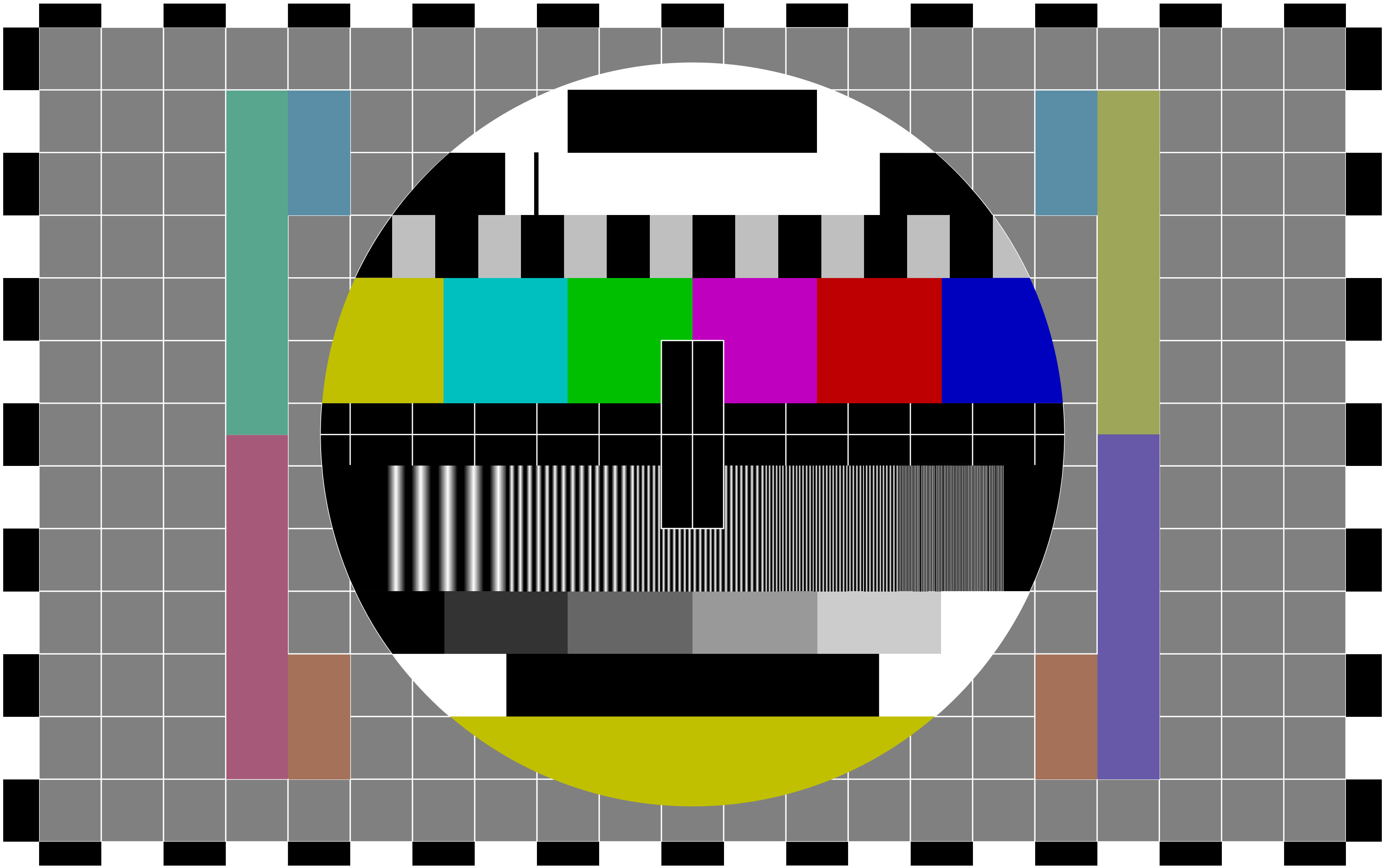


Scary Acronyms (and Super Creeps)

A take on OIBITs, HRTBs, and other charming abbreviations

Patryk Wychowaniec

Past

Quite undoubtedly, many interesting things happened in the past.

1752

For instance, on September 2nd, 1752 **six and a half million** Britons went to bed and woke up on September 14th.

1752

For instance, on September 2nd, 1752 **six and a half million** Britons went to bed and woke up on September 14th.

The reason was: **Calendar (New Style) Act 1750**.

1582

This guy is Pope Gregory XIII.

1582

This guy is Pope Gregory XIII.

In 1582 he was 10 years into his reign as a leader of the Catholic church.

1582

This guy is Pope Gregory XIII.

In 1582 he was 10 years into his reign as a leader of the Catholic church.

… and he had a problem with **Easter**.

1582

To understand why, you've gotta remember that in 1582, **Julian calendar** was (still) all the hype.

1582

To understand why, you’ve gotta remember that in 1582, **Julian calendar** was (still) all the hype.

It measured a year as **365 days** and **6 hours long**…

1582

To understand why, you've gotta remember that in 1582, **Julian calendar** was (still) all the hype.

It measured a year as **365 days** and **6 hours long**…

… which was *close*, but not exactly, **365 days, 5 hours** and **49 minutes**.

1582

Pope Gregory XIII, afraid that "Earth days" (and thus holidays) have diverged over time, declared that countries under the Catholic dominionship should skip a few days to catch up.

1582

Most countries agreed

1582

Most countries agreed

Britain did not

1582

Most countries agreed

Britain did not

... until 1752

1752

In 1752 Britain eventually legislated **Calendar (New Style) Act 1750**, cutting **11 days** from everyone’s lives.

Fast-forward

Let’s fast-forward a few years…

2014

What happened in 2014?

