TD: Introduction to Rust

Jacques-Henri Jourdan, MPRI 2-4

2022/02/09

1 Binary search trees

We use the following definition of dictionaries implemented as binary search trees in Rust (the generic parameters `K` and `V` stand for the type of keys, and values, respectively):

```
struct Node<K, V> {
    left: BST<K, V>,
    key: K,
    val: V,
    right: BST<K, V>
}

enum BST<K, V> {
    Leaf,
    Node(Box<Node<K, V>>)
}
use BST::*;
```

On this type, we maintain the invariant that values of this type have the binary search tree property with respect to the order defined by the `Ord` trait on keys, and that no key appear twice in the same tree.

Exercise 1. In an inherent implementation block for the `BST` type, write a function `BST::new` which takes no parameter and returns an empty tree.

Exercise 2. Write a method `find_copy_rec`, which searches for a key in a dictionary, and returns a copy of the corresponding value if found. It should work recursively and require `K: Ord` in order to compare keys. You can use the following template code:

```
impl<K: Ord, V: Copy> BST<K, V> {
    fn find_copy_rec(&self, k: K) -> Option<V> {
        // Write code here
    }
}
```

Exercise 3. Write a method `find_copy`, which does the same, but using a loop instead of performing a recursive call.

Exercise 4. Write a method `find`, which does the same, but returns a shared borrow to the value instead of a copy and does not require `V: Copy`.

Exercise 5. Write a method `insert`, which adds an entry to the dictionary, and returns the previously stored value, if it exists. You can use the following template code:

```
impl<K: Ord, V> BST<K, V> {
    fn insert<'a>(&'a mut self, k: K, v: V) -> Option<V> {
        // Write code here
    }
}
```

Hint: to solve this question, one may use the function `std::mem::replace` from Rust's standard library. Its documentation can be found online.

We are now going to implement various iterators for these dictionaries.

Exercise 6. Write a *draining iterator*, which consumes the given binary search tree and produces the pairs of keys and values, ordered by keys. You can use the following iterator type:

```
struct IterMove<K, V> {
    cur: BST<K, V>,
    stack: Vec<(K, V, BST<K, V>)>
}
```

The field `cur` contains the next subtree to be iterated over, while the field `stack` contains the keys and values and subtrees hanging on right of the path to the subtree `cur` in the full tree.

To fulfill Rust ownership rules, you can use the `std::mem::replace` function of the standard library.

Write the corresponding instance:

```
impl<K, V> IntoIterator for BST<K, V> {...}
```

Exercise 7. Implement an iterator over pairs of borrows to keys and values (of type `(&K, &V)`).

Write the corresponding instance:

```
impl<'a, K, V> IntoIterator for &'a BST<K, V> {...}
```

Exercise 8. Write a variant of the previous question that produces a mutable borrow to the value (i.e., it produces values of type `(&K, &mut V)`, and the corresponding `IntoIterator` instance:

```
impl<'a, K, V> IntoIterator for &'a mut BST<K, V> {...}
```

Hint: to satisfy Rust's ownership requirements, you can change the `cur` field to an `Option` type.

Why don't we want this iterator to produce mutable borrows of keys too?

2 Variance

Exercise 9. What are the variances of the types `Box`, `&`, `&mut` and `Vec`, with respect to the lifetime parameter and the type parameter?