TD: Interior Mutability and Concurrency

Jacques-Henri Jourdan, MPRI 2-4

2023/02/15

1 Persistent Arrays

We would like to implement a library of *persistent arrays* in Rust. This library will be able to represent arrays, but in a persistent manner, that is, without observationally supporting in-place modifications. In particular, the function `set` of this library will not (observationally) mutate the array given in parameter, but will instead return a new persistent array representing the updated version.

Internally, a persistent array is either represented as a traditional mutable array, or as a specific value representing and update of another persistent array. That is, a persistent array will be represented as a sequence of updates to a mutable array stored traditionally in the memory. Accesses to the array will change the internal representation of the persistent array values to make sure that values used recently have a low number of successive updates to the physical array. In the literature, this is known as the "Baker's trick".

In Rust, we will use a `parray` module for this library, containing the following type definitions:

```
mod parray {
    use std::cell::RefCell;
    use std::rc::Rc;

    enum PAState<T> {
        Arr(Vec<T>),
        Diff(usize, T, PArray<T>),
        Invalid
    }
    use self::PAState::*;

    pub struct PArray<T>(Rc<RefCell<PAState<T>>>);

    impl<T> PArray<T> {
        fn reroot(&self) {
            panic!()
        }
    }
}
```

The public type of persistent arrays is `PArray`. It contains a reference-counted pointer (to enable easy sharing) to a `RefCell` (to enable the change of the internal representation) of a value of type `PAState<T>`. The type `PAState<T>` is an `enum` representing the two different possible representation of a persistent array: either `Arr` for a traditional array,

or `Diff` for an update over another persistent array. The `Invalid` constructor can be used as a default value when one wants to take ownership of the content of a `RefCell<PAState<T>>`.

The associated function `reroot`, whose implementation will be filled later, aims at *rerooting* the internal representation of a persistent array: while keeping the publicly visible value of persistent arrays, it changes the internal representation of persistent array so that `self` directly points to a traditional array.

A template source file is available on the course website. It also contains few helper functions, `get_arr`, `get_arr_mut` and `from_diff` which you can use if needed to answer the exercises.

Exercise 1. Write an associated function `new` which creates a new persistent array from a given content:

```
impl<T> PArray<T> {
    pub fn new(v: Vec<T>) -> Self {
        // TODO
    }
}
```

Exercise 2. Implement the `Clone` trait for `PArray<T>`, so that persistent arrays can be shared at low cost.

Exercise 3. Write two functions `get` and `set` which perform array accesses:

```
impl<T> PArray<T> {
    pub fn get(&self, i: usize) -> T where T: Copy {
        // TODO
    }

    pub fn set(&self, i: usize, x: T) -> Self {
        // TODO
    }
}
```

The functions should *rerooted arrays* as explained above: `get` should reroot its parameter, `set` should return a rerooted array. They can use the `reroot` function without implementing it.

Exercise 4. Implement the `reroot` function, without allocating a new array. Test the implementation, by running the compiled program, with the option `-O`. The tests should pass, and the benchmarks should run in a few tenths of a second. (Note: a recursive implementation of `reroot` will trigger a stack overflow when running the benchmark. Under Linux, one can expand the size of the stack using the command `limit -s unlimited`).

Exercise 4' (bonus). Give another implementation of `reroot`, which is in-place and not recursive.

2 Concurrency

Exercise 5. Give examples of types which are:

- both `Send` and `Sync`,
- `Send` but not `Sync`,
- neither `Send` nor `Sync`,
- `Sync` but not `Send`.

Exercise 6. Have a look at the API of the `RwLock` type in the standard library. What other standard library type is it close to (apart from `Mutex`)? Explain why this type and its satellites are (or not) `Send` and `Sync`, and in which case.

Exercise 7. A classical operator to write concurrent programs is the parallel composition: it takes two programs, run them concurrently and finishes when both programs finished. Give the type that such a function could have in Rust. Compare with the type of `spawn` that we have seen during the course.

Exercise 8. Have a look at the API of the `std::sync::mpsc` module in the standard library. What data structure does it implement? Is this an instance of interior mutability? Comment the instances of `Send` and `Sync` in this module.

3 Misc

Exercise 9. Recall the types of `RefMut::map` and `Ref::map`:

```
impl<'b, T> RefMut<'b, T> {
    pub fn map<U, F>(orig: RefMut<'b, T>, f: F) -> RefMut<'b, U>
        where F: FnOnce(&mut T) -> &mut U
    { ... }
}

impl<'b, T> Ref<'b, T> {
    pub fn map<U, F>(orig: Ref<'b, T>, f: F) -> Ref<'b, U>
        where F: FnOnce(&T) -> &U
    { ... }
}
```

What are the lifetimes of the borrows taken and returned by the closures `F`? Where are they bound? Why is this important for type soundness?