2014

In 2014, there was a FIFA World Cup:

2014

In 2014, Marek Sawicki was appointed to the position of minister of Agriculture and Rural Development in Poland:

2014

Also, this document happened:

Tree: **8fa971a670** ▾ **rfcs** / text / **0019-opt-in-builtin-traits.md** — Find file | Copy path

pnkfelix Fixed typos and format inconsistencies in headers of various RFCs. `f9f030e` on 8 Oct 2014

1 contributor

`531 lines (420 sloc) | 23.7 KB` — Raw | Blame | History

- Start Date: 2014-09-18
- RFC PR #: rust-lang/rfcs#19, rust-lang/rfcs#127
- Rust Issue #: rust-lang/rust#13231

Summary

The high-level idea is to add language features that simultaneously achieve three goals:

1. move `Send` and `Share` out of the language entirely and into the standard library, providing mechanisms for end users to easily implement and use similar "marker" traits of their own devising;
2. make "normal" Rust types sendable and sharable by default, without the need for explicit opt-in; and,
3. continue to require "unsafe" Rust types (those that manipulate unsafe pointers or implement special abstractions) to "opt-in" to sendability and sharability with an unsafe declaration.

These goals are achieved by two changes:

OIBITs

OIBITs

OIBITs

OIBITs

OIBITs

OIBITs

OIBITs

OIBITs

OIBITs

To understand OIBITs, let’s see them at work.

OIBITs

Let's create our very-own struct:

```rust
struct StrWrapper(&'static str);
```

RUST

OIBITs

Now, let's create a variable holding an instance of it:

RUST

```
struct StrWrapper(&'static str);

fn main() {
    let text = StrWrapper(
        "c-rustacean is a rust programmer who likes c
better"
    );
}
```

OIBITs

And, just for the kicks, let's send it into another thread:

RUST

```
struct StrWrapper(&'static str);

fn main() {
    let text = StrWrapper(
        "c-rustacean is a rust programmer who likes c better"
    );

    std::thread::spawn(move || {
        println!("{}", text.0);
    }).join().unwrap();
}
```

OIBITs

So, why does this code compile?

OIBITs

Not all values can be safely sent across thread boundaries - for instance we can't send `Rc`, because it's not thread-safe:

RUST

```
use std::rc::Rc;

fn main() {
    let num = Rc::new(123);

    std::thread::spawn(move || {
        println!("{}", num);
    }).join().unwrap();
}
```

OIBITs

```
error[E0277]: `Rc<i32>` cannot be sent between threads safely
    |
    |       thread::spawn(move || {
    |  _____^^^^^^^^^^^^^_-
    | |     |
    | |     `Rc<i32>` cannot be sent between threads safely
    | |         println!("{}", num);
    | |     });
    | |_____- within this `[closure]`
```

OIBITs

RUST

```
use std::rc::Rc;

fn main() {
    let num = Rc::new(123);
    let mut num2 = Rc::clone(&num);

    std::thread::spawn(move || {
        // err: race read
        println!("{}", num);
    }).join().unwrap();

    // err: race write
    *Rc::get_mut(&mut num2).unwrap() += 1;
}
```

OIBITs

To distinguish between values (types) that can be sent across thread boundaries, and those which can't, Rust uses the `Send` trait.

OIBITs

To distinguish between values (types) that can be sent across thread boundaries, and those which can’t, Rust uses the `Send` trait.

In other words: only when a type implements `Send`, can it be safely transferred into another thread.

OIBITs

We can confirm this by inspecting the definition of `std::thread::spawn()`:

```rust
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

OIBITs

Going back to my original question:

Why does this code compile, if we don't have `impl Send for StrWrapper { }` anywhere?

RUST

```
struct StrWrapper(&'static str);

fn main() {
    let text = StrWrapper(
        "c-rustacean is a rust programmer who likes c better"
    );

    std::thread::spawn(move || {
        println!("{}", text.0);
    }).join().unwrap();
}
```

OIBITs

OIBIT stands for: **opt-in built-in trait.**

OIBITs

OIBIT stands for: **opt-in built-in trait**.

There are two vital things you have to know about opt-in built-in traits:

OIBITs

OIBIT stands for: **opt-in built-in trait**.

There are two vital things you have to know about opt-in built-in traits:

- **they aren't opt-in** (mostly),

OIBITs

OIBIT stands for: **opt-in built-in trait**.

There are two vital things you have to know about opt-in built-in traits:

- **they aren't opt-in** (mostly),
- **they aren't built-in** (mostly).

OIBITs

OIBIT stands for: **opt-in built-in trait**.

There are two vital things you have to know about opt-in built-in traits:

- **they aren’t opt-in** (mostly),
- **they aren’t built-in** (mostly).

The feature was later renamed into **auto traits**, so from this point forward we’re going to stick to the new terminology.

Auto traits

When you have a regular trait, you have to implement it yourself (`opt-in`):

RUST

```rust
struct StrWrapper(&'static str);

impl fmt::Display for StrWrapper {
    /* ... */
}
```

Auto traits

On the other hand, **auto traits** are implemented for you *automatically*, unless you explicitly `opt-out` of them:

RUST

```rust
struct StrWrapper(&'static str);

impl !Send for StrWrapper { }
//   ^ notice the exclamation mark
```

Auto traits

RUST

```
struct StrWrapper(&'static str);

impl !Send for StrWrapper { } // here

fn main() {
    let text = StrWrapper(
        "c-rustacean is a rust programmer who likes c better"
    );

    std::thread::spawn(move || {
        println!("{}", text.0);
    }).join().unwrap();
}
```

Auto traits

```
error[E0277]: `StrWrapper` cannot be sent between threads safely
    |
    |       std::thread::spawn(move || {
    |  _____^^^^^^^^^^^^^^^^^^_-
    | |     |
    | |     `StrWrapper` cannot be sent between threads safely
    | |         println!("{}", text.0);
    | |     }).join().unwrap();
    | |_____- within this `[closure]`
```

Auto traits

Generally, the rule is:

Type `T` automatically implements auto trait `X` when **all** fields of that type implement `X` too.

Auto traits

RUST

```rust
pub struct Word {
    word: String,
    synonyms: Vec<String>,
    antonyms: Vec<String>,
}

fn assert_is_send<T: Send>() { }

fn main() {
    assert_is_send::<Word>();
}
```

Since:

- `String` already implements `Send`,
- `Vec<T>` implements `Send` when `T` does,

… compiler automatically deducts that it’s safe to `impl Send` for this struct too.

Auto traits

RUST

```rust
pub struct Word {
    word: String,
    synonyms: Vec<Rc<String>>, // here
    antonyms: Vec<String>,
}

fn assert_is_send<T: Send>() { }

fn main() {
    assert_is_send::<Word>();
           // error: ^^^^ `Word` cannot be sent between
           //             threads safely
}
```

Since `Rc` implements `!Send`, compiler automatically deducts that our `Word` is `!Send` too.

Auto traits

RUST

