A Persistent Union-Find Data Structure

Sylvain Conchon Jean-Christophe Filliâtre

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Parc Orsay Université, F-91893
{conchon,filliatr}@lri.fr

Abstract

The problem of disjoint sets, also known as *union-find*, consists in maintaining a partition of a finite set within a data structure. This structure provides two operations: a function *find* returning the class of an element and a function *union* merging two classes. An optimal and imperative solution is known since 1975. However, the imperative nature of this data structure may be a drawback when it is used in a backtracking algorithm. This paper details the implementation of a persistent union-find data structure as efficient as its imperative counterpart. To achieve this result, our solution makes heavy use of imperative features and thus it is a significant example of a data structure whose side effects are safely hidden behind a persistent interface. To strengthen this last claim, we also detail a formalization using the Coq proof assistant which shows both the correctness of our solution and its observational persistence.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures; D.2.4 [*Software/Program Verification*]: Correctness proofs

General Terms Algorithms, Verification

Keywords Union-Find, Persistence, Formal Verification

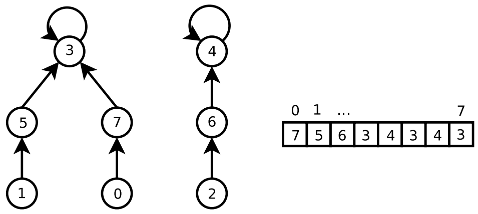
1. Introduction

The problem of disjoint sets, known as *union-find*, consists in maintaining a partition of a finite set within a data structure. Without loss of generality, we can assume that we are considering a partition of the n integers $\{0, 1, \dots, n-1\}$. A typical signature[1] for such a data structure could be the following:

```
module type ImperativeUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → unit
end
```

It provides an abstract data type `t` for the partition and three operations. The `create` operation takes an integer n as input and builds a new partition where each element in $\{0, \dots, n-1\}$ constitutes a class by itself. The `find` operation returns the class of an element, as an integer considered to be the representative of this class. Finally the `union` operation merges two classes of the partition, the data structure being modified in place (hence the return type `unit`).

An optimal solution is known since the 70s. It is attributed to McIlroy and Morris [3] and its complexity was analyzed by Tarjan [17][2]. It is a classic of the algorithmic literature (see for instance [11] chapter 22). The code of this solution is given in appendix. The main idea is to link together the elements of each class. For this purpose, the array `parent` maps each integer i to another integer of the same class. Within each class, these links form an acyclic graph where any path leads to the representative, which is the sole element mapped to itself. The following figure illustrates a situation where the set $\{0, 1, \dots, 7\}$ is partitioned into two classes whose representatives are 3 and 4 respectively.



The `find` operation simply follows the links until it finds the representative. The `union` operation first finds out the two representatives of the given elements and then links one to the other. Then two key improvements are used to reach optimal efficiency. The first one is called *path compression*: any element encountered during the search performed by `find` is re-mapped to the representative. The second one consists in keeping an approximation of the size of each class, called a *rank*, and to choose for the representative of a union the one with the largest rank.

The union-find data structure is used in many different contexts. In most cases, the imperative implementation above is perfectly suited — and even optimal. But as soon as we need to roll back to previous versions of the union-find structure, the imperative nature of this solution is a nuisance. Indeed, providing either an efficient "copy" or "undo" operation turns out to be very difficult, mostly because of path compression. Here are typical examples where union-find is used together with backtracking:

- *Unification*: A typical use of a union-find data structure is unification. For instance, unification is massively used in Prolog where backtracking is precisely a fundamental concept. Unification is also used in ML type inference. In that case, this is not a backtracking algorithm but, within an interactive toplevel,

[1] This article is illustrated with OBJECTIVE CAML [2] code but is obviously translatable to any other programming language.

[2] In the following, we will abusively refer to this solution as Tarjan's algorithm.

ACM SIGPLAN Workshop on ML October 5, 2007, Freiburg, Germany.

destructive unifications of non-generalized type variables must be undone when typing errors are encountered.

- *Decision procedures*: Another example of a union-find data structure used within a backtracking algorithm are decision procedures where the treatment of equality is usually based on such a structure and where the boolean part of formulas is mostly processed using backtracking [13, 16].

A standard solution to the issue of backtracking in data structures is to turn to *persistent* data structures (as opposed to *ephemeral* data structures) [12]. In this case, the `union` operation now returns a new partition and leaves the previous one unchanged. The signature of such a persistent data structure could be the following:

```
module type PersistentUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → t
end
```

It simply differs from signature `ImperativeUnionFind` in the return type of `union`, which is now `t` instead of `unit`. One easy way to implement such a signature is to use purely applicative data structures. An immediate — and probably widely used — solution is to switch from an imperative array to a purely applicative map. Assuming a module `M` implementing maps from integers to integers, as follows

```
module M : sig
  type t
  val empty : t
  val add : int → int → t → t
  val find : int → t → int
end
```

we can directly implement the persistent union-find data structure as a map, that is

```
type t = M.t
```

Then `find` and `union` operations are easily implemented:

```
let rec find m i =
  try find (M.find i m) with Not_found → i

let union m i j =
  let ri = find m i in
  let rj = find m j in
  if ri <> rj then M.add ri rj m else m
```

Unfortunately, path compression cannot be implemented without changing the type of `find`: one would need `find` to return a new partition but this would require the programmer to modify client code. For this reason, such solutions are far less efficient than the imperative implementation.

