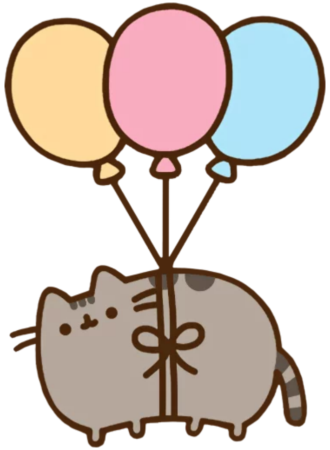
Rust China Tour

×

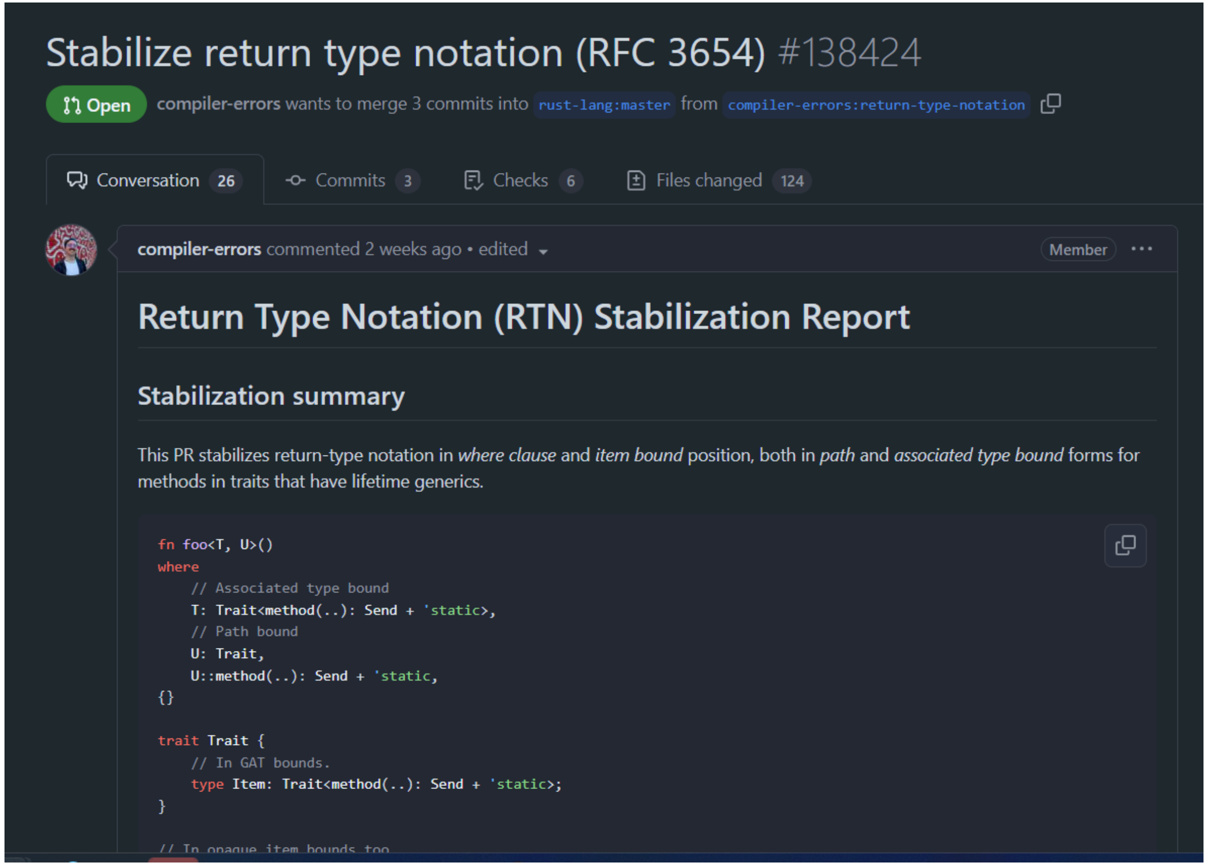
TUNA

New Stuffs in Trait

喵喵



Motivation for this talk…



Stabilize return type notation (RFC 3654) #138424

compiler-errors wants to merge 3 commits into `rust-lang:master` from `compiler-errors:return-type-notation`

