PROCEDURAL MACRO IN THE LINUX KERNEL 🦀

vincenzopalazzo

vincenzopalazzo@member.fsf.org

Spain 17-09-2023

```
➜ ~ whoami
```

“Random Person with Ramdom Side Project!”

SOME OF MY CURRENT INTEREST

SOME OF MY CURRENT INTEREST

- Working on the Rust compiler
  - Project Leader of Macros Working Group
  - Project Member of Async Working Group

SOME OF MY CURRENT INTEREST

- Working on the Rust compiler
    - Project Leader of Macros Working Group
    - Project Member of Async Working Group
- Working in the linux kernel through the Rust for Linux Initiative;

“C macros are difficult to read”

A macro is a name given to a block of C statements as a pre-processor directive.

```rust
while let Some(token) = body_it.next() {
        match token {
            TokenTree::Ident(ident) if ident.to_string() == "fn" => {
                let fn_name = match body_it.next() {
                    Some(TokenTree::Ident(ident)) =>
                          ident.to_string(),
                    _ => continue,
                };
                functions.insert(fn_name);
            }
            // ...
            _ => (),
        }
    }
```

```rust
macro_rules! quote_spanned {
    ($span:expr => $($tt:tt)*) => {{
        let mut tokens;
        #[allow(clippy::vec_init_then_push)]
        {
            tokens = ::std::vec::Vec::new();
            let span = $span;
            quote_spanned!(@proc tokens span $($tt)*);
        }
        ::proc_macro::TokenStream::from_iter(tokens)
    }};
    (@proc $v:ident $span:ident) => {};
    ....
}
```

```
➜ ~ ls -la linux/rust/macros
```

INTO THE KERNEL RIGHT NOW

- All the macros are defined in the same directory; (good for now)

- All the macros are defined in the same directory; (good for now)
- All the macros are parsing almost the same syntax; (impl/struct)

- All the macros are defined in the same directory; (good for now)
- All the macros are parsing almost the same syntax; (impl/struct)

```rust
#[proc_macro]
pub fn foo(body: TokenStream) -> TokenStream {
    for tt in body.into_iter() {
        match tt {
            TokenTree::Ident(_) => eprintln!("Ident"),
            TokenTree::Punct(_) => eprintln!("Punct"),
            TokenTree::Literal(_) => eprintln!("Literal"),
            _ => {}
        }
    }
    return TokenStream::new();
}
```

Drawbacks

- Code duplication

Drawbacks

- Code duplication
- Bigger patch when there is a new syntax to support. (Good for seek jobs)

Drawbacks

- Code duplication
- Bigger patch when there is a new syntax to support. (Good for seek jobs)
- There is no common pattern, so everyone use their own mental pattern for parsing

Drawbacks

- Code duplication
- Bigger patch when there is a new syntax to support. (Good for seek jobs)
- There is no common pattern, so everyone use their own mental pattern for parsing
- Copy and Paste do not work without **eprintln**

```
➜ ~ emacs -nw kernel/kproc_macros/README.md
```

SO, WE ARE FUCK UP?

“Luckily no (maybe)”

The Rust ecosystem has a well known libraries

The Rust ecosystem has a well known libraries

- Parsing the stream of tokens syn

The Rust ecosystem has a well known libraries

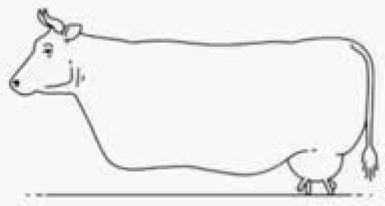
- Parsing the stream of tokens syn
- Formatting the result of the proc macro quote