The main contribution of this paper is the design of a persistent union-find data structure as efficient as the imperative implementation. To achieve this result, we consider a data structure with side effects yet with signature `PersistentUnionFind`, so that client code may use it as if it was a purely applicative data structure. This is thus an example of an imperative data structure whose side effects are *safely hidden* behind a persistent interface. Another contribution of this paper is to strengthen this last claim with a formal proof. Indeed, we used the Coq proof assistant [1, 6] to show both the correctness of our solution and its observational persistence.

There are known techniques to design persistent data structures based on purely applicative programming [14] but there is no efficient solution to the union-find problem in this framework. There are also techniques to make imperative data structures persistent [12] but, again, they were not applied to the union-find data structure. Moreover, no formal proof of the observational persistence of an imperative data structure has ever been done to our knowledge.

This paper is organized as follows. Section 2 details a persistent union-find data structure which is almost as efficient as its imperative counterpart. Then Section 3 analyzes the efficiency of this solution, comparing it to several other implementations. Finally, Section 4 details the formal proof.

2. A Simple Yet Efficient Solution

Our solution to the persistent union-find problem is simultaneously very simple and very efficient. It mostly consists in keeping close to the original algorithm and its path compression, but substituting *persistent arrays* in place of the usual arrays.

2.1 Persistent Arrays

A persistent array is a data structure providing the same operations as a usual array, namely the manipulation of elements indexed from 0 to $n - 1$, with low cost access and update operations, but where the update operation returns a new persistent array and leaves the previous one unchanged. A minimal signature for polymorphic persistent arrays is:

```
module type PersistentArray = sig
  type α t
  val init : int → (int → α) → α t
  val get : α t → int → α
  val set : α t → int → α → α t
end
```

It is easy to implement this signature using purely applicative dictionaries, such as balanced binary search trees from OCAML's standard library. But as demonstrated later by our benchmarks, the logarithmic cost of the access and update operations on such structures is already prohibitive. Fortunately, it is possible to achieve far better efficiency for persistent arrays, as we will show later (Section 2.3). Meanwhile, we are presenting a persistent version of Tarjan's algorithm, independently of the implementation of persistent arrays.

2.2 A Persistent Version of Tarjan's Union-Find Algorithm

To be independent of persistent arrays, we naturally introduce a parameterized module, a so-called *functor*:

```
module Make(A : PersistentArray)
  : PersistentUnionFind
= struct
```

As in the imperative version, the union-find structure is a pair of arrays (`rank` containing the ranks of the representatives and `parent` containing the links) but here we use persistent arrays:

```
type t = {
  rank: int A.t;
  mutable parent: int A.t
}
```

The mutable nature of the second field will be exploited to perform path compression. The creation of a new union-find structure is immediate:

```
let create n = {
  rank = A.init n (fun _ → 0);
  parent = A.init n (fun i → i)
}
```

To implement path compression, the `find` function must perform modifications on the `parent` array on its way back, once the representative has been found. Here this array is persistent and thus a new array must be built. For this purpose, we introduce an auxiliary function `find_aux` returning both the representative and the new version of the `parent` array:

```
let rec find_aux f i =
  let fi = A.get f i in
  if fi == i then
    f, i
  else
    let f, r = find_aux f fi in
    let f = A.set f i r in
    f, r
```

Then we can define a function `find` which calls `find_aux` and then modifies the `parent` field with a side effect, to allow path compression in later accesses of this data structure:

```
let find h x =
  let f,cx = find_aux h.parent x in
  h.parent ← f;
  cx
```

As we can notice, this function indeed has the expected type, that is it returns a single integer, while performing path compression. As in the imperative code, the data structure has been mutated by a side effect but the set of representatives remains unchanged.

To implement the `union` function, we still follow the imperative code but we return a new data structure, namely a new record of type `t`. It contains the new versions of the `rank` and `parent` arrays:

```
let union h x y =
  let cx = find h x in
  let cy = find h y in
  if cx != cy then begin
    let rx = A.get h.rank cx in
    let ry = A.get h.rank cy in
    if rx > ry then
      { h with parent = A.set h.parent cy cx }
    else if rx < ry then
      { h with parent = A.set h.parent cx cy }
    else
      { rank = A.set h.rank cx (rx + 1);
        parent = A.set h.parent cy cx }
  end else
    h
```

We finally get a code which is not longer than its imperative counterpart. The most difficult task remains: to feed this functor with an efficient implementation of persistent arrays.