compiler-errors commented 2 weeks ago • edited

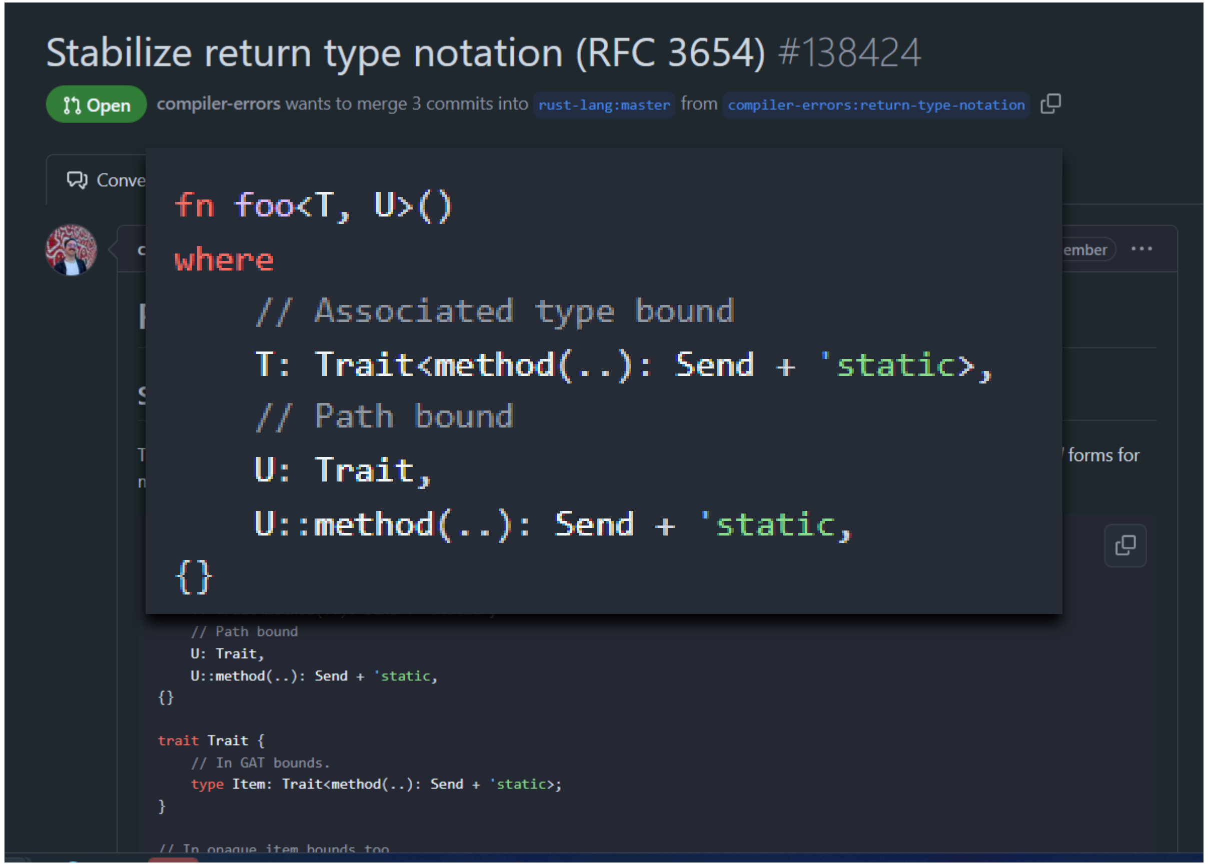
Return Type Notation (RTN) Stabilization Report

Stabilization summary

This PR stabilizes return-type notation in *where clause* and *item bound* position, both in *path* and *associated type bound* forms for methods in traits that have lifetime generics.

```
fn foo<T, U>()
where
    // Associated type bound
    T: Trait<method(..): Send + 'static>,
    // Path bound
    U: Trait,
    U::method(..): Send + 'static,
{}

trait Trait {
    // In GAT bounds.
    type Item: Trait<method(..): Send + 'static>;
}

// In opaque item bounds too
```

Motivation for this talk…



Stabilize return type notation (RFC 3654) #138424

Open compiler-errors wants to merge 3 commits into `rust-lang:master` from `compiler-errors:return-type-notation`

```rust
fn foo<T, U>()
where
    // Associated type bound
    T: Trait<method(..): Send + 'static>,
    // Path bound
    U: Trait,
    U::method(..): Send + 'static,
{}
```

```rust
    // Path bound
    U: Trait,
    U::method(..): Send + 'static,
{}
```

```rust
trait Trait {
    // In GAT bounds.
    type Item: Trait<method(..): Send + 'static>;
}

// In opaque item bounds too
```

???

