation Logic with time credits, considered as resources. This allows us to push our reasoning down to the level of actual source code instead of remaining at the level of idealized algorithms. We implement support for time credits in the tool CFML (Charguéraud, 2013), which translates OCaml code to Coq characteristic formulae, thereby allowing Separation Logic proofs about OCaml code to be carried out in Coq. This yields a practical program verification framework with support for dynamic memory allocation, mutable state, aliasing, and (amortized worst-case) time complexity analysis.
3. We formalize Separation Logic with time credits, in Coq, for an untyped call-by-value λ-calculus, and prove that the logic is sound. Although this proof is not directly about CFML, it helps increase our confidence that the reasoning rules implemented in CFML are sound.
4. We present a realistic, pointer-based, imperative OCaml implementation of the Union-Find data structure and, using CFML and Coq, we carry out a Separation Logic proof of its correctness and time complexity. In other words, we verify that this OCaml code correctly and faithfully implements the abstract Union-Find algorithm.

Our approach is modular in the sense that the Separation Logic specification that we establish can be used to reason about client programs, without knowledge of the Union-Find implementation. There is one caveat, though: our specifications expose hard-coded multiplicative and additive complexity constants, because we have not yet developed the theory and tactics that are required to support the big-O notation. Thus, the specifications shown in this paper do not allow changing the implementation of Union-Find without any impact on the proofs of client programs. Adding support for the big-O notation, and validating our specifications by proving the correctness of illustrative client programs, are left to future work.

In summary, in this work, we carry out, in a unified framework, a mathematical analysis of the correctness and time complexity of a nontrivial data structure and a step-by-step analysis of its OCaml implementation. Our proofs are available online (Charguéraud and Pottier, 2017).

This is an extended and improved version of an earlier conference paper (Charguéraud and Pottier, 2015). In comparison, in this paper:

1. We adopt Alstrup *et al.*'s time complexity analysis (2014), which leads to a slightly simpler and more convenient specification of Union-Find in Separation Logic. In contrast, the conference paper follows Tarjan *et al.*'s analysis (Tarjan, 1999). Our online archive (Charguéraud and Pottier, 2017) in fact contains both analyses, which share a great deal of material.
2. We offer a thorough presentation of a Separation Logic with time credits for an untyped call-by-value λ-calculus. This presentation is backed with a machine-checked proof of soundness, which is also available online (Charguéraud and Pottier, 2017).
3. We extend our OCaml implementation of Union-Find by letting every set carry a piece of user data, which is read and written via two additional operations, `get` and `set`.
4. We add a "ghost" operation, `join`, which causes two disjoint set forests to be regarded as one forest. The fact that disjoint set forests support this operation (at no cost!) seems under-emphasized in the literature, precisely because no code is required. However, in a verified implementation, this fact must be proved. We believe that it is worth attracting some attention to the need for such "ghost" operations in the specifications of certain data structures.

The paper is organized as follows. First, we present an OCaml implementation of Union-Find and give a high-level overview of our approach to specifying and verifying this code, placing particular emphasis on our assumptions (§2).

Then, we introduce a Separation Logic for an untyped call-by-value λ-calculus and proceed to extend this logic with time credits (§3). For this logic, we describe the construction of characteristic formulae (§4), a systematic way of applying the reasoning rules of the logic, so as to obtain proof obligations expressed as ordinary logic formulae. The CFML tool also relies on characteristic formulae to allow Separation Logic proofs about OCaml code to be carried out in Coq.

Coming back to Union-Find, we present a specification of Union-Find in Separation Logic with time credits (§5). We present the purely mathematical part of our proof, which consists in an analysis of the operations on disjoint set forests and of their complexity (§6). We relate our mathematical view of forests with their concrete layout in memory by defining an abstract Separation Logic predicate, named `UF`. We then prove that the OCaml code satisfies its specification (§7). The paper ends with discussions of the related work (§8) and of directions for future research (§9).

2 Overview

2.1 The Union-Find Data Structure: Signature

In OCaml syntax, the Union-Find data structure offers the following signature, where the abstract type `'a elem` is the type of elements:

```
type 'a elem
val make : 'a -> 'a elem
val get  : 'a elem -> 'a
val set  : 'a elem -> 'a -> unit
val eq   : 'a elem -> 'a elem -> bool
val union: 'a elem -> 'a elem -> 'a elem
val find : 'a elem -> 'a elem
```

The Union-Find data structure maintains a collection of disjoint sets and keeps track in each set of a distinguished element, known as the representative of this set. With every set, the data structure attaches a "datum", that is, an OCaml value. The type of these data values, which is chosen by the user, is represented by the type variable `'a`. The type of elements, `'a elem`, is parameterized with `'a`, and the operations are polymorphic in `'a`. There is no type of sets: instead, a set is designated by any of its elements. For instance, the function `get`, whose purpose is to read the datum attached with a set, takes an element as an argument. It finds which set this element lies in, and returns the datum attached with this set.

The function `make` creates a new element and places it in a new singleton set. It expects a value of type `'a`, which it attaches to the newly created set. The functions `get` and `set` consult and update the value attached with a set. The function `eq` tests whether two elements belong in the same set. The function `union` merges two sets. (If `union` is applied to two elements of the same set, nothing happens.) It arbitrarily chooses and keeps one of the two data values attached with these sets; the other value is lost. If desired, `get` and `set` can be used to obtain other behaviors, such as computing a new datum based on the existing two data values. The function `find` returns the representative element of a set. It may seem as though `find` can be of no service to the user: after all, none of the functions that we expose can distinguish between two equivalent elements! Indeed, the main uses of `find` are internal: it is used in the implementation of `union` and `eq`. Nevertheless, there are scenarios where `find` is useful to the user. For instance, exposing `find` is necessary if at some point the user wishes to keep pointers only to representative elements, so that every nonrepresentative element can be reclaimed by the garbage collector.

The type `'a elem` equipped with the operations `make`, `get`, `set`, and `eq` offers the same signature as the built-in type `'a ref` equipped with the operations `ref`, `!`, `:=`, and `==`. In OCaml, as in the other programming languages of the ML family, `'a ref` is the type of "references", or mutable memory cells. Thus, the abstract types `'a ref` and `'a elem` are close cousins. The key operation that the former lacks and the latter supports is `union`. In other words, the Union-Find data structure can be viewed as an efficient implementation of an abstract type of "mergeable memory cells". This is a concise informal way of explaining the purpose of this data structure, which we believe is seldom presented in such a light.

2.2 The Union-Find Data Structure: Implementation

Disjoint set forests were invented by Galler and Fischer (1964). In such a forest, an element either points to another element (its parent) or points to no element and therefore is a root. These pointers form a forest where each tree represents a set and where the root of the tree is the representative element of the set. Thus, out of every node x, there is a unique maximal path, whose endpoint is the root of x's tree, that is, the representative element of x's set.

The `find` operation follows this unique path. For greater efficiency, it performs path compression: every node along the path is updated so as to point directly to the root of the tree. This idea is attributed by Aho *et al.* (1974) to McIlroy and Morris.

The `union` operation updates the root of one tree so as to point to the root of the other tree. To decide which of the two roots should become a child of the other, we apply linking-by-rank (Tarjan, 1975; Tarjan and van Leeuwen, 1984). A natural number, the rank, is associated with every root. The rank of a newly created node is zero. When performing a union, the root of lower rank becomes a child of the root of higher rank. In case of equality, the new root is chosen arbitrarily, and its rank is increased by one.

```
type rank = int

type 'a elem =
  'a content ref
and 'a content =
  | Link of 'a elem
  | Root of rank * 'a

let make v =
  ref (Root (0, v))

let rec find x =
  match !x with
  | Root (_, _) ->
      x
  | Link y ->
      let z = find y in
      x := Link z;
      z

let get x =
  let x = find x in
  match !x with
  | Root (_, v) ->
      v
  | Link _ ->
      assert false
```

```
let set x v =
  let x = find x in
  match !x with
  | Root (r, _) ->
      x := Root (r, v)
  | Link _ ->
      assert false

let link x y =
  if x == y then x else
  match !x, !y with
  | Root (rx, vx), Root (ry, _) ->
      if rx < ry then begin
        x := Link y; y
      end else if rx > ry then begin
        y := Link x; x
      end else begin
        y := Link x;
        x := Root (rx+1, vx); x
      end
  | _, _ ->
      assert false

let union x y = link (find x) (find y)

let eq x y = (find x == find y)
```

Fig. 1 OCaml implementation of Union-Find

We settle on path compression and linking-by-rank because they are simple, standard, and have optimal time complexity. (Either one of them, alone, would not suffice.) There exist other path compression strategies, such as path halving, and other linking strategies, such as linking-by-size. An experimental survey of many variations of Union-Find is given by Patwary *et al.* (2010).

Our OCaml implementation of Union-Find appears in Figure 1. The (mutually recursive) definitions of the types `'a elem` and `'a content` can be read as follows: a node `x` is a reference (that is, a mutable memory cell) whose content is either `Link y`, which means that `x` is a nonroot node whose parent is `y`, or `Root (r, v)`, which means that `x` is a root node whose rank is `r` and which carries the user datum `v`. This is the code whose correctness and complexity we formally establish.

Our code represents ranks as OCaml machine integers, of OCaml type `int`. Depending on the target architecture, these integers are 31 or 63 bits wide. In our proofs, however, for simplicity, an OCaml machine integer is modeled as an ideal (unbounded) integer, whose type in Coq is `Z`. Thus, we do not prove the absence of overflows. Furthermore, if an overflow occurs, then the code could in principle behave incorrectly, even though we have "verified" its correctness. To remedy this problem, two approaches come to mind. One could either modify the code to use arbitrary-precision integers, with relatively minor runtime overhead, and no changes to the specifications and proofs; or, without any change to the code, modify the specifications and proofs so as to impose a bound (dependent on the machine's word size) on the number of elements that can be created. In the particular case of the Union-Find data structure, one can prove (and we have checked in Coq; §6.3) that the rank of every element is at most $\log_2 n$, where n is the total number of elements. Thus, on a 64-

bit architecture, for a rank computation to overflow, one would have to allocate about $2^{2^{62}}$ elements. One might wish to informally argue that this is impossible, by sheer lack of time. Clochard *et al.* (2015) suggest how such a "proof-of-work" argument might be expressed in a formal program verification framework. It would be interesting to find out whether and how our Separation Logic-based approach can accommodate proof-of-work arguments; we leave this question to future work.

2.3 Separation Logic with Time Credits

Separation Logic (Reynolds, 2002) offers a natural framework for proving the correctness of imperative programs that manipulate the heap. In order to formally establish not just the correctness but also the complexity of a program, we extend Separation Logic with time credits.

A time credit is a resource that represents a right to perform one step of computation. (What exactly constitutes one "step" is discussed in §2.7.) The assertion $\$n$, where n is a nonnegative integer, represents the ownership of n time credits, that is, a permission to perform n steps of computation.

A certain number of credits can be explicitly requested as part of the precondition of a function f, and can be used to justify the computation steps performed by f and by its callees. For instance, a function that expects a list l as an argument and traverses this list might require $c \times \mathit{length}\ l$ credits from its caller, where c is an implementation-dependent constant. This would be specified by letting $\$(c \times \mathit{length}\ l)$ appear (among other separated conjuncts) in the precondition of f.

A function f need not consume all of the credits that it receives from its caller. Some credits can be returned by f to its caller: this is specified by letting them appear as part of the postcondition of f. At first, this might seem pointless: why would f request more credits than it really needs, and return those that are left over? The answer is, in combination with abstract predicates, this allows amortized reasoning. The specification of `find`, to be presented shortly (§2.4), illustrates this.

In summary, there are several reasons why Separation Logic with time credits seems particularly attractive. First, it deals with dynamic memory allocation and mutation. Second, it allows reasoning about correctness and complexity at the same time, which means that the properties established in the correctness proof (such as the balancedness of a tree) can be exploited in the complexity proof. Last, it supports amortized complexity analyses.

Time credits, under various forms, have been used in several type systems (Danielsson, 2008; Hoffmann and Hofmann, 2010; Pilkiewicz and Pottier, 2011). Atkey (2011) has argued in favor of viewing credits as predicates in Separation Logic. However, Atkey's work did not go as far as using time credits in a practical, general-purpose program verification framework. In this paper, we present such a framework, and demonstrate its practicality by verifying the Union-Find data structure.

2.4 Formal Specification of Union-Find

In order to express the specification of every operation (`make`, `find`, and so on), we define an abstract Separation Logic predicate, `UF`. This predicate has three parameters, namely D, R, V, where:

- D (for "domain") is the set of all elements created so far;

- R (for "representative") is a total function that maps every element to its representative;
- V (for "value") is a total function that maps every element to its associated datum.

The Separation Logic assertion $\texttt{UF}\,D\,R\,V$ represents the unique ownership of a Union-Find data structure whose current state is summarized by D, R, and V. Every operation requires $\texttt{UF}\,D\,R\,V$ as part of its precondition, and ensures $\texttt{UF}\,D'\,R'\,V'$ as part of its postcondition, where D', R', V' are related with D, R, V in a manner that reflects the effect of this operation.

This is all the user needs to know. In particular, the user need not know that the data structure is represented in memory by a collection of objects (and pointers between them) which form a disjoint set forest. The user need not know which paths exist in the forest, when and how these paths are compressed, and so on. By making `UF` an abstract predicate, we guarantee that the user does not have access to these implementation details, hence cannot depend on them.

