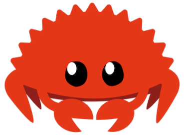
Rust

for JS/TS developers



Miguel Palhas (@naps62) / Subvisual

DAVID HEINEMEIER HANSSON

September 7, 2023

Open source hooliganism and the TypeScript meltdown

Types in the past

```
int main() {
  int x = 0;
  long int y = (long) x;

  println("%d", y);
}
```

1. slow to compile
2. type inference wasn't a thing
3. tooling

```
unresolved external symbol "void __cdecl
importStoredClients(class
std::basic_fstream<char,struct std::char_traits<char> > const
&,class
std::vector<class Client,class std::allocator<class Client> >
&)" (?
importStoredClients@@YAXABV?$basic_fstream@DU?
$char_traits@D@std@@@std@@AAV?
$vector@VClient@@V?$allocator@VClient@@@std@@@2@@Z)
referenced in function
_main DataTracker
```

```
1 + "2" == "12"
```

Undefined is not a function

2. Types of Typings

Structural Typing (Typescript, OCaml, …)

```
type Foo = { x: number, y: string };
type Bar = { x: number };

let x: Foo = { x: 1, y: "hello" };

// works
let y: Bar = x;

console.log(y)
// => { x: 1, y: "hello" }
```

Nominal Typing (Rust, C/C++, ...)

```rust
struct Foo { x: i32, y: String };
struct Bar { x: i32 };

fn main() {
  let x: Foo = Foo { x: 1, y: "hello".to_string() };

  // this would not compile
  // let y: Bar = x;

  // works (explicit conversion needs to be defined)
  let y: Bar = x.into();
}
```

What makes Rust good?

1. Types, but comfortably

```rust
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

```rust
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

```rust
let list: Vec<u32> =
  vec![1, 2, 3]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

integer types are easily inferred (u8, 16, u32, …)

```rust
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

```rust
let list: Vec<u32> =
  vec![1, 2, 3]
    .iter()
    .map(|v| v + 1)
    .collect::<Vec<u32>>()
```

closure arguments are inferred 99% of the time

```rust
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

```rust
let list: Vec<u32> =
  vec![1, 2, 3]
    .iter()
    .map(|v| v + 1)
    .collect::<Vec<_>>()
```

elements in collections are inferred 99% of the time

```rust
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

```rust
let list: Vec<u32> =
  vec![1, 2, 3]
    .iter()
    .map(|v| v + 1)
    .collect()
```

actually, the entire `collect` type and `list` type are redundant

```rust
let list: Vec<u32> =
  vec![1u32, 2u32, 3u32]
    .iter()
    .map(|v: u32| v + 1)
    .collect::<Vec<u32>>()
```

```rust
let list: Vec<_> =
  vec![1, 2, 3]
    .iter()
    .map(|v| v + 1)
    .collect()
```

and the `Vec` element itself is inferred from numbers

Type inference eliminates most noise.

Exceptions: function headers; ambiguity.

```rust
fn increment_and_dedup(v: Vec<u32>) -> HashSet<u32> {
  v.iter().map(|v| v + 1).collect()
}
```

2. Borrow checker

Ownership and References

```typescript
type Foo = { x: number };

function add(foo: Foo) {
  foo.x += 1;
}
```

```rust
struct Foo { x: u32 }

fn add(foo: Foo) {
  foo.x += 1;
}
```

error[E0594]: cannot assign to `foo.x`, as `foo` is not declared as mutable

Ownership and References

```
type Foo = { x: number };

function add(foo: Foo) {
  foo.x += 1;
}
```

```
struct Foo { x: u32 }

fn add(foo: &Foo) {
  foo.x += 1;
}
```

error[E0594]: cannot assign to `foo.x`, which is behind a & reference

Ownership and References

```typescript
type Foo = { x: number };

function add(foo: Foo) {
  foo.x += 1;
}
```

```rust
struct Foo { x: &mut u32 }

fn add(foo: mut Foo) {
  foo.x += 1;
}
```

3. Algebraic types

Product types

when you have one thing AND another thing

```
struct Rectangle {
  width: u32,
  height: u32
}
```

Sum types

when you have one thing OR another thing

```rust
enum Option<T> {
  None,
  Some(T)
}
```

```rust
enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

4. Zero cost abstractions

The ability to use higher-level features without incurring additional runtime cost.

The trade-off: compile-time complexity

“If it compiles, it works”

not to be taken literally

it’s how strongly typed programming **feels**

"Making illegal states unrepresentable"

Aim for compile-time enforcements instead of runtime validations

- Type-drive development;
- Abuse `Option`, `Result`, and `enum`;
- Typestate pattern.

5. Tooling