```
use std::ffi::c_void;

pub struct EnterpriseFizzBuzzFfiWrapper {
    java_handler_object_facade: *const c_void,
}

fn assert_is_send<T: Send>() { }

fn main() {
    assert_is_send::<EnterpriseFizzBuzzFfiWrapper>();
           // error: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
}
```

Auto traits

`Send` isn't magic - it's defined in the standard library:

RUST

```rust
pub unsafe auto trait Send {
    // empty.
}

impl<T: ?Sized> !Send for *const T {}
impl<T: ?Sized> !Send for *mut T {}
impl<T: ?Sized> !Send for Rc<T> {}
// ... and many more
```

Auto traits

No one prevents you from creating your own auto traits:

RUST

```
auto trait Friend { }

impl !Friend for String { }

fn ensure_friend<T: Friend>() { }

fn main() {
    ensure_friend::<&str>();
    ensure_friend::<String>();
        // error: ^^^^^^ the trait `Friend` is not
        //               implemented for
        //               `std::string::String`
}
```

Me, Myself and I

My name's **Patryk Wychowaniec**, a.k.a. **Patryk27**:

keybase.io/patryk27

reddit.com/u/patryk27

github.com/patryk27

4programmers.net (patryk27)

HRTBs

Best way to find HRTBs? Hidden in the plain sight!

HRTBs

RUST

```
struct Movie {
    /* ... */
}
```

HRTBs

RUST

```
struct Movie {
    title: String,
    year: isize, // gotta care about those
                 // pre Christian-Era movies!
}
```

HRTBs

RUST

```rust
struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self) {
        todo!()
    }
}
```

HRTBs

RUST

```
struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        todo!()
    }
}
```

HRTBs

RUST

```rust
struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}
```

HRTBs

RUST

```
struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}

fn main() {
    todo!()
}
```

HRTBs

RUST

```rust
/* ... */

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}

fn main() {
    Movie {
        title: "The Room".into(),
        year: 2003,
    }.print(todo!());
}
```

HRTBs

How should our `Serializer` type look like?

```rust
type Serializer = ?;
```

HRTBs

First of all - it has to be generic over `T`:

RUST

```rust
type Serializer<T> = ?;
```

HRTBs

... we also want it to be a function:

RUST

```
type Serializer<T> = dyn Fn(?) -> ?;
```

HRTBs

… a one returning string:

RUST

```
type Serializer<T> = dyn Fn(?) -> String;
```

HRTBs

… and, obviously, it has to accept the object it wants to serialize:

RUST

```
type Serializer<T> = dyn Fn(&T) -> String;
```

HRTBs

Voilà:

RUST

```rust
type Serializer<T> = dyn Fn(&T) -> String; // here

struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}

fn main() {
    Movie {
        title: "The Room".into(),
        year: 2003,
    }.print(todo!());
}
```

HRTBs

Now, to create some actual serializer, we’re going to use `serde`.

HRTBs

RUST

```rust
use serde::Serialize; // | here

type Serializer<T> = dyn Fn(&T) -> String;

#[derive(Serialize)] // | here
struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}

fn to_json<T>(value: &T) -> String where T: Serialize { // | here
    todo!()                                             // |
}                                                       // |

fn main() {
    Movie {
        title: "The Room".into(),
        year: 2003,
    }.print(to_json); // | here
}
```

HRTBs

RUST

```rust
use serde::Serialize;

type Serializer<T> = dyn Fn(&T) -> String;

#[derive(Serialize)]
struct Movie {
    title: String,
    year: isize,
}

impl Movie {
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}

fn to_json<T>(value: &T) -> String where T: Serialize {
    serde_json::to_string(value) // | here
        .unwrap()                // |
}

fn main() {
    Movie {
        title: "The Room".into(),
        year: 2003,
    }.print(to_json);
}
```

HRTBs

RUST

```rust
fn main() {
    Movie {
        title: "The Room".into(),
        year: 2003,
    }.print(to_json);
//            ^^^^^^^
}
```

```
error[E0308]: mismatched types
   |
26 |     }.print(to_json);
   |             ^^^^^^^
   |             |
   |             expected reference, found fn item
   |             help: consider borrowing here: `&to_json`
   |
   = note: expected reference `&(dyn for<'r> std::ops::Fn(&'r Movie) -> std::string::String + 'static)`
              found fn item `for<'r> fn(&'r _) -> std::string::String {to_json::<_>}`

error: aborting due to previous error
```

HRTBs

expected reference:

```
&(dyn for<'r> Fn(&'r Movie) → String + 'static)
```

HRTBs

expected reference:

```
&(dyn for<'r> Fn(&'r Movie) → String + 'static)
```

(that’s our `Serializer`)

HRTBs

expected reference:

```
&(dyn for<'r> Fn(&'r Movie) → String + 'static)
```

(that's our `Serializer`)

found fn item:

```
for<'r> fn(&'r ) → String {to_json::<>}
```

HRTBs

expected reference:

```
&(dyn for<'r> Fn(&'r Movie) → String + 'static)
```

(that's our `Serializer`)

found fn item:

```
for<'r> fn(&'r ) → String {to_json::<>}
```

(that's our `to_json`)

HRTBs

expected reference:

```
&(dyn for<'r> Fn(&'r Movie) → String + 'static)
```

(that’s our `Serializer`)

found fn item:

```
for<'r> fn(&'r ) → String {to_json::<>}
```

(that’s our `to_json`)

What’s this `dyn for` thingie? We didn’t write it anywhere!

HRTBs

Let’s go back to our type:

```rust
type Serializer<T> = dyn Fn(&T) -> String;
```

RUST

HRTBs

Let’s go back to our type:

RUST

```rust
type Serializer<T> = dyn Fn(&T) -> String;
//                            ^^ so... what's the
//                               lifetime of this?
```

HRTBs

Let’s go back to our type:

RUST

```
type Serializer<T> = dyn Fn(&T) -> String;
//                           ^^ so... what's the
//                              lifetime of this?
//
//                              why is this even
//                              legal?
```

HRTBs

Let’s go back to our type:

RUST