The specification of an operation f takes the form of a Hoare triple $\{H\}\ f(x)\ \{\lambda y.Q\}$ where the Separation Logic assertions H and Q are the precondition and postcondition, x stands for the argument, and y stands for the result. (The variable x occurs free in this triple, and can appear in H and Q. In practice, it is usually explicitly universally quantified in front of such a triple. The variable y is λ-bound and can appear only in Q.) For instance, in Coq syntax, the specification of `find` (also shown in Figure 9) is as follows:

```
Theorem find_spec : ∀D R V x, x ∈ D →
  app UnionFind_ml.find [x]
    PRE (UF D R V ⋆ $(2 ∗ alpha (card D) + 4))
    POST (fun y ⇒ UF D R V ⋆ [ R x = y ]).
```

The specification is universally quantified with respect to D, R, V, and the argument x. It is subject to the requirement that x be a member of D. (This could equivalently be stated as part of the precondition.) The notation `app UnionFind_ml.find [x]` is our concrete syntax for referring to the application of the OCaml function `find` to one argument, namely x. The precondition, which follows the keyword `PRE`, requires $\texttt{UF}\,D\,R\,V$ together with a certain number of time credits (explained further on). The postcondition, which follows the keyword `POST`, guarantees $\texttt{UF}\,D\,R\,V$ again, which means that `find` does not affect the user's view of the data structure. (Note that `find` internally performs path compression, so it does sometimes modify the data structure, but in an unobservable way.) The postcondition also contains the equality $R\,x = y$, which guarantees that the result y is the representative of x.

The definition of the predicate `UF` (which, as explained above, is not exposed to the user) can be summarized as follows:

- There exists in the heap a collection of mutable memory cells of OCaml type `'a elem`.
- The graph formed by these cells and by the pointers between them is a disjoint set forest.
- The set of all vertices is D. There is a path from every vertex x in D to the vertex $R\,x$, which is a root, and which carries the user datum $V\,x$.
- There exist Φ time credits, where Φ is the current potential of the data structure, in Tarjan's terminology (Tarjan, 1985).

In Separation Logic, to know about a resource is to own it, so the first assertion above implies the unique ownership of the memory cells.

For the same reason, the last assertion above implies the unique ownership of Φ credits. These credits are "saved inside the data structure", so to speak. They can be used to pay in part for an operation and therefore decrease its apparent cost. Conversely, if one chooses to advertise an apparent cost that is greater than the operation's actual cost, then the extra credits provided by the caller can be "saved", that is, used to justify an increase of Φ.

Because the invariant is defined in this way, when we formally verify the code, we are naturally required to prove that, for each operation, the initial potential Φ, plus the number of credits brought by the caller, covers the new potential Φ', plus the number of credits consumed during the operation:

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

For instance, according to the specification of `find` (shown earlier and in Figure 9), this function ostensibly requires $2\alpha(|D|)+4$ credits from its caller, where $|D|$ is the cardinality of the set D. This is its apparent cost. Its actual cost (the number of computation steps that it actually performs) may be lower or higher. Indeed, by unfolding the definition of the predicate `UF`, one discovers that (1) in addition to these credits, `find` has access to Φ credits; and (2) upon termination, `find` must arrange for Φ' credits to remain, where Φ' is the potential of the data structure after the operation. (Due to path compression, Φ and Φ' may differ.) In summary, the bound that explicitly appears in the precondition of `find`, namely $2\alpha(|D|)+4$, is a worst-case amortized time complexity bound.

Quite clearly, instead of stating that `find` requires $2 \cdot \alpha(|D|)+4$ credits, it would be preferable to state that it requires $O(\alpha(|D|))$ credits. That would be more readable, more modular, and would carry no less information about physical computation time. Developing support for the big-O notation in our specifications and proofs is ongoing work (§9).

2.5 Verification of Imperative Code using Characteristic Formulae

To prove that the implementation of an operation satisfies its specification, we rely on CFML (Charguéraud, 2010, 2013), which can be described as an implementation of (higher-order) Separation Logic for a subset of OCaml. The CFML tool transforms an OCaml term t into a *characteristic formula*, that is, a higher-order logic formula $[\![t]\!]$, which describes the semantics of t. More precisely, $[\![t]\!]$ denotes the set of all valid specifications of t, as follows: for any precondition H and postcondition Q, the proposition $[\![t]\!]\,H\,Q$ holds if and only if the Hoare triple $\{H\}\ t\ \{Q\}$ is valid. Such a triple implies that, beginning in any state that satisfies H, the term t runs safely, terminates, produces a value v, and leaves the machine in a state that satisfies the assertion $Q\,v$.[1]

The characteristic formula $[\![t]\!]$ is automatically built from the term t by the CFML tool. It is then exploited, in Coq, to formally establish that t satisfies a certain specification. More concretely, we run CFML on the OCaml file `UnionFind.ml`, which contains the code for Union-Find, as shown in Figure 1. We obtain a Coq file, `UnionFind_ml.v`, which contains a series of axioms. For each toplevel OCaml function, we get two axioms: one asserts that this function exists (and names it), while the other provides a means (based on this function's characteristic formula) of establishing a Hoare triple about this function. For instance, for the OCaml function `find`, the following two axioms are generated:

```
Parameter find : func.
Parameter find__cf :
  ∀A_ : Type,
  ∀x : elem_ A_,
```

[1] This explanation is deliberately slightly informal. It is written as though an OCaml value v could be viewed as a Coq value, and avoids mentioning the type of v. In reality, CFML includes machinery for translating an OCaml type τ to a Coq type $[\![\tau]\!]$ and reflecting an OCaml value v as a Coq value $[\![v]\!]$ of type $[\![\tau]\!]$. For more details about this translation, the reader is referred to Charguéraud's work (2013).

```
∀H : hprop,
∀Q : elem_ A_ → hprop,
... (* find's characteristic formula, applied to H and Q *) →
app find [x] H Q.
```

The value `find` is viewed in Coq as an object of abstract type `func` (§4.1). The type `elem_ A_` is the Coq counterpart of the OCaml type `'a elem` (§7). `hprop` is a shorthand for `Heap` → `Prop`: it is the type of a Separation Logic assertion (§3.2).

The axiom `find__cf` states that, to establish a Hoare triple of the form `app find [x] H Q`, it suffices to prove that the characteristic formula, applied to `H` and `Q`, holds.

The theorem `find_spec`, whose statement was shown earlier, can be proved by applying this axiom (and cannot be proved in any other way). We place the statement and proof of `find_spec`, and of other theorems like it, in a hand-written Coq file, `UnionFind_proof.v`. The fact that `find`'s characteristic formula is large and unwieldy is hidden from the user. CFML provides a library of Coq notations and tactics that allow us to carry out the proof of `find_spec` in a natural style. As is usual in Coq, the proof is developed interactively, using a mixture of native Coq tactics, such as induction, and CFML-specific tactics, including tactics for applying the reasoning rules of Separation Logic and tactics for proving Separation Logic entailments.

2.6 Trusted Computing Base

Our approach to program verification involves a number of layers, so we wish to clarify what needs to be trusted at the transition from each layer to the next. There are three layers:

1. We prove Separation Logic specifications, in Coq, by relying on characteristic formulae.
2. One level down, we wish to argue that this yields a guarantee about the behavior of the OCaml source code. This requires characteristic formulae to be sound with respect to the semantics of OCaml.
3. One level down, we wish to argue this yields a guarantee about the behavior of the compiled code. This requires the compiler to be semantics- and complexity-preserving.

Let us look in greater detail at the topmost layer. We express our specifications, and carry out our proofs, using the Coq proof assistant. Therefore, we must trust Coq's meta-theory and implementation. Furthermore, for added convenience, our Coq proofs rely on a number of standard logical axioms, including functional extensionality, propositional extensionality, the law of excluded middle, and Hilbert's ε operator[2]. The latter allows, e.g., defining "the minimum element" of a subset of $\mathbb{N}$ before this subset has been proved nonempty (§6.4), or referring to "the parent" of a node before it has been established that this node has a parent (§6.5). In principle, we do not see any obstacle to carrying out our proofs without these axioms, though with slightly higher effort.

Consider now the transition down to the second layer. We use the CFML tool to produce characteristic formulae what we accept as axioms in our Coq proofs. Therefore, we must trust CFML's meta-theory and implementation.[3] The implementation of CFML is currently not verified. The first author has studied the meta-theory of CFML and proved, on

[2] Hilbert's ε operator applies to a predicate P of type $A \to \mathsf{Prop}$, where A is an inhabited type. It produces a value v of type A such that $\exists x.P\,x$ implies $P\,v$.

[3] One known source of unsoundness in CFML is the gap between OCaml machine integers and Coq ideal integers, which we have discussed already (§2.2).

paper, that the characteristic formulae produced by CFML offer a sound means of reasoning about OCaml code (Charguéraud, 2010, 2013). Furthermore, it is possible to formally verify the metatheory and implementation of a characteristic formula generator. The results that we present in Sections 3 and 4 of this paper are verified in Coq (Charguéraud and Pottier, 2017). Also, in recent work, Guéneau *et al.* (2017) present a verified characteristic formula framework for CakeML. Because the CakeML compiler and characteristic formula generator are both verified, and agree on an operational semantics of CakeML, neither of them need be trusted.

Last, consider the transition down to the third layer. We write OCaml code, and analyze its complexity at the source level. Thus, we must trust the OCaml compiler to be correct, that is, to produce machine code that has the same behavior and the same time complexity (up to a program-dependent constant factor) as the source code. Whether this property holds, and how it could be formally established, is what we examine next.

2.7 From High-Level Time Complexity Bounds to Physical Time Consumption

There are several levels at which the time complexity of a program can be measured. From most abstract to least abstract, here are several levels that we can think of:

1. One can count how many operations of a specific kind are executed at the source level (that is, when the source program is executed according to the operational semantics of the source language). For instance, one can count how many function calls are executed at the source level.
2. One can count how many operations of any kind are executed at the source level, that is, how many reduction steps are taken when the source program is executed according to the operational semantics of the source language.
3. One can count how many instructions are executed at the machine level, that is, how many program instructions must be executed by a machine when the compiled program is run. This measure depends on which target instruction set, compiler, and runtime system are used.
4. One can measure how much physical time is spent when the compiled program is run. This measure additionally depends on which machine, which operating system, and which measurement methodology are used.

We attack the problem at the most abstract level, that is, level 1 above. We posit that one function call consumes one credit. Using Separation Logic, we bound how many credits are consumed, and therefore bound the number of β-reduction steps involved in the execution of the OCaml source code, according to the operational semantics of OCaml.

Now, the question is, are the four levels above related to one another? Ideally, one would hope that these four levels of measurement yield the same notion of time complexity, up to constant factors, so that, when we establish an asymptotic time complexity bound at the highest level, the same bound holds at the lowest level, too.

The fact that level 1 and level 2 above yield the same measure (up to a constant factor, which depends linearly on the program size) is a property of the source language and of its operational semantics. It could be formally stated and verified, if desired. (We have not done so.) An intuitive argument why this property holds is that, in a fixed source program, every function body contains at most a constant number of instructions; so, the number of reduction steps that are executed between two function calls is bounded by a constant[4].

This idea, namely the idea that "it suffices to count the function calls", is quite old. For example, in the setting of a first-order functional programming language, Le Métayer (1988) notes that "the asymptotic complexity and the number of recursive calls necessary for the evaluation of the program are of the same order-of-magnitude". In a higher-order functional programming language, such as OCaml, it is possible to call a statically unknown function. This mechanism can be used to define a fixpoint combinator, that is, to construct recursive functions without explicitly using recursion. For this reason, every function call must be counted, even if it is not visibly a recursive call. (It is possible to relax this rule slightly by not counting the calls to a fixed set of built-in functions whose execution time is known to be constant.) Danielsson (2008) describes a type system which enforces such a discipline.

The fact that level 2 and level 3 above are related (up to a constant factor, which depends linearly on the program size) is a property of the target instruction set, compiler, and runtime system. This property can in principle be formally stated and (for a fixed target instruction set, compiler, and runtime system) verified. Blelloch and Greiner (1995) and Sands *et al.* (2002) independently address this question for call-by-value λ-calculus and prove that counting reduction steps is a valid cost model. We assume that this property holds of the OCaml compiler and runtime system. This has not been formally verified; however, if it did not hold, that would arguably constitute a bug.

An intuitive argument why this property likely holds is that, ignoring for a moment the memory allocation and garbage collection code, every source instruction is translated by the OCaml compiler to at most a constant number of machine instructions. In particular, the cost of evaluating a λ-abstraction (that is, building a closure) and the cost of evaluating a variable (that is, accessing a field in a closure) are both bounded by a constant. In the OCaml compilers, a flat representation of closures is employed, so the cost of building a closure is (up to a constant) the number of free variables of this closure, which can itself be regarded as a constant, as it depends only on the source program, not on the input data. Accessing a field in a closure is just a memory read, so has constant cost as well.