2.3 Efficient Implementation of Persistent Arrays

An efficient solution to the problem of persistent arrays has actually been known for a long time. It seems that it is due to H. Baker [4] who was using it to implement closures in a Lisp runtime.

2.3.1 Main Idea

The base idea is to use a usual array[3] for the last version of the persistent array and indirections for previous versions. For this purpose, we introduce the following mutually recursive data types:

```
type α t = α data ref
and α data =
  | Arr of α array
  | Diff of int × α × α t
```

The type `α t` is the type of persistent arrays. It is a reference on a value of type `α data` which indicates its nature: either an immediate value `Arr` a with an array a, or an indirection `Diff`(i, v, t) standing for a persistent array which is identical to the persistent array t everywhere except at index i where it contains v. The reference may seem superfluous but it is actually crucial. Creating a new persistent array is immediate:

```
let init n f = ref (Arr (Array.init n f))
```

The access function `get` is also straightforward. Either the persistent array is an immediate array, or we need to consider the indirection and possibly to recursively access another persistent array:

```
let rec get t i = match !t with
  | Arr a →
      a.(i)
  | Diff (j, v, t') →
      if i == j then v else get t' i
```

All the subtlety is concentrated into the `set` function. The idea is to keep the efficiency of a usual array on the very last version of the persistent array, while possibly decreasing the efficiency of previous versions. When updating the persistent array `t`, there are two possible cases:

- either `t` is a reference to an object of shape `Arr` a; in that case, we are going to replace `t` with an indirection (which is possible since it is a reference and not a value of type `α data`), modify the array a in place and return a new reference pointing to `Arr` a.
- or `t` is already an indirection, that is pointing to a `Diff` node; then we simply create and return a new indirection.

This is done in the following code:

```
let set t i v = match !t with
  | Arr a as n →
      let old = a.(i) in
      a.(i) ← v;
      let res = ref n in
      t := Diff (i, old, res);
      res
  | Diff _ →
      ref (Diff (i, v, t))
```

As we can notice, a value of the shape `Arr` a can only be created by the `init` function. Thus a sequence of updates only allocates a single array and an additional space which is proportional to the number of updates (since `set` clearly runs in $O(1)$ space and time). If we consider the following definition of four persistent arrays `a0`, `a1`, `a2` and `a3`

```
let a0 = create 7 0
let a1 = set a0 1 7
let a2 = set a1 2 8
let a3 = set a1 2 9
```

then the situation right after these declarations is illustrated in Figure 1, where each reference is displayed as a circle and a `Diff` block by a labelled edge. Generally speaking, we have the following invariant: the graph of references of type `α t` for the various versions of a persistent array is acyclic and from any of these references there is a unique path to the `Arr` node.

Such persistent arrays achieve good results when we always access the last version but efficiency greatly decreases when we access previous versions. Indeed, a sequence of several updates creates a linked list of `Diff` nodes and then any access from this list has a cost which is proportional to its length, that is to the

[3] From now on, we use the term "array" for a usual array, thus modified in place, and the term "persistent array" otherwise.

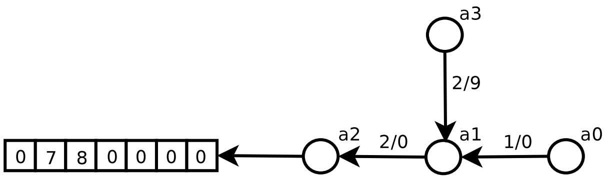


Figure 1. Illustrating persistent arrays (1/2)

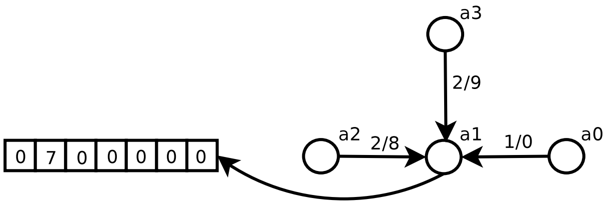


Figure 2. Illustrating persistent arrays (2/2)

number of updates. This can be dramatic in a context where we need to backtrack. But this is precisely the reason why we were building persistent arrays. Fortunately, there exists a very simple way to improve this first implementation.