Trait

- Typeclasses (for static dispatch and ML folks)
- Interfaces (for dynamic dispatch and Java folks)

Trait

- Typeclasses (for static dispatch and ML folks)
- Interfaces (for dynamic dispatch and Java folks)

```
trait Animal {
  fn eat(&mut self);
}
trait Cat : Animal {
  fn meow(&self) -> String;
}
struct Ouroboros;
impl Animal for Ouroboros {
  fn eat(&mut self) { self.eat(); }
}
```

Trait

```
fn pat<M: Cat>(meow: &mut M) { meow.meow(); }
```

Trait

```
fn pat<M: Cat>(meow: &mut M) { meow.meow(); }

fn pat<M>(meow: &mut M) where M: Cat { meow.meow(); }
```

Trait

```
fn pat<M: Cat>(meow: &mut M) { meow.meow(); }

fn pat<M>(meow: &mut M) where M: Cat { meow.meow(); }

fn pat(meow: &mut dyn Cat) { meow.meow(); }
```

Trait

```
fn pat<M: Cat>(meow: &mut M) { feed(meow); }

fn pat<M>(meow: &mut M) where M: Cat { feed(meow); }

fn pat(meow: &mut dyn M) { feed(meow); }


fn feed_bound<A: Animal>(meow: &mut dyn A) { /* ... */ }
fn feed_dyn(meow: &mut dyn Animal) { /* ... */ }
```

Trait

```
fn pat<M: Cat>(meow: &mut M) { feed(meow); }

fn pat<M>(meow: &mut M) where M: Cat { feed(meow); }

fn pat(meow: &mut dyn M) { feed(meow); }


fn feed_bound<A: Animal + ?Sized>(meow: &mut dyn A) { /* */ }
fn feed_dyn(meow: &mut dyn Animal) { /* ... */ }
```

Trait

```
fn pat<M: Cat>(meow: &mut M) { feed(meow); }

fn pat<M>(meow: &mut M) where M: Cat { feed(meow); }

fn pat(meow: &mut dyn M) { feed(meow); }
```

```
fn feed_bound<A: Animal + ?Sized>(meow: &mut dyn A) { /* */ }
fn feed_dyn(meow: &mut dyn Animal) { /* ... */ }
```

```
error[E0658]: cannot cast `dyn Cat` to `dyn Animal`, trait
upcasting coercion is experimental
```

Upcast

Upcast

```
dyn Derived -> dyn Base
```

Upcast

`dyn Derived -> dyn Base`

```cpp
#include <cstdint>
#include <iostream>

struct Base {
  uint64_t var;
};
struct Left : Base {
  uint64_t get() { return var; }
};
struct Right : Base {
  void set(uint64_t i) { var = i; }
};
struct Center : public Left, public Right {};
```

Upcast

There is data stored in…

Upcast

There is data stored in…

-pointers

… namely the vtable

Trait object upcasting support

- New vtable format s.t. subtraits can navigate to vtable of supertraits from their own vtable
- New unsized coercion rules: `dyn T -> dyn U` where `T: U`
  - Allows `&dyn T -> &dyn U`, `Box<dyn T> -> Box<dyn U>`, so on.

Trait object upcasting support

- New vtable format s.t. subtraits can navigate to vtable of supertraits from their own vtable
- New unsized coercion rules: `dyn T -> dyn U` where `T: U`
  - Allows `&dyn T -> &dyn U`, `Box<dyn T> -> Box<dyn U>`, so on.