```
type Serializer<T> = dyn Fn(&T) -> String;
//                            ^^ so... what's the
//                               lifetime of this?
//
//                               why is this even
//                               legal?
```

Answer: **Lifetime elision**

HRTBs

Lifetime elision

To make common lifetime patterns more ergonomic, Rust sometimes allows for lifetimes to be *elided* (i.e. ignored, skipped).

HRTBs

Lifetime elision

To make common lifetime patterns more ergonomic, Rust sometimes allows for lifetimes to be *elided* (i.e. ignored, skipped).

Our tiny example actually used this mechanism **thrice**!

HRTBs

Lifetime elision (1/3)

RUST

```rust
impl Movie {
    //           v                 v
    pub fn print(&self, serialize: &Serializer<Self>) {
        println!("{}", serialize(self));
    }
}
```

HRTBs

Lifetime elision (1/3)

RUST

```rust
impl Movie {
    pub fn print<'a, 'b>(
        &'a self,
        serialize: &'b Serializer<Self>,
    ) {
        println!("{}", serialize(self));
    }
}
```

HRTBs

Lifetime elision (2/3)

RUST

```rust
//                   v
fn to_json<T>(value: &T) -> String
where T: Serialize {
    serde_json::to_string(value)
        .unwrap()
}
```

HRTBs

Lifetime elision (2/3)

RUST

```rust
fn to_json<'a, T>(value: &'a T) -> String
where T: Serialize {
    serde_json::to_string(value)
        .unwrap()
}
```

HRTBs

Lifetime elision (3/3)

RUST

```rust
type Serializer<T> = dyn Fn(&T) -> String;
```

HRTBs

Lifetime elision (3/3)

```rust
type Serializer<T> = dyn Fn(&T) -> String;
```

RUST

What we **want** is a function that will work *for any* lifetime.

We don't care how long `&T` lives, as long as we can access it during the function call.

HRTBs

Lifetime elision (3/3)

We *could* do...

RUST

```rust
type Serializer<'a, T> = dyn Fn(&'a T) -> String;
```

HRTBs

Lifetime elision (3/3)

We *could* do...

RUST

```rust
type Serializer<'a, T> = dyn Fn(&'a T) -> String;
```

... but that would be a bit cumbersome to use (and, in a few places, *impossible* to apply).

... plus we've already said that we want our serializer to work for **any** lifetime, not a specific one.

HRTBs

Lifetime elision (3/3)

Here come **HRTBs**!

HRTBs

Lifetime elision (3/3)

Here come **higher-ranked trait bounds**!

HRTBs

Lifetime elision (3/3)

RUST

```rust
type Serializer<T> = dyn for<'a> Fn(&'a T) -> String;
//                       ^-----^
```

The underlined part is the way we form a higher-ranked trait bound.

What it means is basically: I don’t care about the precise lifetime, make it work for *every one*.

HRTBs

Thus the name: `higher-ranked` as if `not limited to specific lifetime`, `lifted above the ordinary types`™.

HRTBs

By the way, it might be tempting to create types such as:

RUST

```rust
type Wat1 = for<T> T;
type Wat2 = for<'a, T> &'a T;
type Wat3 = for<T> Vec<T>;
type Wat4 = for<T> Vec<Box<T>>;
```

HRTBs

By the way, it might be tempting to create types such as:

RUST

```rust
type Wat1 = for<T> T;
type Wat2 = for<'a, T> &'a T;
type Wat3 = for<T> Vec<T>;
type Wat4 = for<T> Vec<Box<T>>;
```

Worry no more - they are all **illegal**:

```
error: only lifetime parameters can be used in this context
  |
1 | type Wat4 = for<T> Vec<Box<T>>;
  |                 ^
```

HRTBs

Let’s go find another HRTB in the wild.

HRTBs

Let's create a function:

RUST

```rust
fn call_me_maybe() {

}
```

HRTBs

Let’s make our function create an object *inside* it:

RUST

```
fn call_me_maybe() {
    let motto = String::from("existential crisis");
}
```

HRTBs

And, eventually, let's make it accept a closure that will get invoked with a *reference* to that object:

RUST

```rust
fn call_me_maybe(callback: impl Fn(&String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

HRTBs

Now for a quick test:

RUST

```
fn call_me_maybe(callback: impl Fn(&String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}

fn main() {
    call_me_maybe(|motto| {
        println!("motto: {}", motto);
    });
}
```

HRTBs

It works:

RUST

```
fn call_me_maybe(callback: impl Fn(&String)) {
    let motto = String::from("existential
crisis");
    callback(&motto);
}

fn main() {
    call_me_maybe(|motto| {
        println!("motto: {}", motto);
    });
}
```

```
motto: existential crisis
```

HRTBs

... but:

RUST

```rust
//                                  v what's this
//                                  v lifetime, exactly?
fn call_me_maybe(callback: impl Fn(&String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

HRTBs

... but:

RUST

```
//                                 v what's this
//                                 v lifetime, exactly?
fn call_me_maybe(callback: impl Fn(&String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

Once again, **lifetime elision** kicked-in - let's try to desugar our code and see what's happening underneath.

HRTBs

Our first thought may be:

RUST

```rust
fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

HRTBs

… but, unfortunately:

HRTBs

```
error[E0597]: `motto` does not live long enough
```

HRTBs

```
error[E0597]: `motto` does not live long enough
  |
1 | fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
  |                  -- lifetime `'a` defined here
```

HRTBs

```
error[E0597]: `motto` does not live long enough
  |
1 | fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
  |                 -- lifetime `'a` defined here
3 |     callback(&motto);
  |     ---------^^^^^^-
  |     |
  |     argument requires that `motto` is borrowed for `'a`
4 | }
```

HRTBs

```
error[E0597]: `motto` does not live long enough
  |
1 | fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
  |                  -- lifetime `'a` defined here
3 |     callback(&motto);
  |     ---------^^^^^^-
  |     |        |
  |     |        borrowed value does not live long enough
  |     argument requires that `motto` is borrowed for `'a`
4 | }
  | - `motto` dropped here while still borrowed
```

HRTBs

What the compiler is *trying* to say is that our `&motto` doesn’t necessarily live for `'a`, as we’ve tried to persuade it.

HRTBs

What the compiler is *trying* to say is that our `&motto` doesn’t necessarily live for `'a`, as we’ve tried to persuade it.

And, to no one’s surprise, that’s *true*!

To see why, let’s move on to the **call site**.

HRTBs

RUST