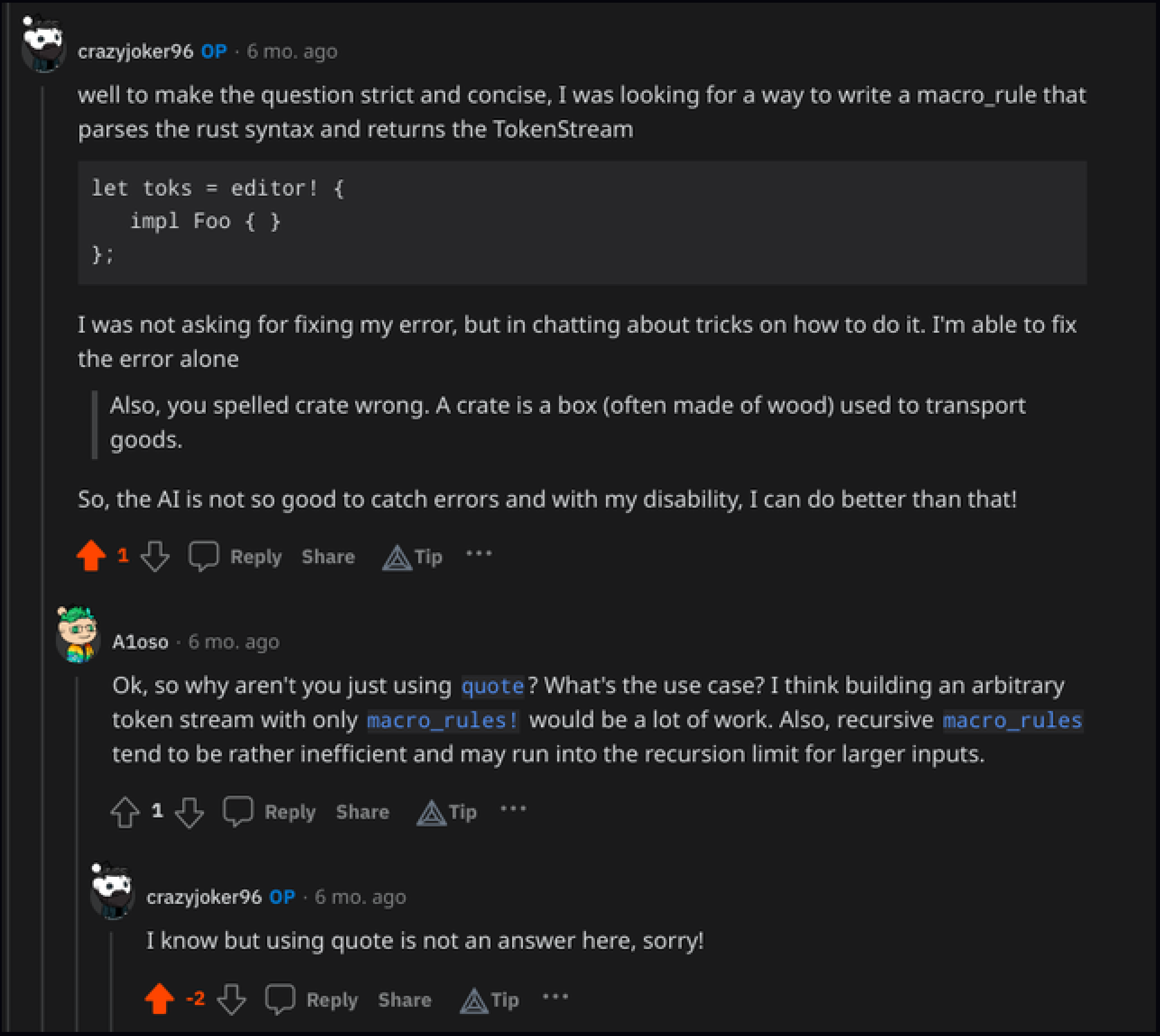
The Rust ecosystem has a well known libraries

- Parsing the stream of tokens syn
- Formatting the result of the proc macro quote

10 years of rust just 2 library?

- If your code works fine don't touch it
+ my code:




crazyjoker96 OP · 6 mo. ago

well to make the question strict and concise, I was looking for a way to write a macro_rule that parses the rust syntax and returns the TokenStream

```
let toks = editor! {
   impl Foo { }
};
```

I was not asking for fixing my error, but in chatting about tricks on how to do it. I'm able to fix the error alone

> Also, you spelled crate wrong. A crate is a box (often made of wood) used to transport goods.

So, the AI is not so good to catch errors and with my disability, I can do better than that!

1 Reply Share Tip ···

A1oso · 6 mo. ago

Ok, so why aren't you just using `quote`? What's the use case? I think building an arbitrary token stream with only `macro_rules!` would be a lot of work. Also, recursive `macro_rules` tend to be rather inefficient and may run into the recursion limit for larger inputs.

1 Reply Share Tip ···

crazyjoker96 OP · 6 mo. ago

I know but using quote is not an answer here, sorry!

-2 Reply Share Tip ···

SO, IN THE KERNEL WE SHOULD USE SYN AND QUOTE?

SO, IN THE KERNEL WE SHOULD USE SYN AND QUOTE?

“Eventually yes, but why we do not take the time experiment with a new lib?”

SO, IN THE KERNEL WE SHOULD USE SYN AND QUOTE?

“Eventually yes, but why we do not take the time experiment with a new lib?”

“No, we can just use quote”

There is a PR in the kernel #1007 +78,232 -25

There is a PR in the kernel #1007 +78,232 -25

That import also a wrapper of the rust API **proc_macro2** (for no reason for the kernel)

```
➜ ~ git commit -S -s -m 'rust: use kproc-macros every.."'
```

RFC: INTRODUCE AN NEW DEVELOPED LIBRARY

“Following the pattern of the kernel we call it kproc_macros”

3th iteration later ..