```
cargo build
cargo run --package serve
cargo +nightly clippy
cargo fmt
cargo test
cargo build --target wasm32-unknown-unknown
cargo audit
bacon
```

Clippy is awesome

suspicious_arithmetic_impl

What it does

Lints for suspicious operations in impls of arithmetic operators, e.g. subtracting elements in an Add impl.

Why is this bad?

This is probably a typo or copy-and-paste error and not intended.

Example

```
impl Add for Foo {
    type Output = Foo;

    fn add(self, other: Foo) -> Foo {
        Foo(self.0 - other.0)
    }
}
```

[rust-analyzer live demo]

Why NOT Rust?

Compilation times?

Half-true

- Incremental compilation is great(ish)
- Not quite instant-reload, but rather close
- Release builds are more painful
- You should `cargo check` instead of `cargo build`

Refactoring is a slog?

Closed • 858 total votes

643	Rafactoring in Rust is Easy
215	Refactoring in Rust is Hard

Voting closed 4 days ago

Best-Idiot • 11 days ago

When you change anything, a strong type system tells you what places still need to be fixed to adapt to the change

79 Reply Share ···

Sw429 • 10 days ago

Compare this to something like python, where you won't know that you missed changing something until you get a runtime exception for it, which sometimes might be missed until it's live in production.

31 Reply Share ···

Not suitable for quick prototyping?

Only true if you haven’t been fully indoctrinated yet

What’s “built with Rust”?

just

```
alias b := build

host := `uname -a`

# build main
build:
    cc *.c -o main

# test everything
test-all: build
    ./test --all

# run a specific test
test TEST: build
    ./test --test {{TEST}}
```

```
: just -l                                        ~
Available recipes:
    build     # build main
    b         # alias for `build`
    test TEST # run a specific test
    test-all  # test everything
:                                                ~
```

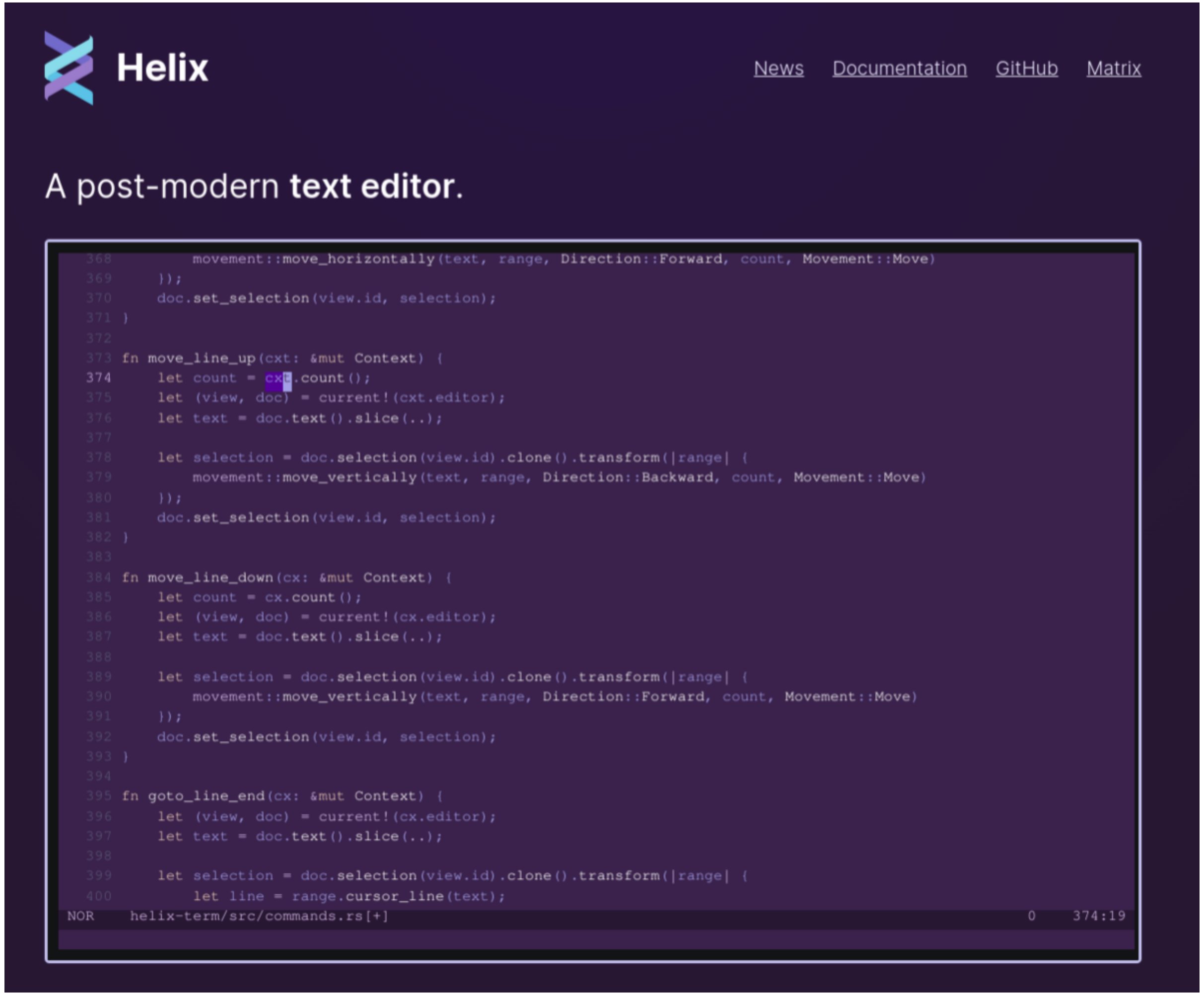
delta