Stablized on Feb 8, next stable

The coloring problem

```rust
fn opt<T>(pred: bool, v: T) -> Option<T> {
    if pred { Some(v) } else { None }
}
```

```rust
const fn opt<T>(pred: bool, v: T) -> Option<T> {
    if pred { Some(v) } else { None }
}
```

```rust
const fn opt<T>(pred: bool, v: T) -> Option<T> {
    if pred { Some(v) } else { None }
}
```

```
error[E0493]: destructor of `T` cannot be evaluated at compile-time
```

```
#![feature(const_destruct)]
#![feature(const_trait_impl)]

fn opt<T>(pred: bool, v: T) -> Option<T>
  where T: ~const std::marker::Destruct
{
    if pred { Some(v) } else { None }
}
```

`~const Trait`

Const implementable traits

```
#[const_trait]
trait Tr {
    fn meow(self);
}
struct M;
impl const Tr for M {
    fn meow(self) {}
}

const fn test<T: ~const Tr>(v: T) {
    v.meow()
}
```

“Part of the trait is const”

[1] https://rustc-dev-guide.rust-lang.org/effects.html

[2] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

“Part of the trait is const”

- ~~Too bad, try zig~~
-
-

[3] https://rustc-dev-guide.rust-lang.org/effects.html
[4] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

“Part of the trait is const”

- ~~Too bad, try zig~~
- Recall that `const` means “**CAN** be evaluated at compile time”
-

[5] https://rustc-dev-guide.rust-lang.org/effects.html

[6] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

“Part of the trait is const”

- ~~Too bad, try zig~~
- Recall that `const` means “**CAN** be evaluated at compile time”
- Wait for keyword generics (`#![feature(effects)]`), maybe stablized in Rust 2099.

[7] https://rustc-dev-guide.rust-lang.org/effects.html

[8] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

“Part of the trait is const”

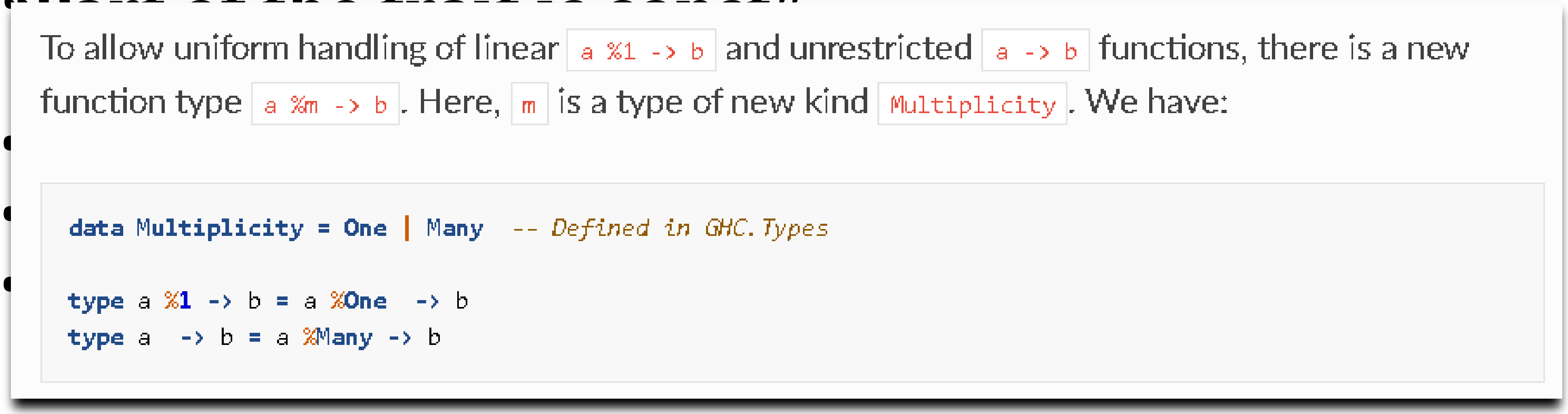
- ~~Too bad, try zig~~
- Recall that `const` means “**CAN** be evaluated at compile time”
- Wait for keyword generics (`#![feature(effects)]`), maybe stablized in Rust 2099.

Compiler dev guide[9] & “Extending Rust’s Effect System” by Yoshua Wuyts[10]

[9] https://rustc-dev-guide.rust-lang.org/effects.html

[10] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

“Part of the trait is const”



To allow uniform handling of linear `a %1 -> b` and unrestricted `a -> b` functions, there is a new function type `a %m -> b`. Here, `m` is a type of new kind `Multiplicity`. We have:

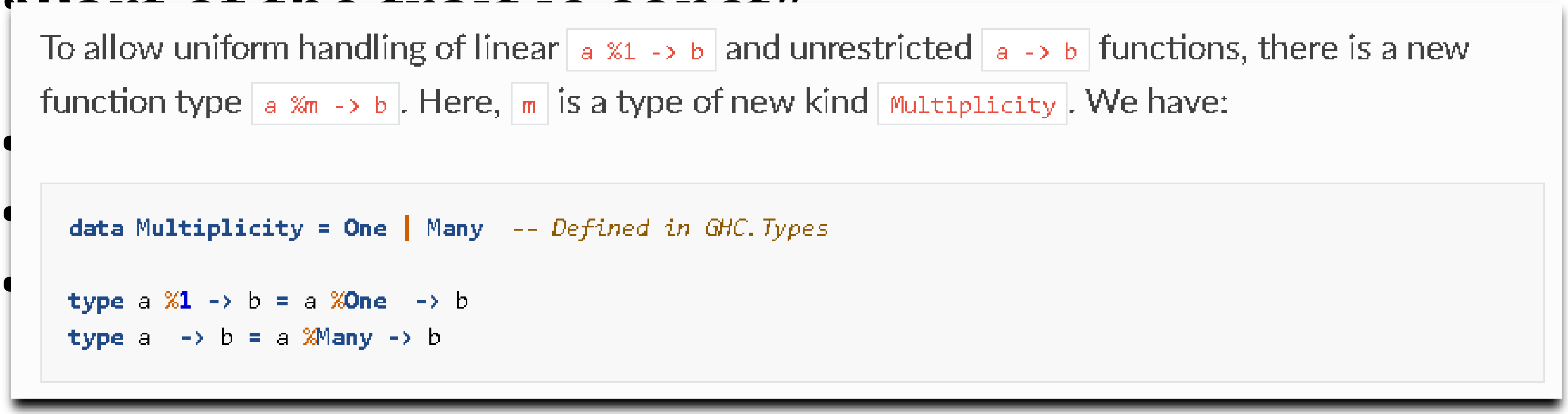
```
data Multiplicity = One | Many  -- Defined in GHC.Types

type a %1 -> b = a %One  -> b
type a  -> b = a %Many -> b
```

Compiler dev guide[11] & “Extending Rust’s Effect System” by Yoshua Wuyts[12]

[11] https://rustc-dev-guide.rust-lang.org/effects.html

[12] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

“Part of the trait is const”



To allow uniform handling of linear `a %1 -> b` and unrestricted `a -> b` functions, there is a new function type `a %m -> b`. Here, `m` is a type of new kind `Multiplicity`. We have:

```
data Multiplicity = One | Many  -- Defined in GHC.Types

type a %1 -> b = a %One  -> b
type a  -> b = a %Many -> b
```

Compiler dev guide[12] & “Extending Rust’s Effect System” by Yoshua W

New keyword -> New sort -> Polymorphism!

