Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation

Arthur Charguéraud[1] and François Pottier[2]

[1] Inria & LRI, Université Paris Sud, CNRS
[2] Inria

Abstract. Union-Find is a famous example of a simple data structure whose amortized asymptotic time complexity analysis is non-trivial. We present a Coq formalization of this analysis. 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. Reasoning in Coq about imperative OCaml code relies on the CFML tool, which is based on characteristic formulae and Separation Logic, 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.

1 Introduction

The Union-Find data structure, also known as a disjoint set forest [12], is widely used in the areas of graph algorithms and symbolic computation. 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 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 (and returns the representative of the new set); *equiv* tests whether two elements belong to the same set. In OCaml syntax, this data structure offers the following signature, where the abstract type `elem` is the type of elements:

```
type elem
val make : unit -> elem
val find : elem -> elem
val union : elem -> elem -> elem
val equiv : elem -> elem -> bool
```

One could generalize the above signature by attaching a datum of type `'a` to every set, yielding a type `'a elem`. We have not done so for the moment.

Disjoint set forests were invented by Galler and Fischer [15]. In such a forest, an element either points to another element or points to no element (i.e., is a

```
type rank = int

type elem = content ref

and content = Link of elem | Root of rank

let make () = ref (Root 0)

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

let link x y =
  if x == y then x else
  match !x, !y with
  | Root rx, 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); x
      end
  | _, _ -> assert false

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

Fig. 1. OCaml implementation of Union-Find

root). These pointers form a forest, where each tree represents a set, and where the root of the tree is the representative of the set. There is a unique path from every node in a tree to the root of this tree.

The *find* operation follows this unique path. For 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.* [1] 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* [29,28]. 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.

A complete OCaml implementation appears in Figure 1.

Union-Find is among the simplest of the classic data structures, yet requires one of the most complicated complexity analyses. Tarjan [29] and Tarjan and van Leeuwen [28] established that the worst-case time required for performing m operations involving 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. The original analysis was significantly simplified over the years [21], ultimately resulting in a 2.5 page proof that appears in a set of course notes by Tarjan [27] and in the textbook *Introduction to Algorithms* [12].

In this paper, we present a machine-checked version of this mathematical result. In addition, we establish a machine-checked connection between this result and the code shown in Figure 1. Our proofs are available online [8].

To assess the asymptotic time complexity of an OCaml program, we rely on the assumption that "it suffices to count the function calls". More precisely, if one ignores the cost of garbage collection and if one encodes loops as tail-recursive functions, then the number of function calls performed by the source program is an accurate measure of the number of machine instructions executed by the compiled program, up to a constant factor, which may depend (linearly) on the size of the program. Relying on this assumption is an old idea. For example, in the setting of a first-order functional programming language, Le Métayer [22] 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 our higher-order setting, every function call is potentially a "recursive" call, so must be counted. Danielsson [13] does this. The cost of constructing and looking up the environment of a closure is not a problem: as noted by Blelloch and Greiner [5], it is a constant, which depends on the number of variables in the program.

We do not prove that the OCaml compiler respects our assumption. It is very likely that it does. If it did not, that would be a bug, which should and very likely could be fixed. On a related theme, the CerCo project [3] has built compilers that not only produce machine code, but also determine the actual cost (according to a concrete machine model) of every basic block in the source program. This allows carrying out concrete worst-case-execution-time analysis at the source level.

In order to formally verify the correctness and complexity of a program, we rely on an extension of Separation Logic with *time credits*. Separation Logic [26] offers a natural framework for reasoning about imperative programs that manipulate the heap. A time credit is a resource that represents a *right to perform one step of computation*. In our setting, every function call consumes one credit. A number of credits can be explicitly requested as part of the precondition of a function f, and can be used to justify the function calls performed by f and by its callees. A time credit can be stored in the heap for later retrieval and consumption: that is, amortization is permitted.

In short, the combination of Separation Logic and time credits is particularly attractive because it allows (1) reasoning about correctness and complexity at the same time, (2) dealing with dynamic memory allocation and mutation, and (3) carrying out amortized complexity analyses.

