Future Feature Fun

Josh Triplett

2026-09-16

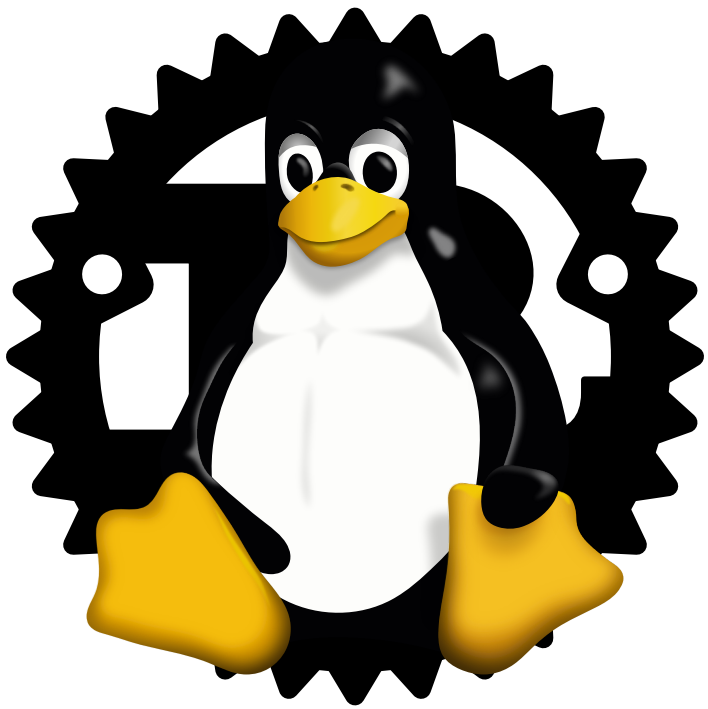
josh@joshtriplett.org

RfL has a long list of necessary features

https://github.com/Rust-for-Linux/linux/issues/2

This is about all the *other* features

We are Kangrejos!



¡Somos Kangrejos!

Example of what I left out:
Trait evolution and other “evolution” features

Not needed in the kernel: “stable API nonsense”

Useful in the standard library and other crates

All of these features are works in progress!

All of these features are works in progress!

Some more than others!

Intentionally glossing over fine details

If you're excited about these,
please speak up!

If you want to work on them,
get in touch!

View types: borrowing **part** of a struct

```
let a = &mut my_struct.a;
let b = &mut my_struct.b;
b.method_b();
a.method_a();
```

```
impl MyStruct {
   fn operate_on_b(&mut self) {
      self.b.method_b();
   }
}
```

```
impl MyStruct {
  fn operate_on_b(&mut self) {
    self.b.method_b();
  }
}


let a = &mut my_struct.a;
my_struct.operate_on_b(); // ERROR
a.method_a();
```

This discourages modular code
and helper methods

Example: FUSE and Git

Example: FUSE and Git

Built a test app to mount a git tree

Example: FUSE and Git

Built a test app to mount a git tree

Then tried to wrap it in an object with methods

Example: FUSE and Git

Built a test app to mount a git tree

Then tried to wrap it in an object with methods

Almost made me bounce off of Rust!

```
impl MyStruct {
   fn operate_on_b(&mut{b} self) {
      self.b.method_b();
   }
}
```

```
impl MyStruct {
  fn operate_on_b(&mut{b} self) {
    self.b.method_b();
  }
}

let a = &mut my_struct.a;
my_struct.operate_on_b(); // OK
a.method_a();
```

View types status: in progress, tracking issue

Pattern types

Pattern types

Encode your invariants in the type system

Pattern types

Encode your invariants in the type system

Generalization of `NonZero`/`NonNull`

```
type Errno = i32 is 0..4096;

type Nanoseconds = u32 is 0..1_000_000_000;

fn fail(...) -> Result is Err(_)
```

(Eventually) literals can have pattern types

What type is `42u64`?

`u64 is 42`

Safely construct a `NonZero` without `unwrap`,
because the compiler knows it isn't `0`

Pattern types are on nightly.

Pattern types do need help!

Turbofishing `Into`

```
x.into::<u64>();
```

Turbofishing `Into`

```
x.into::<u64>();
```

Library team hoping to add this in the next edition
(Approach requires no language change)

Linear types / “unforgettable” types

Linear types / “unforgettable” types

“Hot potato”, can’t let it implicitly drop

Linear types / “unforgettable” types

“Hot potato”, can’t let it implicitly drop

Database transaction: must either commit or roll back

Linear types: In progress

Was blocked on type system work.
Less blocked now that we have the new trait solver.
Types team can always use help!

