rust-lang / **rust**

Tracking issue for specialization (RFC 1210) #31844

New issue

Open **nikomatsakis** opened this issue on 23 Feb 2016 · 130 comments

nikomatsakis commented on 23 Feb 2016 • edited ▾ Contributor

This is a tracking issue for specialization (rust-lang/rfcs#1210).

Major implementation steps:

- [x] Land #30652 =)
- [ ] Restrictions around lifetime dispatch (currently a **soundness hole**)
- [ ] `default impl` (#37653)
- [ ] Integration with associated consts
- [ ] Bounds not always properly enforced (#33017)
- [ ] Should we permit empty impls if parent has no `default` members? #48444
- [ ] implement "always applicable" impls #48538

Unresolved questions from the RFC:

- Should associated type be specializable at all?
- When should projection reveal a `default type`? Never during typeck? Or when monomorphic?
- Should default trait items be considered `default` (i.e. specializable)?
- Should we have `default impl` (where all items are `default`) or `partial impl` (where `default` is opt-in)
- How should we deal with lifetime dispatchability?

👍 19

nikomatsakis assigned **aturon** on 23 Feb 2016

nikomatsakis added **A-traits** **B-RFC-approved** **T-lang** **B-unstable** labels on 23 Feb 2016

nikomatsakis referenced this issue in **rust-lang/rfcs** on 23 Feb 2016

RFC: impl specialization #1210 Merged

aturon commented on 24 Feb 2016 Member

Some additional open questions:

- Should we revisit the orphan rules in the light of specialization? Are there ways to make things more flexible now?
- Should we extend the "chain rule" in the RFC to something more expressive, like the so-called "lattice rule"?
- Related to both of the above, how does negative reasoning fit into the story? Can we recover the negative reasoning we need by a clever enough use of specialization/orphan rules, or should we make it more first-class?

arielb1 commented on 24 Feb 2016 Contributor

I am not sure that specialization changes the orphan rules:

- The "linking" orphan rules must stay the same, because otherwise you would not have safe linking.
- I don't think the "future compatibility" orphan rules should change. Adding a non-specializable impl under you would still be a breaking change.

Worse than that, the "future compatibility" orphan rules keep cross-crate specialization under pretty heavy control. Without them, default-impls leaving their methods open becomes much worse.

Assignees

aturon

Labels

A-traits
B-RFC-approved
B-RFC-implemented
B-unstable
C-tracking-issue
T-lang

Projects

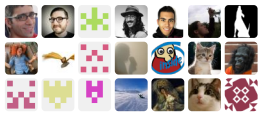
None yet

Milestone

No milestone

Notifications

44 participants



and others

I never liked explicit negative reasoning. I think the total negative reasoning specialization provides is a nice compromise.

SSheldon referenced this issue in **SSheldon/rust-objc** on 5 Mar 2016

Implement Encode for more types #26 Open

sgrif commented on 20 Mar 2016 Contributor

Should this impl be allowed with specialization as implemented? Or am I missing something?
http://is.gd/3Ul0pe

sgrif commented on 20 Mar 2016 Contributor

Same with this one, would have expected it to compile: http://is.gd/RyFIEl

sgrif commented on 20 Mar 2016 Contributor

Looks like there's some quirks in determining overlap when associated types are involved. This compiles: http://is.gd/JBPzIX, while this effectively identical code doesn't: http://is.gd/0ksLPX

SergioBenitez commented on 23 Mar 2016 Contributor

Here's a piece of code I expected to compile with specialization:

http://is.gd/3BNbfK

```
#![feature(specialization)]

use std::str::FromStr;

struct Error;

trait Simple<'a> {
    fn do_something(s: &'a str) -> Result<Self, Error>;
}

impl<'a> Simple<'a> for &'a str {
     fn do_something(s: &'a str) -> Result<Self, Error> {
        Ok(s)
    }
}

impl<'a, T: FromStr> Simple<'a> for T {
    fn do_something(s: &'a str) -> Result<Self, Error> {
        T::from_str(s).map_err(|_| Error)
    }
}

fn main() {
    // Do nothing. Just type check.
}
```

Compilation fails with the compiler citing implementation conflicts. Note that `&str` doesn't implement `FromStr`, so there shouldn't be a conflict.

SergioBenitez added a commit to SergioBenitez/Rocket that referenced this issue on 23 Mar 2016

`Now support Result responses.` ··· ✓ `cddc92f`

aturon commented on 23 Mar 2016 Member

@sgrif

I had time to look at the first two examples. Here are my notes.

Example 1

First case, you have:

- `FromSqlRow<ST, DB> for T where T: FromSql<ST, DB>`
- `FromSqlRow<(ST, SU), DB> for (T, U) where T: FromSqlRow<ST, DB>, U: FromSqlRow<SU, DB>,`

The problem is that these impls overlap but neither is more specific than the other:

- You can potentially have a `T: FromSql<ST, DB>` where `T` is not a pair (so it matches the first impl but not the second).
- You can potentially have a `(T, U)` where:
  - `T: FromSqlRow<ST, DB>` ,
  - `U: FromSqlRow<SU, DB>` , but *not*
  - `(T, U): FromSql<(ST, SU), DB>`
  - (so the second impl matches, but not the first)
- The two impls overlap because you can have a `(T, U)` such that:
  - `T: FromSqlRow<ST, DB>`
  - `U: FromSqlRow<SU, DB>`
  - `(T, U): FromSql<(ST, SU), DB>`

This is the kind of situation that lattice impls would allow -- you'd have to write a third impl for the overlapping case, and say what it should do. Alternatively, negative trait impls might give you a way to rule out overlap or otherwise tweak which matches are possible.

Example 2

You have:

- `Queryable<ST, DB> for T where T: FromSqlRow<ST, DB>`
- `Queryable<Nullable<ST>, DB> for Option<T> where T: Queryable<ST, DB>`

These overlap because you can have `Option<T>` where:

- `T: Queryable<ST, DB>`
- `Option<T>: FromSqlRow<Nullable<ST>, DB>`

But neither impl is more specific:

- You can have a `T` such that `T: FromSqlRow<ST, DB>` but `T` is not an `Option<U>` (matches first impl but not second)
- You can have an `Option<T>` such that `T: Queryable<ST, DB>` but not `Option<T>: FromSqlRow<Nullable<ST>, DB>`

aturon commented on 23 Mar 2016 **Member**

@SergioBenitez

> Compilation fails with the compiler citing implementation conflicts. Note that `&str` doesn't implement `FromStr` , so there shouldn't be a conflict.

The problem is that the compiler is conservatively assuming that `&str` might come to implement `FromStr` in the future. That may seem silly for this example, but in general, we add new impls all the time, and we want to protect downstream code from breaking when we add those impls.

This is a conservative choice, and is something we might want to relax over time. You can get the background here:

- http://smallcultfollowing.com/babysteps/blog/2015/01/14/little-orphan-impls/
- rust-lang/rfcs#1023
- rust-lang/rfcs#1053
- rust-lang/rfcs#1148

sgrif commented on 23 Mar 2016 Contributor

Thank you for clarifying those two cases. It makes complete sense now

On Tue, Mar 22, 2016, 6:34 PM Aaron Turon notifications@github.com wrote:

> **@SergioBenitez** https://github.com/SergioBenitez
>
> Compilation fails with the compiler citing implementation conflicts. Note that &str doesn't implement FromStr, so there shouldn't be a conflict.

> The problem is that the compiler is conservatively assuming that &str might come to implement FromStr in the future. That may seem silly for this example, but in general, we add new impls all the time, and we want to protect downstream code from breaking when we add those impls.
>
> This is a conservative choice, and is something we might want to relax over time. You can get the background here:
>
> -
>
> http://smallcultfollowing.com/babysteps/blog/2015/01/14/little-orphan-impls/
>
> - rust-lang/rfcs#1023 rust-lang/rfcs#1023
> - rust-lang/rfcs#1053 rust-lang/rfcs#1053
> - rust-lang/rfcs#1148 rust-lang/rfcs#1148
>
> —
> You are receiving this because you were mentioned.
> Reply to this email directly or view it on GitHub
> #31844 (comment)

SergioBenitez commented on 23 Mar 2016 Contributor

@aturon

> The problem is that the compiler is conservatively assuming that &str might come to implement FromStr in the future. That may seem silly for this example, but in general, we add new impls all the time, and we want to protect downstream code from breaking when we add those impls.

Isn't this exactly what specialization is trying to address? With specialization, I would expect that even if an implementation of `FromStr` for `&str` were added in the future, the direct implementation of the `Simple` trait for `&str` would take precedence.

sgrif commented on 23 Mar 2016 Contributor

@SergioBenitez you need to put `default fn` in the more general impl. Your example isn't specializable.

On Tue, Mar 22, 2016, 6:54 PM Sergio Benitez notifications@github.com wrote:

> **@aturon** https://github.com/aturon
>
> The problem is that the compiler is conservatively assuming that &str might come to implement FromStr in the future. That may seem silly for this example, but in general, we add new impls all the time, and we want to protect downstream code from breaking when we add those impls.
>
> Isn't this exactly what specialization is trying to address? With specialization, I would expect that even if an implementation of FromStr for &str were added in the future, the direct implementation for the trait for &str would take precedence.
>
> —
> You are receiving this because you were mentioned.
> Reply to this email directly or view it on GitHub
> #31844 (comment)

burdges commented on 1 Apr 2016

I think "default" trait items being automatically considered `default` sounds confusing. You might want both parametricity for a trait like in Haskell, etc. along side with easing the `impl` s. Also you cannot easily `grep` for them like you can for `default` . It's not hard to both type the `default` keyword and give a default implementation, but they cannot be separated as is. Also, if one wants to clarify the language, then these "default" trait items could be renamed to "trait proposed" items in documentation.

Stebalien commented on 15 Apr 2016 • edited Contributor

Note from #32999 (comment): if we do go with the lattice rule (or allow negative constraints), the "use an intermediate trait" trick to prevent further specialization of something will no longer work.

arielb1 commented on 15 Apr 2016 Contributor

@Stebalien

Why won't it work? The trick limits the specialization to a private trait. You can't specialize the private trait if you can't access it.

