Cannot return mutable reference to member from a match arm

Asked 1 year, 4 months ago
Active 1 year, 4 months ago
Viewed 241 times

1

1

As an exercise in rust borrowing and lifetimes, I want to implement a simple binary tree. However, I am stuck on something. Consider this:

```
struct Node {
    key: i32,
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

struct BinaryTree {
     root: Option<Box<Node>>,
}

impl BinaryTree {

    fn find_mut(&mut self, key: i32) -> &mut Option<Box<Node>> {
        let mut node = &mut self.root;
        loop {
            match node {
                Some(box_node) if box_node.key != key => {
                    node = if box_node.key < key {
                        &mut box_node.right
                    } else {
                        &mut box_node.left
                    }
                },
                other => return other
            }
        }
    }
}
```

The above fails to compile with:

```
error[E0505]: cannot move out of `node` because it is borrowed
  --> src/main.rs:40:17
   |
29 |     fn find_mut(&mut self, key: i32) -> &mut Option<Box<Node>> {
   |                 - let's call the lifetime of this reference `'1`
...
33 |                 Some(box_node) if box_node.key != key => {
   |                      -------- borrow of `node.0` occurs here
...
40 |                 other => return other
   |                 ^^^^^           ----- returning this value requires that `node.0` is borrowed for `'1`
   |                 |
   |                 move out of `node` occurs here
```

I tried explicitly setting the lifetime for `self` and the output. I also tried to expand the `Some(_)` arm and match for `None` instead of `other` as well.

(Edit 2): The purpose of `find_mut` is to return a ref to the object where a new node should be created (in case the key is not found) or where the existing node is.

What is the reason for the compile error, in more detail? How should I go to fix it? Is what I am trying to do even a good practice, (edit 1) i.e. return ref to the Optional where the modification should be at (assuming this is not a public method)?

rust borrow-checker ownership

Share
Improve this question
Follow

edited Aug 15 '20 at 18:51

asked Aug 15 '20 at 17:20

Ivan Kalchev
325 ● 3 ● 9

2

I think you found a known limitation of the borrow checked, related to the lifetimes of returned borrows. FWIW, it compiles with `-Z polonius` in nightly, so I expect a future version of the compiler will accept it. Something like this github issue.
– rodrigo Aug 15 '20 at 18:12

Add a comment

1 Answer

Active Oldest Votes

2

The reason why Rust compiler issues error is because `Some(expr)` pattern matches the whole expression in its owned form, that is, `expr` is moved.

Usually, that is easily solved by matching on expression as a borrow `Some(ref expr)`, or mutable borrow `Some(ref mut expr)`, but that is not the case here.

If you look at standard library, you will often see that `as_mut()` / `as_ref()`, when value may not exist, always returns `Option<&mut T>` rather than `&mut Option<T>`. That's because you really want to access the value, not any innards of the data structure, which constructs like `Option<Box<None>>` are.

Following that, I came up with this:

```
struct Node {
    key: i32,
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

struct BinaryTree {
     root: Option<Box<Node>>,
}

impl BinaryTree {
    fn find_mut(&mut self, key: i32) -> Option<&mut Node> {
        // &mut Option<Box<Node>> -> Option<&mut Box<Node>> -> Option<&mut Node>
        let mut node = self.root.as_mut().map(|boxed| boxed.as_mut());
        loop {
            match node {
                Some(box_node) if box_node.key != key => {
                    node = if box_node.key < key {
                        box_node.right.as_mut().map(|boxed| boxed.as_mut())
                    } else {
                        box_node.left.as_mut().map(|boxed| boxed.as_mut())
                    }
                },
                other => return other
            }
        }
    }
}
```

There might be a nicer way to write this, but I am not aware of it at the moment.

Note that this solves both the issue with ownership, since now `&mut Node` is what is being moved here, and makes the API nicer at the same time.

As to whether its good practice, given its double meaning, yes and no.

It helps you learn how to deal with borrows; on the other hand, we already have Vec::binary_search , and BTreeMap / BTreeSet , and its likely that on crates.io there are other implementations that should cover all but the most extreme cases and there's little point in making a search tree yourself.

Share
Improve this answer
Follow

answered Aug 15 '20 at 18:21

Yamirui
149 ● 6

It would be nice to understand *why* mutable borrow like `Some(ref mut box_node)` doesn't help here. I agree that returning `Option<&mut T>` is nicer than returning `&mut Option<T>`, but it's not intuitively clear to me why the latter wouldn't work. After all, the OP's data structures do consistently contain `Option<Box<Node>>`, so it looks ok to return a reference to one of them, with the lifetime of the tree.
– user4815162342
Aug 15 '20 at 19:02

@user4815162342 I'm not sure how to explain it so anyone can understand, but essentially `acquire borrow` -> `acquire borrow that is referencing data through a borrow before` -> `initial borrow drops (but you want to keep a second borrow, that depends on first one)`, which is pretty much what compiler will tell you if you try to modify original code and use `Some(ref mut box_node)`. Now, that wouldn't be an issue with pointers, "sadly", references inherit lifetime bounds from their ancestors and that doesn't quite work as you'd want it to.
– Yamirui
Aug 15 '20 at 19:11

I thought the second borrow would just inherit the lifetime of the first borrow (in this case the lifetime of the whole tree). Still, I tried to modify the code to use `Some(ref mut box_node)`, but the compiler just complained about moving out of the other match arm. When I added the `ref mut` to the other match arm, the compiler started to complain about two mutable borrows, which I don't quite understand as the two should never exist in parallel.
– user4815162342
Aug 15 '20 at 19:29

@user4815162342 the issue, as I mentioned, is that a borrow from a borrow inherits lifetime of a former borrow, and in this case, former borrow exists purely in this method, or rather, inside the match, and cannot outlive it. If there's a safe way to tell rust that it's not the case I'm not aware of it (you can look at what `as_mut` does), that is why instead of double borrow, I extracted a single final borrow preemptively and everything just clicked into their places.
– Yamirui
Aug 15 '20 at 21:20

1

You can also use `Option::as_deref_mut(&mut Option<Box<Node>>) -> Option<&mut Node>`
– Jakub Dąbek
Aug 16 '20 at 13:49

Show 2 more comments

Your Answer

Post Your Answer

By clicking “Post Your Answer”, you agree to our terms of service, privacy policy and cookie policy

Not the answer you're looking for? Browse other questions tagged rust borrow-checker ownership or ask your own question.

The Overflow Blog

- Sequencing your DNA with a USB dongle and open source code
- Don’t push that button: Exploring the software that flies SpaceX rockets and...

Featured on Meta

- Providing a JavaScript API for userscripts
- Congratulations to the 59 sites that just left Beta

Related

Cannot move out of borrowed content / cannot move out of behind a shared reference

6
cannot move out of borrowed content when unwrapping a member variable in a &mut self method

How do I return a reference to something inside a RefCell without breaking encapsulation?

Why doesn't the lifetime of a mutable borrow end when the function call is complete?

Method not compatible with trait with confusing error message

Why do I get expected type `()` when trying to obtain a reference to a boxed value?

How do I return a mutable reference to an Optional boxed Trait stored in a struct member

Const array of closures taking a mutable reference to a struct with lifetime parameter in Rust

Compiler thinks a borrow is alive when it isn't

Hot Network Questions

Question on OEIS A000085
Why is my reasoning incorrect - probability?
I've got a material setup that blends two shaders decently, but how would I apply it to a circular target?
Using a friend to move cash into my checking account
Tax implications of large gift
more hot questions

Question feed