`is` operator

let chaining:

```
while let Some(x) = func().await && x.cond() {
```

let chaining:

```
while let Some(x) = func().await && x.cond() {
```

Convenient!

let chaining:

```
while let Some(x) = func().await && x.cond() {
```

Convenient!

...kinda inside-out, isn't it?

```
while func().await is Some(x) && x.cond() {
```

```rust
while func().await is Some(x) && x.cond() {
```

```rust
opt.is_none()
opt.is_some()
opt.is_some_and(|v| v.cond())
```

```
while func().await is Some(x) && x.cond() {
```

```
opt.is_none()
opt.is_some()
opt.is_some_and(|v| v.cond())
```

```
opt is None
opt is Some(_)
opt is Some(v) && v.cond()
```

```
while func().await is Some(x) && x.cond() {
```

```
opt.is_none()
opt.is_some()
opt.is_some_and(|v| v.cond())
```

```
opt is None
opt is Some(_)
opt is Some(v) && v.cond()
```

```
my_enum is MyVariant
```

`is` operator: lang experiment in progress

Polonius - next generation borrow checker

```rust
let a = &mut my_struct.a;
let b = &mut my_struct.b;
b.method_b();
a.method_a();

println!("{my_struct:?}");
```

Did not work on Rust 1.0.

```rust
let a = &mut my_struct.a;
let b = &mut my_struct.b;
b.method_b();
a.method_a();

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

Did not work on Rust 1.0.

```rust
let a = &mut my_struct.a;
let b = &mut my_struct.b;
b.method_b();
a.method_a();

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

Did not work on Rust 1.0.

Works today because of Non-Lexical Lifetimes (NLL)

```rust
fn find_one<'m>(
  m: &'m mut HashMap<u32, u32>,
  keys: &[u32]
) -> Option<&'m mut u32> {
  for k in keys {
    if let Some(v) = m.get_mut(k) {
      return Some(v);
    }
  }
  None
}
```

Stable Rust:

```
for k in keys {
    if let Some(v) = m.get_mut(k) {
                     ^ `*m` was mutably borrowed here in
                       the previous iteration of the loop
        return Some(v)
               ------- returning this value requires
                       that `*m` is borrowed for `'m`
```

Knowing this is safe depends on control flow

```rust
fn find_one<'m>(
  m: &'m mut HashMap<u32, u32>,
  keys: &[u32]
) -> Option<&'m mut u32> {
  for k in keys {
    if let Some(v) = m.get_mut(k) {
      return Some(v);
    }
  }
  None
}
```

NLL handled borrows not defined by lexical scope

Polonius handles control flow

Polonius Alpha - enabled on nightly Rust

This code compiles on nightly Rust!

Needs performance optimization before stabilizing.

comptime and macro fn

`const fn` can run at compile time or runtime

`comptime fn` can **only** run at compile time

`const fn` can run at compile time or runtime

`comptime fn` can **only** run at compile time

This means we can talk to the compiler

`macro_rules!` is its own declarative language.

No loops, no structures, no conditions
only AST matching and recursion

`macro_rules!` is its own declarative language.

No loops, no structures, no conditions
only AST matching and recursion

What if you could write them in normal Rust
with Rust control flow and data structures?

`macro_rules!` is its own declarative language.

No loops, no structures, no conditions
only AST matching and recursion

What if you could write them in normal Rust
with Rust control flow and data structures?

Without proc macros?

`macro fn`

Compile-Time Function Evaluation (CTFE)
while we're still expanding macros

`macro fn`

Compile-Time Function Evaluation (CTFE)
while we're still expanding macros

Credit to Oli for suggesting this was possible

Currently a project goal

`comptime fn` in progress

Plan to work on `macro fn` once comptime ships

Guaranteed tail calls

Requires changing drop order to drop **before** tail call

Still need to nail down the syntax

A few more:

- `#[loop_match]` and `#[const_continue]`

A few more:

- `#[loop_match]` and `#[const_continue]`
- no panic/sleep/etc, via function “coloring”

A few more:

- `#[loop_match]` and `#[const_continue]`
- no panic/sleep/etc, via function “coloring”
- “full” const generics (numeric operations, enums)

https://joshtriplett.org/

Message me on the Rust Zulip:
https://rust-lang.zulipchat.com/

Image Credits:
https://github.com/MariaLetta/free-ferris-pack
https://commons.wikimedia.org/wiki/File:Rust_for_Linux_logo.svg

100% human-written presentation