Stebalien commented on 15 Apr 2016 Contributor

@arielb1 Ah. Good point. In my case, the trait isn't private.

arielb1 commented on 15 Apr 2016 • edited Contributor

I don't think the "externals can't specialize because orphan forward-compatibility + coherence rulea" reasoning is particularly interesting or useful. Especially when we don't commit to our specific coherence rules.

LukasKalbertodt referenced this issue in **rust-lang/rfcs** on 25 Apr 2016

PartialEq between reference and non-reference type? #1332 Open

burdges referenced this issue in **rust-lang/rfcs** on 7 May 2016

Design By Contract #1077 Open

burdges commented on 7 May 2016

Is there a way to access an overridden `default impl` ? If so, this could aid in constructing tests. See Design By Contract and libhoare.

rphmeier commented on 7 May 2016 • edited Contributor

Allowing projection of default associated types during type-checking will allow enforcing type inequality at compile-time: https://gist.github.com/7c081574958d22f89d434a97b626b1e4

```
#![feature(specialization)]

pub trait NotSame {}

pub struct True;
pub struct False;

pub trait Sameness {
    type Same;
}

mod internal {
    pub trait PrivSameness {
        type Same;
    }
}

use internal::PrivSameness;

impl<A, B> Sameness for (A, B) {
    type Same = <Self as PrivSameness>::Same;
}

impl<A, B> PrivSameness for (A, B) {
    default type Same = False;
}
impl<A> PrivSameness for (A, A) {
    type Same = True;
}
```

```
impl<A, B> NotSame for (A, B) where (A, B): Sameness<Same=False> {}

fn not_same<A, B>() where (A, B): NotSame {}

fn main() {
    // would compile
    not_same::<i32, f32>();

    // would not compile
    // not_same::<i32, i32>();
}
```

edited per **@burdges**' comment

burdges commented on 7 May 2016

Just fyi **@rphmeier** one should probably avoid is.gd because it does not resolve for Tor users due to using CloudFlare. GitHub works fine with full URLs. And play.rust-lang.org works fine over Tor.

SimonSapin commented on 7 May 2016 Contributor

@burdges FWIW play.rust-lang.org itself uses is.gd for its "Shorten" button.

It can probably be changed, though: https://github.com/rust-lang/rust-playpen/blob/9777ef59b/static/web.js#L333

zitsen commented on 17 May 2016

use like this(https://is.gd/Ux6FNs):

```
#![feature(specialization)]
pub trait Foo {}
pub trait Bar: Foo {}
pub trait Baz: Foo {}

pub trait Trait {
    type Item;
}

struct Staff<T> { }

impl<T: Foo> Trait for Staff<T> {
    default type Item = i32;
}

impl<T: Foo + Bar> Trait for Staff<T> {
    type Item = i64;
}

impl<T: Foo + Baz> Trait for Staff<T> {
    type Item = f64;
}

fn main() {
    let _ = Staff { };
}
```

Error :

```
error: conflicting implementations of trait `Trait` for type `Staff<_>`: [--explain
E0119]
  --> <anon>:20:1
20 |> impl<T: Foo + Baz> Trait for Staff<T> {
   |> ^
note: conflicting implementation is here:
  --> <anon>:16:1
16 |> impl<T: Foo + Bar> Trait for Staff<T> {
   |> ^

error: aborting due to previous error
```

Does feture `specialization` support this, and is there any other kind of implementations currently?

aturon commented on 17 May 2016 **Member**

@zitsen

These impls are not allowed by the current specialization design, because neither `T: Foo + Bar` nor `T: Foo + Baz` is more specialized than the other. That is, if you have some `T: Foo + Bar + Baz`, it's not clear which impl should "win".

We have some thoughts on a more expressive system that would allow you to *also* give an impl for `T: Foo + Bar + Baz` and thus disambiguate, but that hasn't been fully proposed yet.

👍 1

rphmeier commented on 17 May 2016 Contributor

If negative trait bounds `trait Baz: !Bar` ever land, that could also be used with specialization to prove that the sets of types that implement Bar and those that implement Baz are distinct and individually specializable.

👍 2 ❤️ 2

zitsen commented on 18 May 2016 • edited

~~Seems @**rphmeier**'s reply is what I exactly want, impls for `T: Foo + Bar + Baz` would also help.~~

Just ignore this, I still have something to do with my case, and always exciting for the `specialization` and other features landing.

Thanks **@aturon** **@rphmeier** .

tinco referenced this issue on 23 May 2016

Rustc asks for Reflect, which is unstable, when it could be asking for Any which is stable #33807 Closed

kylewlacy commented on 24 May 2016 • edited

I've been playing around with specialization lately, and I came across this weird case:

```
#![feature(specialization)]

trait Marker {
    type Mark;
}

trait Foo { fn foo(&self); }

struct Fizz;

impl Marker for Fizz {
    type Mark = ();
}

impl Foo for Fizz {
    fn foo(&self) { println!("Fizz!"); }
}

impl<T> Foo for T
    where T: Marker, T::Mark: Foo
{
    default fn foo(&self) { println!("Has Foo marker!"); }
}

struct Buzz;

impl Marker for Buzz {
    type Mark = Fizz;
}

fn main() {
    Fizz.foo();
    Buzz.foo();
}
```

Compiler output:

```
error: conflicting implementations of trait `Foo` for type `Fizz`: [--explain E0119]
  --> <anon>:19:1
19 |> impl<T> Foo for T
   |> ^
note: conflicting implementation is here:
  --> <anon>:15:1
15 |> impl Foo for Fizz {
   |> ^
```

playpen

I believe that the above *should* compile, and there's two interesting variations that actually do work-as-intended:

1. Removing the `where T::Mark: Fizz` bound:

```
impl<T> Foo for T
    where T: Marker //, T::Mark: Fizz
{
    // ...
}
```

playpen

2. Adding a "trait bound alias":

```
trait FooMarker { }
impl<T> FooMarker for T where T: Marker, T::Mark: Foo { }

impl<T> Foo for T where T: FooMarker {
    // ...
}
```

playpen

(Which *doesn't* work if `Marker` is defined in a separate crate (!), see this example repo)

I also believe that this issue might be related to #20400 somehow

EDIT: I've opened an issue about this: #36587

jan-hudec referenced this issue in **vitiral/strfmt** on 16 Jun 2016

Duplicate effort? #3 Closed

tomaka commented on 2 Jul 2016 • edited Contributor

I'm encountering an issue with specialization. Not sure if it's an implementation problem or a problem in the way specialization is specified.

```
use std::vec::IntoIter as VecIntoIter;

pub trait ClonableIterator: Iterator {
    type ClonableIter;

    fn clonable(self) -> Self::ClonableIter;
}

impl<T> ClonableIterator for T where T: Iterator {
    default type ClonableIter = VecIntoIter<T::Item>;

    default fn clonable(self) -> VecIntoIter<T::Item> {
        self.collect::<Vec<_>>().into_iter()
    }
}

impl<T> ClonableIterator for T where T: Iterator + Clone {
    type ClonableIter = T;

    #[inline]
    fn clonable(self) -> T {
        self
    }
}
```

(playpen)
(by the way, it would be nice if this code eventually landed in the stdlib one day)

This code fails with:

```
error: method `clonable` has an incompatible type for trait:
 expected associated type,
    found struct `std::vec::IntoIter` [--explain E0053]
  --> <anon>:14:5
   |>
14 |>     default fn clonable(self) -> VecIntoIter<T::Item> {
   |>     ^
```

Changing the return value to `Self::ClonableIter` gives the following error:

```
error: mismatched types [--explain E0308]
  --> <anon>:15:9
   |>
15 |>         self.collect::<Vec<_>>().into_iter()
   |>         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected associated type, found
struct `std::vec::IntoIter`
note: expected type `<T as ClonableIterator>::ClonableIter`
note:    found type `std::vec::IntoIter<<T as std::iter::Iterator>::Item>`
```

Apparently you can't refer to the concrete type of a defaulted associated type, which I find quite limiting.

Aatch commented on 6 Jul 2016 Contributor

@tomaka it should work, the RFC text has this:

```
impl<T> Example for T {
    default type Output = Box<T>;
    default fn generate(self) -> Box<T> { Box::new(self) }
}

impl Example for bool {
    type Output = bool;
    fn generate(self) -> bool { self }
}
```

(https://github.com/rust-lang/rfcs/blob/master/text/1210-impl-specialization.md#the-default-keyword)

Which seems similar enough to your case to be relevant.

rphmeier commented on 6 Jul 2016 Contributor

@Aatch that example doesn't seem to compile with the intuitive definition for the example trait:
https://play.rust-lang.org/?gist=97ff3c2f7f3e50bd3aef000dbfa2ca4e&version=nightly&backtrace=0

the specialization code explicitly disallows this -- see #33481, which I initially thought was an error but turned out to be a diagnostics issue. My PRs to improve the diagnostics here went unnoticed, and I haven't maintained them to the latest master for quite some time.

Aatch commented on 7 Jul 2016 Contributor

@rphmeier the RFC text suggests that it should be allowed though, that example is copied from it.

KodrAus referenced this issue in **elastic-rs/elastic** on 17 Jul 2016
More Aggresive Autoderiving of Traits #137 Closed

tupshin referenced this issue in **rust-lang-nursery/rustfmt** on 1 Aug 2016
rustfmt incorrectly strips the "default" keyword from speciallized functions #1112 Closed

hauleth referenced this issue in **rust-num/num** on 8 Aug 2016
Add `is_one` method to `One` trait. #215 Closed

nrc added the **B-RFC-implemented** label on 29 Aug 2016

dtolnay referenced this issue in **serde-rs/serde** on 30 Aug 2016
Implementing custom type serializing #529 Closed

nrc commented on 18 Sep 2016 Member

I had a play with some code that could benefit from specialization. I strongly think we should go for the lattice rule rather chaining - it feels natural and was the only way to get the flexibility I needed (afaict).

If we went for `default` on the `impl` as well as individual items, could we enforce that if any item is overridden then they all must be? That would allow us to reason based on the precise type of a default assoc type (for example) in the other items, which seems like a useful boost in expressivity.

bluss commented on 18 Sep 2016 • edited Contributor

Should the following be allowed? I want to specialize a type so that ArrayVec is `Copy` when its element type is Copy, and that it otherwise has a destructor. I'm trying to accomplish it by using an internal field that is replaced by specialization.