[12] https://rustc-dev-guide.rust-lang.org/effects.html

[12] https://blog.yoshuawuyts.com/extending-rusts-effect-system/

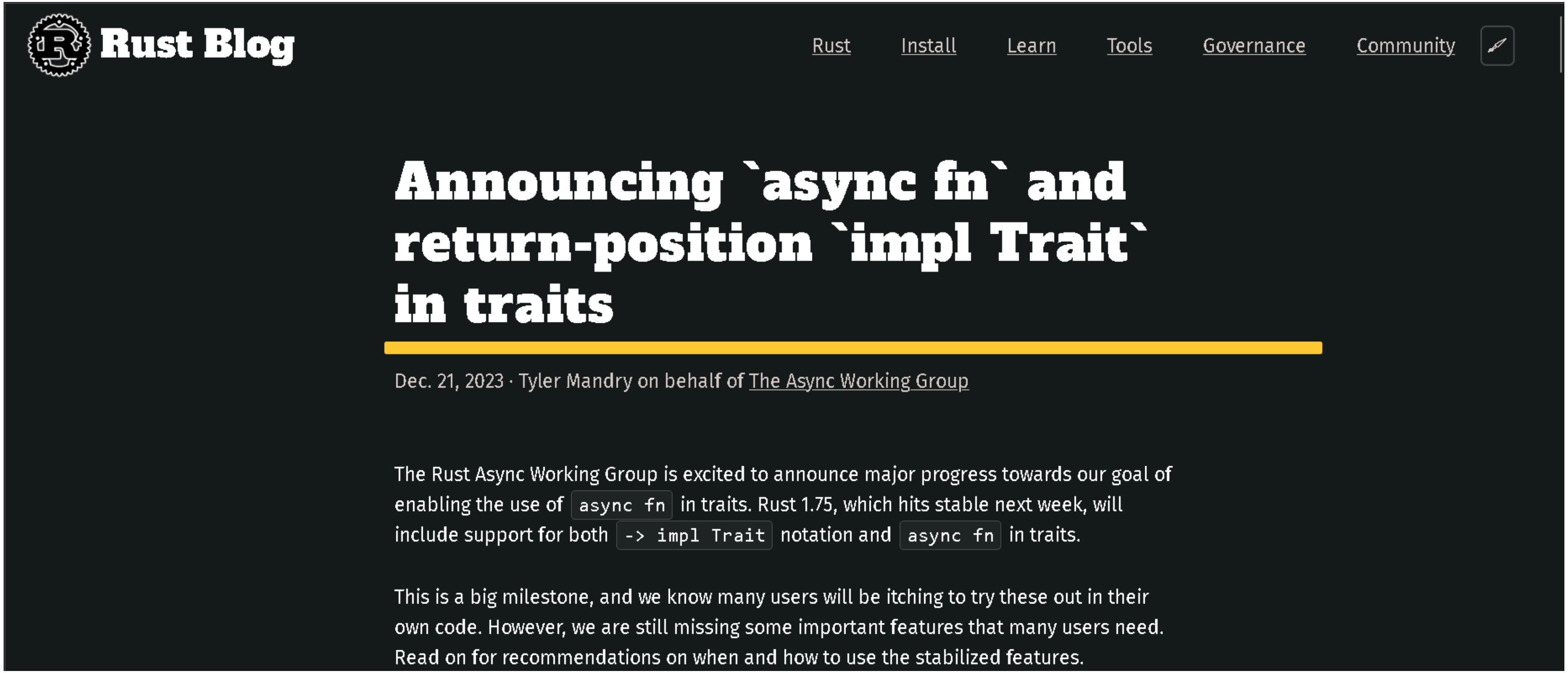
What about async

Traditionally...

```rust
trait Bad {
  async fn bad(&self) -> i32;
}

trait Good {
  fn bad(&self) -> Box<dyn Future<Output = i32>>;
}
```

What about async

What about async



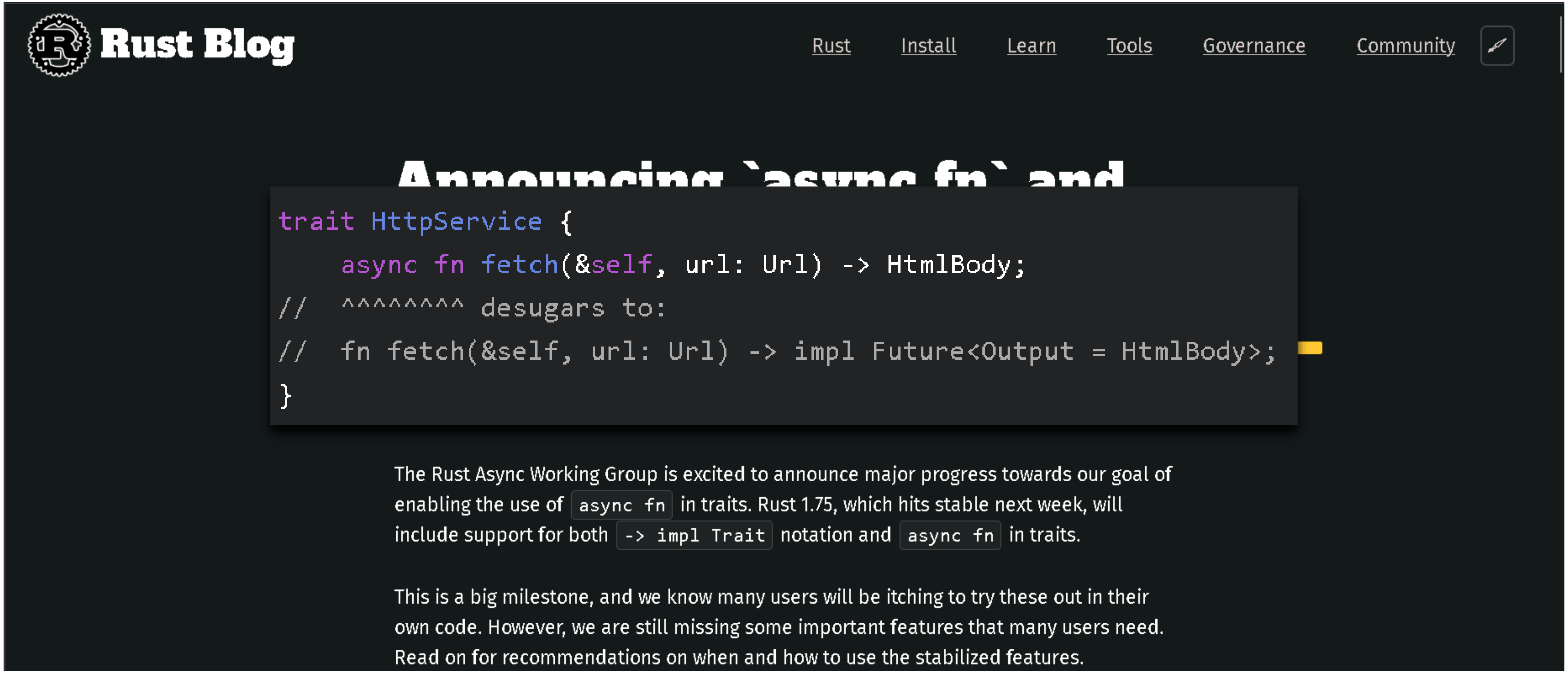
Announcing `async fn` and return-position `impl Trait` in traits