```
fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}

fn main() {
    //                    v------------v
    call_me_maybe::<'some_lifetime>(|motto| {
        println!("motto: {}", motto);
    });
}
```

From the `main` ' s point of view, **what's this lifetime**?

HRTBs

RUST

```rust
fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}

fn main() {
    //                v-----------v
    call_me_maybe::<'some_lifetime>(|motto| {
        println!("motto: {}", motto);
    });
}
```

This lifetime depends on *nothing* inside the `main` function, so what sense does it even make here?

HRTBs

Why do we even declared our function as **generic** over a lifetime `'a`, if there's just **one** lifetime that could ever possibly match?

RUST

```
fn call_me_maybe<'a>(callback: impl Fn(&'a String)) {
    { // lifetime 'motto starts here

        let motto = String::from("existential crisis");

        callback(&motto); // callback must use this
                          // "internal" 'motto lifetime

    } // lifetime 'motto ends here
}
```

HRTBs

So, similarly to the case we'd had before, we want for `call_me_maybe()` to invoke a callback *without* caring for / naming the actual lifetime.

HRTBs

So, similarly to the case we'd had before, we want for `call_me_maybe()` to invoke a callback *without* caring for / naming the actual lifetime.

Higher-ranked trait bounds come to the rescue.

HRTBs

RUST

```rust
fn call_me_maybe(callback: impl Fn(&String)) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

HRTBs

RUST

```
fn call_me_maybe(
    callback: impl for<'a> Fn(&'a String)
) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

HRTBs

RUST

```
fn call_me_maybe(
    callback: impl for<'a> Fn(&'a String)
) {
    let motto = String::from("existential crisis");
    callback(&motto);
}
```

Lo and behold, it actually works.

GATs

GATs

Y'all know the `Iterator` trait, right?

RUST

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

GATs

Y’all know the `Iterator` trait, right?

RUST

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Pros:

GATs

Y’all know the `Iterator` trait, right?

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Pros:

- really simple & tidy

GATs

Y'all know the `Iterator` trait, right?

RUST

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Pros:

- really simple & tidy
- does its job

GATs

Y'all know the `Iterator` trait, right?

RUST

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Pros:

- really simple & tidy
- does its job
- with us since, like, forever

GATs

Y'all know the `Iterator` trait, right?

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Pros:

- really simple & tidy
- does its job
- with us since, like, forever

Cons:

GATs

Y'all know the `Iterator` trait, right?

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Pros:

- really simple & tidy
- does its job
- with us since, like, forever

Cons:

- how do I return an item that borrows from the iterator?

GATs

Now, that *might* seem like a weird question at first, so let’s get our hands on some code that would benefit from such iterator.

GATs

RUST

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let file = File::open("test.txt")
        .unwrap();

    let lines = BufReader::new(file)
        .lines();

    for line in lines {
        println!("{}", line.unwrap());
    }
}
```

GATs

What's wrong with this code?

GATs

What’s wrong with this code?

It’s alright~ish, but not perfect, because it’s **suboptimal**.

GATs

RUST

```
fn main() {
    /* ... */

    for line in lines {
        // For each line, `BufReader` has to allocate a
        // brand-new `String`.
        //
        // Ideally, `BufReader` would just return
        // `Iterator<Item=&str>`, re-using the same
        // `String` underneath.

        println!("{}", line.unwrap());
    }
}
```

GATs

Naturally, a question arises:

Why can't `Lines` (i.e. the object you get by invoking `.lines()`) be `Iterator<Item = &str>` right now?

GATs

Naturally, a question arises:

Why can't `Lines` (i.e. the object you get by invoking `.lines()`) be `Iterator<Item = &str>` right now?

Is it because some big Rust-pharma doesn't want you to know about *truly* zero-cost abstractions?

GATs

Naturally, a question arises:

Why can't `Lines` (i.e. the object you get by invoking `.lines()`) be `Iterator<Item = &str>` right now?

Is it because some big Rust-pharma doesn't want you to know about *truly* zero-cost abstractions?

To find out, let's try to create such iterator!

GATs

Starting from the top:

RUST

```rust
struct SmartLines {
    /* ... */
}
```

GATs

For maximum pleasure & re-usability, we're going to be generic over everything that's `Read`:

RUST

```rust
use std::io::Read;

struct SmartLines<R: Read> {
    /* ... */
}
```

GATs

As for the fields - since what we're creating is a *wrapper*, we'll for sure need to store the underlying reader:

RUST

```rust
use std::io::Read;

struct SmartLines<R: Read> {
    reader: R,
}
```

GATs

Since what we're creating is *smart*, we'll for sure need to store the line-buffer too:

RUST

```rust
use std::io::Read;

struct SmartLines<R: Read> {
    reader: R,
    line: String,
}
```

GATs

We could use some constructor:

RUST

```rust
/* ... */

impl<R: Read> SmartLines<R> {
    pub fn new(reader: R) -> Self {
        Self {
            reader,
            line: String::new(),
        }
    }
}
```

GATs

And, finally, the `impl Iterator` - we’re so, *so* close!

RUST

```
/* ... */

impl<R: Read> Iterator for SmartLines<R> {
    /* ... */
}
```

GATs

We're going to yield `&str`, so:

RUST

```rust
/* ... */

impl<R: Read> Iterator for SmartLines<R> {
    type Item = &str;

    fn next(&mut self) -> Option<Self::Item> {
        todo!()
    }
}
```

GATs

We’re going to yield `&str`, so:

RUST

```rust
/* ... */

impl<R: Read> Iterator for SmartLines<R> {
    type Item = &str;

    fn next(&mut self) -> Option<Self::Item> {
        todo!()
    }
}
```

… oh, right…

GATs

```
error[E0106]: missing lifetime specifier
   |
   |     type Item = &str;
   |                 ^ expected named lifetime parameter
```

GATs

RUST

```rust
impl<R: Read> Iterator for SmartLines<R> {
    type Item = &str;
              // ^ we can't name this lifetime here...

        // v ... because it's not known up to
        // v     the point here
    fn next(&mut self) -> Option<Self::Item> {
        todo!()
    }
}
```

GATs

But - *d’oh!* - why don’t we just implement the `Iterator` *for a reference*?