I hoped this would compile, i.e that it deduces the copyability of `ArrayVec<A>` 's fields from the field types that are selected by the `A: Copy + Array` bound (compilable snippet on playground).

```
impl<A: Copy + Array> Copy for ArrayVec<A>
    //where <A as Repr>::Data: Copy
{ }
```

The commented-out where clause is not wanted because it exposes a private type `Repr` in the public interface. (It also ICEs anyway).

Edit: I had forgotten I reported issue #33162 about this already, I'm sorry.

nrc commented on 18 Sep 2016 **Member**

Follow up on my comment, my actual use case:

```
// Ideal version

trait Scannable {}

impl<T: FromStr> Scannable for T {}
impl<T: FromStr> Scannable for Result<T, ()> {}

// But this doesn't follow from the specialisation rules because Result: !FromStr
// Lattice rule would allow filling in that gap or negative reasoning would allow
specifying it.

// Second attempt

trait FromResult {
    type Ok;
    fn from(r: Result<Self::Ok, ()>) -> Self;
}

impl<T> Scannable for T {
    default type Ok = T;
    default fn from(r: Result<T, ()>) -> Self {...} // error can't assume Ok == T, could
do this if we had `default impl`
}

impl<T> Scannable for Result<T, ()> {
    type Ok = T;
    default fn from(r: Result<T, ()>) -> Self { r }
}

fn scan_from_str<T: FromResult>(x: &str) -> T
    where <T as FromResult>::Ok: FromStr  // Doesn't hold for T: FromStr because of the
default on T::Ok
{ ... }

// Can also add the FromStr bound to FromResult::Ok, but doesn't help

// Third attempt
trait FromResult<Ok> {
    fn from(r: Result<Ok, ()>) -> Self;
}

impl<T> FromResult<T> for T {
    default fn from(r: Result<Self, ()>) -> Self { ... }
}

impl<T> FromResult<T> for Result<T, ()> {
    fn from(r: Result<T, ())>) -> Self { r }
}


fn scan_from_str<U: FromStr, T: FromResult<U>>(x: &str) -> T { ... }

// Error because we can't infer that U == String
let mut x: Result<String, ()> = scan_from_str("dsfsf");
```

kylewlacy referenced this issue on 19 Sep 2016

Specialization doesn't work when an impl has an associated type bound Closed
#36587

eddyb referenced this issue on 20 Sep 2016

Add traits w/ auto-deriving for soundly serializing/inspecting/transforming rustc types. #36588 Open

giannicic referenced this issue on 20 Sep 2016

specialisation error 502 is misleading #36553 Closed

nikomatsakis commented on 24 Sep 2016 • edited Contributor

@tomaka @Aatch

The problem there is that you are not allowed to rely on the value of other default items. So when you have this impl:

```
impl<T> ClonableIterator for T where T: Iterator {
    default type ClonableIter = VecIntoIter<T::Item>;

    default fn clonable(self) -> VecIntoIter<T::Item> {
    //                           ^^^^^^^^^^^^^^^^^^^^
        self.collect::<Vec<_>>().into_iter()
    }
}
```

At the spot where I highlighted, `clonable` is relying on `Self::ClonableIter`, but because `CloneableIter` is declared as default, you can't do that. The concern is that someone might specialize and override `CloneableIter` but *not* `clonable`.

We had talked about some possible answers here. One of them was to let you use `default` to group together items where, if you override one, you must override all:

```
impl<T> ClonableIterator for T where T: Iterator {
    default {
        type ClonableIter = VecIntoIter<T::Item>;
        fn clonable(self) -> VecIntoIter<T::Item> { ... }
    }
}
```

This is ok, but a bit "rightward-drift inducing". The `default` also looks like a naming scope, which it is not. There might be some simpler variant that just lets you toggle between "override-any" (as today) vs "override-all" (what you need).

We had also hoped we could get by by leveraging `impl Trait`. The idea would be that this most often comes up, as is the case here, when you want to customize the return type of methods. So perhaps if you could rewrite the trait to use `impl Trait`:

```
pub trait ClonableIterator: Iterator {
    fn clonable(self) -> impl Iterator;
}
```

This would effectively be a kind of shorthand when implemented for a default group containing the type and the fn. (I'm not sure if there'd be a way to do that purely in the impl though.)

PS, sorry for the long delay in answering your messages, which I see date from *July*.

sgrif commented on 28 Sep 2016 Contributor

While impl Trait does help, there is no RFC that has been accepted or implemented which allows it to be used with trait bodies in any form, so looking to it for this RFC feels a bit odd.

AtheMathmo referenced this issue in **AtheMathmo/rulinalg** on 16 Oct 2016

BaseMatrix and BaseMatrixMut operations should not require same type #70 Closed

lfairy referenced this issue in **lfairy/maud** on 18 Oct 2016

Stable support #17 Open

giannicic commented on 4 Nov 2016 Contributor

I'm interested in implementing the `default impl` feature (where all items are `default`).
Would you accept a contribution on that?

aturon commented on 5 Nov 2016 Member

@giannicic Definitely! I'd be happy to help mentor the work as well.

👍 1

marcianx commented on 7 Nov 2016 • edited

Is there currently a conclusion on whether associated types should be specializable?

The following is a simplification of my use-case, demonstrating a need for specializable associated types.
I have a generic data structure, say `Foo`, which coordinates a collection of container trait objects (`&trait::Property`). The trait `trait::Property` is implemented by both `Property<T>` (backed by `Vec<T>`) and `PropertyBits` (backed by `BitVec`, a bit vector).
In generic methods on `Foo`, I would like to be able to determine the right underlying data structure for `T` via associated types, but this requires specialization to have a blanket impl for non-special cases as follows.

```
trait ContainerFor {
    type P: trait::Property;
}

impl<T> ContainerFor for T {
    default type P = Property<T>; // default to the `Vec`-based version
}

impl ContainerFor for bool {
    type P = PropertyBits; // specialize to optimize for space
}

impl Foo {
    fn add<T>(&mut self, name: &str) {
        self.add_trait_obj(name, Box::new(<T as ContainerFor>::P::new()));
    }
    fn get<T>(&mut self, name: &str) -> Option<&<T as ContainerFor>::P> {
        self.get_trait_obj(name).and_then(|prop| prop.downcast::<_>());
    }
}
```

🎉 2

giannicic commented on 7 Nov 2016 Contributor

Thanks **@aturon** !
Basically I'm doing the work by adding a new "defaultness" attribute to the `ast::ItemKind::Impl` struct (and then use the new attribute together with the impl item "defaultness" attribute) but there is also a quick and easy
possibility consisting on setting default to all the impl items of the `default impl` during parsing.
To me this isn't a "complete" solution since we lost the information that the "defaultness" is related to the impl and not to each item of the impl,
additionally if there is a plan to introduce a `partial impl` the first solution would already provide an attribute that can be used to store `default` as well as `partial`. But just to be sure and
not wasting time, what do you think about?

nikomatsakis commented on 8 Nov 2016 Contributor

@giannicic **@aturon** may I propose we create a specific issue to discuss `default impl` ?

nikomatsakis referenced this issue on 8 Nov 2016
support `default impl` for specialization #37653 Open
1 of 3 tasks complete

nikomatsakis commented on 8 Nov 2016 Contributor

Never mind, I created one: #37653

AndyBarron referenced this issue in **anima-engine/mrusty** on 12 Nov 2016

Don't match macros on exact type names #96 Open

gnzlbg commented on 14 Nov 2016 • edited Contributor

Would the lattice rule allow me to, given:

```
trait Foo {}

trait A {}
trait B {}
trait C {}
// ...
```

add implementations of `Foo` for subset of types that implement some combination of `A`, `B`, `C`, ...:

```
impl Foo for T where T: A { ... }
impl Foo for T where T: B { ... }
impl Foo for T where T: A + B { ... }
impl Foo for T where T: B + C { ... }
// ...
```

and allow me to "forbid" some combinations, e.g., that `A + C` should never happen:

```
impl Foo for T where T: A + C = delete;
```

?

Context: I landed into wanting this when implementing an `ApproxEqual(Shape, Shape)` trait for different kinds of shapes (points, cubes, polygons, ...) where these are all traits. I had to work around this by refactoring this into different traits, e.g., `ApproxEqualPoint(Point, Point)`, to avoid conflicting implementations.

nikomatsakis commented on 14 Nov 2016 Contributor

@gnzlbg

> and allow me to "forbid" some combinations, e.g., that A + C should never happen:

No, this is not something that the lattice rule would permit. That would be more the domain of "negative reasoning" in some shape or kind.

> Context: I landed into wanting this when implementing an ApproxEqual(Shape, Shape) trait for different kinds of shapes (points, cubes, polygons, ...) where these are all traits. I had to work around this by refactoring this into different traits, e.g., ApproxEqualPoint(Point, Point), to avoid conflicting implementations.

So **@withoutboats** has been promoting the idea of "exclusion groups", where you can declare that a certain set of traits are mutually exclusive (i.e., you can implement at most one of them). I envision this as kind of being like an enum (i.e., the traits are all declared together). I like the idea of this, particularly as (I think!) it helps to avoid some of the more pernicious aspects of negative reasoning. But I feel like more thought is needed on this front -- and also a good writeup that tries to summarize all the "data" floating around about how to think about negative reasoning. Perhaps now that I've (mostly) wrapped up my HKT and specialization series I can think about that...

gnzlbg commented on 14 Nov 2016 Contributor

@nikomatsakis :

> So **@withoutboats** has been promoting the idea of "exclusion groups", where you can declare that a certain set of traits are mutually exclusive (i.e., you can implement at most one of them). I envision this as kind of being like an enum (i.e., the traits are all declared together). I like the idea of this, particularly as (I think!) it helps to avoid some of the more pernicious aspects of negative reasoning. But I feel like more thought is needed on this front -- and also a good writeup that tries to summarize all the "data" floating around about how to think about negative reasoning. Perhaps now that I've (mostly) wrapped up my HKT and specialization series I can think about that...