Time credits, under various forms, have been used previously in several type systems [13,17,25]. Atkey [4] 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 general-purpose program verification framework.

We express the specification of every operation (*make*, *find*, *union*, etc.) in terms of an abstract predicate $\texttt{UF}\,N\,D\,R$, where N is an upper bound on the number of elements, D is the set of all elements, and R is a mapping of every element to its representative. The predicate `UF` captures: (1) the existence (and ownership) in the heap of a collection of reference cells; (2) the fact that the graph formed by these cells is a disjoint set forest; (3) the connection between this graph and the parameters N, D, and R, which the client uses in her reasoning; and (4) the ownership of a number of time credits that corresponds to the current "total potential", in Tarjan's terminology, of the data structure.

The precondition of an operation tells how many credits this operation requires. This is its amortized cost. Its actual cost may be lower or higher, as the operation is free to store credits in the heap, or retrieve credits from the heap, as long as it maintains the invariant encoded in the predicate `UF`. For instance, the precondition of *find* contains $\$(\alpha(N) + 2)$, which means "$\alpha(N) + 2$ credits". We note that it might be preferable to state that *find* requires $O(\alpha(N))$ credits.

However, we have not yet developed an infrastructure for formalizing the use of the big-O notation in our specifications.

To prove that the implementation of an operation satisfies its specification, we rely on the tool CFML [7,6], which is based on Separation Logic and on *characteristic formulae*. The characteristic formula of an OCaml term t is a higher-order logic formula $[\![t]\!]$ which describes the semantics of t. (This obviates the need for embedding the syntax of t in the logic.) More precisely, $[\![t]\!]$ denotes the set of all valid specifications of t, in the following sense: for any precondition H and postcondition Q, if the proposition $[\![t]\!]\,H\,Q$ can be proved, then the Separation Logic triple $\{H\}\ t\ \{Q\}$ holds. The characteristic formula $[\![t]\!]$ is built automatically from the term t by the CFML tool. It can then be used, in Coq, to formally establish that the term t satisfies a particular specification.

Our proofs are carried out in the Coq proof assistant. This allows us to perform in a unified framework a mathematical analysis of the Union-Find data structure and a step-by-step analysis of its OCaml implementation.

In addition to Coq, our trusted foundation includes the meta-theory and implementation of CFML. Indeed, the characteristic formulae produced by CFML are accepted as axioms in our Coq proofs. We trust these formulae because they are generated in a systematic way by a tool whose soundness has been proved on paper [7,6]. For convenience, we also rely on a number of standard logical axioms, including functional extensionality, predicate extensionality, the law of excluded middle, and Hilbert's ϵ operator. The latter allows, e.g., defining "the minimum element" of a subset of $\mathbb{N}$ before this subset has been proved nonempty (§4.4), or referring to "the parent" of a node before it has been established that this node has a parent (§4.5).

In summary, our contribution is to present:

- the first practical verification framework with support for heap allocation, mutation, and amortized complexity analysis;
- the first formalization of the potential-based analysis of Union-Find;
- the first integrated verification of correctness and time complexity for a Union-Find implementation.

The paper is organized as follows. We describe the addition of time credits to Separation Logic and to CFML (§2). We present a formal specification of Union-Find, which mentions time credits (§3). We present a mathematical analysis of the operations on disjoint set forests and of their complexity (§4). We define the predicate `UF`, which relates our mathematical view of forests with their concrete layout in memory, and we verify our implementation (§5). Finally, we discuss related work (§6) and future work (§7).

2 Time Credits, Separation, and Characteristic Formulae

2.1 Time Credits in Separation Logic

In Separation Logic, a heap predicate has type $\mathsf{Heap} \to \mathsf{Prop}$ and characterizes a portion of the heap. The fundamental heap predicates are defined as follows,

where h is a heap, H is a heap predicate, and P is a Coq proposition.

$$\begin{array}{lcl} [P] & \equiv & \lambda h.\ h = \varnothing \ \wedge\ P \\ H_1 \star H_2 & \equiv & \lambda h.\ \exists h_1 h_2.\ h_1 \perp h_2 \ \wedge\ h = h_1 \uplus h_2 \ \wedge\ H_1\, h_1 \ \wedge\ H_2\, h_2 \\ \exists x.\, H & \equiv & \lambda h.\ \exists x.\ H\, h \end{array}$$

The pure heap predicate $[P]$ characterizes an empty heap and at the same time asserts that P holds. The empty heap predicate $[\,]$ is sugar for $[\mathsf{True}]$. The separating conjunction of two heap predicates takes the form $H_1 \star H_2$ and asserts that the heap can be partitioned in two disjoint parts, of which one satisfies H_1 and the other satisfies 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. Existential quantification is also lifted to the level of heap predicates, taking the form $\exists x.\, H$.

Logical entailment between two heap predicates, written $H_1 \rhd H_2$, is defined by $\forall h.\ H_1\, h \ \Rightarrow\ H_2\, h$. This relation is used in the construction of characteristic formulae (§2.2) and also appears in specifications (§3).