The cost of memory allocation and garbage collection may naively seem uncontrollable, as it is well-known that garbage collection can cause long pauses at unpredictable times. However, it is not difficult to argue that certain collection algorithms, such as stop-and-copy, have constant amortized per-allocation cost, provided there is "enough" free space, that is, provided the ratio of heap size to live data size remains above a constant k, where k must be strictly greater than 1 (Appel, 1992, Chapter 16). There are also "real-time" collection algorithms with constant worst-case per-allocation cost, such as Baker's (1978).

The CerCo project (Amadio and Régis-Gianas, 2011; Ayache et al, 2012; Amadio et al, 2014) has built compilers for C and ML which (roughly speaking) annotate every basic block in the source program with the maximum actual cost (according to a concrete model of the target machine) of the corresponding compiled basic block. Furthermore, these compilers are verified, so that these annotations are guaranteed to be correct upper bounds. This allows carrying out a worst-case-execution-time analysis in terms of the source program and nevertheless obtain a concrete upper bound that holds at the machine level (that is, level 3 above). Furthermore, this proves that counting how many basic blocks are entered when the source program is executed yields the same result, up to a constant factor (which depends

[4] This argument assumes that the programming language has neither iteration statements (`while`, `for`, etc.) nor unstructured control statements (`break`, `continue`, `goto`, etc.). One way of dealing with these statements is to encode them in terms of recursive functions, so that, after this encoding, "counting every function call" is indeed sufficient.

$$\begin{array}{rcl} v & := & x \mid n \mid () \mid (v,v) \mid l \mid \mu f.\lambda x.t \mid \mathsf{ref} \mid \mathsf{get} \mid \mathsf{set} \\ t & := & v \mid \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid (v\ v) \end{array}$$

Fig. 2 Syntax of values and terms

$$\text{EVAL-VAL}\quad \frac{}{v/m \Downarrow^{0} v/m}$$

$$\text{EVAL-IF}\quad \frac{\begin{array}{l} \quad\ v \neq 0\ \wedge\ t_1/m \Downarrow^{n} v'/m' \\ \vee\ \ v = 0\ \wedge\ t_2/m \Downarrow^{n} v'/m' \end{array}}{(\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)/m \Downarrow^{n} v'/m'}$$

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

$$\text{EVAL-APP}\quad \frac{[\mu f.\lambda x.t/f]\,[v/x]\,t/m \Downarrow^{n} v'/m'}{((\mu f.\lambda x.t)\ v)/m \Downarrow^{(n+1)} v'/m'}$$

$$\text{EVAL-REF}\quad \frac{l \notin \mathrm{dom}(m)}{(\mathsf{ref}\ v)/m \Downarrow^{0} l/m \uplus (l \mapsto v)}$$

$$\text{EVAL-GET}\quad \frac{l \notin \mathrm{dom}(m)}{(\mathsf{get}\ v)/m \uplus (l \mapsto v) \Downarrow^{0} v/m \uplus (l \mapsto v)}$$

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

Fig. 3 Cost-instrumented big-step operational semantics

on the program), as measuring the cost of executing the compiled program. In other words, these compilers preserve asymptotic complexity: when they are used, levels 1, 2 and 3 above yield the same measure, up to a program-dependent constant factor.

Finally, the fact that levels 3 and 4 above are related (up to a constant factor) may sound plausible, but would be difficult to prove, as that would require formal models of the machine and operating system. Certainly the constant factor would have to be quite large, so as to take into account the possibility of cache misses, page faults, interrupts, and so on.

3 Separation Logic with Time Credits

3.1 Source language syntax and semantics

We consider an imperative call-by-value λ-calculus (Figure 2). The syntactic categories are values v and terms t. The values include variables x (also f), integer literals n, the unit value $()$, pairs (v_1, v_2), memory locations l, recursive functions $\mu f.\lambda x.t$, and the three primitive operations for allocating, reading, and writing references. The terms include a construct for returning a value, a conditional construct, sequencing, and function invocation.

Following common practice, the calculus is presented in "administrative normal form". That is, "$\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$" is the sole sequencing construct: no sequencing is implicit in any other construct. For instance, the conditional construct "$\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$" cannot be expressed directly and must be encoded as "$\mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$". This presentation is intended to avoid redundancy in the reasoning rules.

The operational semantics of the calculus is defined by a big-step evaluation judgment. Because we wish to reason about time complexity, we adopt a cost-instrumented semantics. The judgment $t/m \Downarrow^{n} v/m'$ asserts that the evaluation of the term t in the initial memory m terminates after n "steps" of computation, producing the value v and the final memory m'. (A memory is a finite map of memory locations to values.) This judgment is inductively defined in Figure 3. These rules are standard. As discussed earlier (§2.7), the only "steps" that we wish to count are function calls, so EVAL-APP is the only rule where the counter n is

$$\begin{array}{lcl}
[P] & \equiv & \lambda h.\ h = \emptyset \land P \\
\top & \equiv & \lambda h.\, \mathsf{True} \\
l \hookrightarrow v & \equiv & \lambda h.\ h = (l \mapsto v) \\
H_1 \star H_2 & \equiv & \lambda h.\ \exists h_1 h_2.\ h_1 \perp h_2 \land h = h_1 \uplus h_2 \land H_1\, h_1 \land H_2\, h_2 \\
\exists x. H & \equiv & \lambda h.\ \exists x.\, H\, h
\end{array}$$

Fig. 4 Separation Logic assertions: syntax and interpretation

incremented. A simpler judgement $t/m \Downarrow v/m'$, which does not carry a cost annotation, is defined as $\exists n.\ (t/m \Downarrow^n v/m')$.

3.2 Ordinary Separation Logic

In Separation Logic, an assertion H has type $\mathsf{Heap} \to \mathsf{Prop}$ and describes a heap fragment. Such an assertion has an ownership reading. If "we" know that H holds of a certain heap fragment, then "we" are the unique owner of this heap fragment: that is, "we" have read-write access to it, while "others" have no access to it. A heap (or heap fragment) h is just a memory, that is, a finite map of locations to values: $\mathsf{Heap} \equiv \mathsf{Memory}$. This definition is revisited in §3.3, where time credits are introduced.

Several fundamental forms of assertions are defined in Figure 4. These definitions remain in force in §3.3, where time credits are introduced.

The assertion $[P]$ is true of an empty heap, provided the proposition P holds. (P has type Prop.) This assertion is "pure" in the sense that it represents pure information and no ownership. The "empty heap" assertion $[\,]$ can be viewed as syntactic sugar for $[\mathsf{True}]$.

The "top" assertion $\top$ is true of any heap. It asserts that "we" may own a nonempty heap fragment, but we know nothing about its contents. It is used in the interpretation of Separation Logic triples (Definition 1, coming up) to describe a heap fragment that the user has decided to discard.

The "points-to" assertion $l \hookrightarrow v$ is true of the singleton heap fragment $(l \mapsto v)$. This assertion means that the reference cell at location l exists, is uniquely owned by "us", and currently contains the value v.

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

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

Entailment, written $H_1 \preceq H_2$ and pronounced "H_1 entails H_2", is defined as follows: $\forall h.\ H_1\, h \Rightarrow H_2\, h$. Entailment is used in the reasoning rules of Separation Logic (Figure 5, coming up) and in the construction of characteristic formulae (§4). Entailment between postconditions, written $Q \preceq Q'$, is sugar for $\forall x. (Q\,x) \preceq (Q'\,x)$.

A Separation Logic triple $\{H\}\ t\ \{Q\}$ is defined as follows:

Definition 1 (Separation Logic Triple) A triple $\{H\}\ t\ \{Q\}$ is short for the proposition:

$$\forall m, H'.\ (H \star H')\, m \ \Rightarrow\ \exists v, m'.\ \begin{cases} t/m \Downarrow v/m' \\ (Q\, v \star H' \star \top)\, m' \end{cases}$$

In order to understand this definition, it is useful to first read the special case where H' is the empty heap assertion. One finds that, for the Hoare triple $\{H\}\ t\ \{Q\}$ to hold, starting

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

CONSEQUENCE
$$\frac{H \preceq H' \qquad \{H'\}\; t\; \{Q'\} \qquad Q' \preceq Q}{\{H\}\; t\; \{Q\}}$$

DISCARD-POST
$$\frac{\{H\}\; t\; \{Q \star H'\}}{\{H\}\; t\; \{Q\}}$$

EXTRACT-PROP
$$\frac{P \Rightarrow \{H\}\; t\; \{Q\}}{\{[P] \star H\}\; t\; \{Q\}}$$

EXTRACT-EXISTS
$$\frac{\forall x.\; \{H\}\; t\; \{Q\}}{\{\exists x. H\}\; t\; \{Q\}}$$

VAL
$$\{[\,]\}\; v\; \{\lambda y.\; [y = v]\}$$

IF
$$\frac{\begin{array}{c} n \neq 0 \Rightarrow \{H\}\; t_1\; \{Q\} \\ n = 0 \Rightarrow \{H\}\; t_2\; \{Q\} \end{array}}{\{H\}\; (\mathsf{if}\; n\; \mathsf{then}\; t_1\; \mathsf{else}\; t_2)\; \{Q\}}$$

LET
$$\frac{\begin{array}{c} \{H\}\; t_1\; \{Q'\} \\ \forall x.\; \{Q'\, x\}\; t_2\; \{Q\} \end{array}}{\{H\}\; (\mathsf{let}\; x = t_1\; \mathsf{in}\; t_2)\; \{Q\}}$$

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

REF
$$\{[\,]\}\; (\mathsf{ref}\; v)\; \{\lambda x.\; \exists l.\; [x = l] \star l \hookrightarrow v\}$$

GET
$$\{l \hookrightarrow v\}\; (\mathsf{get}\; l)\; \{\lambda x.\; [x = v] \star l \hookrightarrow v\}$$

SET
$$\{l \hookrightarrow v'\}\; (\mathsf{set}\; (l, v))\; \{\lambda x.\; l \hookrightarrow v\}$$

Fig. 5 Reasoning rules for ordinary Separation Logic; rule APP is revised in Figure 6

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

Fig. 6 Revised reasoning rules for Separation Logic with Time Credits

from any initial heap m that satisfies the precondition H, the term t must run without error and terminate, producing some value v and some final heap m', a fragment of which[5] must satisfy the postcondition $Q\, v$.

The universal quantification over an arbitrary heap predicate H' effectively "builds the frame rule" into the interpretation of triples. That is, for $\{H\}\; t\; \{Q\}$ to hold, executing t must not affect the heap outside of the fragment described by the precondition H. To see why Definition 1 indeed imposes this heap-preservation property, let m_2 denote an arbitrary heap fragment, and instantiate H' with the predicate "$= m_2$", which characterizes the heaps that are exactly the same as m_2. Then, according to Definition 1, starting from any initial heap m of the form $m_1 \uplus m_2$, where m_1 satisfies H, the term t must run without error and terminate, producing some value v and some final heap m' of the form $m'_1 \uplus m_2$, where a fragment of m'_1 satisfies $Q\, v$.

This is a logic of total correctness: in a valid triple, the precondition must be strong enough to ensure termination. Anyway, when we extend the logic with time credits (§3.3), we impose the even stronger requirement that the computation must terminate within a certain time bound, which is encoded in the precondition and postcondition.

The reasoning rules of Separation Logic appear in Figure 5. The first part of the rules, from FRAME up to EXTRACT-EXISTS, are non-syntax-directed: they can be applied to any term t.

[5] Requiring $(Q\, v \star \top)\; m'$ is equivalent to requiring that $Q\, v$ be satisfied by some fragment of m'. This condition is more permissive than $Q\, v\, m'$. This yields extra flexibility: the postcondition need not describe all of the final heap; it may describe just a part of it. This interpretation allows validating the "garbage collection" rule, that is, rule DISCARD-POST in Figure 5.

The remaining rules are syntax-directed: there is one rule for each construct in the syntax of terms. We do not explain these rules, as they are standard (Reynolds, 2002).

Although these reasoning principles are presented in the form of inference rules, they should not be read as an inductive definition of Hoare triples. Instead, each of them is really a separate lemma, whose validity can be established, based on Definition 1. Thus:

Theorem 1 (Soundness of Separation Logic) *Each of the reasoning rules in Figure 5 is valid with respect to the interpretation of triples given by Definition 1.*

The proof of this theorem is straightforward, and has been machine-checked (Charguéraud and Pottier, 2017).

The meaning of Separation Logic triples (Definition 1), as well as one of the reasoning rules (rule APP in Figure 5), are revisited in §3.3, where time credits are introduced.

3.3 Separation Logic with Time Credits

We now extend Separation Logic with time credits. To do this, we must revisit the definition of the type Heap of heap fragments. Instead of defining Heap as a synonym for Memory, as in the previous section (§3.2), we let a heap h be a pair (m,c) of a memory m and a natural number c, which represents a number of time credits that "we" own and are allowed to spend. Thus, we posit:

$$\mathsf{Heap} \;\equiv\; \mathsf{Memory} \times \mathbb{N}.$$

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

$$\$n \quad\equiv\quad \lambda(m,c).\ m = \emptyset \wedge c = n$$

The definitions of the standard Separation Logic assertions (Figure 4) are unchanged, provided some of our notation is lifted from the level of memories m to the levels of heaps h. In the first equation, the empty heap $\emptyset$ must be understood as a pair $(\emptyset,0)$ of the empty memory and zero time credits. Similarly, in the third equation, the singleton heap $(l \mapsto v)$ must be understood as the pair $((l \mapsto v),0)$ of the singleton memory $(l \mapsto v)$ and zero time credits:

$$\begin{array}{rcl} \emptyset & \equiv & (\emptyset,0) \\ (l \mapsto v) & \equiv & ((l \mapsto v),0) \end{array}$$

In the fourth equation, the notions of disjointness of heaps and disjoint union of heaps are defined in terms of disjointness of memories and disjoint union of memories, as follows:

$$\begin{array}{rcl} (m_1,c_1) \perp (m_2,c_2) & \equiv & m_1 \perp m_2 \\ (m_1,c_1) \uplus (m_2,c_2) & \equiv & (m_1 \uplus m_2,\ c_1 + c_2) \end{array}$$

In other words, the partial commutative monoid $(\mathsf{Heap},\uplus)$ is the product of the partial commutative monoids $(\mathsf{Memory},\uplus)$ and $(\mathbb{N},+)$.