Goals

Goals

- Do not replace **syn**;

Goals

- Do not replace **syn**;
- Do not make an solution similar to **syn**;

Goals

- Do not replace **syn**;
- Do not make an solution similar to **syn**;
- Made experimentation on how improve the proc macro in general.

Goals

- Do not replace **syn**;
- Do not make an solution similar to **syn**;
- Made experimentation on how improve the proc macro in general.

Goals

Goals

- Be able to trace the macro parsing (debugging, understanding);

Goals

- Be able to trace the macro parsing (debugging, understanding);
- Import from the parser what we need, (useless now, but with different subsystem may be helpful);

Goals

- Be able to trace the macro parsing (debugging, understanding);
- Import from the parser what we need, (useless now, but with different subsystem may be helpful);
- Be able to build quote in the language (already in nightly), or

Goals

- Be able to trace the macro parsing (debugging, understanding);
- Import from the parser what we need, (useless now, but with different subsystem may be helpful);
- Be able to build quote in the language (already in nightly), or
- Be able to have a version of quote build with kproc-macro itself (needs reseach)

Goals

- Be able to trace the macro parsing (debugging, understanding);
- Import from the parser what we need, (useless now, but with different subsystem may be helpful);
- Be able to build quote in the language (already in nightly), or
- Be able to have a version of quote build with kproc-macro itself (needs reseach)
- Be able to remove proc_macro2 only in tests, and use rust proc-macro API

Goals

- Be able to trace the macro parsing (debugging, understanding);
- Import from the parser what we need, (useless now, but with different subsystem may be helpful);
- Be able to build quote in the language (already in nightly), or
- Be able to have a version of quote build with kproc-macro itself (needs reseach)
- Be able to remove proc_macro2 only in tests, and use rust proc-macro API
- Be able to cache proc macro metadata around proc-macro. See 44034

How the user code looks like

```rust
#[derive(RustBuilder)]
pub struct BooLifetimeDyn<'a> {
    #[allow(dead_code)]
    attr: String,
    #[allow(dead_code)]
    self_ref: u32,
    #[allow(dead_code)]
    gen: Vec<&'a dyn GenTrait>,
}
```

How the proc macro looks like

```rust
struct Tracer;
impl KParserTracer for Tracer {
    fn log(&self, msg: &str) {
        eprintln!("\x1b[93mkproc-tracing\x1b[1;97m {msg}");
    }
}

#[proc_macro_derive(RustBuilder, attributes(build))]
pub fn derive_rust(input: TokenStream) -> TokenStream {
    let tracer = DummyTracer {};
    let parser = RustParser::with_tracer(&tracer);
    let ast = parser.parse_struct(&input);
    let toks = generate_impl(&ast);
    trace!(tracer, "{}", toks);
    toks
}
```

Open Problems

- How the generate_impl looks like? (Into a string or with quote like solution)

Open Problems

- How the generate_impl looks like? (Into a string or with quote like solution)
- How I can print errors while parsing? or while generating the code?

```
➜ ~ cat kproc_macros/exaperiments/README.md
```

DOGFOOTING

```
let plugin = plugin! {
    state: State::new(),
    dynamic: true,
    notification: [],
    methods: [],
    hooks: [],
};
plugin.start();
```

```rust
#[rpc_method(
    rpc_name = "foo_macro",
    description = "This is a simple and short description"
)]
pub fn foo_rpc(plugin: &mut Plugin<State>, request: Value) -> Result<Va
    let response = json!({"is_dynamic": plugin.dynamic, "rpc_request":
    Ok(response)
}
```

User library: lexopt-derive

```rust
pub fn generate_impl(struct_tok: &StructToken) -> TokenStream {
    let gen = if let Some(str_gen) = &struct_tok.generics {
        format!("{}", str_gen)
    } else {
        "".to_owned()
    };
    let name_attr = &struct_tok.fields[0].identifier;
    let ty = struct_tok.fields[0].ty.to_string();
    let code = format!(
        "impl{} {}{} {{ \
                    fn get_{name_attr}(&self) -> {ty} {{ \
                       return self.{name_attr}.clone()\
                    }} \
                       \
                    fn set_{name_attr}(&self, inner: {ty}) {{ }}
                }}",
        gen, struct_tok.name, gen
```

```
editor!{
   @foreach ${attributes} {
     println!("{}", ${ir});
   }
}
```

```
editor!{
   @foreach ${attributes} {
     println!("{}", ${ir});
   }
}
```

Or just finish to implement quote in the std

Please complain at https://github.com/rsmicro/kproc-macros

THANKS!