3 Persistent Arrays

We warmly thank Arthur Charguéraud at Inria for proposing this challenge.

A persistent array is a data structure that provides the same operations as an array, except that the operation that updates the array returns a new persistent array and leaves the previous one unchanged. A minimal OCaml signature for the operations on polymorphic persistent arrays is

```
val create : int → 'a → 'a parray
val get : 'a parray → int → 'a
val set : 'a parray → int → 'a → 'a parray
```

We consider a version of persistent arrays based on Baker's arrays [1] that provides better performance guarantees for *head persistent* versions.

A persistent array is said to be *head persistent* if it has not been provided as argument to any operation since it has been created (by either `create` or `set`). Such a *head persistent array* implicitly "degrades" to being just a persistent array after it is provided as argument to another operation.

We are interested in a structure where `get` and `set` operations on head persistent arrays are $O(1)$ amortized, and where operations on other persistent arrays are $O(n)$ in the worst-case, where n denotes the length of the array. In contrast, Baker's array have operations of unbounded costs.

3.1 Implementation

Our persistent arrays are represented as follows. We explain the ingredients next.

```
type 'a parray = {
  mutable desc : 'a parray_desc;
  mutable dist : int (* length of the incoming chain *)
}

and 'a parray_desc =
| Base of { elements: 'a array }
| Diff of { data: 'a parray; index: int; value: 'a }
```

Following Baker's algorithm [1], a persistent array is either a plain array in the case of the head persistent version, or a "diff" on another persistent array `data`. Such a "diff" describes an array whose value should be the same as that of `data`, except at one specific `index`, where the contents should be replaced with a specific `value`. These two cases correspond to the constructors `Base` and `Diff`, respectively, in the type definition.

The full implementation of the data structure is given in Fig. 1. We briefly describe it below.

Consider a *head persistent array*, i.e., a `parray` described as a `Base`. This constructor carries a pointer to an ephemeral array, call it `a`. To read a value in such a head persistent array, it suffices to read in the array `a`.

Consider now an update operation setting the value `v` at index `i`. The result of the update consists of a fresh parray node, of type `'a parray`; this node becomes the new head persistent version. The array `a` is effectively modified by performing a write of the value `v` at index `i`. The fresh node is described as a `Base` carrying the pointer to `a`, which has been updated. To preserve the representation of the original node, we set its description to a `Diff`, pointing onto

the fresh node, and storing the index `i` and the values that was stored at index `i` in the array `a` prior to the write of `v`.

With Baker's persistent arrays [1], the chain of diffs may have unbounded length. To ensure worst-case bounds, we consider a variant where each persistent array version stores an upper bound on the length of the (unique) chain that reaches this node. When performing an update operation, if the length of the chain of diffs is about to exceed the length of the array, then begin the construction of an independent chain. In other words, rather than modifying the persistent array at hand, we instead allocate an independent ephemeral array on which to apply the update operation.

Consider now a non-head persistent array. Different strategies are possible for reading and updating such older versions. We consider the following strategy: whenever a non-head version is accessed, whether for a read or an update operation, we first transform that version into a head version. To that end, we allocate a fresh array to store the relevant data, and "cut the chain of diffs" by replacing a `Diff` node with a `Base` node. Computing the contents for the fresh array can be performed by traversing the chain of diffs backwards from the base array, and applying the modifications described in the diffs.

Tasks

(a) Define the persistent array data structure in your verification tool of choice and implement the `set`, `get`, and `create` operations.

(b) Prove that the implementations of the operations are memory-safe / crash-free. In particular, you should prove that all pointer accesses are safe and that all array accesses are within the bounds of the array.

(c) Prove the functional correctness of the three operations.

(d) Show that get has cost at most $O(n)$; this amounts to bounding the lengths of diff chains.

Extra Tasks

(e) Formalize the notion of head persistent version and show that operations on such versions are $O(1)$.

(f) Extend the implementation so that it is thread-safe. That is, threads should be able to call operations concurrently on the same persistent array or two persistent arrays that share parts of their memory representation. Verify the memory safety and functional correctness of the concurrent versions of the operations.

References

[1] Henry G. Baker. Shallow binding makes functional arrays fast. 26(8):145–147, 1991.

```
let create n d =
 let a = Array.make n d in
 { desc = Base {elements = a}; dist = 0 }

let rec to_array pa =
 match pa.desc with
 | Base { elements } → Array.copy elements
 | Diff {data; index; value} →
  let a = to_array data in
  a.(index) ← value;
  a

let rebase_and_get_array pa =
 match pa.desc with
 | Base { elements } → elements
 | Diff _ →
  let a = to_array pa in
  pa.desc ← Base {elements = a};
  a

let get pa i =
 let a = rebase_and_get_array pa in
 a.(i)

let set pa i x =
 let a = rebase_and_get_array pa in
 if pa.dist = Array.length a then begin
  let b = Array.copy a in
  b.(i) ← x;
  { desc = Base {elements = b}; dist = 0 }
 end else begin
  let v = a.(i) in
  a.(i) ← x;
  let pb = { desc = Base {elements = a}; dist = pa.dist + 1 } in
  pa.desc ← Diff {data = pb; index = i; value = v};
  pb
 end
```

Figure 1: Full implementation of persistent array data structure