I thought about exclusions groups while writing this (you mentioned it in the forums the other day), but I don't think they can work since in this particular example not all traits implementations are exclusive. The most trivial example is the `Point` and `Float` traits: a `Float` *can* be a 1D point, so `ApproxEqualPoint(Point, Point)` and `ApproxEqualFloat(Float, Float)` cannot be exclusive. There are other examples like `Square` and `Polygon`, or `Box` | `Cube` and `AABB` (axis-aligned bounding box) where the "trait hierarchy" actually needs more complex constraints.

> No, this is not something that the lattice rule would permit. That would be more the domain of "negative reasoning" in some shape or kind.

I would at least be able to implement the particular case and put an `unimplemented!()` in it. That would be enough, but obviously I would like it more if the compiler would statically catch those cases in which I call a function with an `unimplemented!()` in it (and at this point, we are again in negative reasoning land).

withoutboats commented on 14 Nov 2016 • edited Contributor

@gnzlbg lattice specialization would allow you to make that impl panic, but the idea of doing that makes me 😢.

The idea of "exclusion groups" is really just negative supertrait bounds. One thing we haven't explored too thoroughly is the notion of reverse polarity specialization - allowing you to write a specialized impl that is of reversed polarity to its less specialized impl. For example, in this case you would just write:

```
impl<T> !Foo for T where T: A + C { }
```

I'm not fully sure what the implications of allowing that are. I think it connects to the issues Niko's already highlighted about how specialization is sort of conflating code reuse with polymorphism right now.

glaebhoerl commented on 15 Nov 2016 Contributor

With all this discussion of negative reasoning and negative impls, I feel compelled to bring up the old Haskell idea of "instance chains" again (paper, paper, GHC issue tracker, Rust pre-RFC), as a potential source of inspiration if nothing else.

Essentially the idea is that anywhere you can write a `trait` impl, you can also write any number of "else if clauses" specifying a different `impl` that should apply in case the previous one(s) did not, with an optional final "else clause" specifying a negative impl (that is, if none of the clauses for `Trait` apply, then `!Trait` applies).

gnzlbg commented on 15 Nov 2016 • edited Contributor

@withoutboats

> The idea of "exclusion groups" is really just negative supertrait bounds.

I think that would be enough for my use cases.

> I think it connects to the issues Niko's already highlighted about how specialization is sort of conflating code reuse with polymorphism right now.

I don't know if these can be untangled. I want to have:

- polymorphism: a single trait that abstracts different implementations of an operation for lots of different types,
- code reuse: instead of implementing the operation for each type, I want to implement them for groups of types that implement some traits,
- performance: be able to override an already existing implementation for a particular type or a subset of types that has a more specific set of constraints that the already existing implementations,
- productivity: be able to write and test my program incrementally, instead of having to add a lots of

`impl`s for it to compile.

Covering all cases is hard, but if the compiler forces me to cover all cases:

```
trait Foo {}
trait A {}
trait B {}

impl<T> Foo for T where T: A { ... }
impl<T> Foo for T where T: B { ... }
// impl<T> Foo for T where T: A + B { ... }  //< compiler: need to add this impl!
```

and also gives me negative impls:

```
impl<T> !Foo for T where T: A + B { }
impl<T> !Foo for T where T: _ { } // _ => all cases not explicitly covered yet
```

I would be able to incrementally add impls as I need them and also get nice compiler errors when I try to use a trait with a type for which there is no impl.

> I'm not fully sure what the implications of allowing that are.

Niko mentioned that there are problems with negative reasoning. FWIW the only thing negative reasoning is used for in the example above is to state that the user knows that an impl for a particular case is required, but has explicitly decided not to provide an implementation for it.

dtolnay commented on 28 Nov 2016 **Member**

I just hit #33017 and don't see it linked here yet. It is marked as a soundness hole so it would be good to track here.

dtolnay commented on 28 Nov 2016 • edited **Member**

For dtolnay/quote#7 I need something similar to this example from the RFC which doesn't work yet. cc **@tomaka** **@Aatch** **@rphmeier** who commented about this earlier.

```
trait Example {
    type Output;
    fn generate(self) -> Self::Output;
}

impl<T> Example for T {
    default type Output = Box<T>;
    default fn generate(self) -> Box<T> { Box::new(self) }
}

impl Example for bool {
    type Output = bool;
    fn generate(self) -> bool { self }
}
```

I stumbled upon the following workaround which gives a way to express the same thing.

```
#![feature(specialization)]

use std::fmt::{self, Debug};

////////////////////////////////////////////////////////////////////////////////

trait Example: Output {
    fn generate(self) -> Self::Output;
}

/// In its own trait for reasons, presumably.
trait Output {
    type Output: Debug + Valid<Self>;
}

fn main() {
    // true
    println!("{:?}", Example::generate(true));
```

```
    // box("s")
    println!("{:?}", Example::generate("s"));
}

////////////////////////////////////////////////////////////////////////////

/// Instead of `Box<T>` just so the "{:?}" in main() clearly shows the type.
struct MyBox<T: ?Sized>(Box<T>);

impl<T: ?Sized> Debug for MyBox<T>
    where T: Debug
{
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "box({:?})", self.0)
    }
}

////////////////////////////////////////////////////////////////////////////

/// Return type of the impl containing `default fn`.
type DefaultOutput<T> = MyBox<T>;

impl Output for bool {
    type Output = bool;
}

impl<T> Example for T where T: Pass {
    default fn generate(self) -> Self::Output {
        T::pass({
            // This is the impl you wish you could write
            MyBox(Box::new(self))
        })
    }
}

impl Example for bool {
    fn generate(self) -> Self::Output {
        self
    }
}

////////////////////////////////////////////////////////////////////////////
// Magic? Soundness exploit? Who knows?

impl<T: ?Sized> Output for T where T: Debug {
    default type Output = DefaultOutput<T>;
}

trait Valid<T: ?Sized> {
    fn valid(DefaultOutput<T>) -> Self;
}

impl<T: ?Sized> Valid<T> for DefaultOutput<T> {
    fn valid(ret: DefaultOutput<T>) -> Self {
        ret
    }
}

impl<T> Valid<T> for T {
    fn valid(_: DefaultOutput<T>) -> Self {
        unreachable!()
    }
}

trait Pass: Debug {
    fn pass(DefaultOutput<Self>) -> <Self as Output>::Output;
}

impl<T: ?Sized> Pass for T where T: Debug, <T as Output>::Output: Valid<T> {
    fn pass(ret: DefaultOutput<T>) -> <T as Output>::Output {
        <T as Output>::Output::valid(ret)
    }
}
```

❤️ 2

dtolnay commented on 29 Nov 2016 **Member**

I am still working on dtolnay/quote#7 and needed a diamond pattern. Here is my solution. cc **@zitsen** who asked about this earlier and **@aturon** and **@rphmeier** who responded.

```
#![feature(specialization)]
```

```
/// Can't have these impls directly:
///
///  - impl<T> Trait for T
///  - impl<T> Trait for T where T: Clone
///  - impl<T> Trait for T where T: Default
///  - impl<T> Trait for T where T: Clone + Default
trait Trait {
    fn print(&self);
}

fn main() {
    struct A;
    A.print(); // "neither"

    #[derive(Clone)]
    struct B;
    B.print(); // "clone"

    #[derive(Default)]
    struct C;
    C.print(); // "default"

    #[derive(Clone, Default)]
    struct D;
    D.print(); // "clone + default"
}

trait IfClone: Clone { fn if_clone(&self); }
trait IfNotClone { fn if_not_clone(&self); }

impl<T> Trait for T {
    default fn print(&self) {
        self.if_not_clone();
    }
}

impl<T> Trait for T where T: Clone {
    fn print(&self) {
        self.if_clone();
    }
}

impl<T> IfClone for T where T: Clone {
    default fn if_clone(&self) {
        self.clone();
        println!("clone");
    }
}

impl<T> IfClone for T where T: Clone + Default {
    fn if_clone(&self) {
        self.clone();
        Self::default();
        println!("clone + default");
    }
}

impl<T> IfNotClone for T {
    default fn if_not_clone(&self) {
        println!("neither");
    }
}

impl<T> IfNotClone for T where T: Default {
    fn if_not_clone(&self) {
        Self::default();
        println!("default");
    }
}
```

dtolnay referenced this issue in **dtolnay/quote** on 2 Dec 2016

Support use of non-repeating tokens in repeating blocks #7 Open

ipetkov commented on 4 Dec 2016 Contributor

Hit a bug (or at least unexpected behavior from my perspective) with specialization and type inference: #38167

Popog referenced this issue in **bluss/indexmap** on 7 Dec 2016

Move mutable key access to a trait #7 Closed

sgrif commented on 10 Dec 2016 • edited Contributor

These two impls should be expected to be valid with specialization, right? It seems to not be successfully picking it up.

```
impl<T, ST, DB> ToSql<Nullable<ST>, DB> for T where
    T: ToSql<ST, DB>,
    DB: Backend + HasSqlType<ST>,
    ST: NotNull,
{
    ...
}

impl<T, ST, DB> ToSql<Nullable<ST>, DB> for Option<T> where
    T: ToSql<ST, DB>,
    DB: Backend + HasSqlType<ST>,
    ST: NotNull,
{
    ...
}
```

$R_\emptyset$ **rotty** referenced this issue in **erickt/rust-zmq** on 10 Dec 2016

Wrap zmq_send_const? #119 Open

dtolnay commented on 21 Dec 2016 Member

I filed #38516 for some unexpected behavior I ran into while working on building specialization into Serde. Similar to #38167, this is a case where the program compiles without the specialized impl and when it is added there is a type error. cc **@bluss** who was concerned about this situation earlier.

withoutboats commented on 27 Dec 2016 • edited Contributor

What if we allowed specialization without the `default` keyword within a single crate, similar to how we allow negative reasoning within a single crate?

My main justification is this: "the iterators and vectors pattern." Sometimes, users want to implement something for all iterators and for vectors:

```
impl<I> Foo for I where I: Iterator<Item = u32> { ... }
impl Foo for Vec<u32> { ... }
```

(This is relevant to other situations than iterators and vectors, of course, this is just one example.)

Today this doesn't compile, and there is tsuris and gnashing of teeth. Specialization solves this problem:

```
default impl<I> Foo for I where I: Iterator<Item = u32> { ... }
impl Foo for Vec<u32> { ... }
```

But in solving this problem, you have added a public contract to your crate: it is possible to overide the iterator impl of `Foo`. Maybe we don't want to force you to do that - hence, local specialization without `default`.