2.3.2 A Major Improvement

To lower the cost of operations on previous versions of persistent arrays, H. Baker introduces a very simple improvement [5]: as soon as we try to access a persistent array which is not an immediate array we first reverse the linked list leading to the `Arr` node, to move it in front of the list, that is precisely where we want to access. This operation, that Baker calls *rerooting*, can be coded by the following `reroot` function which takes a persistent array as argument and returns nothing; it simply modifies the structure of pointers, without modifying the contents of the persistent arrays.

```
let rec reroot t = match !t with
  | Arr _ → ()
  | Diff (i, v, t') →
      reroot t';
      begin match !t' with
        | Arr a as n →
            let v' = a.(i) in
            a.(i) ← v;
            t := n;
            t' := Diff (i, v', t)
        | Diff _ → assert false
      end
```

After calling this function, we have the property that `t` now points to a value of the shape `Arr`. Thus we can modify the access function so that it now calls the `reroot` function as soon as the persistent array is a `Diff` node:

```
let rec get t i = match !t with
  | Arr a →
      a.(i)
  | Diff _ →
      reroot t;
      begin match !t with
        | Arr a → a.(i)
        | Diff _ → assert false
      end
```

We can modify the `set` function in a similar way:

```
let set t i v =
  reroot t;
  match !t with
  | Arr a as n → ... as previously ...
  | Diff _ → assert false
```

Going back to the situation of Figure 1, let us assume that we now try to access `a1`. Then `reroot` is going to be called on `a1`. This results in `a1` now pointing to the `Arr` node and `a2` pointing to an indirection, the values 0 and 8 at index 2 being swapped between the array and the indirection. This new situation is illustrated in Figure 2.

The `reroot` operation has a cost proportional to the number of `Diff` nodes that must be followed to reach the `Arr` node, but it is only performed the first time we access to an old version of a persistent array. Any subsequent access will be performed in constant time. To put it otherwise, we pay only once the cost of coming back to a previous version. In a backtracking context, this is a perfect solution. If the number of array operations is far greater than the number of backtracks, then the amortized complexity of the `get` and `set` operations will be $O(1)$ in space and time.

It is important to notice, however, that if we manipulate simultaneously several versions of a single persistent array then efficiency will decrease since the `reroot` function is going to spend much time in list reversals.

Finally, we can also notice that the `reroot` function is not tail-recursive. This can be an issue when we manipulate persistent arrays on which many updates have been performed. It is however easy to solve this issue by rewriting the code into continuation passing style (CPS), without any real loss of efficiency. The tests which are presented later are performed using such a CPS version.

2.3.3 Final Improvements

It is possible to further improve this implementation. The first idea is to optimize a call to `set` $t\ i\ v$ when i is already mapped to v in t. Then we save a useless indirection, and thus the allocation of two blocks. This is especially efficient in the context of the union-find data structure, since path compression quickly maps all the elements to the representative and then subsequent `set` operations become useless. (We could equivalently unroll the `find_aux` function and make a special treatment for paths of length 1.)

The second idea is to notice that in a context where we *only perform backtracking*, it is useless to maintain the contents of persistent arrays that become unreachable when we go back to a previous point. Indeed, these values are going to be immediately reclaimed by the garbage collector. Thus we can still improve the efficiency of our persistent arrays, in the particular case where we are using them, that is where we go back to a previous version t of a persistent array without keeping any pointer to younger versions of t.

The first modification is to introduce an `Invalid` node denoting a persistent array where it is no more possible to access:

```
type α t = α data ref
and α data =
  | Arr of int array
  | Diff of int × α × α t
  | Invalid
```

Then we modify the `reroot` function so that it does not reverse the list of pointers but simply updates the contents of the array:

```
let rec reroot t = match !t with
  | Arr _ → ()
  | Diff (i, v, t') →
```

```
      reroot t';
      begin match !t' with
        | Arr a as n →
            a.(i) ← v;
            t := n;
            t' := Invalid
        | Diff _ | Invalid → assert false
      end
  | Invalid → assert false
```

As we can notice, we save the allocation of a Diff node. The remaining of the code is unchanged but is adapted to fail if we try to access to an Invalid node.

It is striking to notice that the final data structure we get is actually nothing more than a usual array together with an undo stack, that is the backtracking design pattern of the imperative programmer. But contrary to an imperative programming style where the stack is made explicit inside the main algorithm, it is here hidden behind an abstract data type which creates the *illusion* of persistence.

3. Performance

We tried to test the efficiency of our solution in a situation as realistic as possible. For this purpose we looked at the use made by the Ergo decision procedure [8] of its internal union-find data structure. We distinguish three parameters:

- the number of backtracks;
- the total number of `find` and `union` operations between two branchings, denoted N;
- the proportion of `union` operations with respect to `find` operations, denoted p.

On the tests we made with the decision procedure, it happens that the number of backtracks is small and that the proportion of `union` operations is also quite small. Thus we chose to make some tests following the branchings of a full binary search of height 4. Between each node and its two sons we perform exactly N operations. This traversal is executed for $N = 20000$, $N = 100000$ and $N = 500000$, with proportion p of `union` being equal to 5, 10 and 15%.

Figure 3 displays the timings[4] for various implementations. The first line corresponds to the imperative implementation, as given in appendix. It is incorrectly used here — we keep the current version when backtracking —- but it is shown as a reference. The next four lines are the successive refinements of our solution: the first version of Section 2.3.2, the two improvements of Section 2.3.3 and finally a manually defunctorized version of the final solution (defun.). The last two lines are purely applicative solutions, in a purpose of comparison: the first (naïve) is an AVL-based implementation without path compression nor ranks, as described in the introduction, and the second is the application of the functor from Section 2.2 to persistent arrays implemented as AVLs.