To assert the existence (and ownership) of a memory cell and to describe its content, Separation Logic introduces a heap predicate of the form $l \hookrightarrow v$, which asserts that the cell at location l contains the value v. Assuming the heap is a store, i.e., a map of locations to values, the predicate $l \hookrightarrow v$ is defined as $\lambda h.\ h = (l \mapsto v)$, where $(l \mapsto v)$ denotes the singleton map of l to v. More details on these definitions are given in the first author's description of CFML [6].

To accommodate time credits, we give a new interpretation of the type Heap. In traditional Separation Logic, a heap is a store, i.e., a map from location to values: $\mathsf{Heap} \equiv \mathsf{Store}$. We reinterpret a heap h as a pair (m, c) of a store and of a natural number: $\mathsf{Heap} \equiv \mathsf{Store} \times \mathbb{N}$. The second component represents a number of time credits that are available for consumption.

This new interpretation of Heap allows us to define the heap predicate $\$\,n$, which asserts the ownership of n time credits. Furthermore, the definitions of $\uplus$, $\perp$, $\varnothing$, and $l \hookrightarrow v$, are lifted as shown below.

$$\begin{array}{rcl} \$\,n & \equiv & \lambda(m,c).\ m = \varnothing \ \wedge\ c = n \\ l \hookrightarrow v & \equiv & \lambda(m,c).\ m = (l \mapsto v) \ \wedge\ c = 0 \\ (m_1, c_1) \perp (m_2, c_2) & \equiv & m_1 \perp m_2 \\ (m_1, c_1) \uplus (m_2, c_2) & \equiv & (m_1 \uplus m_2,\ c_1 + c_2) \\ \varnothing_{:\mathsf{Heap}} & \equiv & (\varnothing_{:\mathsf{Store}}, 0) \end{array}$$

In short, we view Heap as the product of the monoids Store and $(\mathbb{N}, +)$. The definitions of the fundamental heap predicates, namely $[P]$, $H_1 \star H_2$ and $\exists x.\, H$, are unchanged.

Two provable equalities are essential when reasoning about credits:

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

The left-hand equation, combined with the fact that the logic is affine, allows weakening $\$\,n$ to $\$\,n'$ when $n \geqslant n'$ holds. Technically, this exploits CFML's "garbage collection" rule [6], and can be largely automated using tactics.

2.2 Characteristic Formulae

Let t be an OCaml term. Its characteristic formula $[\![t]\!]$ is a higher-order predicate such that, for every precondition H and postcondition Q, if $[\![t]\!]\, H\, Q$ can be proved (in Coq), then the Separation Logic triple $\{H\}\ t\ \{Q\}$ holds. This implies that, starting in any state that satisfies H, the execution of t terminates and produces a value v such that the final state satisfies the heap predicate $Q\, v$. This informal sentence assumes that an OCaml value can be reflected as a Coq value; for the sake of simplicity, we omit the details of this translation. In the following, we also omit the use of a predicate transformer, called local [6], which allows applying the structural rules of Separation Logic. Up to these simplifications, characteristic formulae for a core subset of ML are constructed as follows:

$$[\![v]\!] \equiv \lambda H Q.\ H \rhd Q\, v \tag{1}$$

$$[\![\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2]\!] \equiv \lambda H Q.\ \exists Q'.\ [\![t_1]\!]\, H\, Q' \ \wedge\ \forall x.\ [\![t_2]\!]\, (Q'\, x)\, Q \tag{2}$$

$$[\![f\, v]\!] \equiv \lambda H Q.\ \mathsf{App}\, f\, v\, H\, Q \tag{3}$$

$$[\![\mathsf{let}\, f = \lambda x.\, t_1\, \mathsf{in}\, t_2]\!] \equiv \lambda H Q.\ \forall f.\, P \ \Rightarrow\ [\![t_2]\!]\, H\, Q \tag{4}$$

$$\text{where } P \equiv \big(\forall x H' Q'.\ [\![t_1]\!]\, H'\, Q' \ \Rightarrow\ \mathsf{App}\, f\, x\, H'\, Q'\big)$$

In order to read equations (3) and (4), one must know that an OCaml function is reflected in the logic as a value of abstract type func. Such a value is opaque: nothing is known a priori about it. The abstract predicate App is used to assert that a function satisfies a certain specification. Intuitively, $\mathsf{App}\, f\, v\, H\, Q$ stands for the triple $\{H\}\ f\, v\ \{Q\}$. When a function is defined, an App assumption is introduced (4); when a function is called, an App assumption is exploited (3). In short, equation (4) 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. Equation (3) states that the only way of reasoning about a function call is to exploit such an assumption.

2.3 Combining Time Credits and Characteristic Formulae

To ensure that a time credit effectively represents "a right to perform one function call", we must enforce spending one credit at every function call. In principle, this can be achieved without any modification of the reasoning rules. All we need to do is transform the program before constructing its characteristic formula. We insert a call to an abstract function, pay, at the beginning of every function body (and loop body). This is analogous to Danielsson's "tick" [13]. We equip `pay` with a specification that causes one credit to be consumed when `pay` is invoked:

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

Here, tt denotes the unit argument and unit result of pay. The precondition $\$\, 1$ requests one time credit, while the postcondition $[\,]$ is empty. When reasoning about a call to pay, the user has no choice but to exploit the above specification and give away one time credit.

In practice, in order to reduce clutter, we simplify the characteristic formula for a sequence that begins with a call to pay. The simplified formula is as follows:

$$[\![\mathsf{pay}() \,;\, t]\!] \;\equiv\; \lambda H Q.\; \exists H'.\; H \rhd \$\,1 \star H' \;\wedge\; [\![t]\!]\, H'\, Q$$

If desired, instead of performing a program transformation followed with the generation of a characteristic formula, one can alter equation (4) above to impose the consumption of one credit at the beginning of every function body:

$$[\![\mathsf{let}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2]\!] \;\equiv\; \lambda H Q.\; \forall f.\, P \;\Rightarrow\; [\![t_2]\!]\, H\, Q$$
$$\text{where } P \equiv (\forall x H' H'' Q'.\; H' \rhd \$\,1 \star H'' \;\wedge\; [\![t_1]\!]\, H''\, Q' \;\Rightarrow\; \mathsf{App}\, f\, x\, H'\, Q')\,.$$

2.4 Meta-Theory

We revisit the informal soundness theorem for characteristic formulae [7] so as to account for time credits. The new theorem relies on a cost-annotated semantics of the programming language (a subset of OCaml). The judgment $t_{/m} \Downarrow^n v_{/m'}$ means that the term t, executed in the initial store m, terminates after n function calls and produces a value v and a final store m'. Our theorem is as follows.

Theorem 1 (Soundness of characteristic formulae with time credits).

$$\forall mc.\; \begin{cases} [\![t]\!]\, H\, Q \\ H\,(m,c) \end{cases} \;\Rightarrow\; \exists n v m' m'' c' c''.\; \begin{cases} t_{/m} \Downarrow^n v_{/m' \uplus m''} & (1) \\ m' \perp m'' & (2) \\ Q\, v\, (m', c') & (3) \\ c = n + c' + c'' & (4) \end{cases}$$

Suppose we have proved $[\![t]\!]\, H\, Q$. Pick an initial heap (m, c) that satisfies the precondition H. Here, m is an initial store, while c is an initial number of time credits. (Time credits are never created out of thin air, so one must assume that they are given at the beginning.) Then, the theorem guarantees, the program t runs without error and terminates (1). The final heap can be decomposed into (m', c') and (m'', c'') (2), where (m', c') satisfies the postcondition $Q\, v$ (3) and (m'', c'') represents resources (memory and credits) that have been abandoned during reasoning by applying the "garbage collection" rule [7]. Our accounting of time is exact: the initial number of credits c is the sum of n, the number of function calls that have been performed by the program, and $c' + c''$, the number of credits that remain in the final heap (4). In other words, every credit either is spent to justify a function call, or is still there at the end. In particular, equation (4) implies $c \geqslant n$: the number of time credits that are initially supplied is an upper bound on the number of function calls performed by the program.

The proof of Theorem 1 follows the exact same structure as that of the original soundness theorem for characteristic formulae [7]. We have carried out the extended proof on paper [8]. As expected, only minor additions are required.

3 Specification of Union-Find

Our specification of the library is expressed in Coq. It relies on an abstract type `elem` and an abstract representation predicate `UF`. Their definitions, which we present later on (§5), are not publicly known. As far as a client is concerned, `elem` is the type of elements, and `UF` $N\,D\,R$ is a heap predicate which asserts the existence (and ownership) of a Union-Find data structure, whose current state is summed up by the parameters N, D and R. The parameter D, whose type is `elem` $\rightarrow$ Prop, is the domain of the data structure, that is, the set of all elements. The parameter N is an upper bound on the cardinality of D. The parameter R, whose type is `elem` $\rightarrow$ `elem`, maps every element to its representative.

Because R maps every 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. We advertise this to the client via the first theorem in Figure 2. Recall that $\rhd$ is entailment of heap predicates. Thus, `UF_properties` states that, if one possesses `UF` $N\,D\,R$, then certain logical properties of N, D and R hold.

The next theorem, `UF_create`, asserts that out of nothing one can create an empty Union-Find data structure. `UF_create` is a "ghost" operation: it does not exist in the OCaml code. Yet, it is essential: without it, the library would be unusable, because `UF` appears in the pre-condition of every operation. When one applies this theorem, one commits to an upper bound N on the number of elements. N remains fixed thereafter.

The need for N is inherited from the proof that we follow [27,12]. Kaplan *et al.* [20] and Alstrup *et al.* [2] have carried out more precise complexity analyses, which lead to an amortized complexity bound of $\alpha(n)$, as opposed to $\alpha(N)$, where n is the cardinality of D. In the future, we would like to formalize Alstrup *et al.*'s argument, as it seems to require relatively minor adjustments to the proof that we have followed. This would remove the need for fixing N in advance and would thus make our specification easier to use for a client.

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 (recall §2.2). It states that `UnionFind_ml.make` satisfies a certain specification, thereby describing the behavior of `make`. The condition `card D` $<$ `N` indicates that new elements can be created only as long as the number of elements remains under the limit N. Then comes an application of the predicate App to the value `UnionFind_ml.make`, to the unit value `tt`, and to a pre- and postcondition. The precondition is the conjunction of `UF` $N\,D\,R$, which describes the pre-state, and of $\$\,1$, which indicates that `make` works in constant time. (We view the OCaml function `ref`, which appears in the implementation of `make`, as a primitive operation, so its use does not count as a function call.) In the postcondition, x denotes the element returned by `make`. The postcondition describes the post-state via the heap predicate `UF` $N\,(D \cup \{x\})\,R$. It also asserts that x is new, that is, distinct from all previous elements.

The next theorem provides a specification for `find`. The argument x must be a member of D. In addition to `UF` $N\,D\,R$, the precondition requires $\alpha(N) + 2$