Based on these definitions, the following equalities between assertions can be proved:

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

These equalities play an essential role in proofs of programs. In particular, thanks to the left-hand equality, if at a function call the caller has m credits at hand and the callee requires

n credits, then, provided $m \geq n$ holds, one can transform $\$m$ into $\$n \star \$(m-n)$. The first conjunct is used to justify the call, while the second conjunct can be framed out during the call, so $n-m$ credits remain after the call. Furthermore, the left-hand equality, combined with rule DISCARD-POST in Figure 5, allows weakening $\$m$ to $\$n$ when $m \geq n$ holds, throwing away $m-n$ unneeded credits. This is typically useful when the branches of a conditional construct require different amounts of credits.

At this point, credits are purely an accounting device: one can carry them around, split them, join them, and so on, but they do not (yet) have any connection with the consumption of actual computational resources, such as space, energy, or time. There remains to revise the definition of Separation Logic triples (Definition 1) so as to connect the notion of credit with the number of computation steps that are taken by the program, therefore allowing credits to be interpreted as "time credits".

Three steps are required to adapt Definition 1. First, the precondition $H \star H'$ now applies not just to a memory m, but to a heap, which is now a pair (m,c) of a memory and a number of credits. Similarly, the postcondition $Q\ v \star H' \star \top$ now applies to a pair (m',c') of a memory and a number of credits. Second, the evaluation judgment $t/m \Downarrow v/m'$ is replaced with its cost-instrumented counterpart, written $t/m \Downarrow^n v/m'$, where n denotes the number of computation steps. Last but not least, we wish to keep track of a relationship between the newly-introduced variables c, c', and n. Therefore, we add the constraint $c = n + c'$, which captures the fact that the number of initially available credits is exactly the sum of the number of computation steps that are taken and the number of credits that remain. The revised definition is thus as follows.

Definition 2 (Triples in Separation Logic with Time Credits) A triple $\{H\}\ t\ \{Q\}$ is short for the following proposition:

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

In the particular case where H' is instantiated as the empty heap assertion, one finds that the Hoare triple $\{H\}\ t\ \{Q\}$ implies the following guarantee:

"*If $H\,(m,c)$ holds, then there exist n, v, and m' such that $t/m \Downarrow^n v/m'$ and $c \geq n$*".

That is, if (m,c) satisfies the precondition H, then the program t, executed in the initial memory m, runs safely and terminates after at most c computation steps. In other words, the number of credits that one (fictionally) initially supplies is an upper bound for the cost of running the program.

All of the reasoning rules shown in Figure 5 except APP can be proved sound with respect to the revised definition of triples. APP, as shown in Figure 5, is unsound. Indeed, a function call costs 1 at runtime: this must be reflected in the reasoning rule by consuming one credit. Therefore, we replace APP with APP-PAY (Figure 6).

Theorem 2 (Soundness of Separation Logic with Time Credits) *Each of the reasoning rules in Figure 5 except* APP *is valid with respect to the interpretation of triples given by Definition 2. So is rule* APP-PAY *in Figure 6.*

The proof of this theorem is again straightforward (Charguéraud and Pottier, 2017).

$$\begin{aligned}
[\![v]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ H \preceq Q\,v) && (1)\\
[\![\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\quad (b = \mathsf{true} \Rightarrow [\![t_1]\!]\,H\,Q)\ ) && (2)\\
&\qquad\qquad\qquad\quad \wedge\ (b = \mathsf{false} \Rightarrow [\![t_2]\!]\,H\,Q) &&\\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ \exists Q'.\ [\![t_1]\!]\,H\,Q' \wedge \forall x.\ [\![t_2]\!]\,(Q'\,x)\,Q) && (3)\\
[\![v_1\ v_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ \mathsf{App}\,v_1\,v_2\,H\,Q) && (4)\\
[\![\mathsf{let}\ f = (\mu f.\lambda x.t_1)\ \mathsf{in}\ t_2]\!] &\equiv \mathsf{local}\,(\lambda HQ.\ \forall f.(\forall x.\ ([\![t_1]\!] \subseteq \mathsf{App}\,f\,x)) \Rightarrow [\![t_2]\!]\,H\,Q) && (5)
\end{aligned}$$

$$\text{where}\quad F_1 \subseteq F_2 \ \equiv\ \forall HQ.\ F_1\,H\,Q \Rightarrow F_2\,H\,Q$$

Fig. 7 Construction rules for characteristic formulae

$$\begin{array}{llll}
\forall v. & \mathsf{App\ ref}\ v & [\,] & (\lambda r.\ (r \hookrightarrow v))\\
\forall rv. & \mathsf{App\ get}\ r & (r \hookrightarrow v) & (\lambda x.\ (r \hookrightarrow v) \star [x = v])\\
\forall rvv'. & \mathsf{App\ set}\ (r, v) & (r \hookrightarrow v') & (\lambda\_.\,(r \hookrightarrow v))
\end{array}$$

Fig. 8 The specifications of the primitive operations, expressed as axioms about App

4 Characteristic Formulae

We now give a characteristic-formula presentation of the Separation Logic described in the previous section (§3). We first recall the construction of characteristic formulae in the absence of time credits (§4.1), then adapt this construction slightly so as to account for time credits (§4.2).

4.1 Ordinary Characteristic Formulae

As explained earlier (§2.5), the characteristic formula $[\![t]\!]$ is intended to denote the set of all valid specifications of the term t. That is, for any precondition H and postcondition Q, the proposition $[\![t]\!]\,H\,Q$ should hold if and only if the Hoare triple $\{H\}\ t\ \{Q\}$ is valid. The left-to-right implication is Theorem 3, further on. The right-to-left implication corresponds to completeness: it is established in the first author's Ph.D. thesis (Charguéraud, 2010), but is not essential (and not proved) in the present paper.

The characteristic formula $[\![t]\!]$ is constructed in a compositional manner, by induction over the syntax the term t. It is defined by the equations in Figure 7.

Two auxiliary predicates appear in characteristic formulae. The predicate $\mathsf{App}\,f\,v\,H\,Q$ is defined as $\{H\}\ f\,v\ \{Q\}$. As far as the end user is concerned, it is abstract: it is never necessary to unfold it (or to unfold the definition of a Hoare triple) in the course of reasoning about a program. The predicate transformer local is explained further on and can be ignored for the moment.

The equations in Figure 7 provide a syntax-directed reformulation of the reasoning rules. For example, equation (3) mimics the two premises of the reasoning rule LET. Naturally, it involves an existential quantification over the intermediate assertion Q', which appears in the premises of LET, but not in its conclusion. Equations (4) and (5) concern function definitions and function applications. When a function is defined, an App assumption is introduced (5); when a function is called, an App assumption is exploited (4). In short, equation (5) states that if the body t_1 of the function f satisfies a specification $\{H'\}\cdot\{Q'\}$, then one can assume that a call $f\,x$ satisfies the same specification.

The characteristic formula for a recursive function definition, given in (5), does not build in an induction principle. This is in contrast with the traditional Hoare logic rule for reasoning about a recursive function, which requires the user to supply a specification (as well as a termination measure) and requires proving that the function body satisfies this specification under the hypothesis that every recursive call satisfies it. Here, the characteristic formula is constructed without any input from the user. In order to prove that a recursive function satisfies a certain specification, the user must state this fact as a lemma and prove it by induction, using the induction facility of the host system—in our case, Coq. Such an induction is typically conducted over an argument of the function, or over an auxiliary variable that appears in the specification. Thus, the induction hypothesis allows reasoning about recursive calls on “smaller” arguments.

Because functions are values, equation (4) overlaps with the combination of equations (3) and (1). This overlap is not problematic: all of the equations are correct anyway. That said, one should give higher priority to equation (4), when applicable. Indeed, after App is made abstract, equation (4) becomes more informative than the combination of equations (3) and (1), as it introduces a hypothesis about App f that cannot otherwise be proved outside of the abstraction barrier.

When App is made abstract, the reasoning rules REF, GET, and SET of Figure 5 must be exposed in the forms of axioms about App, shown in Figure 8. These properties must be exposed at this point because, outside of the abstraction barrier, they cannot be proved.

The predicate transformer local, which appears at the head of every node in a characteristic formula, allows applying the structural rules of Separation Logic at any point during reasoning. For example, to apply the FRAME rule while reasoning, one may exploit the following property of local: $[\![t]\!]\, H_1\, Q_1 \;\Rightarrow\; \mathsf{local}\, [\![t]\!]\, (H_1 \star H_2)\, (Q_1 \star H_2)$. For each of the other structural rules, namely CONSEQUENCE, DISCARD-POST, EXTRACT-PROP, and EXTRACT-EXISTS, local enjoys a corresponding property. Should several structural rules be applied successively, local can be duplicated as many times as needed, thanks to the logical equivalence $\mathsf{local}\, [\![t]\!]\, H\, Q = \mathsf{local}\, (\mathsf{local}\, [\![t]\!])\, H\, Q$. Should no structural rule be applied, local may simply be eliminated, thanks to the implication $[\![t]\!]\, H\, Q \;\Rightarrow\; \mathsf{local}\, [\![t]\!]\, H\, Q$. These properties follow from the definition of local, which, for reference, is provided below:

$$\mathsf{local}\, F\, H\, Q \quad \equiv \quad \forall h.\, H\, h \Rightarrow \exists H_1 H_2 Q_1.\, \begin{cases} (H_1 \star H_2)\, h \\ F\, H_1\, Q_1 \\ Q_1 \star H_2 \preceq Q \star \top \end{cases}$$

The CFML tool parses OCaml source code and produces its characteristic formula, expressed as a Coq proposition. Furthermore, CFML comes with a library of Coq tactics which allows the user to reason at a high level, that is, in terms of the reasoning rules of Separation Logic. There is no need for the user to read or understand characteristic formulae.

The manner in which CFML constructs characteristic formulae for OCaml code closely follows the above description, with one major addition, namely, a transparent type-directed lifting of OCaml values as Coq values. In the above presentation, every value v is viewed in Coq as an object of type Val. Thus, every postcondition Q is viewed in Coq as a formula of type Val $\rightarrow$ Heap $\rightarrow$ Prop. This forces the end user to explicitly work with predicates that classify values, e.g., “this value represents an integer”, “this value represents a list of integers”, and so on. By constrast, CFML exploits the type information present in OCaml programs so as to transparently lift every OCaml value into a Coq value of suitable type. Thus, an OCaml program of type `int list` is described in Coq using a postcondition of type

$(\mathsf{list}\,\mathsf{int}) \rightarrow \mathsf{Heap} \rightarrow \mathsf{Prop}$. Thus, the end user is presented with the illusion that the program produces "a list of integers", as opposed to "a value that represents a list of integers".

Through this lifting, every OCaml function is lifted as a Coq value of abstract type func. By default, nothing is known about this type or its inhabitants. The only way for the user to reason about functions is via the predicate App, whose type is $\forall A\,B.\ \mathsf{func} \rightarrow A \rightarrow (\mathsf{Heap} \rightarrow \mathsf{Prop}) \rightarrow (B \rightarrow \mathsf{Heap} \rightarrow \mathsf{Prop}) \rightarrow \mathsf{Prop}$.

For more information on CFML, the reader is referred to Charguéraud's paper (2013).

4.2 Characteristic formulae with Time Credits

To ensure that a time credit effectively represents "a right to perform one function call", we must enforce the spending of one credit at every function call. One way of achieving this is by modifying the characteristic formula generator. We alter equation (5) above to impose the consumption of one credit at the beginning of every function body:

$$[\![\mathsf{let}\ f = (\mu f.\lambda x.t_1)\ \mathsf{in}\ t_2]\!] \ \equiv\ \mathsf{local}\,(\lambda HQ.\ \forall f.\,(\forall x.\ (\mathsf{Tick}\,[\![t_1]\!] \subseteq \mathsf{App}\,f\,x)) \Rightarrow [\![t_2]\!]\,H\,Q)$$

where

$$\mathsf{Tick}\,F \ \equiv\ \lambda HQ.\ \exists H'.\ H \preceq \$1 \star H' \ \wedge\ F\,H'\,Q$$

The calculus presented here does not have iteration constructs. If it did, then a similar modification would be required so as to enforce the spending of one credit in every loop iteration.

An alternative way of enforcing spending, which does not require any change to the characteristic formula generator, is to preprocess the source program. The transformation inserts a call to an abstract function, named pay, at the beginning of every function body. pay has type `unit -> unit`. It is analogous to Danielsson's "tick" (2008).