GATs

RUST

```
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    /* ... */
}
```

GATs

RUST

```rust
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    type Item = &'a str;

    /* ... */
}
```

GATs

RUST

```
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    type Item = &'a str;

    fn next(&mut self) -> Option<Self::Item> {
        todo!()
    }
}
```

GATs

RUST

```
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    type Item = &'a str;

    fn next(&mut self) -> Option<Self::Item> {
        Some(&self.line)
    }
}
```

GATs

```
error[E0495]: cannot infer an appropriate lifetime for borrow expression due to conflicting requirements
   |
   |          Some(&self.line)
   |               ^^^^^^^^^^
   |
note: first, the lifetime cannot outlive the anonymous lifetime #1 defined on the method body at 20:5...
   |
   | /     fn next(&mut self) -> Option<Self::Item> {
   | |         Some(&self.line)
   | |     }
   | |_____^
note: ...so that reference does not outlive borrowed content
   |
   |          Some(&self.line)
   |               ^^^^^^^^^^
note: but, the lifetime must be valid for the lifetime `'a` as defined on the impl at 17:6...
   |
   | impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
   |      ^^
note: ...so that the types are compatible
   |
   |       fn next(&mut self) -> Option<Self::Item> {
   |  ______________________________________________^
   | |         Some(&self.line)
   | |     }
   | |_____^
```

GATs

What the compiler is trying to say is that `&mut self` doesn't necessarily live for `'a`, because they are two separate lifetimes:

RUST

```rust
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    type Item = &'a str;

        // v doesn't necessarily predecease 'a
    fn next(&mut self) -> Option<Self::Item> {
        Some(&self.line)
    }
}
```

GATs

We could try fixing this by annotating the lifetime we *expect* to be there:

RUST

```rust
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    type Item = &'a str;

          // vv here
    fn next(&'a mut self) -> Option<Self::Item> {
        Some(&self.line)
    }
}
```

GATs

We could try fixing this by annotating the lifetime we *expect* to be there:

RUST

```rust
impl<'a, R: Read> Iterator for &'a mut SmartLines<R> {
    type Item = &'a str;

          // vv here
    fn next(&'a mut self) -> Option<Self::Item> {
        Some(&self.line)
    }
}
```

… but, as you might have guessed, **that doesn't work**

GATs

```
error[E0308]: method not compatible with trait
   |
   |     fn next(&'a mut self) -> Option<Self::Item> {
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |                                 lifetime mismatch
   |
   = note: expected fn pointer
             `fn(&mut &'a mut SmartLines<R>) -> Option<_>`
           found fn pointer
             `fn(&'a mut &'a mut SmartLines<R>) -> Option<_>`
```

GATs

The proper solution, as it turns out, requires a magic of **GATs**.

GATs

The proper solution, as it turns out, requires a magic of **generic associated types**.

GATs

Let's go back to the definition of our iterator:

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

GATs

Let's go back to the definition of our iterator:

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

The issue with current design is that we cannot possibly name or *provide* the lifetime for the `Item` associated type.

GATs

Let’s go back to the definition of our iterator:

RUST

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

The issue with current design is that we cannot possibly name or *provide* the lifetime for the `Item` associated type.

Solution? Let’s make the `Item` generic (at least over lifetimes)!

GATs

RUST

```rust
trait StreamingIterator {
    type Item<'a>;
    fn next(&mut self) -> Option<Self::Item<'_>>;
}
```

GATs

RUST

```
impl<R: Read> StreamingIterator for SmartLines<R> {
    type Item<'a> = &'a str;

    fn next(&mut self) -> Option<Self::Item<'_>> {
        todo!()
    }
}
```

GATs

RUST

```
impl<R: Read> StreamingIterator for SmartLines<R> {
    type Item<'a> = &'a str;

    fn next(&mut self) -> Option<Self::Item<'_>> {
        todo!()
    }
}
```

It’s been already possible for a while on nightly, although the feature itself is very much work-in-progress.

GATs

At this point we can create associated types generic solely over lifetimes:

RUST

```rust
trait Foo {
    type Bar<'a, 'b, 'c>
        where 'a: 'b;
}
```

GATs

At this point we can create associated types generic solely over lifetimes:

RUST

```rust
trait Foo {
    type Bar<'a, 'b, 'c>
        where 'a: 'b;
}
```

... but, as the name of the feature suggests, eventually we'll be able to construct arbitrarily-generic associated types.

GATs

Thanks to GATs, in the future we'll be able to create structures generic over - for instance - pointer types:

RUST

```
trait PointerFamily {
    type Pointer<T>: Deref<Target = T>;
}
```

GATs

RUST

```
trait PointerFamily {
    type Pointer<T>: Deref<Target = T>;
}

struct ArcFamily;

impl PointerFamily for ArcFamily {
    type Pointer<T> = Arc<T>;
}

struct RcFamily;

impl PointerFamily for RcFamily {
    type Pointer<T> = Rc<T>;
}
```

GATs

RUST

```rust
trait PointerFamily {
    type Pointer<T>: Deref<Target = T>;
}

struct ArcFamily;

impl PointerFamily for ArcFamily {
    type Pointer<T> = Arc<T>;
}

struct RcFamily;

impl PointerFamily for RcFamily {
    type Pointer<T> = Rc<T>;
}

struct Foo<P: PointerFamily> {
    bar: P::Pointer<String>,
}
```

(example from `RFC 1598` @ https://github.com/rust-lang/rfcs/pull/1598)

GATs

Bonus acronym: initially this feature was called `associated type constructors` (**ATCs**).

ZSTs

What do you think will be the output of this code?

