Writing a Rust program

WITHOUT A SINGLE LINE OF CODE

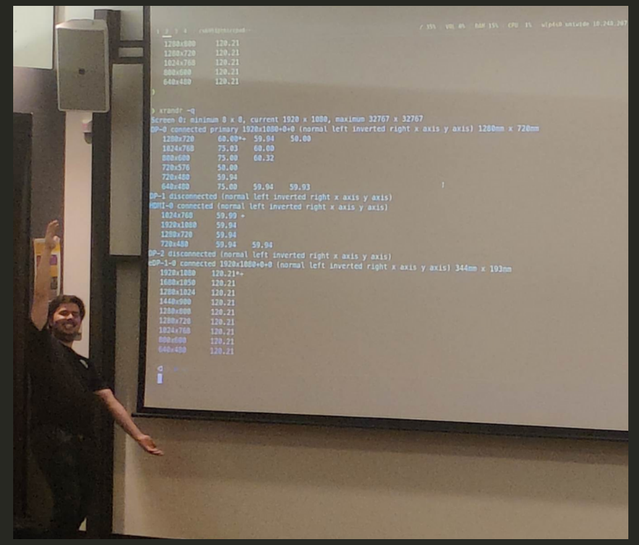
ZAC KOLOGLU

@INSOU22

Hi!

I teach

COMP6991 @ UNSW



I engineer

Backend software

I car

(sometimes)

What is a line of code?

```
let x = 42;
```

```
let x = 42;
```

```
struct Person {
    name: String,
    age: u8,
    height_cm: f32,
}
```

```
struct Person {
    name: String,
    age: u8,
    height_cm: f32,
}
```

```
println!("Hello, World!");
```

```
println!("Hello, World!");
```

```
trait Hello {
  type World;
}
```

```
trait Hello {
    type World;
}
```

```rust
impl Default for Person {
  fn default() -> Self {
    Self {
      name: String::from("Ferris"),
      age: 8,
      height_cm: 16.4,
    }
  }
}
```

```rust
impl Default for Person {
  fn default() -> Self {
    Self {
      name: String::from("Ferris"),
      age: 8,
      height_cm: 16.4,
    }
  }
}
```

```rust
impl Default for Person {
  fn default() -> Self {
    Self {
      name: String::from("Ferris"),
      age: 8,
      height_cm: 16.4,
    }
  }
}
```

```
impl Hello for Person {
    type World = Self;
}
```

```
impl Hello for Person {
    type World = Self;
}
```

So how can we write code?


HELL
WELCOME TO HELL.
HUH. LESS FIRE THAN I THOUGHT.
HELL IS UNIQUE TO EACH INDIVIDUAL.
OK. WHAT'S MINE?
CLICK
YOU HAVE TO WRITE RUST, ONLY WITH TYPES
struct False;
struct True;
impl T for Fal
where T: Foo {
type Output
#13
HEY BUDDY COMICS

Let's Implement OR with types

```
struct False;
struct True;
```

structs are our values

```
trait Or {
    type Output;
}
```

traits are our functions

```
impl Or for (False, False) {
    type Output = False;
}

impl Or for (False, True) {
    type Output = True;
}

impl Or for (True, False) {
    type Output = True;
}

impl Or for (True, True) {
    type Output = True;
}
```

impl's implement our functions

```rust
type FnCall1 = <(False, False) as Or>::Output;
type FnCall2 = <(False, True ) as Or>::Output;

fn main() {
    println!("{}", std::any::type_name::<FnCall1>());
    println!("{}", std::any::type_name::<FnCall2>());
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.27s
     Running `target/debug/playground`

playground::False
playground::True
```

Type resolution calls functions

Hold up…

```
type FnCall1 = <(False, False) as Or>::Output;
type FnCall2 = <(False, True ) as Or>::Output;

fn main() {
    println!("{}", std::any::type_name::<FnCall1>());
    println!("{}", std::any::type_name::<FnCall2>());
}
```

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.27s
     Running `target/debug/playground`

playground::False
playground::True
```

That's definitely code

Type resolution calls functions

Fine, watch this...

```
trait Never {}
```

a trait that is never impl'd...