One, and only one, specification is provided for pay. It causes one credit to be consumed when pay is invoked:

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

The precondition $\$1$ demands one time credit, while the postcondition $[\,]$ is empty. Thus, when reasoning about a call to pay, the user has no choice but to exploit this specification and give away one time credit.

The two approaches described above are logically equivalent; we choose the former, but the latter would work equally well.

The first author has proved that ordinary characteristic formulae are sound with respect to ordinary Separation Logic (Charguéraud, 2010). We extend this theorem so as to account for time credits.

Theorem 3 (Soundness of Characteristic Formulae)

$$\forall t, H, Q.\quad [\![t]\!]\,H\,Q \ \Rightarrow\ \{H\}\,t\,\{Q\}$$

The proof is straightforward and has been machine-checked (Charguéraud and Pottier, 2017).

```
(* The type "data" corresponds to the type variable "'a" in the OCaml code. *)
(* The type "elem" corresponds to the type "'a elem" in the OCaml code. *)

Implicit Types D : set elem.
Implicit Types R : elem → elem.
Implicit Types V : elem → data.

Definition UF D R V : heap → Prop :=
  ... (* "UF D R V" should be viewed as an abstract assertion. *)

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

Theorem UF_create : ∀V, [ ] ⊳ UF ∅ id V.

Theorem UF_join : ∀D1 R1 V1 D2 R2 V2,
  UF D1 R1 V1 ⋆ UF D2 R2 V2 ⊳
  UF
    (D1 ∪ D2)
    (fun x ⇒ If x ∈ D1 then R1 x else R2 x)
    (fun x ⇒ If x ∈ D1 then V1 x else V2 x)
  ⋆ [ D1 ∩ D2 = ∅ ].

Theorem make_spec : ∀D R V v,
  app UnionFind_ml.make [v]
    PRE (UF D R V ⋆ $3)
    POST (fun x ⇒
     UF (D ∪ {x} ) R (fun z ⇒ If R z = R x then v else V z)
     ⋆ [ x ∉ D ]).

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

Theorem get_spec : ∀D R V x, x ∈ D →
  app UnionFind_ml.get [x]
    PRE (UF D R V ⋆ $(2 ∗ alpha (card D) + 5))
    POST (fun v ⇒ UF D R V ⋆ [ v = V x ]).

Theorem set_spec : ∀D R V x v, x ∈ D →
  app UnionFind_ml.set [x v]
    PRE (UF D R V ⋆ $(2 ∗ alpha (card D) + 5))
    POST (# UF D R (fun z ⇒ If R z = R x then v else V z)).

Theorem eq_spec : ∀D R V x y, x ∈ D → y ∈ D →
  app UnionFind_ml.eq [x y]
    PRE (UF D R V ⋆ $(4 ∗ alpha (card D) + 9))
    POST (fun b ⇒ UF D R V ⋆ [ b = true ↔ R x = R y ]).

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

Fig. 9 Complete specification of Union-Find

5 Specification of Union-Find

Our formal specification of the Union-Find library, shown in Figure 9, is expressed in Coq. Using Coq sections and notations, we set things up so that the Coq types `data` and `elem` respectively correspond to the OCaml types `'a` and `'a elem` in the code of Figure 1. The type `data` is a parameter: the specification is polymorphic with respect to it, and it is up to the client to instantiate it. The type `elem`, on the other hand, is abstract: the client is not supposed to know how it is defined.

The specification begins with the declaration of a representation predicate, `UF`. Like `elem`, it is abstract: the client is not supposed to know how it is defined. (We present and explain its definition in §7.) The Separation Logic assertion $\mathtt{UF}\,D\,R\,V$ claims the existence (and unique ownership) of a Union-Find data structure, whose current state is summed up by the parameters D, R and V.

The parameter D, whose type is $\mathtt{elem} \to \mathsf{Prop}$, is the domain of the data structure, that is, the set of all elements. The parameter R, whose type is $\mathtt{elem} \to \mathtt{elem}$, maps every element to its representative. The parameter V, whose type is $\mathtt{elem} \to \mathtt{data}$, maps every element to the datum that is carried by its equivalence class. In Coq, we assume that the type `data` is inhabited. This benign assumption allows us to view V as a total function whose behavior outside of D is irrelevant.

Since R maps an element to its representative, we expect it to be an idempotent function of the set D into itself. Furthermore, although this is in no way essential, we decide that R should be the identity outside D. Finally, because V maps an element to a datum that is associated with its equivalence class, we expect V to be compatible with R. These properties are advertised to the client via the first theorem in Figure 9, namely `UF_properties`. Its statement is of the form $\mathtt{UF}\,D\,R\,V \preceq \mathtt{UF}\,D\,R\,V \star [P]$, which means that, if one owns $\mathtt{UF}\,D\,R\,V$, then one can deduce that P holds. (Recall that $\preceq$ denotes entailment of assertions and that $[P]$ denotes a pure assertion.) Thus, `UF_properties` states that, if one owns $\mathtt{UF}\,D\,R\,V$, then D, R and V must enjoy the properties listed above.

The next theorem, `UF_create`, asserts that out of nothing one can create an empty Union-Find data structure. Its statement is, again, an entailment. It can be exploited via the rule of CONSEQUENCE of Separation Logic (Figure 5). Creating an empty Union-Find data structure is a "ghost" operation: indeed, the OCaml code does not explicitly offer such an operation. Yet, it is essential: without it, the library would be unusable, since `UF` appears in the precondition of every operation other than `UF_create`.

In our conference paper (Charguéraud and Pottier, 2015), which follows the classic proof (Tarjan, 1999; Cormen et al, 2009), `UF_create` takes a parameter N, which becomes a (fixed) upper bound on the number of elements that can ever exist in this Union-Find data structure. The parameter N must be fixed at this time because it appears in the definition of the potential function Φ. In contrast, in the present paper, we follow Alstrup *et al.*'s proof (2014). This removes the need to fix N in advance and makes our specification easier to understand and to use.

The third theorem, `UF_join`, represents another "ghost" operation. It asserts that two disjoint (therefore independent) Union-Find data structures can be viewed as a single Union-Find data structure. (The construct `If P then e1 else e2`, where `P` is in Prop, is a nonconstructive conditional. It is defined using the law of excluded middle and Hilbert's ε operator.) Quite remarkably, the Union-Find data structure supports merging at zero cost! Although we do not have enough experience to tell whether this theorem might be useful to some clients, we believe that it deserves to be stated and proved. Its proof is conceptually trivial, and does not require any change to the potential function, but still requires a nontrivial amount of work, as one must dig deep down into the definition of every internal invariant and check that it is preserved by joining.

In principle, it should be possible to offer the converse ghost operation: that is, to allow conceptually splitting a Union-Find data structure into two independent Union-Find data structures, provided the split respects the equivalence classes. We have not done so at this time.

Next comes the specification of the OCaml function `make`. The theorem `make_spec` refers to `UnionFind_ml.make`, which is defined for us by CFML and has type func (§4.1). It states that `UnionFind_ml.make` satisfies a certain triple, thereby describing the behavior of `make`. The precondition is the conjunction of $\mathtt{UF}\,D\,R\,V$, which describes the pre-state, and of $\$3$, which indicates that `make` works in constant time. In the postcondition, x denotes the element returned by `make`. The postcondition describes the post-state via the assertion $\mathtt{UF}\,D'\,R\,V'$, where D' is $D \cup \{x\}$, as the domain has grown, and where V' maps x to the user-supplied datum v and elsewhere coincides with V. The postcondition also asserts that x is new, that is, distinct from all previous elements. The new element is its own representative: by our convention, R must be the identity outside D, so $R\,x$ must be x.

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

The specifications of `get` and `set` are analogous to that of of `find`. The function `get` returns the datum v associated with the element x. This is expressed by the equation $v = V\,x$ in the postcondition. The function `set` updates the datum associated with (the equivalence class of) x. This is expressed by replacing V in the postcondition with V', where V' maps every element in the equivalence class of x to v and elsewhere coincides with V.

The specification of `eq` indicates that this function returns a Boolean result, which tells whether the elements x and y are in the same equivalence class. Its cost is one (for calling `eq`) plus the cost of two calls to `find`.

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

The function `link` is internal, so its specification (given in §7) is not public.

6 Mathematical Analysis of Disjoint Set Forests

We carry out a mathematical analysis of disjoint set forests. This is a Coq formalization of the classic proof found in textbooks (Tarjan, 1999; Cormen et al, 2009) and of Alstrup *et al.*'s proof (2014). The two proofs have the same overall structure, differing only in the details of the potential function, so a large part of the effort is shared between them. Our conference paper shows the classic proof; in this paper, we present Alstrup *et al.*'s version. This mathematical analysis is independent of the OCaml code (Figure 1) and of the content of the previous sections (§3–§5). For brevity, we elide many details; we focus on the main definitions and highlight a few lemmas.

6.1 Disjoint Set Forests as Graphs

We model a disjoint set forest as a graph. The nodes of the graph inhabit a type V which is an implicit parameter throughout this section (§6). As in §5, the domain of the graph is represented by a set D of nodes. The edges of the graph are represented by a relation F between nodes. Thus, $F\,x\,y$ indicates that there is an edge from node x to node y.

The predicate `path` F is the reflexive, transitive closure of F. Thus, `path` $F\,x\,y$ indicates the existence of a path from x to y. A node x is a *root* if it has no successor. A node x is *represented by* a node r if there is a path from x to r and r is a root.

```
Definition is_root F x := ∀y, ¬ F x y.
Definition is_repr F x r := path F x r ∧ is_root F r.
```

Several properties express the fact that the graph represents a disjoint set forest. First, the relation F is *confined* to D: whenever there is an edge from x to y, the nodes x and y are members of D. Second, the relation F is *functional*: every node has at most one parent. Finally, the relation `is_repr` F is *defined*: every node x is represented by some node r. This ensures that the graph is acyclic. The predicate `is_dsf` is the conjunction of these three properties:

```
Definition confined D F :=
  ∀x y, F x y → x ∈ D ∧ y ∈ D.
Definition functional F :=
  ∀x y z, F x y → F x z → y = z.
Definition defined R :=
  ∀x, ∃y, R x y.
Definition is_dsf D F :=
  confined D F ∧ functional F ∧ defined (is_repr F).
```

6.2 Correctness of Path Compression

The first part of our mathematical analysis is mostly concerned with the functional correctness of linking and path compression. Here, we highlight a few results on compression. Compression assumes that there is an edge between x and y and a path from y to a root z. It replaces this edge with a direct edge from x to z. We write `compress` $F\,x\,z$ for the relation that describes the edges after this operation: in set-theoretic terms, it is defined as $F \setminus \{(x,\_)\} \cup \{(x,z)\}$.

We prove that, although compression destroys some paths in the graph (namely, those that used to go through x), it preserves the relationship between nodes and roots. More precisely, if v has representative r in F, then this still holds in the updated graph `compress` $F\,x\,z$.

```
Lemma compress_preserves_is_repr : ∀D F x y z v r,
  is_dsf D F → F x y → is_repr F y z →
  is_repr F v r → is_repr (compress F x z) v r.
```

It is then easy to check that compression preserves `is_dsf`.

6.3 Ranks

In order to perform linking-by-rank and to reason about it, we attach a rank to every node in the graph. To do so, we introduce a function K of type $V \to \mathbb{N}$. This function satisfies a number of interesting properties. First, because linking makes the node of lower rank a child of the node of higher rank, and because the rank of a node can increase only when this node is a root, ranks increase along graph edges. Furthermore, a rank never exceeds $\log |D|$. Indeed, if a root has rank p, then its tree has at least 2^p elements. We record these properties using a new predicate, called `is_rdsf`, which extends `is_dsf`. This predicate also records the fact that D is finite. Finally, we find it convenient to impose that the function K be uniformly zero outside of the domain D.

```
Definition is_rdsf D F K :=
   is_dsf D F ∧
   (∀ x y, F x y → K x < K y) ∧
   (∀ r, is_root F r → 2^(K r) ≤ card (descendants F r)) ∧
   finite D ∧
   (∀ x, x ∉ D → K x = 0).
```

It may be worth noting that, even though at runtime only roots carry a rank (Figure 1), in the mathematical analysis, the function K maps every node to a rank. The value of K at nonroot nodes can be thought of as "ghost state".

6.4 Ackermann's Function and its Inverse

For the amortized complexity analysis, we need to introduce Ackermann's function, written $A_k(x)$, where $k \geq 0$. Let us write $f^{(i)}$ for the i-th power of a function f, which is defined by $f^{(0)}(x) = x$ and $f^{(i+1)}(x) = f(f^{(i)}(x))$. According to Tarjan (1999), Ackermann's function is defined as follows:

$$A_0(x) = x+1 \qquad\qquad A_{k+1}(x) = A_k^{(x+1)}(x)$$

Informally, $A_{k+1}(x)$ can also be written $A_k(A_k(...A_k(x)...))$, where there are $x+1$ nested applications of A_k.

This is transcribed in Coq as follows. We write `iter` $i\,f$ for $f^{(i)}$.