The results show that our final solution (defun.) is almost as efficient as the imperative implementation. Note that the incorrect use of the imperative code makes path compression on one branch to be effective on all successive branches, resulting in maximal path compression eventually. Results for the naïve implementation shows that it does not scale at all. The last line emphasizes that path compression and ranking alone are not sufficient to achieve good results, but that an efficient implementation of persistent arrays is mandatory.

[4] Timings are CPU time measured in seconds on a Pentium IV 2.4 GHz running Linux.

4. A Formal Proof of Correctness

Even if our solution is conceptually simple, the data structures are somewhat complex due to the massive use of (hidden) side effects, in both persistent arrays and the persistent implementation of Tarjan's algorithm. For this reason, we decided to formally verify the correctness of these two data structures and to show their observational persistence. This section gives an overview of this formalization, conducted in the Coq proof assistant [1]. The whole development can be found online[5].

Section 4.1 briefly presents program verification using Coq, introducing the notations used in the remainder. Section 4.2 describes our Coq modeling of ML references. Then Section 4.3 presents the verification of persistent arrays and Section 4.4 that of the persistent union-find data structure.

4.1 Program Verification using Coq

Coq is an interactive proof assistant based on the Calculus of Inductive Constructions, a higher-order logic with polymorphism, dependent types and a primitive notion of inductive types [10, 15]. In particular, this logic contains the purely applicative fragment of ML and thus can be used to to define ML programs and to show their correctness. Here is a short example on Peano natural numbers. First, we define the datatype `nat` with two constructors `O` and `S` as we would do in ML:

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat
```

`nat` has type `Set` which is the sort of datatypes. Predicates may also be defined inductively. Here is the definition of a unary predicate `even` on natural numbers:

```
Inductive even : nat → Prop :=
  | evenO : even O
  | evenSS : ∀n:nat, even n → even (S (S n)).
```

`even` has type `nat → Prop` where `Prop` is the sort of propositions. Such an inductive predicate is equivalent to the following inference rules:

$$\frac{}{\texttt{even O}}(\texttt{evenO}) \qquad \frac{\texttt{even}\ n}{\texttt{even (S (S}\ n\texttt{))}}(\texttt{evenSS})$$

ML-like functions are defined in a straightforward way:

```
Definition plus3 : nat → nat :=
  fun n:nat ⇒ S (S (S n)).
```

Then it is possible to state properties about functions, such as "for any even natural number n, the result of $(\texttt{plus3}\ n)$ is not even":

```
Lemma even_plus3 :
  ∀n:nat, even n → ~(even (plus3 n)).
```

Such a lemma declaration must be followed by a proof, which is a list of tactic invocations. Proofs are omitted in this paper.

The richness of the Coq type system actually allows the user to combine a function definition together with its correctness proof. To do so, the function is given a type which contains both types for its arguments and result and a specification. Here is such a *definition-as-proof* of a function f taking an even number n as argument and returning a result m greater than n and even:

```
Definition f :
  ∀n:nat, even n → { m:nat | m > n ∧ even m }.
```

The type $\{\ x : T \mid P(x)\ \}$ is a dependent pair of a value x of type T and a proof of $P(x)$. The definition body for f is omitted and

[5] http://www.lri.fr/~filliatr/puf/

p / N	5% 20000	5% 100000	5% 500000	10% 20000	10% 100000	10% 500000	15% 20000	15% 100000	15% 500000
Tarjan	0.31	2.23	12.50	0.33	2.34	12.90	0.34	2.36	13.20
2.3.2	0.52	3.03	17.10	0.81	4.78	26.80	1.16	6.78	37.90
2.3.3a	0.36	2.08	12.30	0.46	2.76	15.90	0.64	3.58	20.70
2.3.3b	0.34	2.01	11.70	0.42	2.54	14.90	0.52	3.21	18.70
defun.	0.33	1.90	11.30	0.41	2.45	14.40	0.52	3.14	17.80
naïve	0.76	5.28	37.50	1.22	9.14	63.80	40.40	>10mn	>10mn
maps	1.52	10.60	67.90	2.45	17.50	116.00	3.42	24.70	167.00

Figure 3. Performances

replaced by a proof script, which is both the function definition and its correctness proof. Once the proof is completed, a mechanism allows the user to extract an ML program from the definition-as-proof.

Coq is naturally suited for the verification of purely applicative programs. To deal with imperative programs, we need to model the memory and to interpret imperative features as memory transformers. This is similar to the formalization of operational semantics.

4.2 Modelling ML References

To model references, we introduce an abstract type `pointer` for the values of references:

```
Parameter pointer : Set.
```

Then the set of all (possibly aliased) references of a given type is modelled as a dictionary mapping each reference to the value it is pointing at. This dictionary is simply axiomatized as a module called `PM` declaring a polymorphic abstract type `t`:

```
Module PM.
  Parameter t : Set → Set.
```

Thus `PM.`t a is the type of a dictionary for references of type a. Three operations are provided on this type:

```
  Parameter find : ∀a, t a → pointer → option a.
  Parameter add : ∀a, t a → pointer → a → t a.
  Parameter new : ∀a, t a → pointer.
```

`find` returns the value possibly associated to a pointer (that is `None` if there is no associated value and `Some` v otherwise); `add` adds a new mapping; and finally `new` returns a fresh reference (that is a reference which is not yet mapped to any value). Three axioms describe the behavior of these three operations:

```
  Axiom find_add_eq :
    ∀a, ∀m:t a, ∀p:pointer, ∀v:a,
    find (add m p v) p = Some v.
  Axiom find_add_neq :
    ∀a, ∀m:t a, ∀p p':pointer, ∀v:a,
    ~p'=p → find (add m p v) p' = find m p'.
  Axiom find_new :
    ∀a, ∀m:t a, find m (new m) = None.
End PM.
```

This axiomatization is obviously consistent, since we could realize it using natural numbers for pointers and a finite map (*e.g.* an association list) for the dictionary.

Then the heap can be viewed as a set of such dictionaries, since ML typing prevents aliasing between references of different types (we are not considering polymorphic references here). The next two sections use this memory model to verify both persistent arrays and persistent union-find data structures.

4.3 Verifying Persistent Arrays

The type of persistent arrays is directly modelled as the `pointer` type for references. The `data` type is the same sum type as in the OCAML code:

```
Inductive data : Set :=
  | Arr : data
  | Diff  : Z → Z → pointer → data.
```

We use the fact that there is a single instance of the `Arr` node to make it a constant constructor. Indeed, we model the OCAML heap by the pair made of a dictionary mapping pointers to values of type `data` and of the contents of the array designated by `Arr`, which is here modelled as a function from `Z` to `Z`:

```
Record mem : Set := { ref : PM.t data; arr : Z→Z }.
```

It is clear that we only model the part of the heap relative to a single persistent array and its successive versions, but this is enough for this formal proof (since there is no operation taking several persistent arrays as arguments, which would require a more complex model).

As defined above, our model already includes much more possible situations than allowed by the sole `create` and `set` operations (exactly as the OCAML type from Section 2.3.1 does not exclude *a priori* the construction of cyclic values for instance). To introduce the structural invariant of persistent arrays, we introduce the following inductive predicate, `pa_valid`, which states that a given reference is a valid persistent array:

```
Inductive pa_valid (m: mem) : pointer → Prop :=
  | array_pa_valid :
      ∀p, PM.find (ref m) p = Some Arr →
      pa_valid m p
  | diff_pa_valid :
      ∀p i v p',
      PM.find (ref m) p = Some (Diff i v p') →
      pa_valid m p' → pa_valid m p.
```

This definition says that the reference points either to a value `Arr` or to a value `Diff` i v p' with p' being itself a valid persistent array. Note that it implies the absence of cycles due to its inductive nature.

To express the specifications of the `get` and `set` functions, we relate each persistent array to the function over $\{0, 1, \ldots, n-1\}$ that it represents. Such a function is directly modelled as a value of type `Z→Z`. We introduce the following inductive predicate relating a reference to a function:

```
Inductive pa_model (m: mem)
  : pointer → (Z → Z) → Prop :=
  | pa_model_array :
      ∀p, PM.find (ref m) p = Some Arr →
      pa_model m p (arr m)
  | pa_model_diff :
```

```
    ∀p i v p′,
    PM.find (ref m) p = Some (Diff i v p′) →
    ∀f, pa_model m p′ f →
    pa_model m p (upd f i v).
```

where upd is the pointwise update of a function, defined as

```
Definition upd (f:Z→Z) (i:Z) (v:Z) :=
  fun j ⇒ if Z_eq_dec j i then v else f j.
```

As we can notice, the definition of pa_model is analogous to the one of pa_valid. It is even clear that pa_valid m p holds as soon as pa_model m p f does for any function f. But it is convenient to distinguish the two notions, as we will see in the following.

We can now give specifications to the get and set functions. Opting for a definition-as-proof, a possible type for get is:

```
Definition get :
  ∀m, ∀p, pa_valid m p →
  ∀i, { v:Z | ∀f, pa_model m p f → v = f i }.
```

The precondition states that the reference p must designate a valid persistent array and the postcondition states that the returned value v must be f v for any function f modelled by p through the memory layout m. The correctness proof of get does not seem much difficult, due to the tautological nature of its specification. Though it raises a serious issue, namely its termination. Indeed, get only terminates because we assumed p to be a valid persistent array. If not, we could have a circularity on the heap (such as the one resulting from `let rec p = ref (Diff (0,0,p)))` and some calls to get would not terminate. To express the termination property, *i.e.* that reference p eventually leads to the Arr node, we introduce the predicate dist m p n which states that the distance from p to the Arr node is n. Then we can introduce the relation $R\,m$, for a given memory m, stating that a pointer is closer to the Arr node than another one:

$$R\, m\, p_1\, p_2 \equiv \exists n_1,\, \exists n_2,\, \texttt{dist}\, m\, p_1\, n_1 \wedge \texttt{dist}\, m\, p_2\, n_2 \wedge n_1 < n_2$$

Then get can be defined using a well-founded induction over this relation. Since it is easy to show that if p points to Diff i v p' then R m p' p holds, under the hypothesis pa_valid m p, the termination of get follows.

The set function has a more complex specification since it must not only express the assignment but also the persistence of the initial argument. A possible specification is the following:

```
Definition set :
  ∀m:mem, ∀p:pointer, ∀i v:Z,
  pa_valid m p →
  { p′:pointer & { m′:mem |
    ∀f, pa_model m p f →
       pa_model m′ p f ∧
       pa_model m′ p′ (upd f i v) } }.
```

Here the function is returning both the resulting reference p' but also the new memory layout m'. The precondition is still the validity of the persistent array passed as argument. The postcondition states that for any function f that p was modelling in the initial memory m then p keeps modelling f in the new memory m' and p' is modelling the result of the assignment, that is the function upd f i v.

The proof of correctness for set only contains a single difficulty: we need to show that the allocation of a new reference (through the function PM.new) does not modify the allocated values on the heap. This is expressed by the following separation lemma:

```
Lemma pa_model_sep :
  ∀m, ∀p, ∀d, ∀f,
  pa_model m p f→
  pa_model
    (Build_mem
       (PM.add (ref m) (PM.new (ref m)) d)
       (arr m))
    p f.
```

The proofs of correctness for get and set represent a total of 140 lines of Coq script, including all auxiliary lemmas and definitions. This proof does not include the verification of function reroot (which is only an optimization). In our formalization, this function would have the following type:

```
Definition reroot :
  ∀m:mem, ∀p:pointer, pa_valid m p →
  { m′:mem |
     ∀f, pa_model m p f → pa_model m′ p f }.
```

4.4 Verifying Persistent Union-Find

To model the union-find data structure, we set the number of elements once and for all, as a parameter N of type Z. We can omit the management of ranks without loss of generality, since they only account for the complexity, not for the correctness. Thus the union-find data structure reduces to a single array (the parent array). We can reuse the previous memory model and we can model a union-find data structure as a value of type pointer.

As we did for persistent arrays, we first define a validity predicate. Indeed, a union-find data structure is not made of any persistent array but of one modelling a function mapping each integer in $[0, N-1]$ to a representative, in one or several steps. In particular, this property will ensure the termination. To define this validity predicate, we introduce the notion of representatives, for a function f modelling the persistent array parent, as a relation repr f i j meaning that the representative of i is j:

```
Inductive repr (f: Z→Z) : Z→Z→Prop :=
  | repr_zero :
      ∀i, f i = i → repr f i i
  | repr_succ :
      ∀i j r, f i = j → 0≤j<N →
      ~j=i → repr f j r → repr f i r.
```

Then we can define the validity notion as both the validity of the persistent array and the existence of a representative for any integer in $[0, N-1]$:

```
Definition reprf (f:Z→Z) :=
  (∀i, 0≤i<N → 0≤f i<N) ∧
  (∀i, 0≤i<N → ∃j, repr f i j).
Definition uf_valid (m:mem) (p:pointer) :=
  pa_valid m p ∧
  ∀f, pa_model m p f → reprf f.
```

Specifying the find function is more complex than simply saying that the returned value is the representative. Indeed, the find function modifies the memory, due to path compression, and thus we need to express the invariance of classes throughout this compression. We first define the property for two functions to define the same set of representatives:

```
Definition same_reprs f1 f2 :=
  ∀i, 0≤i<N → ∀j, repr f1 i j ↔ repr f2 i j.
```

Then we can specify the find function (actually the find_aux function returning the new persistent array, modelled here as a pointer p'):

```
Definition find :
  ∀m, ∀p, uf_valid m p →
  ∀x, 0≤x<N →
  { r:Z & { p′:pointer & { m′:mem |
```

```
    uf_valid m' p' ∧
    ∀f, pa_model m p f → repr f x r ∧
    ∀f', pa_model m' p' f' → same_reprs f f' } } }.
```

Once again the proof requires a well-founded induction, here on the distance from x to its representative.

To specify the union function, we need a notion of equivalent elements, *i.e.* belonging to the same class:

```
Definition equiv f x y :=
  ∀cx cy, repr f x cx → repr f y cy → cx=cy.
```

Then we can specify union (the assignment `h.parent <- f` is here modelled by the returned pointer p'):