RUST

```rust
use std::mem::size_of;

struct Struct;

enum Enum { }

fn main() {
    println!("{}", size_of::<Struct>());
    println!("{}", size_of::<Enum>());
    println!("{}", size_of::<()>());
    println!("{}", size_of::<!>());
}
```

ZSTs

Yeah, correct:

```
error[E0658]: the `!` type is experimental
   |
   |     println!("{}", size_of::<!>());
   |                              ^
   |
   = note: see issue #35121 for more information
   = help: add `#![feature(never_type)]` to the crate attributes to
enable
```

ZSTs

RUST

```rust
#![feature(never_type)]

use std::mem::size_of;

struct Struct;

enum Enum { }

fn main() {
    println!("{}", size_of::<Struct>());
    println!("{}", size_of::<Enum>());
    println!("{}", size_of::<()>());
    println!("{}", size_of::<!>());
}
```

ZSTs

Yeah, all those types are **literally empty**:

```
0
0
0
0
```

ZSTs

ZST stands for **zero-sized type**.

ZSTs

ZST stands for **zero-sized type**.

... and they are hella useful!

ZSTs

For instance, the `()` (called `unit type`) is used by the Rust's standard library to implement `HashSet`, reusing code from `HashMap`:

RUST

```rust
pub struct HashSet<T, S = RandomState> {
    map: HashMap<T, (), S>,
}
```

Since both Rust and LLVM know that such map contains only keys, all the additional code gets striped out - yay **zero-cost abstractions**!

ZSTs

By the way, `()` is both a value, and a type:

RUST

```rust
fn main() {
                // vv value
    let foo: () = ();
            // ^^ type

    println!("{:?}", foo); // ()
}
```

ZSTs

There exists a similar type, `!` (called `never type`), which serves a similar purpose, with one difference: you can't obtain a value of this type.

ZSTs

Let’s talk: `Result<String, ()>`.

ZSTs

Let's talk: `Result<String, ()>`:

RUST

```
fn print_me(val: Result<String, ()>) {
    match val {
        Ok(val) => println!("ok: {:?}", val),
        Err(val) => println!("err: {:?}", val),
    }
}

fn main() {
    print_me(Ok("pancake".into())); // ok: "pancake"
    print_me(Err(()));             // err: ()
}
```

ZSTs

Let’s talk: `Result<String, !>`.

ZSTs

Let’s talk: `Result<String, !>`:

RUST

```
fn print_me(val: Result<String, !>) {
    match val {
        Ok(val) => println!("ok: {:?}", val),
        Err(val) => println!("err: {:?}", val),
    }
}

fn main() {
    print_me(Ok("pancake".into()));
    print_me(Err(!)); // compile-time error
}
```

For `Result<String, !>` there’s *no way* to construct the `Err` variant.

ZSTs

As an example, we can use `!` to implement a non-failing `FromStr`:

RUST

```rust
use std::str::FromStr;

struct Person(String);

impl FromStr for Person {
    type Err = !;

    fn from_str(str: &str) -> Result<Self, !> {
        Ok(Person(
            str.into()
        ))
    }
}

fn main() {
    let Ok(person) = Person::from_str("Tommy Wiseau");

    // ^ no need to `.unwrap()`, because Rust understands
    // ^ that the `Err` variant cannot be possibly constructed

    println!("Oh hi, {}!", person.0);
}
```

ZSTs

As another example, we *will be able to* use `!` (called `never type`) to implement a non-failing `FromStr`:

RUST

```rust
use std::str::FromStr;

struct Person(String);

impl FromStr for Person {
    type Err = !;

    fn from_str(str: &str) -> Result<Self, !> {
        Ok(Person(
            str.into()
        ))
    }
}

fn main() {
    let Ok(person) = Person::from_str("Tommy Wiseau");

    // ^ no need to `.unwrap()`, because Rust understands
    // ^ that the `Err` variant cannot be possibly constructed

    println!("Oh hi, {}!", person.0);
}
```

ZSTs

Currently the compiler cannot yet fully reason about the `!`:

```
error[E0005]: refutable pattern in local binding: `Err(_)` not
covered
    |
    |     let Ok(person) = Person::from_str("Tommy Wiseau");
    |         ^^^^^^^^^^ pattern `Err(_)` not covered
```

The work on this feature is still ongoing though, so fingers crossed it gets merged soon!

DSTs

DST stands for **dynamically-sized type**.

DSTs

DST stands for **dynamically-sized type**.

You've for sure had the chance to use them tons of times:

- `str` (but not `&str` or `String`),
- `[T]` (but not `[T; n]`, `&[T]` or `Vec<T>`),
- `dyn Trait` (but not `&dyn Trait`).

DSTs

DST stands for **dynamically-sized type**.

You’ve for sure had the chance to use them tons of times:

- `str` (but not `&str` or `String`),
- `[T]` (but not `[T; n]`, `&[T]` or `Vec<T>`),
- `dyn Trait` (but not `&dyn Trait`).

… but there’s also *one* more.

DSTs

What's the size of this type?

RUST

```rust
struct NamedSlice<'a, T> {
    name: String,
    slice: &'a [T],
}
```

DSTs

What’s the size of this type?

RUST

```rust
struct NamedSlice<'a, T> {
    name: String,
    slice: &'a [T],
}
```

24 bytes for `String` + 8 bytes for `&[T]` + padding = **40 bytes**.

(counted using `std::mem::size_of()` on a x86-64)

DSTs

What's the size of *this* type?

RUST

```rust
struct NamedSlice<T> {
    name: String,
    slice: [T], // look, ma! no reference
}
```

DSTs

What's the size of *this* type?

RUST

```
struct NamedSlice<T> {
    name: String,
    slice: [T],
}
```

First things first: **this is legal**; it's fine for a struct's *last* field to be unsized.

DSTs

What's the size of *this* type?

```rust
struct NamedSlice<T> {
    name: String,
    slice: [T],
}
```

First things first: **this is legal**; it's fine for a struct's *last* field to be unsized.

Second things second: **this struct is** `!Sized`.

DSTs

RUST

```rust
use std::mem::size_of;

struct NamedSlice<T> {
    name: String,
    slice: [T],
}

fn main() {
    println!("{}", size_of::<NamedSlice<String>>());
}
```

DSTs

```
error[E0277]: the size for values of type `[String]` cannot be
              known at compilation time
    |
    |     println!("{}", size_of::<NamedSlice<String>>());
    |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |                    doesn't have a size known at
    |                    compile-time
    |
```

DSTs

RUST

```rust
struct NamedSlice<T> {
    name: String,
    slice: [T],
}

fn main() {
    let ns = NamedSlice {
        name: "named".into(),
        slice: [1, 2, 3] as _,
    };
}
```

DSTs

```
error[E0277]: the size for values of type `[_]` cannot be known
              at compilation time
   |
   |       let ns = NamedSlice {
   |  ______________^
   | |         name: "named".into(),
   | |         slice: [1, 2, 3] as _,
   | |     };
   | |_____^ doesn't have a size known at compile-time
```

ICE CTA

What do you think the compiler will say about this code?

