How to `BorrowMut` a trait impl without `dyn`? (nested generic)

help

overlookmotel March 10, 2023, 10:31pm 1

I have a trait `Storage` which is implemented by various types:

```
pub trait Storage {
 fn new() -> Self;
 fn push(&mut self, n: u32) -> ();
}

pub struct HeapStorage { inner: Vec<u32> }
impl Storage for HeapStorage {
 fn new() -> Self {
   Self { inner: Vec::with_capacity(100) }
 }
 fn push(&mut self, n: u32) { self.inner.push(n); }
}

pub struct StackStorage { inner: [u32; 100], pos: usize }
impl Storage for StackStorage {
 fn new() -> Self {
   Self { inner: [0; 100], pos: 0 }
 }
 fn push(&mut self, n: u32) {
   self.inner[self.pos] = n;
   self.pos += 1;
 }
}
```

And a generic type `Loader` which operates on `Storages`:

```
pub struct Loader<S: Storage> { storage: S }
impl<S: Storage> Loader<S> {
 pub fn new() -> Self {
   Self { storage: S::new() }
 }
 pub fn load(&mut self) {
   for n in 0..100 {
     self.storage.push(n);
   }
 }
}
```

So far so good! But what I'd really like is for `Loader` to contain a `BorrowMut<Storage>`, and provide methods to either create a new `Loader` with its own owned `Storage`, or to provide an existing `&mut Storage`:

```
pub struct BetterLoader<S: BorrowMut<Storage>> { storage: S }
impl<S: Storage> BetterLoader<S> {
 pub fn new() -> Self {
   Self { storage: S::new() }
 }
 pub fn from_storage(storage: S) -> Self {
   Self { storage }
 }
 pub fn load(&mut self) {
   for n in 0..100 {
     self.storage.borrow_mut().push(n);
   }
 }
}
```

The compiler doesn't like the above example:

```
error[E0782]: trait objects must include the `dyn` keyword
  |
58 | pub struct BetterLoader<S: BorrowMut<Storage>> {
```

```
  |                                       ^^^^^^^
help: add `dyn` keyword before this trait
  |
58 | pub struct BetterLoader<S: BorrowMut<dyn Storage>> {
```

It's not a trait object I'm after ideally.

I've also tried various other permutations, and always hit an error one way or another. Is there any way to achieve something like this?

I'm sure this question has come up before, but I've been unable to find it - probably because I don't know what this pattern would be called (nested generic?), and so what search terms to use. Please excuse the repetition.

kpreid March 11, 2023, 12:46am 2

You need to introduce a second type parameter to specify which *concrete* type you will borrow as. (This is not just syntactic: a type could implement `BorrowMut` for two different storage types.)

This addition compiles:

```
use core::borrow::BorrowMut;
use std::marker::PhantomData;

pub struct BetterLoader<T, S> {
   storage: T,
   _phantom: PhantomData<S>,
}

impl<T: BorrowMut<S>, S: Storage> BetterLoader<T, S> {
   pub fn new() -> Self
   where
       T: Default,
   {
       Self {
           storage: T::default(),
           _phantom: PhantomData,
       }
   }
   pub fn from_storage(storage: T) -> Self {
       Self {
           storage,
           _phantom: PhantomData,
       }
   }
   pub fn load(&mut self) {
       for n in 0..100 {
           self.storage.borrow_mut().push(n);
       }
   }
}
```

Note that:

- I had to add a `PhantomData` field — because otherwise the compiler is missing information about exactly how the parameter `S` should relate to the type.
- `BetterLoader::new()` can't use `Storage::new()` since `T` doesn't implement `Storage`. But in general, it is often not a good idea to put `new()` methods in your *traits*, precisely because of situations like this, and situations where particular implementations can't be constructed with no inputs. Instead, I used the `Default` trait, and you can and should do this for the entire code (using a `Default + Storage` bound where you need it, and only where you need it).

1 Like

quinedot March 11, 2023, 12:52am 3

I also suggest using some meaningful names for the two type parameters if you go this route.

An alternative may be to implement `Storage` for `&mut T` where `T: Storage`, and then you can do things with `Loader<SomeConcreteS>` or `Loader<&mut SomeConcreteS>`.

1 Like

overlookmotel March 13, 2023, 1:05pm 4

Amazing. That works a treat. Thank you @ kpreid.

Two further questions, if you have time:

1. `PhantomData`

Does `PhantomData<S>` imply anything else about the relationship between `BetterLoader` and `S: Storage`? Does it cause the compiler to do anything different re: optimization?

My understanding of `PhantomData` was it was to indicate that a type conceptually owns another type where that's not evident from the type def. e.g.:

```
struct Owner<Owned> {
 // a `*const Owned` stored as `usize` (bad idea, just for illustration)
 ptr: usize,
 _phantom: PhantomData<Owned>,
}
```

But that's not the relationship between `BetterLoader` and S in example above. The point of using `BorrowMut<S>` is precisely that `BetterLoader` *doesn't* need to own S.

2. @quinedot's suggestion

@quinedot Do you mean a blanket implementation?

```
impl<S: Storage> Storage for &mut S {
 fn new() -> Self {
   // What goes here?
 }
}
```

Sorry if I'm being dumb, but I can't figure out how this would work.

kpreid March 13, 2023, 2:55pm 5

overlookmotel:

> Does `PhantomData<S>` imply anything else about the relationship between `BetterLoader` and `S: Storage`? Does it cause the compiler to do anything different re: optimization?

It has no effects on optimization. It *does* have effects on auto traits and variance. It would be more precise to write:

```
pub struct BetterLoader<T, S> {
   storage: T,
   _phantom: PhantomData<fn(&mut T) -> &mut S>,
}
```

which expresses that the relationship of `T` and `S` is exactly the one that `BorrowMut::borrow_mut` gives us.

However, I think this can only make a difference in the case where *`T` doesn't* own an `S`.

overlookmotel:

> @quinedot Do you mean a blanket implementation?
>
> ```
> impl<S: Storage> Storage for &mut S {
>  fn new() -> Self {
>    // What goes here?
>  }
> }
> ```
>
> Sorry if I'm being dumb, but I can't figure out how this would work.

This can't work and that is precisely an example of why you **should not** put `new()` in traits that are doing other things unless it is fundamentally necessary, as I already mentioned.

1 Like

quinedot March 13, 2023, 6:46pm 6

Like @kpreid said, you'd have to drop or restrict the `new` method.

play.rust-lang.org

Rust Playground

A browser interface to the Rust compiler to experiment with the language

zirconium-n March 14, 2023, 4:28am 7

kpreid:

```
pub struct BetterLoader<T, S> {
   storage: T,
   _phantom: PhantomData<fn(&mut T) -> &mut S>,
}
```

I think this is inaccurate. It should be just `PhantomData<S>` instead. With putting `&mut T` into `fn` args, it will make `BetterLoader` invariant over `T` which I don't think that's what you want.

system Closed June 12, 2023, 4:29am 8

This topic was automatically closed 90 days after the last reply. We invite you to open a new topic if you have further questions or comments.

Related topics

Topic		Replies	Views	Activity
Different impl of trait for T, &T, and &mut T help	13	2698	August 22, 2022	
Lifetime on trait and mutable borrow help	16	767	May 2, 2023	
&mut self borrows nested struct from generic type help	8	192	October 7, 2025	
Confusing reborrow needed for trait implementation. Is there a better way? 	4	535	January 12, 2023	
Generic over trait and Borrow type, PhantomData help	5	1245	January 12, 2023	