```
Definition A k := iter k (fun f x ⇒ iter (x+1) f x) (fun x ⇒ x+1).
```

That is, A_k is the k-th power of $\lambda f.\lambda x.f^{(x+1)}(x)$, applied to A_0. The leftmost use of `iter` is at type $\mathbb{N} \to \mathbb{N}$; the rightmost one is at type $\mathbb{N}$. This definition is well-known; see, for instance, Hutton (1999, §5.2) or Bertot and Castéran (2004, §4.3.3.2).

The inverse of Ackermann's function, written $\alpha(n)$, maps n to the smallest k such that $A_k(1) \geq n$ holds. Below, `mmin le` denotes the minimum element of a nonempty subset of $\mathbb{N}$. The function `mmin` is defined using Hilbert's ε operator.

```
Definition alpha n := mmin le (fun k ⇒ A k 1 ≥ n).
```

Alstrup *et al.* (2014) define $\alpha_r(n)$ as one plus the smallest k such that $A_k(r) \geq n+1$ holds.

```
Definition alphar r n := mmin le (fun k ⇒ A k r ≥ n + 1) + 1.
```

In the case where r is 1, which is of particular interest to us, one finds $\alpha_1(n) = \alpha(n+1)+1$. Unfortunately, Alstrup *et al.* do not index Ackermann's function A_k in the same way as Tarjan and we do. They let k count from 1 and up, whereas, following Tarjan, we let k count from 0 and up. For this reason, our definitions differ slightly from those found in Alstrup *et al.*'s paper.

6.5 Potential

For every node x, Tarjan (1999) and Alstrup *et al.* (2014) write $p(x)$ for the *parent* of x in the forest. If x is not a root, $p(x)$ is uniquely defined. We define $p(x)$ using Hilbert's ε operator:

```
Definition p F x := epsilon (fun y ⇒ F x y).
```

Thus, `p F x` is formally defined for every node `x`, but the characteristic property `F x (p F x)` can be exploited only if one can prove that `x` has a parent.

Alstrup *et al.*'s definition of the potential function (2014, §4.1) relies on a few auxiliary definitions.

First, Alstrup *et al.* introduce an integer parameter r, which must satisfy $r \geq 1$. Later on (§7), we let r be 1, because that is good enough for our purposes. For now, we work with an arbitrary r, because this is more general, allows us to follow Alstrup *et al.* more closely, and requires no extra effort. Alstrup *et al.* define the "adjusted rank" of a node x as follows: $rank_r(x)$ is obtained by adding r to the rank of x.

Then, they define the *level* and *index* of a nonroot node x. These definitions involve the adjusted ranks of x and of its parent, as follows:

1. The level of x, written $k(x)$, is one plus the largest k such that $rank_r(p(x)) \geq A_k(rank_r(x))$ holds. It lies in the semi-open interval $[1, \alpha_r(rank_r(p(x))))$.
2. The index of x, written $i(x)$, is the largest i such that $rank_r(p(x)) \geq A^{(i)}_{k(x)-1}(rank_r(x))$ holds. It lies in the interval $[1, rank_r(x)]$.

The formal definitions, shown below, rely on the function `mmax`, which is the dual of `mmin` (§6.4). The parameters r, F, K are hidden by the use of Coq sections and notations.

```
Definition rankr x :=
  K x + r.
Definition k x :=
  mmax le (fun k ⇒ rankr (p x) ≥ A k (rankr x)) + 1.
Definition i x :=
  mmax le (fun i ⇒ rankr (p x) ≥ iter i (A (k x − 1)) (rankr x)).
```

The potential of a node is written $\phi(x)$. Following Alstrup *et al.*, if x is a root, then $\phi(x)$ is $\alpha_r(rank_r(x)) \cdot (rank_r(x)+1)$. If x is not a root, then $\phi(x)$ is $(\alpha_r(rank_r(x)) - k(x)) \cdot rank_r(x) - i(x) + 1$ if the condition $\alpha_r(rank_r(x)) = \alpha_r(rank_r(p(x)))$ holds, and 0 otherwise.

```
Definition phi x :=
  If is_root F x then
     alphar (rankr x) ∗ (rankr x + 1)
  else If alphar (rankr x) = alphar (rankr (p x)) then
    (alphar (rankr x) − k x) ∗ (rankr x) − i x + 1
  else
    0.
```

The total potential Φ is obtained by summing ϕ over all nodes in the domain D.

```
Definition Phi := fold (monoid_ plus 0) phi D.
```

6.6 Potential Analysis

The definition of the representation predicate `UF`, which we present later on (§7), explicitly mentions Φ. It states that, between two operations, we have Φ time credits at hand. Thus, when we try to prove that every operation preserves `UF`, as claimed earlier (§5), we are naturally faced with the obligation to prove that the initial potential Φ, plus the number of credits brought by the caller, covers the new potential Φ', plus the number of credits consumed during the operation:

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

We check that this property holds for all operations. The two key operations are linking and path compression. In the latter case, we consider not just one step of compression (that is, updating one graph edge, as in §6.2), but "iterated path compression", that is, updating the parent of every node along a path, as performed by `find` in Figure 1. To model iterated path compression, we introduce the predicate $\texttt{ipc}\,F\,x\,d\,F'$, which means that, if the initial graph is F and if one performs iterated path compression starting at node x, then one performs d individual compression steps and the final graph is F'.

$$\frac{\texttt{is\_root}\,F\,x}{\texttt{ipc}\,F\,x\,0\,F} \qquad\qquad \frac{F\,x\,y \qquad \texttt{is\_repr}\,F\,y\,z \qquad \texttt{ipc}\,F\,y\,d\,F' \qquad F'' = \texttt{compress}\,F'\,x\,z}{\texttt{ipc}\,F\,x\,(d+1)\,F''}$$

On the one hand, the predicate `ipc` is a paraphrase, in mathematical language, of the recursive definition of `find` in Figure 1, so it is easy to argue that "`find` implements `ipc`". This is done as part of the verification of `find` (§7). On the other hand, by following Alstrup *et al.*'s proof (2014), we establish the following key lemma, which sums up the amortized complexity analysis of iterated path compression:

```
Lemma amortized_cost_of_iterated_path_compression: ∀D F K x,
  is_rdsf D F K →
  x ∈ D →
  ∃d F',
  ipc F x d F' ∧
  Phi D F K + 2 ∗ alpha (card D) + 4 ≥ Phi D F' K + d + 1.
```

(Here, the parameter r of Alstrup *et al.*'s analysis has been fixed; it is 1.) This lemma states that, in any initial state described by D, F, K and for any node x, (1) iterated path compression is possible, requiring a certain number of steps d and taking us to a certain final state F', and (2) more interestingly, the inequality "$\Phi +$advertised cost $\geq \Phi' +$actual cost" holds. Indeed, $2\alpha(|D|)+4$ is the advertised cost of `find` (Figure 9), whereas $d+1$ is its actual cost, because iterated path compression along a path of length d involves $d+1$ calls to `find`.

7 Verifying the Code

To prove that the OCaml code in Figure 1 satisfies the specification in Figure 9, we first define the predicate `UF`, then establish each of the theorems in Figure 9.

7.1 Definition of the representation predicate

In our OCaml code (Figure 1), we have defined `'a elem` as a synonym for `'a content ref`, and we have defined `'a content` as an algebraic data type whose data constructors are `Link of 'a elem | Root of rank * 'a`. These definitions are mutually recursive. The CFML tool automatically mirrors these type definitions in Coq as follows. First, it defines `content_ A_` as an inductive data type. Then, it defines `elem_ A_` in terms of `content_ A_`.

```
Inductive content_ A_ :=
  | Link : ref_ (content_ A_) → content_ A_
  | Root : rank_ → A_ → content_ A_.
Definition elem_ A_ :=
  ref_ (content_ A_).
```

The type `ref_ A_` is defined in CFML's library as a synonym for `loc`, an abstract type of memory locations. In this definition, the type parameter `A_` is unused. This explains why Coq accepts the definition of the type `content_ A_`. This type is not truly recursive: it is just a sum type. Its left injection, `Link`, carries a memory location; its right injection, `Root`, carries an integer rank and a datum of type `A_`.

The type `data` of §5 corresponds to `A_` in the above definitions. The type `elem` of §5 is an abbreviation for `elem_ A_`, which itself is convertible with `loc`.

To define the assertion `UF` $D\,R\,V$, we need two auxiliary predicates, `Inv` and `Mem`.

The auxiliary predicate `Inv D F K R V` relates our mathematical view of the data structure, which is encoded by the parameters D, F, K, V, and the view that is exposed to the client, which is encoded by D, R, V. In the client's view, K disappears altogether, because the client does not need to know about ranks. Furthermore, F is replaced with R, because the client cares about the mapping from elements to representatives, but does not need a description of the entire graph. The two views are related as follows:

```
Definition Inv D F K R V :=
  is_rdsf D F K ∧
  (∀ x, is_repr F x (R x)) ∧
  (∀ x, V x = V (R x)).
```

The first conjunct is the invariant `is_rdsf D F K`, which describes a ranked disjoint set forest (§6.3). The second conjunct requires the function R and the relation `is_repr` F to agree. The last conjunct requires the function V to be compatible with the equivalence relation induced by R, so that V can be thought of as a mapping of equivalence classes to user data.

The auxiliary predicate `Mem D F K V M` relates a model of a disjoint set forest, represented by D, F, K, V, and a model of the memory, represented by a finite map M of memory locations to values of type `content_ data`. (This view of memory is imposed by CFML.) M maps a location x to either `Link` y for some location y or `Root` $k\,v$ for some integer k and user datum v. The predicate `Mem D F K V M` explains how to interpret the contents of memory as a disjoint set forest. It asserts that M has domain D, that $M(x) =$ `Link` y implies the existence of an edge of x to y, and that $M(x) =$ `Root` $k\,v$ implies that x is a root, that x has rank k, and that the user datum associated with x is v.

```
Definition Mem D F K V M :=
  dom M = D ∧
  ∀x, x ∈ D →
  match M[x] with
```

```
Lemma link_spec : ∀D R V x y,
  x ∈ D → y ∈ D → R x = x → R y = y →
  app UnionFind_ml.link [x y]
    PRE (UF D R V ⋆ $3)
    POST (fun z ⇒
          UF D (fun w ⇒ If R w = R x ∨ R w = R y then z else R w)
               (fun w ⇒ If R w = R x ∨ R w = R y then V z else V w)
          ⋆ [ z = x ∨ z = y ]
    ).
```

Fig. 10 Specification of the function `link`

```
  | Link y ⇒ F x y
  | Root k v ⇒ is_root F x ∧ k = K x ∧ v = V x
  end.
```

At last, we are ready to define the assertion UF $N\,D\,R$:

```
Definition UF D R V := ∃F K M,
  Group Ref M ⋆
  [ Inv D F K R V ] ⋆
  [ Mem D F K V M ] ⋆
  $(Phi D F K).
```

The first conjunct asserts the existence in the heap of a group of reference cells, collectively described by the map M. (The predicates `Group` and `Ref` are provided by CFML's library. `Group` is an iterated separating conjunction.) The second conjunct constrains the graph (D,F,K) to represent a valid disjoint set forest whose roots are described by R, while the third conjunct relates the graph (D,F,K) with the contents of memory (M). These two conjuncts are pure assertions. The last conjunct asserts that we have Φ time credits at hand. The definition is existentially quantified over F, K, and M, which are not exposed to the client.

7.2 Verification

There remains to prove that every public operation meets its specification, that is, to establish each of the theorems in Figure 9. To this end, we must also provide a specification for the internal function `link`; its statement appears in Figure 10. It is essentially the same as the specification of `union`. The differences are that `link` must be applied to two roots and has constant cost. It is worth noting that `link` requires three time credits, even though it may seem that one credit should be enough, as `link` makes no function calls. The extra two credits are used to cover a possible increase in the potential of the data structure.

The proofs of these theorems involve a mix of standard Coq tactics and tactics provided by CFML to prove Separation Logic entailments and apply the reasoning rules of Separation Logic. These proofs are rather short, as all we need to do at this point is to establish a correspondence between the operations performed by the imperative code and the mathematical analysis that we have already carried out. The structure of these proofs follows the structure of the OCaml code, so (in our experience) it is relatively easy to update a proof when the code evolves.

The CFML library is about 5Kloc. (We count nonblank, noncomment lines of code.) The mathematical analysis of Union-Find (§6) is about 4.3Kloc. The specification of Union-Find (Figures 9 and 10) and the verification of the code (§7) together take up about 0.8Kloc. Both CFML and our proofs about Union-Find rely on Charguéraud's TLC library, a complement to Coq's standard library. Everything is available online (Charguéraud and Pottier, 2017).

8 Related Work

8.1 Union-Find and its analysis

Disjoint set forests and linking-by-size are due to Galler and Fischer (1964), while path compression is attributed by Aho *et al.* (1974) to McIlroy and Morris. Hopcroft and Ullman (1973) study linking-by-size and path compression and establish an amortized bound of $O(\log^* N)$ per operation, where N is the length of the sequence of operations under consideration and $\log^*$ is the iterated logarithm function. Tarjan (1975) studies linking-by-rank and path compression and establishes the first amortized bound in $O(\alpha(N))$. After several simplifications (Tarjan and van Leeuwen, 1984; Kozen, 1992), this leads to a streamlined proof that appears in Tarjan's online course notes (1999) and in Cormen *et al.*'s textbook (2009).

Kaplan *et al.* (2002) and Alstrup *et al.* (2014) establish a "local" bound. That is, they bound the amortized cost of $\mathit{find}(x)$ by $O(\alpha(n))$, where n is the size of x's set. We follow Alstrup *et al.*'s proof, but let n stand for the total number of elements in the data structure, that is, the cardinal of the domain D, as this leads to a slightly simpler statement for the end user, and should still be strong enough for all practical purposes. Alstrup *et al.*'s proof has the same overall structure as the textbook proof, but offers the advantage that the definition of the potential function Φ does not depend on N, so N does not need to be fixed when the Union-Find data structure is created.

Unfortunately, to the best of our understanding, none of the modern proofs conveys much intuition, as the potential function seems to be pulled out of a hat, so to speak (§6.5). More intuition is offered by Harfst and Reingold (2000) and by Seidel and Sharir (2005), whose analysis is based on recurrence equations, and offers more and more precise bounds as one refines the analysis.

Galil and Italiano (1991) offer a survey of the literature on Union-Find and its variants.

8.2 Machine-checked proofs of Union-Find

We know of a few machine-checked proofs of the functional correctness of Union-Find. None of them deals with the algorithm's time complexity.

Conchon and Filliâtre (2007) present an efficient persistent Union-Find algorithm, based on two persistent arrays, which respectively store link and rank information. They verify both the Union-Find algorithm and the underlying persistent array data structure in Coq. The proofs are not based on the actual OCaml code, but on a (manually-constructed) Coq model of the OCaml code. This approach is analogous to ours, except CFML automates the process of modeling in Coq the meaning of an OCaml program.

The Ynot distribution (Chlipala et al, 2011) includes an imperative Union-Find implementation, based on two arrays, which respectively store link and rank information. The code and its proof (which is carried out in Separation Logic) are both written in Coq.

Lammich and Meis (2012, Chapter 18) establish the correctness of a similar Union-Find implementation. They use two arrays, which respectively store link and size information.

Krishnaswami (2012, Chapter 5) presents a (paper) proof of Union-Find. Like us, he uses heap-allocated vertices, and therefore supports the dynamic creation of new vertices. He performs path compression and linking by rank. His specification of Union-Find uses an abstract Separation Logic predicate $H(\phi)$, where ϕ is presented as a "formula" in a "little logic" whose syntax is $\phi ::= I \mid \mathrm{elt}(x,y) \mid \phi \otimes \phi$. The atom $\mathrm{elt}(x,y)$ means that the representative of x is y; the tensor $\otimes$ allows constructing conjunctions of several such facts; and I is the unit of the tensor. A moment's thought reveals that this is just a syntactic presentation of our function R, which maps every element to its representative. So, Krishnaswami's specification is, in spirit at least, relatively close to ours.

We note that Union-Find is part of the VACID-0 suite of benchmark verification problems (Leino and Moskal, 2010). However, we did not find any solution to this particular benchmark problem online.

8.3 Formal approaches to reasoning about program complexity

The idea of using a machine to assist in the complexity analysis of a program goes back at least as far back as Wegbreit (1975). He extracts recurrence equations from the program and bounds their solution. More recent work along these lines includes Le Métayer's (1988) and Danner *et al.*'s (2013). Although Wegbreit aims for complete automation, he notes that one could "allow the addition to the program of performance specifications by the programmer, which the system then checks for consistency". We follow this route.

Okasaki (1999) explains how to reason about suspensions (also known as thunks; a simple data structure that offers a form of memoization) in terms of "debits". He uses this approach in the study of many purely functional data structures. The advantage of the debit discipline is that it does not come with an affinity restriction: whereas duplicating a credit is unsound, duplicating a debit is sound. Danielsson (2008) formalizes Okasaki's idea in a type system. Pilkiewicz and Pottier (2011) show how suspensions and debits can be implemented as a library on top of mutable state and credits. Madhavan *et al.* (2017) present an automated system that infers and verifies resource bounds for higher-order functional programs with suspensions and (more generally) with memoization tables. They transform the source program to an instrumented form where the cache is explicit, so that its state can be described monotone assertions, which remain valid as the cache grows. Thus, it is possible to assert that a suspension has been forced already, so that forcing it again will have zero cost. This seems analogous in Okasaki's terminology to asserting that a suspension has zero debits, also a monotone assertion. In contrast with previous work, Madhavan *et al.* are able to keep track of aliasing (that is, equalities) between suspensions. This is required in order to obtain precise complexity bounds for lazy data structures that involve scheduling.

Type systems with support for controlling time complexity have been proposed, among others, by Dornic, Jouvelot, and Gifford (1992), by Reistad and Gifford (1994), and by Crary and Weirich (2000). In the former two systems, a function type is annotated with a "latent cost", which may depend on the size of the procedure's arguments. For instance, a function whose execution requires three steps of computation could have type $int \rightarrow^3 int$. In Crary and Weirich's system, there is a global "virtual clock", and a function type specifies the function's beginning and ending clock readings. A function whose execution requires three steps of computation could have type $\forall n.(int, n+3) \rightarrow (int, n)$. These type systems do not involve linearity (or affinity), therefore do not have a first-class notion of "time credit" and do not allow reasoning about amortized time complexity.

A Hoare logic with support for reasoning about certain measures, including instruction counts, maximum stack space, and the number of heap allocations, has been proposed by Aspinall *et al.* (2007). This is not a separation logic, though, so there are no "time credits" and no amortized reasoning. The CerCo project (Amadio and Régis-Gianas, 2011; Ayache et al, 2012; Amadio et al, 2014), mentioned earlier (§2.7), pursues closely related goals.

A line of work by Hofmann *et al.* (Hofmann and Jost, 2003; Hoffmann and Hofmann, 2010; Hoffmann et al, 2012) aims to infer amortized time and space bounds. Because emphasis is on automation, these systems are limited in the bounds that they can infer (e.g., polynomial bounds) and/or in the programs that they can analyze (e.g., without side effects; without higher-order functions). In recent work, Hoffmann, Das and Weng (2017) automatically infer tight polynomial amortized complexity bounds for nontrivial OCaml programs. Higher-order functions are supported, as far as we understand, by repeating their analysis when they are applied. Side effects are supported, albeit with the severe restriction that the time complexity of the code is not allowed to depend on mutable state.

McCarthy *et al.* (2016) present a Coq monad that allows the time complexity of a Coq computation to be expressed in its type. This approach requires the code to be correctly instrumented with "pay" instructions: this can be done either manually or automatically, using an external tool, implemented in Racket. This is analogous to the manner in which CFML instruments the OCaml code with `pay` instructions. In McCarthy *et al.*'s approach, the source code is written in Coq, which implies that it must be side-effect free, and that great care must be taken to ensure that the time complexity monad is erased when the code is extracted to OCaml. Our approach, in contrast, is based on OCaml source code, so extraction is not a problem. Furthermore, our use of Separation Logic with time credits allows us to deal with mutable state and to carry out amortized complexity arguments.

Nipkow (2015) presents machine-checked amortized analyses of several data structures, including skew heaps, splay trees and splay heaps. As he seems to be mainly interested in the mathematical analysis of a data structure, as opposed to the verification of an actual implementation, he manually derives from the code a "timing function", which represents the actual time consumed by an operation.

8.4 Separation Logic and time credits

There is a huge body of work on program verification using Separation Logic. We are particularly interested in embedding Separation Logic into an interactive proof assistant, such as Coq, where it is possible to express arbitrarily complex specifications and to perform arbitrarily complex proofs. The Union-Find case study exploits this power in several ways: we find that (say) reasoning about a complex potential function (§6) or working with iterated separating conjunctions (§7) is not significantly more difficult in Coq than on paper. In contrast, supporting iterated separating conjunctions in an automated program verifier is possible but challenging (Müller et al, 2016). The interactive approach to program verification has been explored in several projects, including Ynot (Nanevski et al, 2007, 2008a,b; Chlipala et al, 2009), Bedrock (Chlipala, 2013), and CFML (Charguéraud, 2010, 2013).

Separation Logic is modular and allows independent verification of the implementation of a data structure and of its clients, based on a common specification. Separation Logic allows expressing the fact that a data structure is uniquely-owned and (therefore) must be used in a single-threaded manner. This is particularly important for certain data structures whose amortized complexity guarantees can be subverted if this restriction is violated (Okasaki, 1999).

The idea of extending a type system with time credits, viewed as affine resources, has been proposed by Pilkiewicz and Pottier (2011). Atkey (2011) applies to the same idea to Separation Logic. The extension of Separation Logic with time credits is very modest; in fact, if the source program is explicitly instrumented by inserting calls to `pay`, no modification of the verification toolchain is required. We believe that we are the first to follow this approach in practice to perform a modular verification of functional correctness and complexity for a nontrivial data structure.

Although we have exploited one particular implementation of Separation Logic, namely CFML, the idea of time credits could equally well be implemented on top of other flavors of Separation Logic, such as Hoare Type Theory (Nanevski et al, 2007, 2008a,b; Chlipala et al, 2009), and Iris (Jung et al, 2015, 2016; Krebbers et al, 2017). In particular, because Iris has monotonic state and "hidden state" (that is, shared invariants), we believe that Pilkiewicz and Pottier's analysis of thunks in terms of time credits (2011) could be reconstructed and machine-checked in an extension of Iris with time credits.

Hoffmann, Marmar and Shao (2013) use time credits in Concurrent Separation Logic. They use the logic to prove the lock-freedom of several concurrent data structures. They show that the logic can express a "quantitative compensation scheme" whereby a thread that makes progress in an operation logically provides time credits to other threads to compensate for possible interference it may have caused. The beauty of this approach is that no ad hoc reasoning rules are needed: the only alteration with respect to ordinary Concurrent Separation Logic is in the reasoning rule for `while` loops, which states that each iteration consumes one credit.

8.5 Space

We focus on time complexity. Would it be more difficult to reason about space complexity? Whereas time, once spent, is lost forever, space can be reclaimed and reused. If it is clear in the source code where allocation and deallocation take place, then bounding space should be no more difficult than bounding time.

Stack space, for instance, seems relatively easy to reason about, because it is evident in the source code where stack frames are allocated and deallocated. The Hoare logic by Aspinall *et al.* (2007) can reportedly keep track of the maximum stack space required by a program. Carbonneaux *et al.* (2014) describe a quantitative Hoare logic which can be used to verify a stack-space bound at the level of C source code. The required constants, such as stack frame sizes, are automatically provided by the compiler. The concrete bound thus obtained is guaranteed to hold of the machine code produced by the compiler.

Heap space is similarly relatively easy to control, provided it is clear, by inspection of the source code, or by inspection of a type derivation, where deallocation or re-use take place. However, for explicit deallocation to be safe, an ad hoc type discipline must be adopted. For instance, a linear type system for guaranteeing that a functional program runs in a fixed-size heap has been proposed by Hofmann (2000) and subsequently improved and implemented (MacKenzie and Wolverson, 2003; Aspinall et al, 2008).