RUST

```rust
fn main() {
    break rust;
}
```

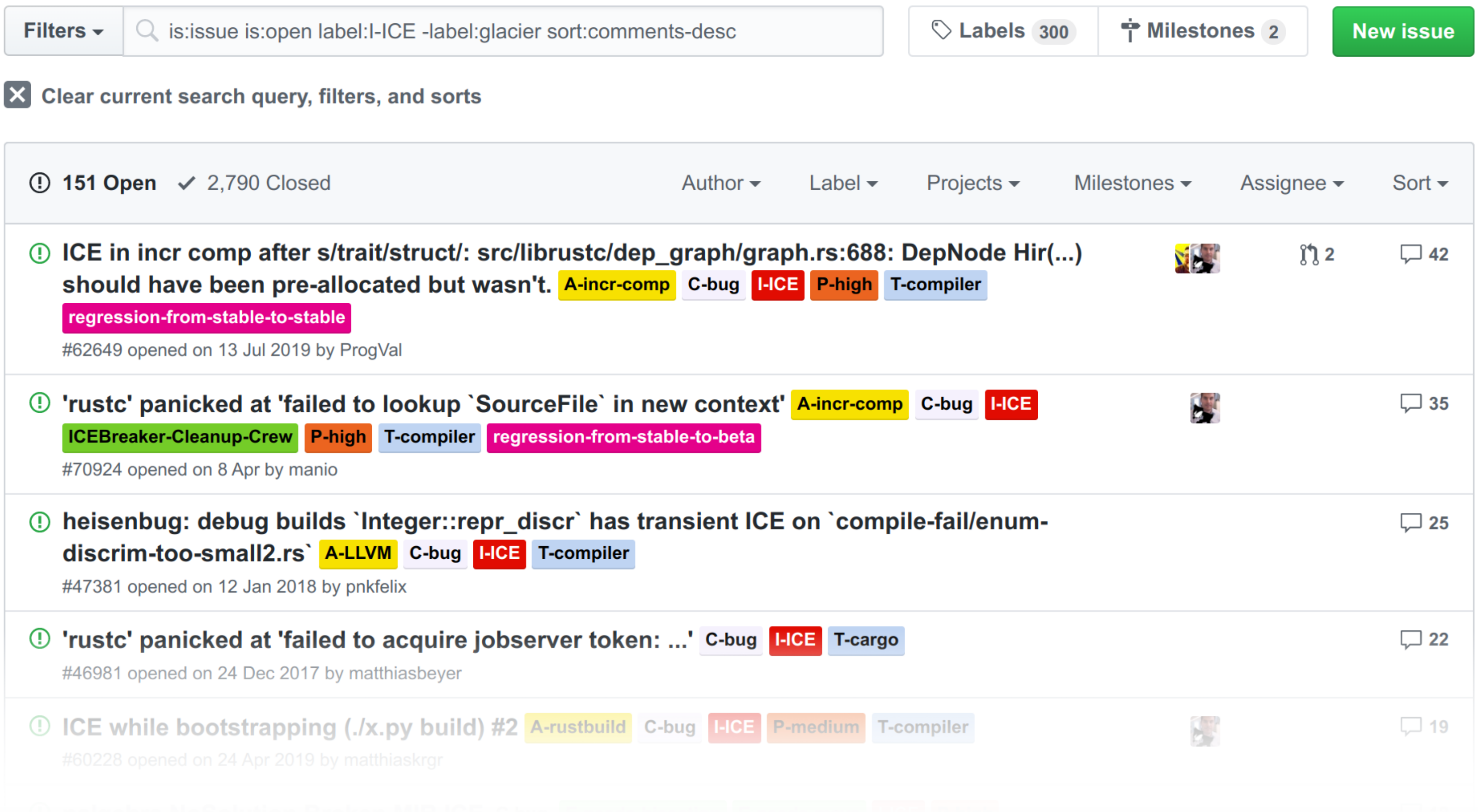
ICE CTA

Yes, `rustc` has easter eggs:

```
❯ rustc test.rs
error[E0425]: cannot find value `rust` in this scope
 --> test.rs:2:11
  |
2 |     break rust;
  |           ^^^^ not found in this scope

error[E0268]: `break` outside of a loop
 --> test.rs:2:5
  |
2 |     break rust;
  |     ^^^^^^^^^^ cannot `break` outside of a loop

error: internal compiler error: It looks like you're trying to break rust; would you like some ICE?

note: the compiler expectedly panicked. this is a feature.

note: we would appreciate a joke overview: https://github.com/rust-lang/rust/issues/43162#issuecomment-320764675

note: rustc 1.44.0-nightly (dbf8b6bf1 2020-04-19) running on x86_64-unknown-linux-gnu

error: aborting due to 3 previous errors

Some errors have detailed explanations: E0268, E0425.
For more information about an error, try `rustc --explain E0268`.
```

ICE CTA

And, yes, `rustc` has bugs too:



ICE CTA

ICE: internal compiler error

ICE CTA

ICE: internal compiler error

CTA: call to action

My *iced-tea* for you

Try contributing to `rustc`, `cargo`, `rustfmt` or any other project you find useful - all of them could use a little bit more love!

Bonus points for fixing an actual ICE in the compiler, but really: even a single, small commit can improve your (or someone else's!) workflow and make (your or someone else's) life better.

To sum up

Scary acronym	Comforting expansion
ATC	associated type constructor (→ GAT)
CTA	call to action
DST	dynamically-sized type
GAT	generic associated type
HRTB	higher-rank trait bound
ICE	internal compiler error
OIBIT	opt-in built-in trait (→ auto trait)
ZST	zero-sized type

Scary Acronyms (and Super Creeps)

~ Patryk Wychowaniec, 2020

Thank you!