Dec. 21, 2023 · Tyler Mandry on behalf of The Async Working Group

The Rust Async Working Group is excited to announce major progress towards our goal of enabling the use of `async fn` in traits. Rust 1.75, which hits stable next week, will include support for both `-> impl Trait` notation and `async fn` in traits.

This is a big milestone, and we know many users will be itching to try these out in their own code. However, we are still missing some important features that many users need. Read on for recommendations on when and how to use the stabilized features.

What about async

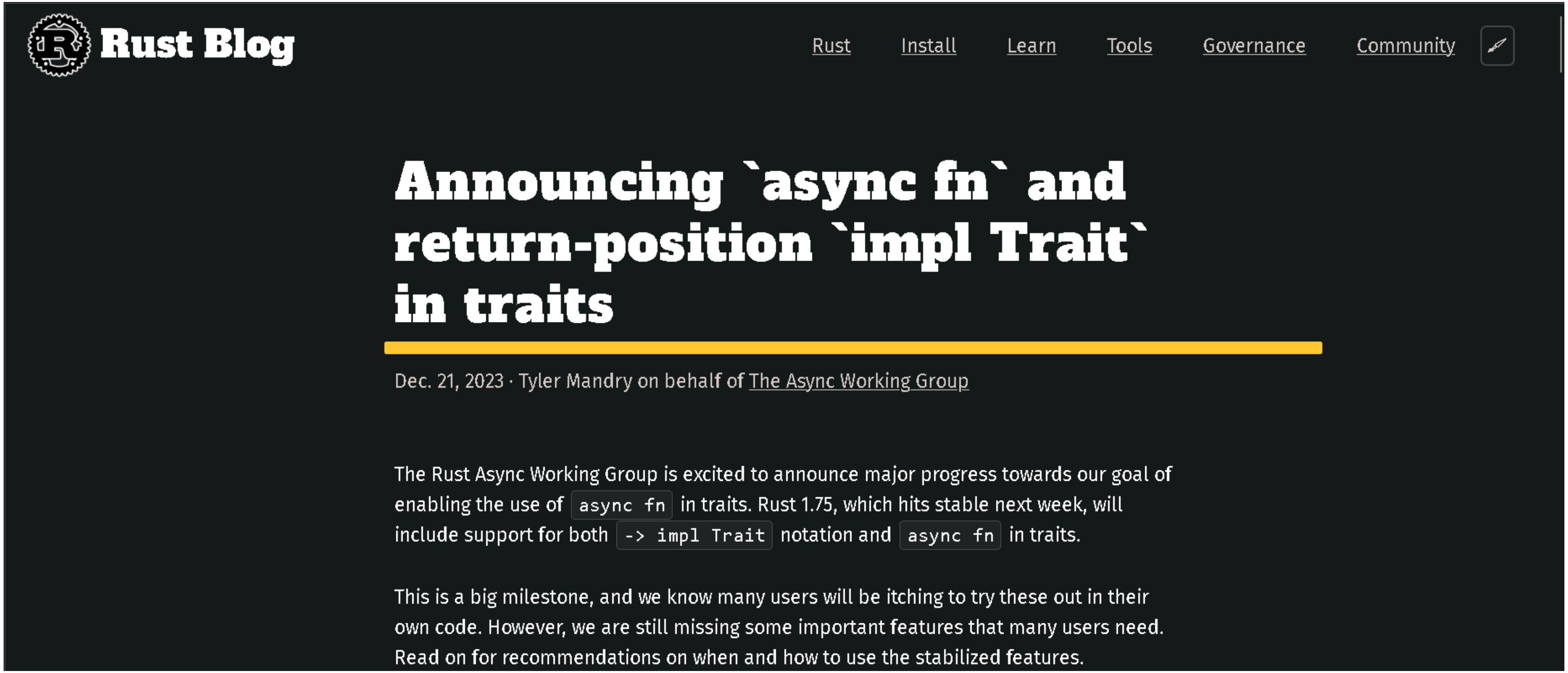


```
trait HttpService {
    async fn fetch(&self, url: Url) -> HtmlBody;
//  ^^^^^^^^ desugars to:
//  fn fetch(&self, url: Url) -> impl Future<Output = HtmlBody>;
}
```