The question I suppose is, what exactly is the role of `default`. Requiring `default` was, I think, originally a gesture toward explicitness and self-documenting code. Just as Rust code is immutable by default, private by default, safe by default, it should also be final by default. However, because "non-finality" is a global property, *I* cannot specialize an item unless I let *you* specialize an item.

👍 5

spinda referenced this issue on 27 Dec 2016

Specialization doesn't trigger for these cases #38642 Closed

canndrew commented on 28 Dec 2016

Contributor

> Requiring `default` was, I think, originally a gesture toward explicitness and self-documenting code. However [..] I cannot specialize an item unless I let you specialize an item.

Is that really so bad though? If you want to specialize an impl then maybe other people want to aswell.

I worry because just thinking about this RFC is already giving me PTSD flashbacks of working in C++ codebases which use obscene amounts of overloading and inheritance and having no idea wtf is going on in any line of code which has a method call in it. I really appreciate the lengths that **@aturon** has gone to to make specialization explicit and self-documenting.

👍 4

Wyverald commented on 31 Dec 2016

> Is that really so bad though? If you want to specialize an impl then maybe other people want to aswell.

If other people only "maybe" want to specialize it too, and if there are good cases where we wouldn't want them to, we shouldn't make it *impossible* to specify this. (a bit similar to encapsulation: you want to access some data and maybe some other people want to as well -- so you explicitly mark *this data* public, instead of defaulting all data to be public.)

> I worry because just thinking about this RFC is already giving me PTSD flashbacks ...

But how would disallowing this specification prevent these things from happening?

canndrew commented on 31 Dec 2016

Contributor

> if there are good cases where we wouldn't want them to, we shouldn't make it impossible to specify this.

It's not necessarily a good idea to give users a power whenever they might have a good usecase for it. Not if it also enables users to write confusing code.

> But how would disallowing this specification prevent these things from happening?

Say you see `foo.bar()` and you want to look at what `bar()` does. Right now, if you find the method implemented on a matching type and it's not marked `default` you *know* that its the method definition you're looking for. With **@withoutboats**' proposal this will no longer be true - instead you'll never know for sure whether you're actually looking at the code which is getting executed.

withoutboats commented on 31 Dec 2016 • edited

Contributor

> instead you'll never know for sure whether you're actually looking at the code which is getting executed.

This is quite an exaggeration of the effect of allowing specialization of non-default impls for local types. If you are looking at a concrete impl, you know you are looking at the correct impl. And you have access to the entire source of this crate; you can determine if this impl is specialized or not significantly sooner than "never."

Meanwhile, even with `default`, the problem remains when an impl has not been finalized. If the correct impl is actually a `default` impl, you are in the same situation of having difficulty being unsure if this is the correct impl. And of course if specialization is employed, this will quite commonly be the case (for example, this is the case today for nearly every impl of `ToString`).

In fact I do think this is a rather serious problem, but I'm not convinced that `default` solves it. What we need are better code navigation tools. Currently rustdoc makes a very much 'best effort' approach when it comes to trait impls - it doesn't link to their source and it doesn't even list impls that are provided by blanket impls.

I'm not saying this change is a slamdunk by any means, but I think its worth a more nuanced consideration.

👍 1

Wyverald commented on 31 Dec 2016

> It's not necessarily a good idea to give users a power whenever they might have a good usecase for it. Not if it also enables users to write confusing code.

Exactly, I absolutely agree. I think I'm talking about a different "user" here, which is the user of crates you write. You don't want them to freely specialize traits in your crate (possibly affecting the behavior of your crate in a hacky way). On the other hand, we'd be giving more power the "user" you're talking about, namely the crate author, but even without **@withoutboats**' proposal, you'd have to use "default" and run into the same problem.

burdges commented on 31 Dec 2016

I think `default` helps in the sense that if you want to simplify reading a code then you can ask that nobody use `default` or establish rigorous documentation rules for using it. At that point, you need only worry about the `default` s from `std` , which presumably folks would better understand.

I recall the idea that documentation rules could be imposed on usages of specialization contributed to getting the specialization RFC approved.

nrc commented on 4 Jan 2017 Member

@withoutboats am I correct in reading your motivation for loosening of `default` as you want a restricted form of `default` which means "overridable, but only in this crate" (i.e., `pub(crate)` but for `default`)? However, to keep things simple you are proposing changing the semantics of omitting `default` , rather than adding graduations of `default` -ness?

withoutboats commented on 4 Jan 2017 Contributor

Correct. Doing something like `default(crate)` seems like overkill.

burdges commented on 4 Jan 2017

A priori, I'd imagine one could simulate that through what the crate exports though, no? Are there any situations where you could not simply introduce a private helper trait with the `default` methods and call it from your own final `impl` s? You want the user to use your `default` s but not supply any of their own?

nikomatsakis commented on 4 Jan 2017 Contributor

> Correct. Doing something like default(crate) seems like overkill.

I disagree. I really want a restricted form of default. I have been meaning to propose it. My motivation is that sometimes intersection impls etc will force you to add default, but that doesn't mean you want to allow for arbitrary crates to change your behavior. Sorry, have a meeting, I can try to elaborate with an example in a bit.

withoutboats commented on 4 Jan 2017 Contributor

@nikomatsakis I have the same motivation, what I'm proposing is that we just remove the default requirement to specialize in the same crate, as opposed to adding more levers. :-)

burdges commented on 4 Jan 2017

If by chance this non-exported default might be the more common usage, then a `#[default_export]` feature would be easier to remember by analogy with `#[macro_export]` . An intermediate option might be allowing this export feature for `pub use` or `pub mod` lines.

jimmycuadra commented on 4 Jan 2017 Contributor

Using the `pub` keyword would be better, since Macros 2.0 will support macros as normal items and use `pub` instead of `#[macro_use]`. Using `pub` to indicate visibility across the board would be a big win for its consistency.

nikomatsakis commented on 4 Jan 2017 Contributor

@withoutboats regardless, I think sometimes you will **want** to specialize locally but not necessarily open the doors to all

sgrif commented on 4 Jan 2017 Contributor

> Using the pub keyword would be better

Having `pub default fn` mean "publicly export the defaultness of the fn" as opposed to affecting the visibility of the function itself would be super confusing to newcomers.

👍 2

withoutboats commented on 4 Jan 2017 Contributor

@jimmycuadra is that what you meant by using the `pub` keyword? I agree with **@sgrif** that it seems more confusing, and if we're going to allow you to scope defaultness explicitly, the same syntax we decide on for scoping visibility seems like the correct path.

jimmycuadra commented on 5 Jan 2017 Contributor

Probably not `pub default fn` exactly, because that is ambiguous, as you both mention. I was just saying there's value in having `pub` universally mean "expose something otherwise private to the outside." There's probably some formulation of syntax involving `pub` that would be visually different so as not to be confused with making the function itself public.

nrc commented on 5 Jan 2017 Member

Although it is a bit syntaxey, I would not oppose `default(foo)` working like `pub(foo)` - the symmetry between the two marginally outweights the fiddliness of the syntax for me.

👍 4

glaebhoerl commented on 5 Jan 2017 Contributor

Bikeshed warning: have we considered calling it `overridable` instead of `default`? It's more literally descriptive, and `overridable(foo)` reads better to me than `default(foo)` - the latter suggests "this is the default within the scope of `foo`, but something else might be the default elsewhere", while the former says "this is overridable within the scope of `foo`", which is correct.

👍 1

burdges commented on 5 Jan 2017 • edited

I think the first two questions are really : Is exporting or not exporting `default` ness significantly more common? Should *not* exporting `default` ness be the default behavior?

Yes case: You could maximize the similarity with exports elsewhere dictates something like `pub mod mymodule default;` and `pub use mymodule::MyTrait default;`, or maybe with `overridable`. If needed, you could export `default` ness for only some methods with `pub use MyModule::MyTrait::{methoda,methodb} default;`

No case: You need to express privateness, not publicness, which differs considerably from anything else in Rust anyways, so now `default(crate)` becomes the normal way to control these exports.

Also, if exporting and not exporting `default` ness are comparably common, then you guys can probably choose arbitrarily to be in either the yes or no case, so again just picking `pub use MyModule::MyTrait::{methoda,methodb} default;` works fine.

All these notations look compatible anyways. Another option might be some special `impl` that closed off the `default` s, but that sounds complex and strange.

jimmycuadra commented on 5 Jan 2017 Contributor

@burdges Do you have the labels "yes case" and "no case" backwards there, or am I misunderstanding what you're saying?

burdges commented on 5 Jan 2017 • edited

Yup, oops! Fixed!

burdges commented on 7 Jan 2017 • edited

We have `impl<T> Borrow<T> for T where T: ?Sized` so that a `Borrow<T>` bound can treat owned values as if they were borrowed.

I suppose we could use specialization to optimize away calls to `clone` from a `Borrow<T>` , yes?

```
pub trait CloneOrTake<T> {
    fn clone_or_take(self) -> T;
}

impl<B,T> CloneOrTake<T> for B where B: Borrow<T>, T: Clone {
    #[inline]
    default fn clone_or_take(b: B) -> T { b.clone() }
}
impl<T> CloneOrTake<T> for T {
    #[inline]
    fn clone_or_take(b: T) -> T { b };
}
```

I'd think this might make `Borrow<T>` usable in more situations. I dropped the `T: ?Sized` bound because one presumably needs `Sized` when returning `T` .

Another approach might be

```
pub trait ToOwnedFinal : ToOwned {
    fn to_owned_final(self) -> Self::Owned;
}

impl<B> ToOwnedFinal for B where B: ToOwned {
    #[inline]
    default fn to_owned_final(b: B) -> Self::Owned { b.to_owned() }
}
impl<T> ToOwnedFinal for T {
    #[inline]
    fn to_owned_final(b: T) -> T { b };
}
```


nipunn1313 referenced this issue in **servo/rust-smallvec** on 26 Jan 2017

Implement extend_from_slice and insert_from_slice with memmove optimization #29 Merged

withoutboats commented on 28 Jan 2017 Contributor

We've made some possibly troubling discoveries today, you can read the IRC logs here: https://botbot.me/mozilla/rust-lang/

I'm not 100% confident about all of the conclusions we reached, especially since Niko's comments after the fact seem uplifting. For a little while it seemed a bit apocalyptic to me.

