Operator overloading by value results in use of moved value

Asked 6 years, 11 months ago
Active 6 years, 11 months ago
Viewed 961 times

7

2

Compiling the following Rust code that uses operator overloading

```
use std::ops::{Add};

#[derive(Show)]
struct Point {
    x: int,
    y: int
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {x: self.x + other.x, y: self.y + other.y}
    }
}

fn main() {
    let p: Point = Point {x: 1, y: 0};
    let pp = p + p;
}
```

Results in compiler errors due to ownership of p:

```
<anon>:21:18: 21:19 error: use of moved value: `p`
<anon>:21     let pp = p + p;
                       ^
<anon>:21:14: 21:15 note: `p` moved here because it has type `Point`, which is non-copyable
<anon>:21     let pp = p + p;
                   ^
```

The rationale behind it is explained here and led to an RFC that was not accepted (part of due to reasons of the above example). However, later the following RFC still introduced the by-value type signatures for operators.

While I understand the rationale behind the decision. Due to my lack of experience in rust, I'm not sure what the "proper" way would be to allow the above code to work (a) if I do not want to copy or (b) how to make the struct copyable?

copy | operator-overloading | rust | pass-by-value | ownership

Share
Improve this question
Follow

edited Jan 16 '15 at 15:58

Shepmaster
305k • 59 • 824 • 1083

asked Jan 9 '15 at 9:04

zgerd
942 • 6 • 15

Add a comment

2 Answers

Active Oldest Votes

5

If you don't want to copy then, as far as my newbie understanding goes, you need to implement `Add` on references to `Point`.

This would be supported by the RFC:

> Fortunately, there is no loss in expressiveness, since you can always implement the trait on reference types. However, for types that do need to be taken by reference, there is a slight loss in ergonomics since you may need to explicitly borrow the operands with &. The upside is that the ownership semantics become clearer: they more closely resemble normal function arguments.

And indeed it seems to work:

```
use std::ops::{Add};

#[derive(Show)]
struct Point {
    x: i32,
    y: i32
}

impl<'a> Add for &'a Point {
    type Output = Point;

    fn add(self, other: &'a Point) -> Point { //'
        Point {x: self.x + other.x, y: self.y + other.y}
    }
}

fn main() {
    let p: Point = Point {x: 1, y: 0};
    let pp = &p + &p;
    println!("{:?}", pp);
}
```

(playpen)

To make `Point` copyable instead, just replace `#[derive(Show)]` with `#[derive(Show,Copy)]`. Such structs used to be copyable by default, but it changed.

Share
Improve this answer
Follow
edited Jan 15 '15 at 23:36

answered Jan 9 '15 at 10:27

Michał Politowski
4,050 ● 3 ● 30 ● 40

1

The problem with this is that `let pp = &p + &p + &p` is not working.
– SirVer
Jan 15 '15 at 9:43

@SirVer yes, you would have to write something like `let pp = &(&p + &p) + &p`. I guess that the practical thing to do would be to create several implementations, like the answer by Vladimir Matveev suggests (or just derive `Copy` and be done with it).
– Michał Politowski
Jan 15 '15 at 11:31

Add a comment

4

If your structure can't be copied (e.g. it has `Drop` implementation, either itself or for one of its fields), then it may make sense to create several implementations: value+value, value+reference, reference+value and reference+reference. The first three can reuse the storage of one of the operands, and the last one can clone one of the operands and then just delegate to the already existing implementations. This way the user of your library can easily decide whether they want to reuse existing values for optimization or not.

In fact, that's how e.g. `BigInt` or `Complex` types are handled.

Your `Point`, however, can just be made `Copy` as it is cheap to copy it.

Share
Improve this answer
Follow
answered Jan 9 '15 at 18:02

Vladimir Matveev
104k ● 30 ● 254 ● 274

Vladimir, thanks for your reply. What if my type is easy, but expensive to copy, say a `Matrix1000x1000` type? Is the compiler smart enough to avoid copying in chained operations or do I need to write the trivially forwards to the 4 implementations you mentioned?
– SirVer
Jan 16 '15 at 18:12

1

I'm not sure I understand your question. If your type is expensive to copy, don't make it `Copy` and implement operation traits for four variants (self+self, &self+self, self+&self, &self+&self). If your type is that large, you will need to put the data on the heap anyway, so move semantics will make sure that only the structure itself is copied.
– Vladimir Matveev
Jan 16 '15 at 18:25

Add a comment

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 copy operator-overloading rust pass-by-value 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

Visit chat

Linked

1 Could Rust take an immutable reference of a value automatically?

BigUint and "cannot move out of borrowed content" error

Related

7290 Is Java "pass-by-reference" or "pass-by-value"?

Why doesn't Java offer operator overloading?

Operator Overloading with C# Extension Methods

633 What's the difference between passing by reference vs. passing by value?

1582 Is JavaScript a pass-by-reference or pass-by-value language?

Operator overloading in Java

2315 What are the basic rules and idioms for operator overloading?

In Rust, can you own a string literal?

Magic behind match: no implementation for `&std::option::Option<ListNode> == std::option::Option<_>

Hot Network Questions

- 'Cloth shop' and 'Clothes shop'
- I've got a material setup that blends two shaders decently, but how would I apply it to a circular target?
- Seeing oneself in an abstract painting
- Problems that are polynomially "hard" to compute but "easy" to verify
- Company kept previous personal phone number

more hot questions

Question feed

Enterprise

COMPANY

About
Press
Work Here
Legal
Privacy Policy
Terms of Service
Contact Us
Cookie Settings
Cookie Policy

STACK EXCHANGE NETWORK

Technology
Culture & recreation
Life & arts
Science
Professional
Business
API
Data

Blog
Facebook
Twitter
LinkedIn
Instagram