```
(* UF : nat → (elem → Prop) → (elem → elem) → heap → Prop *)

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

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

Theorem make_spec : ∀N D R, card D < N →
  App UnionFind_ml.make tt
    (UF N D R ⋆ $1)
    (fun x ⇒ UF N (D ∪ {x}) R ⋆ [x ∉ D] ⋆ [R x = x]).

Theorem find_spec : ∀N D R x, x ∈ D →
  App UnionFind_ml.find x
    (UF N D R ⋆ $(alpha N + 2))
    (fun y ⇒ UF N D R ⋆ [R x = y]).

Theorem union_spec : ∀N D R x y, x ∈ D → y ∈ D →
  App UnionFind_ml.union x y
    (UF N D R ⋆ $(3*(alpha N)+6))
    (fun z ⇒ UF N D (fun w ⇒ If R w = R x ∨R w = R y then z else R w)
           ⋆ [z = R x ∨z = R y]).

Theorem equiv_spec : ∀N D R, x ∈ D → y ∈ D →
  App UnionFind_ml.equiv x y
    (UF N D R ⋆ $(2*(alpha N) + 5))
    (fun b ⇒ UF N D R ⋆ [b = true ↔ R x = R y]).
```

Fig. 2. Complete specification of Union-Find

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 $\texttt{UF}\,N\,D\,R$ still holds. Even though path compression may update internal pointers, the mapping of elements to representatives, which is all the client knows about, is preserved.

The precondition of `union` requires $\texttt{UF}\,N\,D\,R$ together with $3 \times \alpha(N) + 6$ time credits. The postcondition indicates that `union` returns an element z, which is either x or y, and updates the data structure to $\texttt{UF}\,N\,D\,R'$, where R' updates R by mapping to z every element that was equivalent to x or y. The construct `If P then e1 else e2`, where `P` is in Prop, is a non-constructive conditional. It is defined using the law of excluded middle and Hilbert's ϵ operator.

The postcondition of `equiv` indicates that `equiv` returns a Boolean result, which tells whether the elements x and y have a common representative.

The function `link` is internal, so its specification (given in §5) is not public.

4 Mathematical Analysis of Disjoint Set Forests

We carry out a mathematical analysis of disjoint set forests. This is a Coq formalization of textbook material [27,12]. It is independent of the OCaml code (Figure 1) and of the content of the previous sections (§2, §3). For brevity, we elide many details; we focus on the main definitions and highlight a few lemmas.

4.1 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 (§4). As in §3, 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* iff it has no successor. A node x is *represented by* a node r iff 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 is_dsf D F :=
  confined D F ∧ functional F ∧ defined (is_repr F).
```

4.2 Correctness of Path Compression

The first part of our mathematical analysis is concerned mostly 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: 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`.

