The Cost of Zero Cost

2026-03-04

Florian “Florob” Zeitz

Outline

1 Definitions 3
2 Compilation Flow 6
3 Macros 8
4 println!() 14
5 Iterators 19

1 Definitions

Abstraction

Abstraction

Representation of a high level concept, hiding the low level implementation.

Zero-Cost Abstraction

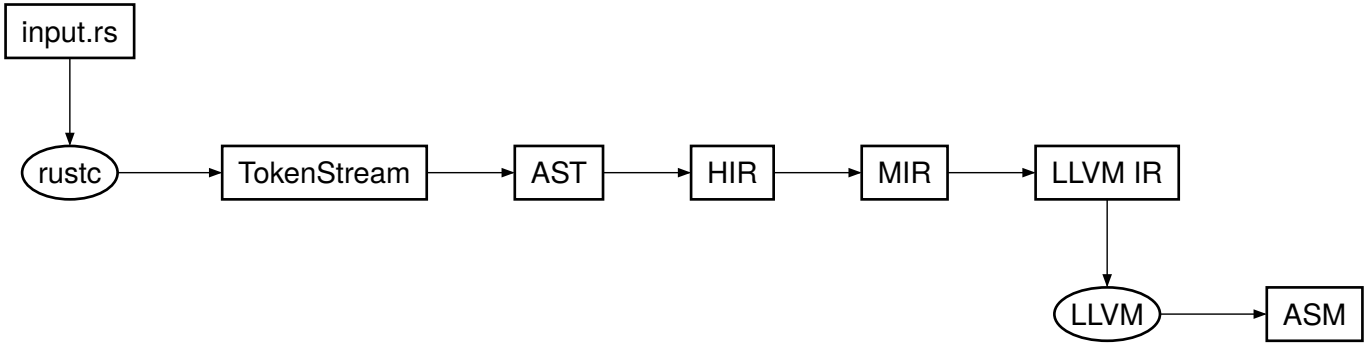
Zero-Cost Abstraction

An abstraction where every operation on it is as efficient as you could have written by hand using lower level concepts.

- No additional CPU overhead
- No additional memory overhead
- No additional code size
- Rust heavily relies on optimization for this

2 Compilation Flow

All the Representations



- Various internal representations
- These have (potentially lossy) textual representations for debugging

3 Macros

Overview

- Way to extend Rust with new functionality and syntax
- Transcribes input tokens to output tokens
- As efficient as their definition

Macros by Example


```
macro_rules! matches {
    ($expression:expr, $pattern:pat $(if $guard:expr)? $(,)?) => {
        match $expression {
            $pattern $(if $guard)? => true,
            _ => false
        }
    };
}
```

```
matches!(expr, Some(x) if x == 3);
// becomes:
match expr {
    Some(x) if x == 3 => true,
    _ => false,
}
```

Procedural Macros

- Transforms a stream of tokens by executing a function on it
- Realized as compiler plugins
- Three types:
  - Function-like: `funcy!(...)`
  - Derive macros: `#[derive(CustomDerive)]`
  - Attribute macros: `#[CustomAttr]`

Compiler Built-ins

- Basically procedural macros part of the main compiler
- Work on the actual AST instead of a token stream
- Has access to additional information

Expanding Macros

- `cargo +nightly rustc -- -Z unpretty=expanded`

E.g. Hello World:

```
#![feature(prelude_import)]
extern crate std;
#[prelude_import]
use std::prelude::rust_2024::*;
fn main() {
    {
        ::std::io::_print(format_args!("Hello, world!\n"));
    };
}
```

4 println!()

Formatting Syntax


```rust
println!("{}/{}/{}", month, day, year);
println!("{1}/{2}/{0} == {0}-{1}-{2}", year, month, day);
println!("{m}/{d}/{y} == {y}-{m}-{d}", y=year, m=month, d=day);
println!("{y}-{m:02}-{d:02}", y=year, m=month, d=day);
```

- Rust uses ICU-style format strings
- `{}` as placeholders
- Arguments can be indexed and named
- Flags, width, precision after a `:`

Formatting flow

- Template string and arguments are passed to `format_args!()` compiler built-in
- `format_args!()` creates a `fmt::Arguments` struct
- `fmt::write(&mut dyn fmt::Write, fmt::Arguments<'_>)` performs the actual formatting

Expanding to HIR

- **H**igh-Level **I**ntermediate **R**epresentation
- Compiler-friendly AST representation
- `cargo +nightly rustc -- -Z unpretty=hir`

```
::std::io::_print({
    super let args = (&year, &month, &day);
    super let args =
        [format_argument::new_display(args.0),
                format_argument::new_display(args.1),
                format_argument::new_display(args.2)];
    unsafe {
        format_arguments::new(
            b"\xc3\x20\x00\x00\x69\x02\x00\x01-\xc3\x20\x00\x00\x69\x02\x00\x01-\
              \xc3\x20\x00\x00\x69\x02\x00\x01\n\x00",
            &args
        )
    }
});
```

Expanding to MIR

- **M**id-Level **I**ntermediate **R**epresentation
- Control-flow graph of basic blocks
- No nested expressions
- All types fully explicit
- Subject to optimization passes
- `cargo +nightly rustc -- -Z unpretty=mir`

5 Iterators

Iterator Protocol


```rust
pub trait Iterator {
    type Item;

    // Required method
    fn next(&mut self) -> Option<Self::Item>;

    // Provided methods
    // ...
}
```

- Iterators are progressed by repeated calls to `next()`
- Everything uses iterators
- For loops just call `next()` unit it returns `None`

for-Loop

```
for i in 0..255 {
    puts(i);
}
```

desugars to (HIR):

```
match into_iter(Range { start: 0, end: 255 }) {
    mut iter => loop {
        match next(&mut iter) {
            None {} => break,
            Some { 0: i } => {
                puts(i);
            }
        }
    },
}
```

Outputting LLVM-IR

- LLVM's IR, produced by rustc, consumed by LLVM
- Static single-assignment form
- Easiest to show with `cargo-show-asm`
- `cargo asm --llvm --everything`

Outputting ASM

- Textual representation of CPU instructions
- Easiest to show with `cargo-show-asm`
- `cargo asm --everything`

Questions?



https://babelmonkeys.de/~florob/talks/RC-2026-03-04-zero-cost.pdf

Thank you for your attention. Any questions?