```
redis-cli.c

static sds cliReadLine(int fd) {

    while(1) {
        char c;
        ssize_t ret;

        if (read(fd,&c,1) == -1) {
        ret = read(fd,&c,1);
        if (ret == -1) {
            sdsfree(line);
            return NULL;
        } else if (c == '\n') {
        } else if ((ret == 0) || (c == '\n')) {
            break;
        } else {
            line = sdscatlen(line,&c,1);
```



Helix
News
Documentation
GitHub
Matrix
A post-modern text editor.
movement::move_horizontally(text, range, Direction::Forward, count, Movement::Move)
});
doc.set_selection(view.id, selection);
}
fn move_line_up(cxt: &mut Context) {
let count = cxt.count();
let (view, doc) = current!(cxt.editor);
let text = doc.text().slice(..);
let selection = doc.selection(view.id).clone().transform(|range| {
movement::move_vertically(text, range, Direction::Backward, count, Movement::Move)
});
doc.set_selection(view.id, selection);
}
fn move_line_down(cx: &mut Context) {
let count = cx.count();
let (view, doc) = current!(cx.editor);
let text = doc.text().slice(..);
let selection = doc.selection(view.id).clone().transform(|range| {
movement::move_vertically(text, range, Direction::Forward, count, Movement::Move)
});
doc.set_selection(view.id, selection);
}
fn goto_line_end(cx: &mut Context) {
let (view, doc) = current!(cx.editor);
let text = doc.text().slice(..);
let selection = doc.selection(view.id).clone().transform(|range| {
let line = range.cursor_line(text);
NOR helix-term/src/commands.rs[+]
0 374:19

Typst

```
#set page(width: 10cm, height: auto)
#set heading(numbering: "1.")

= Fibonacci sequence
The Fibonacci sequence is defined through the
recurrence relation $F_n = F_(n-1) + F_(n-2)$.
It can also be expressed in _closed form:_

$ F_n = round(1 / sqrt(5) phi.alt^n), quad
  phi.alt = (1 + sqrt(5)) / 2 $

#let count = 8
#let nums = range(1, count + 1)
#let fib(n) = (
  if n <= 2 { 1 }
  else { fib(n - 1) + fib(n - 2) }
)

The first #count numbers of the sequence are:

#align(center, table(
  columns: count,
  ..nums.map(n => $F_#n$),
  ..nums.map(n => str(fib(n))),
))
```

1. Fibonacci sequence

The Fibonacci sequence is defined through the recurrence relation $F_n = F_{n-1} + F_{n-2}$. It can also be expressed in *closed form:*

$$F_n = \left\lfloor \frac{1}{\sqrt{5}} \phi^n \right\rceil, \quad \phi = \frac{1 + \sqrt{5}}{2}$$

The first 8 numbers of the sequence are:

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
1	1	2	3	5	8	13	21

TAURI

Build an optimized, secure, and frontend-independent application for multi-platform deployment.

Bash PowerShell Cargo npm Yarn pnpm

```
$ sh <(curl https://create.tauri.app/sh)
```

Quick Start

Rust

for JS/TS developers

Miguel Palhas / @naps62