4.3 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 non-root nodes can be thought of as "ghost state".

4.4 Ackermann's Function and its Inverse

For the amortized complexity analysis, we need to introduce Ackermann's function, written $A_k(x)$. According to Tarjan [27], it is defined as follows:

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

We write $f^{(i)}$ for the i-th iterate of the function f. In Coq, we write `iter i f`. The above definition is transcribed very compactly:

```
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 iterate of $\lambda f.\, \lambda x.\, f^{(x+1)}(x)$, applied to A_0.

The inverse of Ackermann's function, written $\alpha(n)$, maps n to the smallest value of k such that $A_k(1) \geqslant n$. Below, `min_of le` denotes the minimum element of a nonempty subset of $\mathbb{N}$.

```
Definition alpha n := min_of le (fun k ⇒ A k 1 ⩾ n).
```

4.5 Potential

The definition of the potential function relies on a few auxiliary definitions. First, for every node x, Tarjan [27] writes $p(x)$ 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.

Then, Tarjan [27] introduces the *level* and the *index* of a non-root node x. These definitions involve the rank of x and the rank of its parent. The level of x, written $k(x)$, is the largest k for which $K(p(x)) \geqslant A_k(K(x))$ holds. It lies in the interval $[0, \alpha(N))$, if N is an upper bound on the number of nodes. The index of x, written $i(x)$, is the largest i for which $K(p(x)) \geqslant A^{(i)}_{k(x)}(K(x))$ holds. It lies in the interval $[1, K(x)]$. The formal definitions, shown below, rely on the function `max_of`, which is the dual of `min_of` (§4.4).

```
Definition k F K x :=
  max_of le (fun k ⇒ K (p F x) ⩾ A k (K x)).
Definition i F K x :=
  max_of le (fun i ⇒ K (p F x) ⩾ iter i (A (k F K x)) (K x)).
```

The potential of a node, written $\phi(x)$, depends on the parameter N, which is a pre-agreed upper bound on the number of nodes. (See the discussion of `UF_create` in §3.) Following Tarjan [27], if x is a root or has rank 0, then $\phi(x)$ is $\alpha(N) \cdot K(x)$. Otherwise, $\phi(x)$ is $(\alpha(N) - k(x)) \cdot K(x) - i(x)$.

```
Definition phi F K N x :=
  If (is_root F x) ∨(K x = 0)
    then (alpha N) * (K x)
    else (alpha N - k F K x) * (K x) - (i F K x).
```

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

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

4.6 Rank Analysis

The definition of the representation predicate `UF`, which we present later on (§5), 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 (§3), 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:

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

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 (i.e., updating one graph edge, as in §4.2), but "iterated path compression", i.e., 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 `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'.

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

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` (§5). On the other hand, by following Tarjan's proof [27], 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 N,
 is_rdsf D F K → x ∈ D → card D ⩽ N →
 ∃d F', ipc F x d F' ∧ (Phi D F K N + alpha N + 2 ⩾ Phi D F' K N + d + 1).
```

This lemma guarantees 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 "Φ + advertised cost $\geqslant \Phi'$ + actual cost" holds. Indeed, $\alpha(N) + 2$ is the advertised cost of `find` (Figure 2), whereas $d + 1$ is its actual cost, because iterated path compression along a path of length d involves $d + 1$ calls to `find`.

5 Verification of the Code

To prove that the OCaml code in Figure 1 satisfies the specification in Figure 2, we first define the predicate `UF`, then establish each of the theorems in Figure 2.

5.1 Definition of the representation predicate

In our OCaml code (Figure 1), we have defined `elem` as `content ref`, and defined `content` as `Link of elem | Root of rank`. CFML automatically mirrors these type definitions in Coq. It defines `elem` as `loc` (an abstract type of memory locations, provided by CFML's library) and `content` as an inductive data type:

```
Definition elem := loc.
Inductive content := Link of elem | Root of rank.
```

To define the heap predicate `UF` $N\,D\,R$, we need two auxiliary predicates.