One thing I do feel fairly sure about is that requiring the `default` cannot be made compatible with a guarantee that adding new `default` impls is always backward compatible. Here's the demonstration:

crate `parent` v `1.0.0`

```
trait A { }
trait B { }
trait C {
    fn foo(&self);
}

impl<T> C for T where T: B {
    // No default, not specializable!
    fn foo(&self) { panic!() }
}
```

crate `client` (depends on `parent`)

```
extern crate parent;

struct Local;

impl parent::A for Local { }
impl parent::C for Local {
    fn foo(&self) { }
}
```

Local implements `A` and `C` but not `B`. If local implemented `B`, its impl of `C` would conflict with the non-specializable blanket impl of `C for T where T: B`.

crate `parent` v `1.1.0`

```
// Same code as before, but add:
default impl<T> B for T where T: A { }
```

This impl has been added, and is a completely specializable impl, so we've said its a non-breaking change. **However**, it creates a transitive implication - we already had "all B impl C (not specializable)", by adding "all A impl B (specializable)," we've implicitly added the statement "all A impl C (not specializable)". Now the child crate cannot upgrade.

It might be the case that the idea of guaranteeing that adding specializable impls is not a breaking change is totally out the window, because Aaron showed (as you can see in the logs linked above) that you can write impls which make equivalent guarantees regarding defaultness. However, Niko's later comments suggest that such impls may be prohibited (or at least prohibitable) by the orphan rules.

So its uncertain to me if the 'impls are non-breaking' guarantee is salvageable, but it *is* certain that it is not compatible with explicit control over impl finality.

torkleyy commented on 28 Jan 2017 • edited

Is there any plan on allowing this?

```
struct Foo;

trait Bar {
    fn bar<T: Read>(stream: &T);
}

impl Bar for Foo {
    fn bar<T: Read>(stream: &T) {
        let stream = BufReader::new(stream);

        // Work with stream
    }

    fn bar<T: BufRead>(stream: &T) {
        // Work with stream
    }
}
```

So essentially a specialization for a template function which has a type parameter with a bound on `A` where the specialized version has a bound on `B` (which requires `A`).

withoutboats commented on 28 Jan 2017 Contributor

@torkleyy not currently but you can secretly do it by creating a trait which is implemented for both `T: Read` and `T: BufRead` and containing the parts of your code you want to specialize in the impls of that trait. It doesn't even need to be visible in the public API.

👍 2

withoutboats commented on 30 Jan 2017 Contributor

Regarding the backwards compatibility issue, I think thanks to the orphan rules we can get away with these rules:

*An impl is backwards compatible to add **unless**:*

- *The trait being impl'd is an auto trait.*
- *The receiver is a type parameter, and every trait in the impl previously existed.*

That is, I think in all of the problematic examples the added impl is a blanket impl. We wanted to say that fully default blanket impls are also okay, but I think we just have to say that adding of existing blanket impls can be a breaking change.

The question is what guarantee do we want to make in the face of that - e.g. I think it would be a very nice property if at least a blanket impl can only be a breaking change based on the code in your crate, so you can review your crate and know with certainty whether or not you need to increment the major version.

aturon commented on 7 Feb 2017 Member

@withoutboats

> Regarding the backwards compatibility issue, I think thanks to the orphan rules we can get away with these rules:
>
> *An impl is backwards compatible to add **unless**:*
>
> - *The trait being impl'd is an auto trait.*
> - *The receiver is a type parameter, and every trait in the impl previously existed.*
>
> That is, I think in all of the problematic examples the added impl is a blanket impl. We wanted to say that fully default blanket impls are also okay, but I think we just have to say that adding of existing blanket impls can be a breaking change.

A week and many discussions later, this has unfortunately turned out not to be the case.

withoutboats commented on 7 Feb 2017 Contributor

The results we've had are 😿, but I think what I wrote there is the same as your conclusion. Adding blanket impls is a breaking change, no matter what. But only blanket impls (and auto trait impls); as far as I know we've not found a case where a non-blanket impl could break downstream code (and that would be very bad).

I did think at one point that we might be able to relax the orphan rules so that you could implement traits for types like `Vec<MyType>`, but if we did that this situation would then play out in exactly the same way there:

```
//crate A

trait Foo { }

// new impl
// impl<T> Foo for Vec<T> { }


// crate B
extern crate A;
```

```
use A::Foo;

trait Bar {
    type Assoc;
}

// Sadly, this impl is not an orphan
impl<T> Bar for Vec<T> where Vec<T>: Foo {
    type Assoc = ();
}



// crate C

struct Baz;

// Therefore, this impl must remain an orphan
impl Bar for Vec<Baz> {
    type Assoc = bool;
}
```

aturon commented on 7 Feb 2017 **Member**

@withoutboats Ah, I understood your two-bullet list as *or* rather than *and*, which it seems is what you meant?

withoutboats commented on 7 Feb 2017 • edited Contributor

@aturon Yea, I meant 'or' - those are the two cases where it is a breaking change. Any auto trait impl, no matter how concrete, is a breaking change because of the way we allow negative reasoning about them to propogate: https://is.gd/k4Xtlp

That is, unless it contains new names. AFAIK an impl that contains a new name is never breaking.

nikomatsakis commented on 7 Feb 2017 Contributor

@withoutboats I wonder if we can/should restrict people relying on negative logic around auto-traits. That is, if we said that adding new impls of auto traits is a legal breaking change, we might then warn about impls that could be broken by an upstream crate adding `Send`. This would work best if we had:

1. stable specialization, one could overcome the warnings by adding `default` in strategic places (much of the time);
2. some form of explicit negative impls, so that types like `Rc` could declare their intention to **never** be `Send` -- but then we have those for auto traits, so we could take them into account.

withoutboats commented on 7 Feb 2017 Contributor

I don't know I think it depends on whether or not there's strong motivation. It seems especially unlikely you'll realize a type could have an `unsafe impl Send/Sync` after you've already released it; I think most of the time that would be safe, you'll have written a type with the foreknowledge that it would be safe (because that's the point of the type).

sgrif commented on 8 Feb 2017 Contributor

I add `unsafe impl Send/Sync` after the fact all the time. Sometimes because I make it thread safe, sometimes because I realize the C API I'm interfacing with is fine to share across threads, and sometimes it's just because whether something should be `Send` / `Sync` isn't what I'm thinking about when I introduce a type.

sfackler commented on 8 Feb 2017 **Member**

I add them after the fact as well when binding C APIs - often because someone explicitly asks for those bounds so I then go through and check what the underlying library guarantees.

dpc referenced this issue in **slog-rs/slog** on 10 Feb 2017

Implement slog::Serialize for std::net::SocketAddr #109 Closed

withoutboats commented on 13 Feb 2017 • edited Contributor

One thing I don't love about how specializing associated traits works right now, this pattern doesn't work:

```
trait Buffer: Read {
    type Buffered: BufRead;
    fn buffer(self) -> impl BufRead;
}

impl<T: Read> Buffer for T {
    default type Buffered = BufReader<T>;
    default fn buffer(self) -> BufReader<T> {
        BufReader::new(self)
    }
}

impl<T: BufRead> Buffer for T {
    type Buffered = Self;
    fn buffer(self) -> T {
        self
    }
}
```

This is because the current system requires that this impl would be valid:

```
impl Buffer for SomeRead {
    type Buffered = SomeBufRead;
    // no overriding of fn buffer, it no longer returns Self::Buffered
}
```

`impl Trait` in traits would release a lot of desire for this sort of pattern, but I wonder if there isn't a better solution where the generic impl is valid but that specialization doesn't work because it introduces a type error?

aturon commented on 13 Feb 2017 Member

@withoutboats Yeah, this is one of the main unresolved questions about the design (which I'd forgotten to bring up in recent discussions). There's a fair amount of discussion about this on the original RFC thread, but I'll try to write up a summary of the options/tradeoffs soon.

oli-obk referenced this issue in **serde-rs/json** on 13 Feb 2017

Parser cannot read arbitrary precision numbers #18 Closed

withoutboats commented on 13 Feb 2017 Contributor

@aturon Is the current solution the most conservative (forward compatible with whatever we want to do) or is it a decision we have to make before stabilizing?

nikomatsakis commented on 16 Feb 2017 Contributor

I personally think the only real solution to this problem that **@withoutboats** raised is to allow items to be "grouped" together when you specify the `default` tag. It's kind of the better-is-better solution, but I feel like the worse-is-better variant (overriding any means overriding all) is quite a bit worse. (But actually **@withoutboats** the way you wrote this code is confusing. I think in place of using `impl BufRead` as the return type of `Buffer`, you meant `Self::BufReader`, right?)

In that case, the following would be permitted:

```
trait Buffer: Read {
    type Buffered: BufRead;
    fn buffer(self) -> impl BufRead;
}
```

```
impl<T: Read> Buffer for T {
    default {
        type Buffered = BufReader<T>;
        fn buffer(self) -> BufReader<T> {
            BufReader::new(self)
        }
    }
}

impl<T: BufRead> Buffer for T {
    type Buffered = Self;
    fn buffer(self) -> T {
        self
    }
}
```

But perhaps we can infer these groupings? I've not given it much thought, but it seems that the fact that item defaults are "entangled" is visible from the trait definition.

👍 1

withoutboats commented on 16 Feb 2017 Contributor

> But actually **@withoutboats** the way you wrote this code is confusing. I think in place of using impl BufRead as the return type of Buffer, you meant Self::BufReader, right?