Controlling heap space in a garbage-collected programming language, such as OCaml, seems much more challenging. Whereas heap allocation sites are evident in the source code, it is difficult to predict when an object is deallocated, therefore difficult to bound the maximum live heap space that a program requires. Proving an OCaml program correct using

Separation Logic does not evidently help, because ownership and reachability are unrelated concepts. Reachability does not imply ownership: the program may well hold a pointer, yet have no permission to dereference it. Conversely, ownership does not imply reachability: the assertion $\exists l.(l \hookrightarrow v)$ guarantees that the program owns some heap cell, yet the program may well have discarded the address of this cell. Finally, the Separation Logic implemented in CFML imposes unique ownership of mutable memory blocks, but does not keep any track of immutable memory blocks.

9 Conclusion and Future Work

We have demonstrated that the state of the art has advanced to a point where one can (and, arguably, one should) prove not only that a library is correct but also that it meets a certain time complexity bound.

Our current approach permits amortized reasoning by default. A Hoare triple of the form $\{\$n \star I\}\ t\ \{\lambda\_.I\}$, where I is a (possibly abstract) separation logic assertion, can be informally interpreted as "the command t has amortized time complexity n". The actual worst-case cost of this command, however, could be greater than n, if the assertion I controls a certain number of credits. $\mathtt{UF}\,D\,R\,V$ is an example of such an assertion. This gives rise to two questions.

First, is it possible to indicate that a Hoare triple represents a nonamortized guarantee? The answer is positive. To guarantee such a property about the triple $\{\$n \star I\}\ t\ \{\lambda\_.I\}$, it suffices to prove that the assertion I is "creditless", that is, if I holds of a heap (m,c), then c must be zero. If I is creditless, then the execution of the command t requires at most n computation steps.

Second, is it possible, in a single triple, to express both amortized and nonamortized bounds? For instance, can one express the fact that `find` has amortized cost $O(\alpha(|D|))$ and nonamortized cost $O(\log(|D|))$, without duplicating its analysis? The answer is negative in our system as of today, but positive in principle. In order to carry out amortized and non-amortized analyses at the same time, we suggest proceeding as follows. Extend the system with a distinction between two kinds of credits, written (say) $\$1$ and $\boldsymbol{\$1}$, and impose the rule that a computation step costs one credit of each kind: that is to say, `pay` consumes $\$1 \star \boldsymbol{\$1}$. Adopt the convention that "saving" ordinary credits is permitted, where saving "heavy" credits is not. Keep the definition of $\mathtt{UF}\,D\,R\,V$, unchanged: it involves Φ ordinary credits and no heavy credits. Prove and publish the fact that $\mathtt{UF}\,D\,R\,V$ is a heavy-creditless assertion: this guarantees that the above convention is respected. Finally, establish a Hoare triple for `find`, whose precondition might be $\mathtt{UF}\,D\,R\,V \star \$(2\alpha(|D|)+4) \star \boldsymbol{\$}(\log(|D|)+1)$. Because $\mathtt{UF}\,D\,R\,V$ is heavy-creditless, this guarantees that `find` actually requires at most $\log(|D|)+1$ computation steps, and at the same time, states that its amortized cost is $2\alpha(|D|)+4$. Of course, establishing such a triple involves extra work, as one must keep track of two kinds of credits everywhere. Still, the step-by-step analysis of the code is carried out just once.

There are many directions for future work.

One could study different compression strategies (e.g., tail-recursive `find`, path splitting, path halving) and linking strategies (e.g., linking-by-size).

It would be desirable to verify a client of Union-Find, such as a first-order unification algorithm, or Kruskal's minimum spanning tree algorithm, so as to ensure that our specification of Union-Find is usable. Also, it should be possible to publish one or more simplified versions of our specification, which for many clients would be good enough. For instance, we could phrase a simplified specification in terms of a partial equivalence relation E instead of a domain D and a total function R. Indeed, every function except `find` can be given a useful specification in terms of E, and most clients likely do not need `find`. Exposing such a simplified specification would not require a new analysis of the code: it should be just a thin layer on top of our current specification.

Concerning our verification methodology, we would like to close the gap between OCaml machine integers and Coq ideal integers (§2.2). Can proof-of-work arguments (Clochard et al, 2015) be expressed in (a suitable extension of) Separation Logic with time credits?

Finally, we wish to use the big-O notation in specifications. For instance, instead of advertising that `find` requires $2 \cdot \alpha(|D|) + 4$ credits, it would be preferable to state that it requires $O(\alpha(|D|))$ credits, thus avoiding explicit mentions of the multiplicative factor 2 and the additive constant 4. The use of such concrete constants is noisy and compromises modularity, as it contaminates every (direct and indirect) user of the data structure. If, after a change in the code of `find`, the constant 4 must be replaced with 5, then every (direct and indirect) user of the Union-Find data structure must be updated! Furthermore, hiding these constants does not truly cause any loss of information, since the "computation steps" that we are counting are related to physical computation time only up to an unknown constant factor anyway.

Technically, working with the big-O notation poses several challenges. For one thing, although the meaning of this notation is well-defined for functions of one variable, it is more subtle when dealing with functions of several variables (Howell, 2008, 2012). Furthermore, the big-O notation hides several existential quantifiers, so, if used naively, causes many proofs to begin by exhibiting several witnesses, which the user must painfully guess. Engineering work is required so as to build Coq tactics that allow these witness to be built incrementally and transparently, as the proof progresses.

References

Aho AV, Hopcroft JE, Ullman JD (1974) The Design and Analysis of Computer Algorithms. Addison-Wesley

Alstrup S, Thorup M, Gørtz IL, Rauhe T, Zwick U (2014) Union-find with constant time deletions. ACM Transactions on Algorithms 11(1):6:1–6:28

Amadio R, Régis-Gianas Y (2011) Certifying and reasoning on cost annotations of functional programs. In: Foundational and Practical Aspects of Resource Analysis, Springer, Lecture Notes in Computer Science, vol 7177, pp 72–89

Amadio RM, Ayache N, Bobot F, Boender J, Campbell B, Garnier I, Madet A, McKinna J, Mulligan DP, Piccolo M, Pollack R, Régis-Gianas Y, Coen CS, Stark I, Tranquilli P (2014) Certified complexity (CerCo). In: Foundational and Practical Aspects of Resource Analysis, Springer, Lecture Notes in Computer Science, vol 8552, pp 1–18

Appel AW (1992) Compiling with Continuations. Cambridge University Press

Aspinall D, Beringer L, Hofmann M, Loidl H, Momigliano A (2007) A program logic for resources. Theoretical Computer Science 389(3):411–445

Aspinall D, Hofmann M, Konečný M (2008) A type system with usage aspects. Journal of Functional Programming 18(2):141–178

Atkey R (2011) Amortised resource analysis with separation logic. Logical Methods in Computer Science 7(2:17)

Ayache N, Amadio RM, Régis-Gianas Y (2012) Certifying and reasoning on cost annotations in C programs. In: Formal Methods for Industrial Critical Systems, Springer, Lecture Notes in Computer Science, vol 7437, pp 32–46

Baker HG (1978) List processing in real time on a serial computer. Communications of the ACM 21(4):280–294

Bertot Y, Castéran P (2004) Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer

Blelloch GE, Greiner J (1995) Parallelism in sequential functional languages. In: Functional Programming Languages and Computer Architecture (FPCA), pp 226–237

Carbonneaux Q, Hoffmann J, Ramananandro T, Shao Z (2014) End-to-end verification of stack-space bounds for C programs. In: Programming Language Design and Implementation (PLDI), pp 270–281

Charguéraud A (2010) Characteristic formulae for mechanized program verification. PhD thesis, Université Paris 7

Charguéraud A (2013) Characteristic formulae for the verification of imperative programs. Unpublished. http://www.chargueraud.org/research/2013/cf/cf.pdf

Charguéraud A, Pottier F (2015) Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: Interactive Theorem Proving (ITP), Springer, Lecture Notes in Computer Science, vol 9236, pp 137–153

Charguéraud A, Pottier F (2017) Online accompanying material. http://gallium.inria.fr/~fpottier/dev/uf/

Chlipala A (2013) The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In: International Conference on Functional Programming (ICFP), pp 391–402

Chlipala A, Malecha G, Morrisett G, Shinnar A, Wisnesky R (2009) Effective interactive proofs for higher-order imperative programs. In: International Conference on Functional Programming (ICFP), pp 79–90

Chlipala A, Malecha G, Morrisett G, Shinnar A, Sozeau M, Wisnesky R (2011) Ynot. http://ynot.cs.harvard.edu/dist/ynot-20110710.tgz

Clochard M, Filliâtre JC, Paskevich A (2015) How to avoid proving the absence of integer overflows. In: Verified Software: Theories, Tools and Experiments, Springer, Lecture Notes in Computer Science, vol 9593, pp 94–109

Conchon S, Filliâtre J (2007) A persistent union-find data structure. In: ACM Workshop on ML, pp 37–46

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to Algorithms (Third Edition). MIT Press

Crary K, Weirich S (2000) Resource bound certification. In: Principles of Programming Languages (POPL), pp 184–198

Danielsson NA (2008) Lightweight semiformal time complexity analysis for purely functional data structures. In: Principles of Programming Languages (POPL)

Danner N, Paykin J, Royer JS (2013) A static cost analysis for a higher-order language. In: Programming Languages Meets Program Verification (PLPV), pp 25–34

Dornic V, Jouvelot P, Gifford DK (1992) Polymorphic time systems for estimating program complexity. ACM Letters on Programming Languages and Systems 1(1):33–45

Galil Z, Italiano GF (1991) Data structures and algorithms for disjoint set union problems. ACM Computing Surveys 23(3):319–344

Galler BA, Fischer MJ (1964) An improved equivalence algorithm. Communications of the ACM 7(5):301–303

Guéneau A, Myreen MO, Kumar R, Norrish M (2017) Verified characteristic formulae for CakeML. In: European Symposium on Programming (ESOP), Springer, Lecture Notes in Computer Science, vol 10201, pp 584–610

Harfst GC, Reingold EM (2000) A potential-based amortized analysis of the union-find data structure. SIGACT News 31(3):86–95

Hoffmann J, Hofmann M (2010) Amortized resource analysis with polynomial potential. In: European Symposium on Programming (ESOP), Springer, Lecture Notes in Computer Science, vol 6012, pp 287–306

Hoffmann J, Aehlig K, Hofmann M (2012) Multivariate amortized resource analysis. ACM Transactions on Programming Languages and Systems 34(3):14:1–14:62

Hoffmann J, Marmar M, Shao Z (2013) Quantitative reasoning for proving lock-freedom. In: Logic in Computer Science (LICS), pp 124–133

Hoffmann J, Das A, Weng S (2017) Towards automatic resource bound analysis for OCaml. In: Principles of Programming Languages (POPL), pp 359–373

Hofmann M (2000) A type system for bounded space and functional in-place update. Nordic Journal of Computing 7(4):258–289

Hofmann M, Jost S (2003) Static prediction of heap space usage for first-order functional programs. In: Principles of Programming Languages (POPL), pp 185–197

Hopcroft JE, Ullman JD (1973) Set merging algorithms. SIAM Journal on Computing 2(4):294–303

Howell RR (2008) On asymptotic notation with multiple variables. Tech. Rep. 2007-4, Kansas State University

Howell RR (2012) Algorithms: A top-down approach. Draft

Hutton G (1999) A tutorial on the universality and expressiveness of fold. Journal of Functional Programming 9(4):355–372

Jung R, Swasey D, Sieczkowski F, Svendsen K, Turon A, Birkedal L, Dreyer D (2015) Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Principles of Programming Languages (POPL), pp 637–650

Jung R, Krebbers R, Birkedal L, Dreyer D (2016) Higher-order ghost state. In: International Conference on Functional Programming (ICFP), pp 256–269

Kaplan H, Shafrir N, Tarjan RE (2002) Union-find with deletions. In: Symposium on Discrete Algorithms (SODA), pp 19–28

Kozen DC (1992) The design and analysis of algorithms. Texts and Monographs in Computer Science, Springer

Krebbers R, Jung R, Bizjak A, Jourdan JH, Dreyer D, Birkedal L (2017) The essence of higher-order concurrent separation logic. In: European Symposium on Programming (ESOP), Springer, Lecture Notes in Computer Science, vol 10201, pp 696–723

Krishnaswami NR (2012) Verifying higher-order imperative programs with higher-order separation logic. PhD thesis, School of Computer Science, Carnegie Mellon University

Lammich P, Meis R (2012) A separation logic framework for Imperative HOL. Archive of Formal Proofs

Le Métayer D (1988) ACE: an automatic complexity evaluator. ACM Transactions on Programming Languages and Systems 10(2):248–266

Leino KRM, Moskal M (2010) VACID-0: Verification of ample correctness of invariants of data-structures, edition 0, manuscript KRML 209

MacKenzie K, Wolverson N (2003) Camelot and Grail: resource-aware functional programming for the JVM. In: Trends in Functional Programming (TFP), vol 4, pp 29–46

Madhavan R, Kulal S, Kuncak V (2017) Contract-based resource verification for higher-order functions with memoization. In: Principles of Programming Languages (POPL), pp 330–343

McCarthy JA, Fetscher B, New MS, Feltey D, Findler RB (2016) A Coq library for internal verification of running-times. In: Functional and Logic Programming, Springer, Lecture Notes in Computer Science, vol 9613, pp 144–162

Müller P, Schwerhoff M, Summers AJ (2016) Automatic verification of iterated separating conjunctions using symbolic execution. In: Computer Aided Verification (CAV), Springer, Lecture Notes in Computer Science, vol 9779, pp 405–425

Nanevski A, Ahmed A, Morrisett G, Birkedal L (2007) Abstract predicates and mutable ADTs in Hoare type theory. In: European Symposium on Programming (ESOP), Springer, Lecture Notes in Computer Science, vol 4421, pp 189–204

Nanevski A, Morrisett G, Birkedal L (2008a) Hoare type theory, polymorphism and separation. Journal of Functional Programming 18(5–6):865–911

Nanevski A, Morrisett G, Shinnar A, Govereau P, Birkedal L (2008b) Ynot: dependent types for imperative programs. In: International Conference on Functional Programming (ICFP), pp 229–240

Nipkow T (2015) Amortized complexity verified. In: Interactive Theorem Proving (ITP), Springer, Lecture Notes in Computer Science, vol 9236, pp 310–324

Okasaki C (1999) Purely Functional Data Structures. Cambridge University Press

Patwary MMA, Blair J, Manne F (2010) Experiments on union-find algorithms for the disjoint-set data structure. In: International Symposium on Experimental Algorithms (SEA), Springer, Lecture Notes in Computer Science, vol 6049, pp 411–423

Pilkiewicz A, Pottier F (2011) The essence of monotonic state. In: Types in Language Design and Implementation (TLDI)

Reistad B, Gifford DK (1994) Static dependent costs for estimating execution time. In: ACM Symposium on Lisp and Functional Programming (LFP), pp 65–78

Reynolds JC (2002) Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS), pp 55–74

Sands D, Gustavsson J, Moran A (2002) Lambda calculi and linear speedups. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, Springer, Lecture Notes in Computer Science, vol 2566, pp 60–84

Seidel R, Sharir M (2005) Top-down analysis of path compression. SIAM Journal on Computing 34(3):515–525

Tarjan RE (1975) Efficiency of a good but not linear set union algorithm. Journal of the ACM 22(2):215–225

Tarjan RE (1985) Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods 6(2):306–318

Tarjan RE (1999) Class notes: Disjoint set union

Tarjan RE, van Leeuwen J (1984) Worst-case analysis of set union algorithms. Journal of the ACM 31(2):245–281

Wegbreit B (1975) Mechanical program analysis. Communications of the ACM 18(9):528–539