The auxiliary predicate `Inv N D R F K` includes the invariant `is_rdsf D F K`, which describes a ranked disjoint set forest (§4.3), adds the condition `card D ⩽ N` (which appears in the complexity analysis, §4.6), and adds the requirement that the function R and the relation `is_repr` F agree, in the sense that $R\,x = r$ implies `is_repr` $F\,x\,r$. The last conjunct is needed because our specification exposes R, which describes only the mapping from nodes to representatives, whereas our mathematical analysis is in terms of F, which describes the whole graph.

```
Definition Inv N D F K R :=
  (is_rdsf D F K) ∧ (card D ⩽ N) ∧ (rel_of_func R ⊂ is_repr F).
```

```
Theorem link_spec : ∀D R x y, x ∈ D → y ∈ D → R x = x → R y = y →
  App UnionFind_ml.link x y
    (UF D R ⋆ $(alpha N+1))
    (fun z ⇒ UF D (fun w ⇒ If R w = R x ∨R w = R y then z else R w) ⋆ [z = x ∨z = y]).
Proof.
  xcf. introv Dx Dy Rx Ry. credits_split. xpay. xapps. xif.
  { xret. rewrite- root_after_union_same. hsimpl-. }
  { unfold UF at 1. xextract as F K M HI HM. lets HMD: (Mem_dom HM). xapps-. xapps-. xmatch.
    { forwards* (HRx&Kx): Mem_root_inv x HM.
      forwards* (HRy&Ky): Mem_root_inv y HM. xif.
      { forwards* (F'&EF'&HI'&n&EQ): (>> Inv_link_lt D R F K x y). splits*. math.
        xchange (credits_eq EQ). chsimpl. xapp*. xret. unfold UF. chsimpl*. applys* Mem_link. }
      { xif.
        { forwards* (F'&EF'&HI'&n&EQ): (>> Inv_link_gt D R F K x y). splits*. math.
          xchange (credits_eq EQ). chsimpl. xapp*. xret. unfold UF. chsimpl*. applys* Mem_link. }
        { asserts: (rx = ry). math. asserts: (K x = K y). math.
          forwards* (F'&K'&EF'&EK'&HI'&n&EQ): (>> Inv_link_eq D R F K x y). splits*.
          xchange (credits_eq EQ). chsimpl. xapp-. xapp-. apply- map_indom_update_already.
          xret. rewrite H2 in EK'. unfold UF. chsimpl*. applys* Mem_link_incr HM. } } }
    { xfail ((rm C0) (K x) (K y)). fequals; applys* Mem_root. }
Qed.
```

Fig. 3. Verification script for the link function

The auxiliary predicate `Mem D F K M` relates a model of a disjoint set forest, represented by D, F, K, and a model of the memory, represented by a finite map M of memory locations to values of type `content`. (This view of memory is imposed by CFML.) M maps a location x to either `Link` y for some location y or `Root` k for some natural number k. The predicate `Mem D F K 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 implies that x is a root and has rank k.

```
Definition Mem D F K M := (dom M = D) ∧ (∀ x, x ∈ D →
  match M\(x) with Link y ⇒ F x y | Root k ⇒ is_root F x ∧ k = K x end).
```

At last, we are ready to define the heap predicate `UF` N D R:

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

The first conjunct asserts the existence in the heap of a group of reference cells, collectively described by the map M. (The predicates `Group`, `Ref` and `Id` are provided by CFML's library.) 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). Recall that the brackets lift an ordinary proposition as a heap predicate (§2). 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.

5.2 Verification through characteristic formulae

Our workflow is as follows. Out of the file `UnionFind.ml`, which contains the OCaml source code, the CFML tool produces the Coq file `UnionFind_ml.v`, which

contains characteristic formulae. Consider, for example, the OCaml function `link`. The tool produces two Coq axioms for it. The first axiom asserts the existence of a value `link` of type `func` (§2.2). The second axiom, `link_cf`, is a characteristic formula for `link`. It can be exploited to establish that `link` satisfies a particular specification. For instance, in Figure 3, we state and prove a specification for `link`. The proof involves a mix of standard Coq tactics and tactics provided by CFML for manipulating characteristic formulae. It is quite short, because 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 CFML library is about 5Kloc. The analysis of Union-Find (§4) is 3Kloc. The specification of Union-Find (Figure 2) and the verification of the code (§5), together, take up only 0.4Kloc. 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 [8].

6 Related Work

Disjoint set forests as well as linking-by-size are due to Galler and Fischer [15]. Path compression is attributed by Aho *et al.* [1] to McIlroy and Morris. Hopcroft and Ullman [19] study linking-by-size and path compression and establish an amortized bound of $O(log^* N)$ per operation. Tarjan [29] studies linking-by-rank and path compression and establishes the first amortized bound in $O(\alpha(N))$. After several simplifications [28,21], this leads to the proof that we follow [27,12]. Kaplan *et al.* [20] and Alstrup *et al.* [2] establish a "local" bound: they bound the amortized cost of $find(x)$ by $O(\alpha(n))$, where n is the size of x's set.