```
trait Print: Never {}
```

…and a trait for printing

```
type FnCall = <(False, False) as Or>::Output;

impl Print for FnCall {}
```

impl the Print trait for FnCall

```
❯ cargo build
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
error[E0277]: the trait bound `True: Never` is not satisfied
  --> src/lib.rs:29:6
   |
29 | impl Print for FnCall {}
   |      ^^^^^ the trait `Never` is not implemented for `True`
   |
note: required by a bound in `Print`
  --> src/lib.rs:2:14
   |
2  | trait Print: Never {}
   |              ^^^^^ required by this bound in `Print`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `no_code` due to previous error
```

and build prints our value!

```
❯ cargo build
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
error[E0277]: the trait bound `True: Never` is not satisfied
  --> src/lib.rs:29:6
   |
29 | impl Print for FnCall {}
   |      ^^^^^ the trait `Never` is not implemented for `True`
   |
note: required by a bound in `Print`
  --> src/lib.rs:2:14
   |
2  | trait Print: Never {}
   |              ^^^^^ required by this bound in `Print`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `no_code` due to previous error
```

no really, see!?

```
❯ cargo build 2>&1 \
    | sed -En '/^\s*Compiling/p;/\[E0277\]/p' \
    | sed -E 's/.*bound `(.*): Never`.*/\1/'
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
True
```

a bit of sed cleans it up

```rust
impl Or for (False, False) {
    type Output = False;
}

impl Or for (False, True) {
    type Output = True;
}

impl Or for (True, False) {
    type Output = True;
}

impl Or for (True, True) {
    type Output = True;
}
```

```rust
impl<Rhs> Or for (False, Rhs) {
  type Output = Rhs;
}

impl<Rhs> Or for (True, Rhs) {
  type Output = True;
}
```

we can also clean up Or a bit...

lib.rs

```rust
trait Never {}
trait Print: Never {}

struct False;
struct True;

trait Or {
  type Output;
}

impl<Rhs> Or for (False, Rhs) {
  type Output = Rhs;
}

impl<Rhs> Or for (True, Rhs) {
  type Output = True;
}

type FnCall = <(False, True) as Or>::Output;

impl Print for FnCall {}
```

```
❯ cargo build 2>&1 \
    | sed -En '/^\s*Compiling/p;/\[E0277\]/p' \
    | sed -E 's/.*bound `(.*): Never`.*/\1/'
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
True
```

for our first no-code program!

Now, let's sum a list of non-negative integers!

Wait, what's an integer?

maybe u32?

maybe u32?

u32 is the type
of integer values

Our types are our integers

It's time to birth an algebra

```
struct Z;
```

We define the number zero

```
struct Z;
struct S<N>;
```

and a "successor" type

```rust
type Zero  = Z;
type One   = S<Zero>;
type Two   = S<One>;
type Three = S<Two>;

impl Print for Three {}
```

```
❯ cargo build 2>&1 \
    | sed -En '/^\s*Compiling/p;/\[E0277\]/p' \
    | sed -E 's/.*bound `(.*): Never`.*/\1/'
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
S<S<S<Z>>>
```

with just this, we have integers!

```rust
use std::marker::PhantomData;

struct Z;
struct S<N>(PhantomData<N>);
```

aside: PhantomData

Similarly, we can birth a list

```
struct Nil;
```

We define the empty list, Nil

```
struct Nil;
struct Cons<X, Xs>(PhantomData<(X, Xs)>);
```

and Cons, a constructor of
an element X and a list Xs

```
type List = Cons<One, Cons<Two, Cons<Three, Nil>>>;

impl Print for List {}
```

```
❯ cargo build 2>&1 \
    | sed -En '/^\s*Compiling/p;/\[E0277\]/p' \
    | sed -E 's/.*bound `(.*): Never`.*/\1/'
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
Cons<S<Z>, Cons<S<S<Z>>, Cons<S<S<S<Z>>>, Nil>>>
```

and thus we have lists!

Now for integer addition…

```
trait Add {
    type Output;
}
```

Of course, Add is a function...
i.e. a trait!

```rust
impl<Rhs> Add for (Z, Rhs) {
  type Output = Rhs;
}
```

`zero + Rhs` is simply Rhs!

```
impl<N, Rhs, NPlusRhs> Add for (S<N>, Rhs)
where
  (N, Rhs): Add<Output = NPlusRhs>,
{
  type Output = S<NPlusRhs>;
}
```

... and then there's this...

```
impl Print for <(One, Two) as Add>::Output {}
```

```
❯ cargo build 2>&1 \
    | sed -En '/^\s*Compiling/p;/\[E0277\]/p' \
    | sed -E 's/.*bound `(.*): Never`.*/\1/'
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
S<S<S<Z>>>
```

testing it out now!

Finally, we can add up a list

```
trait SumList {
    type Output;
}
```

Create a SumList funct-- trait!