```
Definition union :
  ∀m, ∀p, uf_valid m p →
  ∀x y, 0≤x<N → 0≤y<N →
  { p':pointer & { p1:pointer & { m':mem |
      uf_valid m' p1 ∧ uf_valid m' p'  ∧
      ∀f1, pa_model m p f1 →
      ((∀f2, pa_model m' p1 f2 →
                  same_reprs f1 f2)
       ∧
       (∀f', pa_model m' p' f' →
        ∀a b, 0≤a<N → 0≤b<N →
          (equiv f' a b ↔
          (equiv f1 a b ∨
            (equiv f1 a x ∧ equiv f1 b y) ∨
            (equiv f1 b x ∧ equiv f1 a y))))))
  }}}.
```

The slightly complex postcondition states several properties: first the persistence of the initial structure (it is still valid in the new memory and defines the same set of representatives); then the validity of the new structure; finally the behavioral property, namely that union indeed merges the two classes of x and y. We express this last property by saying that, for any elements a and b, a and b are in the same class after the merge (function f') if and only if they were already in the same class (function f_1) or a and b were both in the initial classes of x and y.

The proofs of correctness of find and union represent a total of 600 lines of Coq script. Note that, since we omitted the management of ranks, `union a b` always appends the class of `b` as a subtree of the representative of `a`. To be fully correct, we should also consider the converse as if union was performing a nondeterministic choice.

5. Conclusion

We have presented a persistent data structure for the union-find problem which is as efficient as the imperative Tarjan's algorithm [17] on realistic benchmarks. In particular, our solution is exactly the same as the imperative one when used linearly *i.e.* without any backtracking.

Our solution is built from a persistent version of Tarjan's algorithm together with an efficient implementation of persistent arrays following an idea from Baker [4, 5]. Though persistent, these two data structures make heavy use of side effects. Contrary to a widely spread idea, persistent data structures are not necessarily purely applicative (even if excellent books such as Okasaki's [14] only focus on purely applicative solutions). As a consequence, it is less obvious to convince oneself of the correctness of the implementation and this is why we also conducted a formal verification of our code.

Another consequence of the imperative nature of this persistent data structure is that it is not thread-safe. Indeed, the assignment in function union is atomic but assigments in function set are not (function reroot makes a lot of assigments all over the data structure).

The most efficient version we finally obtained actually uses arrays that are not fully persistent. Indeed, they must only be used to come back to previous versions (which is the typical use of persistent structures in a backtracking algorithm). This *semi-persistence* currently has a dynamic nature (the data is made invalid when we do the backtrack) but it would be even more efficient if we checked statically the legal use of this semi-persistence. Such a static analysis is work in progress [9].

A. Imperative Union-Find Algorithm

The following code implements the optimal imperative solution for union-find, as described in the introduction. The data structure is a record containing two arrays:

```
type t = { parent : int array; rank : int array }
```

parent links together the elements of each class and rank contains the size of each class. Creation is straightforward, using Array.init to create an array where each element is mapped to itself:

```
let create n =
  { parent = Array.init n (fun i → i);
    rank = Array.create n 0 }
```

The function find recursively follows the links until it finds the representative (that is an element mapped to itself), performing path compression along the way:

```
let rec find uf i =
  let pi = uf.parent.(i) in
  if pi == i then
    i
  else begin
    let ci = find uf pi in
    uf.parent.(i) ← ci; (* path compression *)
    ci
  end
```

Finally, union maps the representative of the smallest class to the one of the largest and updates the rank when necessary:

```
let union ({ parent = p; rank = r } as uf) x y =
  let cx = find uf x in
  let cy = find uf y in
  if cx != cy then begin
    if r.(cx) > r.(cy) then
      p.(cy) ← cx
    else if r.(cx) < r.(cy) then
      p.(cx) ← cy
    else begin
      r.(cx) ← r.(cx) + 1;
      p.(cy) ← cx
    end
  end
```

Acknowledgments

We are grateful to the anonymous reviewers for their helpful comments and suggestions. We also thank the members of the Proval project for many discussions related to the persistent union-find problem. In particular, we are grateful to Claude Marché for mentioning T.-R. Chuang's paper [7] and to Christine Paulin for encouraging us to perform a verification proof in Coq.

References

[1] The Coq Proof Assistant. http://coq.inria.fr/.

[2] The Objective Caml Programming Language. `http://caml.inria.fr/`.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[4] Henry G. Baker. Shallow binding in Lisp 1.5. *Commun. ACM*, 21(7):565–569, 1978.

[5] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Not.*, 26(8):145–147, 1991.

[6] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`.

[7] Tyng-Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *ESOP'92: Symposium proceedings on 4th European symposium on programming*, pages 110–129, London, UK, 1992. Springer-Verlag.

[8] Sylvain Conchon and Evelyne Contejean. Ergo: A Decision Procedure for Program Verification. `http://ergo.lri.fr/`.

[9] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-Persistent Data Structures. Research Report, LRI, Université Paris Sud, 2007. `http://www.lri.fr/~filliatr/publis/spds.ps`.

[10] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[12] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[13] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980.

[14] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[15] Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

[16] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984.

[17] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.