We know of only one machine-checked proof of the functional correctness of Union-Find, due to Conchon and Filliâtre [11]. They reformulate the imperative algorithm in a purely functional style, where the store is explicit. They represent the store as a persistent array and obtain an efficient persistent Union-Find. We note that Union-Find is part of the VACID-0 suite of benchmark verification problems [23]. We did not find any solution to this particular benchmark problem online. We know of no machine-checked proof of the complexity of Union-Find.

The idea of using a machine to assist in the complexity analysis of a program goes back at least as far back as Wegbreit [30]. He extracts recurrence equations from the program and bounds their solution. More recent work along these lines includes Le Métayer's [22] and Danner *et al.*'s [14]. 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.

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. Such an approach has been explored in several projects, such as Ynot [10], Bedrock [9], and CFML [7,6].

This approach allows verifying not only the implementation of a data structure but also its clients. In particular, when a data structure comes with an amortized complexity bound, we verify that it is used in a single-threaded manner.

Nipkow [24] carries out 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.

The idea of extending a type system or program logic with time or space credits, viewed as affine resources, has been put forth by several authors [18,4,25]. The extension is very modest; in fact, if the source program is explicitly instrumented by inserting calls to pay, no extension at all 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.

A line of work by Hofmann *et al.* [18,17,16] 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).

7 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 (and at the same time) that it meets a certain complexity bound.

There are many directions for future work. Concerning Union-Find, we would like to formalize Alstrup *et al.*'s proof [2] that the amortized cost can be expressed in terms of the current number n of nodes, as opposed to a fixed upper bound N. Concerning our verification methodology, we wish to use the big-O notation in our specifications, so as to make them more modular.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Alstrup, S., Thorup, M., Gørtz, I.L., Rauhe, T., Zwick, U.: Union-find with constant time deletions. ACM Transactions on Algorithms 11(1) (2014)
3. Amadio, R.M., Ayache, N., Bobot, F., Boender, J., Campbell, B., Garnier, I., Madet, A., McKinna, J., Mulligan, D.P., Piccolo, M., Pollack, R., Régis-Gianas, Y., Coen, C.S., Stark, I., Tranquilli, P.: Certified complexity (CerCo). In: Foundational and Practical Aspects of Resource Analysis. Lecture Notes in Computer Science, vol. 8552. Springer (2014)
4. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science 7(2:17) (2011)
5. Blelloch, G.E., Greiner, J.: Parallelism in sequential functional languages. In: Functional Programming Languages and Computer Architecture (FPCA) (1995)

6. Charguéraud, A.: Characteristic formulae for the verification of imperative programs (2012), to appear in HOSC
7. Charguéraud, A.: Characteristic Formulae for Mechanized Program Verification. Ph.D. thesis, Université Paris 7 (2010)
8. Charguéraud, A., Pottier, F.: Self-contained archive. `http://gallium.inria.fr/~fpottier/dev/uf/` (2015)
9. Chlipala, A.: The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In: International Conference on Functional Programming (ICFP) (2013)
10. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: International Conference on Functional Programming (ICFP) (2009)
11. Conchon, S., Filliâtre, J.: A persistent union-find data structure. In: ACM Workshop on ML (2007)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (Third Edition). MIT Press (2009)
13. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Principles of Programming Languages (POPL) (2008)
14. Danner, N., Paykin, J., Royer, J.S.: A static cost analysis for a higher-order language. In: Programming Languages Meets Program Verification (PLPV) (2013)
15. Galler, B.A., Fischer, M.J.: An improved equivalence algorithm. Communications of the ACM 7(5) (1964)
16. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Transactions on Programming Languages and Systems 34(3) (2012)
17. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 6012. Springer (2010)
18. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Principles of Programming Languages (POPL) (2003)
19. Hopcroft, J.E., Ullman, J.D.: Set merging algorithms. SIAM Journal on Computing 2(4) (1973)
20. Kaplan, H., Shafrir, N., Tarjan, R.E.: Union-find with deletions. In: Symposium on Discrete Algorithms (SODA) (2002)
21. Kozen, D.C.: The design and analysis of algorithms. Texts and Monographs in Computer Science, Springer (1992)
22. Le Métayer, D.: ACE: an automatic complexity evaluator. ACM Transactions on Programming Languages and Systems 10(2) (1988)
23. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0 (2010), manuscript KRML 209
24. Nipkow, T.: Amortized complexity verified. In: Interactive Theorem Proving (2015)
25. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Types in Language Design and Implementation (TLDI) (2011)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS) (2002)
27. Tarjan, R.E.: Class notes: Disjoint set union (1999)
28. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. Journal of the ACM 31(2) (1984)
29. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM 22(2) (1975)
30. Wegbreit, B.: Mechanical program analysis. Communications of the ACM 18(9) (1975)