The Rust Async Working Group is excited to announce major progress towards our goal of enabling the use of `async fn` in traits. Rust 1.75, which hits stable next week, will include support for both `-> impl Trait` notation and `async fn` in traits.

This is a big milestone, and we know many users will be itching to try these out in their own code. However, we are still missing some important features that many users need. Read on for recommendations on when and how to use the stabilized features.

What about async



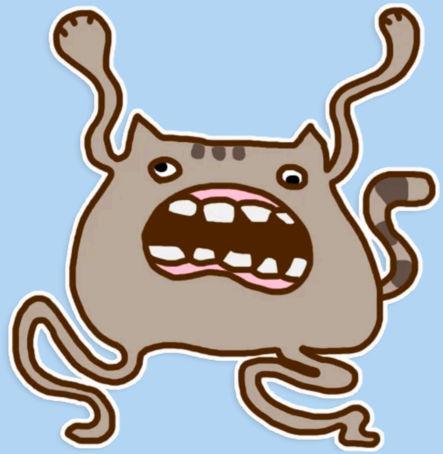
Announcing `async fn` and return-position `impl Trait` in traits

Dec. 21, 2023 · Tyler Mandry on behalf of The Async Working Group

The Rust Async Working Group is excited to announce major progress towards our goal of enabling the use of `async fn` in traits. Rust 1.75, which hits stable next week, will include support for both `-> impl Trait` notation and `async fn` in traits.

This is a big milestone, and we know many users will be itching to try these out in their own code. However, we are still missing some important features that many users need. Read on for recommendations on when and how to use the stabilized features.

Those (types) who cannot be named



Desugaring AFIT

AFIT

```
trait Meow {
  type Item: Copy;
  async fn meow(&self) -> Self::Item;
}
```

Desugaring AFIT

AFIT → RPITIT

```
trait Meow {
  type Item: Copy;
  fn meow(&self) -> impl Future<Output = Self::Item>;
}
```

Desugaring AFIT

AFIT → RPITIT→ Anonymous [G]AT

```
trait Meow {
  type Item: Copy;
  type __fut__: Future<Output = Self::Item>;
  fn meow(&self) -> Self::__fut__;
}
```

Desugared impl

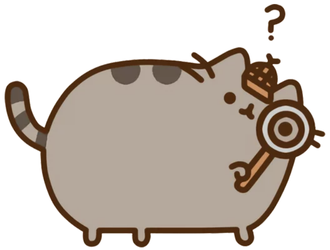
```
impl Meow for T {
  type Item = i32;
  type __fut__ = impl Future<Output = Self::Item>;
  fn meow(&self) -> Self::__fut__ {
    async move { 42 }
  }
}
```

Desugared impl

```
impl Meow for T {
  type Item = i32;
  type __fut__ = impl Future<Output = Self::Item>;
  fn meow(&self) -> Self::__fut__ {
    async move { 42 }
  }
}
```

Wait for **`impl Trait` in associated type**.

Question time!



https://layered.meow.plus