Yes, I had modified the solution to an impl Trait based one & then switched back but missed the return type in the trait.

ncalexan referenced this issue in **Marwes/combine** on 16 Feb 2017

Make it possible for `combine::primitives::ParseError` to be `std::error::Error` for more range types #86 Closed

porky11 commented on 24 Feb 2017

Maybe something like the type system of this language may also be interesting, since it seems to be similar to Rusts, but with some features, that may solve the current problems.
(`A <: B` would in Rust be true when `A` is a struct and implements trait `B` , or when `A` is a trait, and generic implementations for objects of this trait exist, I think)

antoyo commented on 5 Mar 2017 • edited Contributor

It seems there is an issue with the `Display` trait for specialization.
For instance, this example does not compile:

```
use std::fmt::Display;

pub trait Print {
    fn print(&self);
}

impl<T: Display> Print for T {
    default fn print(&self) {
        println!("Value: {}", self);
    }
}

impl Print for () {
    fn print(&self) {
        println!("No value");
    }
}

fn main() {
    "Hello, world!".print();
    ().print();
}
```

with the following error:

```
error[E0119]: conflicting implementations of trait `Print` for type `()`:
```

```
  --> src/main.rs:41:1
   |
35 |   impl<T: Display> Print for T {
   |  _- starting here...
36 | |     default fn print(&self) {
37 | |         println!("Value: {}", self);
38 | |     }
39 | | }
   | |_- ...ending here: first implementation here
40 |
41 |   impl Print for () {
   |  _^ starting here...
42 | |     fn print(&self) {
43 | |         println!("No value");
44 | |     }
45 | | }
   | |_^ ...ending here: conflicting implementation for `()`
```

while this compiles:

```
pub trait Print {
    fn print(&self);
}

impl<T: Default> Print for T {
    default fn print(&self) {
    }
}

impl Print for () {
    fn print(&self) {
        println!("No value");
    }
}

fn main() {
    "Hello, world!".print();
    ().print();
}
```

Thanks to fix this issue.

shepmaster commented on 5 Mar 2017 **Member**

@antoyo are you sure that's because `Display` is special, or could it be because `Display` isn't implemented for tuples while `Default` is?

antoyo commented on 5 Mar 2017 • edited Contributor

@shepmaster
I don't know if it is about `Display`, but the following works with a `Custom` trait not implemented for tuples:

```
pub trait Custom { }

impl<'a> Custom for &'a str { }

pub trait Print {
    fn print(&self);
}

impl<T: Custom> Print for T {
    default fn print(&self) {
    }
}

impl Print for () {
    fn print(&self) {
        println!("No value");
    }
}

fn main() {
    "Hello, world!".print();
    ().print();
}
```

By the way, here is the real thing that I want to achieve with specialization:

```
pub trait Emit<C, R> {
    fn emit(callback: C, value: Self) -> R;
}

impl<C: Fn(Self) -> R, R, T> Emit<C, R> for T {
    default fn emit(callback: C, value: Self) -> R {
        callback(value)
    }
}

impl<C> Emit<C, C> for () {
    fn emit(callback: C, _value: Self) -> C {
        callback
    }
}
```

I want to call a function by default, or return a value if the parameter would be unit.
I get the same error about conflicting implementations.
It is possible (or will this be possible) to do that with specialization?
If not, what are the alternatives?

Edit: I think I figured out why it does not compile:
`T` in `for T` is more general than `()` in `for ()` so the first `impl` cannot be the specialization.
And `C` is more general than `C: Fn(Self) -> R` so the second `impl` cannot be the specialization.
Please tell me if I'm wrong.
But I still don't get why it does not work with the first example with `Display`.

withoutboats commented on 5 Mar 2017 Contributor

This is currently the correct behavior.

In the `Custom` example, those impls do not overlap because of special *local negative reasoning*. Because the trait is from this crate, we can infer that `()`, which does not have an impl of `Custom`, does not overlap with `T: Custom`. No specialization necessary.

However, we do not perform this negative reasoning for traits that aren't from your crate. The standard library could add `Display for ()` in the next release, and we don't want that to be a breaking change. We want libraries to have the freedom to make those kinds of changes. So even though () doesn't impl Display, we can't use that information in the overlap check.

But also, because () doesn't impl Display, it is not more specific than `T: Display`. This is why specialization does not work, whereas in the Default case, `(): Default`, therefore that impl is more specific than `T: Default`.

Impls like this one are sort of in 'limbo' where we can neither assume it overlaps or doesn't. We're trying to figure out a principled way to make this work, but it's not the first implementation of specialization, it's a backwards compatible extension to that feature coming later.

👍 1

dtolnay referenced this issue in **serde-rs/serde** on 9 Mar 2017

Unstable functionality in Serde #812 Open

3 of 5 tasks complete

dtolnay referenced this issue on 16 Mar 2017

Specialization and lifetime dispatch #40582 Open

dtolnay commented on 16 Mar 2017 Member

I filed #40582 to track the lifetime-related soundness issue.

amosonn referenced this issue in **wfraser/fuse-mt** on 17 Mar 2017

Feature: store handler objects instead of fh #6 Open

colin-kiegel referenced this issue on 20 Mar 2017

Make AsRef and AsMut reflexive #39397 Closed

SergioBenitez added a commit to SergioBenitez/Rocket that referenced this issue on 19 Apr 2017

`Now support Result responses.` ··· `90b96a0`

afonso360 commented on 21 Apr 2017

I had an issue trying to use specialization, I don't think its quite the same as what **@antoyo** had, I had filed it as a separate issue #41140, I can bring the example code from that into here if necessary

aturon commented on 25 Apr 2017 Member

@afonso360 No, a separate issue is fine.

As a general point: at this point further work on specialization is blocked on the work on Chalk, which should allow us to tackle soundness issues and is also likely to clear up the ICEs being hit today.

Mark-Simulacrum referenced this issue on 2 May 2017

Conflicting impl of Borrow with templated trait type #32315 Closed

SergioBenitez referenced this issue in **SergioBenitez/Rocket** on 19 May 2017

Compile with stable Rust #19 Open

7 of 18 tasks complete

sgrif commented on 1 Jun 2017 Contributor

Can someone clarify if this is a bug, or something that is purposely forbidden? https://is.gd/pBvefi

rphmeier commented on 1 Jun 2017 • edited Contributor

@sgrif I believe the issue here is just that projection of default associated types is disallowed. Diagnostics could be better though: #33481

sgrif commented on 1 Jun 2017 Contributor

Could you elaborate on why it is expected to be disallowed? We know that no more specific impl could be added, since it would violate the orphan rules.

rphmeier commented on 1 Jun 2017 Contributor

This comment indicates it's to necessary for some cases in order to require soundness (although I don't know why) and in others to force consumers of the interface to treat it as an abstract type:
https://github.com/rust-lang/rust/blob/e5e664f/src/librustc/traits/project.rs#L41

fafhrd91 referenced this issue in **PyO3/pyo3** on 12 Jun 2017

Compile with stable rust #5 Open

2 of 5 tasks complete

hcpl referenced this issue in **phsym/prettytable-rs** on 12 Jun 2017

Cell redesign proposal #58 Open

lxrec referenced this issue in **rust-lang/rfcs** on 21 Jun 2017

eRFC: Experimentally add coroutines to Rust #2033 Merged

Mark-Simulacrum referenced this issue on 23 Jun 2017

ability to opt out of auto-rolling of reflexive From #42861 Closed

rubdos referenced this issue in **serde-rs/serde** on 24 Jun 2017

Serializer-specific struct serialization #968 Closed

joliss referenced this issue on 26 Jun 2017

Rc/Arc equality checking could short-circuit on equal pointers if T: Eq #42655 Open

sgrif commented on 14 Jul 2017 Contributor

Was anyone ever able to look at #31844 (comment) ? Those impls should be valid with specialization as far as I can tell. I believe there is a bug that is preventing them.

Mark-Simulacrum added **C-tracking-issue** and removed **C-enhancement** **C-feature-request** labels on 22 Jul 2017

mehcode referenced this issue in **mehcode/shio-rs** on 24 Aug 2017

Redirect example #9 Merged

SergioBenitez commented on 30 Aug 2017 • edited Contributor

@sgrif I believe the issue with your code there may be similar to the issue in #31844 (comment) which **@withoutboats** explained in #31844 (comment). That being said, based on **@withoutboats**'s comment, it seems that the present local reasoning should allow your example to compile, but perhaps I'm mistaken as to what's expected to work.

As an aside, I tried to implement the following, unsuccessfully:

```
trait Optional<T> {
    fn into_option(self) -> Option<T>;
}

impl<R, T: Into<R>> Optional<R> for T {
    default fn into_option(self) -> Option<R> {
        Some(self.into())
    }
}

impl<R> Optional<R> for Option<R> {
    fn into_option(self) -> Option<R> {
        self
    }
}
```

I intuitively expected `Option<R>` to be more specific than `<R, T: Into<R>> T`, but of course, nothing prevents an `impl<R> Into<R> for Option<R>` in the future.

I'm not sure why this is disallowed, however. Even if an `impl<R> Into<R> for Option<R>` was added in the future, I would still expect Rust to choose the non-`default` implementation, so as far as I can see, allowing this code has no implication on forward-compatibility.

In all, I find specialization very frustrating to work with. Just about everything I expect to work doesn't. The only cases where I've had success with specialization are those that are very simple, such as having an two `impl`s that include `T where T: A` and `T where T: A + B`. I have a hard time getting other things to work, and the error messages don't indicate why attempts to specialize don't work. Of course, there's still a road ahead, so I don't expect very helpful error messages. But there seem to be quite a few cases where I really expect something to work (like above) but it just doesn't, and it's currently quite difficult for me to ascertain if that's because I've misunderstood what's allowed (and more importantly, why), if something is wrong, or if something just hasn't been implemented yet. A nice overview of what's going on with this feature as it stands would be very helpful.

burns47 commented on 2 Sep 2017 • edited

I'm not positive this is in the right place, but we ran into a problem on the users forum that I'd like to mention here.

The following code (which is adapted from the RFC here) does not compile on nightly:

```
#![feature(specialization)]

trait Example {
    type Output;
    fn generate(self) -> Self::Output;
}

default impl<T> Example for T {
    type Output = Box<T>;
    fn generate(self) -> Self::Output { Box::new(self) }
}

impl Example for bool {
    type Output = bool;
    fn generate(self) -> Self::Output { self }
}
```

This doesn't really seem like a glitch but more like a usability problem - if a hypothetical `impl` specialized *only* the associated type in the example above, the `defaulti impl` of `generate` wouldn't typecheck.

Link to the thread here

dtolnay commented on 2 Sep 2017 — Member

@burns47 there is a confusing but useful workaround here: #31844 (comment).

burns47 commented on 2 Sep 2017

@dtolnay Not quite satisfactory - what if we're specializing on traits we don't own (and can't modify)? We shouldn't need to rewrite/refactor trait definitions to do this IMO.

