TD: Interior Mutability and Concurrency

Jacques-Henri Jourdan, MPRI 2-4

2024/02/06

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 an 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 temporary 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 all the persistent arrays existing in memory, it changes the internal representation of the given 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 a function `set` which takes a persistent array, an index and a new value, and return an updated persistent array:

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

This function should not copy the whole content of the array, but rather use the `Diff` constructor.

Exercise 4. Write a function `get`, which reads the content of a persistent array:

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

This function should *reroot* the array given as parameter, as explained above, so that further accesses to that array will be cheap. You can use the `reroot` function without implementing it (yet).

Exercise 5. Implement the `reroot` function. It should change the representation of persistent arrays without changing their observable content.

Be cautious: by modifying the initernal representation, one may change the observable value of the persistent array passed as a parameter, but also the observable value of other

persistent arrays, which happen to also use the memory we are modifying. Rerooting should keep the observable content of all these persistent arrays intact.

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 6. Give examples of types which are:

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

Exercise 7. 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 8. 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 9. 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 Union-find

Union-Find is an important data structure allowing to compute equivalence classes for the reflexive transitive symmetric closure of a relation. It is particularly important in the implementation of unification in a type checker.

Union-find uses trees to represent equivalence classes. Each node has a pointer to its *parent* (instead of pointers to children). We provide two operations to this structure: `find` takes a node and returns a canonical representative of the relation it belongs to (the root of the tree), and `union` takes two nodes and merges their equivalence classes.

In order to implement union-find efficiently, we need sharing and mutation. In Rust, we thus use *interior mutability*.

We use the following type:

```
#[derive(PartialEq, Eq, Debug, Clone)]
struct Node<T>(Rc<RefCell<Inner<T>>>);

#[derive(PartialEq, Eq, Debug)]
enum Inner<T> {
    Root { data: T },
    Link { parent: Node<T> },
}

impl Node<T> {
    fn new(data: T) -> Self {
        Node(Rc::new(RefCell::new(Root { data })))
    }
}
```

The type `Node` defined above is actually a *pointer* to either to the `Root` or a `Link`.

Exercise 10. Implement `Node::naive_find` and `naive_union`. Try to avoid stack overflow for large tests.

The naive implementation of the previous exercise does work, but can be particularly slow when the tree built by the data structure is not balanced. In order to solve this problem, we do an easy optimisation: we give a *rank* to every tree root which represents (more or less) the height of the corresponding tree. When we merge two trees using `union`, we can put the tree with the smallest rank under the on with the largest height.

Exercise 11. Add to `Root` a field `rank: usize`, and update the definitions in the rest of the code to implement this optimization.

An other performance problem that we can encounter with `Union-Find` is that large trees can have long brnaches which are traversed repeatedly when calling `find`. If we can remember immediately the canonical representative of every class we could save a lot of computing time. To that end, we can compress the paths in the tree: if the representative of a parent of a node is not the parent itself, we will replace our parent by this representative directly.

Exercise 12. Implement path compression in your implementation of `find`.

4 Misc

Exercise 13. 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?