👍 1

PlasmaPower referenced this issue in **matthiasbeyer/filters** on 3 Oct 2017

Add into_failable and as_failable methods #22 — Merged

nvzqz referenced this issue in **nvzqz/RandomKit** on 21 Oct 2017

revealing current seed as a public read-only property #44 — Open

bstrie commented on 26 Oct 2017 — Contributor

Can anyone comment as to whether the code in the following issue is intentionally rejected? #45542

Emerentius referenced this issue on 26 Oct 2017

Specialization: cannot specialize an impl of a local trait when the default impl is a blanket impl bounded by a non-local trait #45542 — Open

eternaleye referenced this issue in **serde-rs/bytes** on 17 Nov 2017

Consider not using specialization even after it lands #8 — Open

stbuehler referenced this issue in **rust-lang-nursery/futures-rs** on 28 Nov 2017

Generic Executor traits #661 — Closed

leodasvacas referenced this issue on 29 Nov 2017

Adding a specialized impl can break inference. #46363 — Open

mbrubeck referenced this issue in **servo/rust-smallvec** on 29 Nov 2017

[meta] Wishlist for 1.0 #73 — Open

2 of 7 tasks complete

Rantanen referenced this issue on 20 Dec 2017

Can't specialize `Drop` #46893 Open

Restioson referenced this issue in **rust-lang/rfcs** on 24 Dec 2017

Relax Trait Coherence rules to allow the implementation of a trait on generic types where the type must impl another trait owned by the current crate #1124 Open

stjepang commented on 7 Jan • edited Contributor

Would specialization allow adding something like the following to libcore?

```
impl<T: Ord> Eq for T {}

impl<T: Ord> PartialEq for T {
    default fn eq(&self, other: &Self) -> bool {
        self.cmp(other) == Ordering::Equal
    }
}

impl<T: Ord> PartialOrd for T {
    default fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
```

This way you could implement `Ord` for your custom type and have `Eq`, `PartialEq`, and `PartialOrd` be automatically implemented.

Note that implementing `Ord` and simultaneously deriving `PartialEq` or `PartialOrd` is dangerous and can lead to very subtle bugs! With these default impls you would be less tempted to derive those traits, so the problem would be somewhat mitigated.

Alternatively, we modify derivation to take advantage of specialization. For example, writing `#[derive(PartialOrd)]` above `struct Foo(String)` could generate the following code:

```
impl PartialOrd for Foo {
    default fn partial_cmp(&self, other: &Foo) -> Option<Ordering> {
        self.0.partial_cmp(&other.0)
    }
}

impl PartialOrd for Foo where Foo: Ord {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
```

This way the default impl gets used if `Ord` is not implemented. But if it is, then `PartialOrd` relies on `Ord`. Unfortunately, this doesn't compile: `error[E0119]: conflicting implementations of trait` `std::cmp::PartialOrd` for type `Foo`

scottmcm commented on 7 Jan Member

@stjepang I certainly hope the blankets like that can be added -- `impl<T:Copy> Clone for T` too.

👍 2

jan-hudec commented on 9 Jan

I think

> `impl<T: Ord> PartialEq for T`

should be

`impl<T, U> PartialEq<U> for T where T : PartialOrd<U>`

because `PartialOrd` requires `PartialEq` and can provide it too.

povilasb referenced this issue in **ujh/hamcrest-rust** on 16 Jan

Rename matcher Contains #45 Open

burdges commented on 17 Jan

Right now, one cannot really use associated types to constrain a specialization, both because they cannot be left unspecified and because they trigger uneeded recursion. See dhardy/rand#18 (comment)

Centril commented on 22 Jan Contributor

Eventually, I'ld love to see what I'm calling *specialization groups* with the syntax proposed by **@nikomatsakis** here #31844 (comment) and independently by me. I'ld like to write an RFC on that proposal later when we're closer to stabilizing specialization.

fluffysquirrels referenced this issue in **fluffysquirrels/webdriver_client_rust** on 1 Feb

Support launching multiple Chrome DriverSessions #17 Open

ExpHP referenced this issue on 15 Feb

Bug in rust automatic deref for methods of generic traits #48145 Open

remexre referenced this issue in **remexre/sparkly-rs** on 19 Feb

#[derive(PrettyDebug)] #1 Open

nikomatsakis referenced this issue on 25 Feb

implement "always applicable impls" #48538 Open

nikomatsakis commented on 25 Feb Contributor

Just in case nobody saw it, this blog post covers a proposal to make specialization sound in the face of lifetime-based dispatch.

❤️ 2

jasongrlicky referenced this issue in **pcsm/simulacrum** on 27 Mar

Figure out a way to not have <unknown> appear on Stable when encountering unexpected parameters #22 Open

jan-hudec referenced this issue in **rust-lang/rfcs** on 8 Apr

RFC: Implied #[derive(SuperTrait)] #2385 Closed

jturner314 referenced this issue in **bluss/ndarray** on 14 Apr

Broadcasting limitations / Custom strides? #437 Closed

frewsxcv referenced this issue in **georust/rust-geo** on 15 Apr

Figure out if it's possible to make the algorithm traits generically commutative #73 Open

earthengine commented on 16 Apr • edited

As copy closures were already stablized in Beta, developers have more motivation to stabilizing on specialization now. The reason is that `Fn` and `FnOnce + Clone` represent two overlapping set of closures, and in many case we need to implement traits for both of them.

Just figure out that the wording of rfc 2132 seems to imply that there are only 5 types of closures:

- `FnOnce` (a `move` closure with all captured variables being neither `Copy` nor `Clone`)
- `FnOnce + Clone` (a `move` closure with all captured variables being `Clone`)
- `FnOnce + Copy + Clone` (a `move` closure with all captured variables being `Copy` and so `Clone`)
- `FnMut + FnOnce` (a non-`move` closure with mutated captured variables)
- `Fn + FnMut + FnOnce + Copy + Clone` (a non-`move` closure without mutated captured variables)

So if specification is not available in the near future, maybe we should update our definition of `Fn` traits so `Fn` does not overlapping with `FnOnce + Clone`?

I understand that someone may already implemented specific types that is `Fn` without `Copy/Clone`, but should this be deprecated? I think there is always better way to do the same thing.

plietar referenced this issue in **rust-lang/rfcs** on 5 May

RFC: add futures and task system to libcore #2418 Open

glandium commented on 9 May Contributor

Is the following supposed to be allowed by specialization (note the absence of `default`) or is it a bug?

```
#![feature(specialization)]
mod ab {
    pub trait A {
        fn foo_a(&self) { println!("a"); }
    }

    pub trait B {
        fn foo_b(&self) { println!("b"); }
    }

    impl<T: A> B for T {
        fn foo_b(&self) { println!("ab"); }
    }

    impl<T: B> A for T {
        fn foo_a(&self) { println!("ba"); }
    }
}

use ab::B;

struct Foo;

impl B for Foo {}

fn main() {
    Foo.foo_b();
}
```

without specialization, this fails to build with:

```
error[E0119]: conflicting implementations of trait `ab::B` for type `Foo`:
  --> src/main.rs:24:1
   |
11 |     impl<T: A> B for T {
   |     ------------------ first implementation here
...
24 | impl B for Foo {}
   | ^^^^^^^^^^^^^^^^ conflicting implementation for `Foo`
```

👍 1

gnzlbg commented on 9 May Contributor

@glandium what on earth is going on there? Nice example, here the playground link: https://play.rust-lang.org/?gist=fc7cf5145222c432e2bd8de1b0a425cd&version=nightly&mode=debug

nikomatsakis commented on 10 May Contributor

@glandium that is #48444

glandium commented on 10 May Contributor

is it? there is no empty impl in my example.

MoSal commented on 11 May • edited

@glandium

```
impl B for Foo {}
```

gnzlbg commented on 11 May • edited Contributor

@MoSal but that impl "isn't empty" since `B` adds a method with a default implementation.

alexreg commented on 11 May Contributor

@gnzlbg It is empty by definition. Nothing between the braces.

RyanSquared referenced this issue in **RyanSquared/Syx** on 15 May

Fix optimization on loading u8 from u8 stream #1 Open

MoSal commented 8 days ago

```
#![feature(specialization)]

use std::borrow::Borrow;

#[derive(Debug)]
struct Bla {
    bla: Vec<Option<i32>>
}

// Why is this a conflict ?
impl From<i32> for Bla {
    fn from(i: i32) -> Self {
        Bla { bla: vec![Some(i)] }
    }
}

impl<B: Borrow<[i32]>> From<B> for Bla {
    default fn from(b: B) -> Self {
        Bla { bla: b.borrow().iter().map(|&i| Some(i)).collect() }
    }
}

fn main() {
    let b : Bla = [1, 2, 3].into();
    println!("{:?}", b);
}
```

```
error[E0119]: conflicting implementations of trait `std::convert::From<i32>` for type
`Bla`:
  --> src/main.rs:17:1
   |
11 | impl From<i32> for Bla {
   | ---------------------- first implementation here
...
17 | impl<B: Borrow<[i32]>> From<B> for Bla {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ conflicting implementation for `Bla`
   |
   = note: upstream crates may add new impl of trait `std::borrow::Borrow<[i32]>` for
type `i32` in future versions
```

Wouldn't specialization prevent possible future conflicts?

alexreg commented 8 days ago

Contributor

Goodness me, this is a slow-moving feature! No progress in over two years, it seems (certainly according to the original post). Has the lang team abandoned this?

Centril commented 8 days ago

Contributor

@alexreg see http://aturon.github.io/2018/04/05/sound-specialization/ for the latest development.

👍 3

mark-i-m commented 8 days ago

Contributor

@alexreg It turns out soundness is *hard*. I believe there is some work on the "always applicable impls" idea currently happening, so there is progress. See #49624. Also, I believe that the chalk working group is working on implementing the "always applicable impls" idea too, but I don't know how far that has gotten.

👍 1

Lymia commented 4 days ago • edited

Contributor

After a bit of wrangling, it seems it is possible to effectively implement intersection impls already via a hack using `specialization` and `overlapping_marker_traits`.

https://play.rust-lang.org/?gist=cb7244f41c040db41fc447d491031263&version=nightly&mode=debug