```
impl SumList for Nil {
    type Output = Z;
}
```

An empty list is easy...

```
impl<N, Ns, NsSum, TotalSum> SumList for Cons<N, Ns>
where
  Ns: SumList<Output = NsSum>,
  (N, NsSum): Add<Output = TotalSum>,
{
  type Output = TotalSum;
}
```

The other case is harder :P

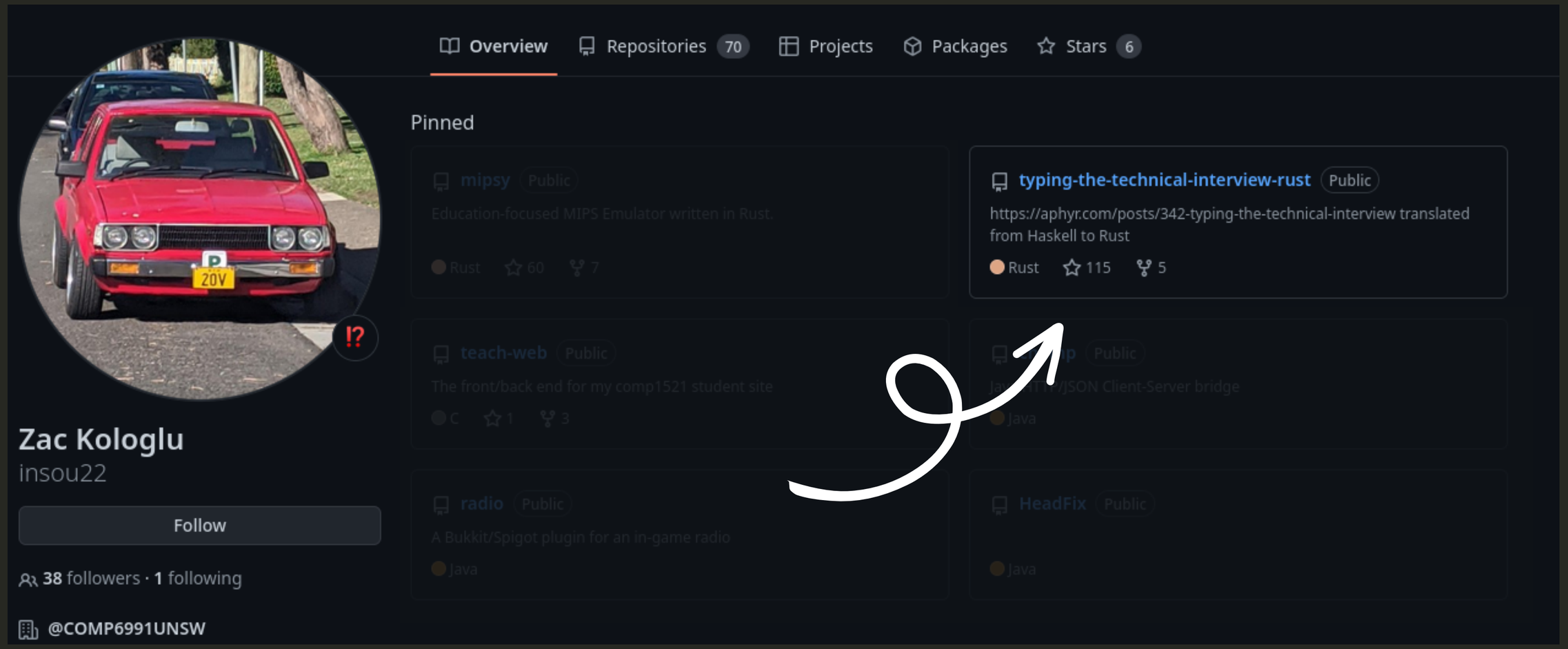
```
type List = Cons<One, Cons<Two, Cons<Three, Nil>>>;

impl Print for <List as SumList>::Output {}
```

```
❯ cargo build 2>&1 \
    | sed -En '/^\s*Compiling/p;/\[E0277\]/p' \
    | sed -E 's/.*bound `(.*): Never`.*/\1/'
   Compiling no_code v0.1.0 (/home/zac/dev/no_code)
S<S<S<S<S<S<Z>>>>>>>
```

but in the end, it works!

Wrapping things up...

How far can this go?

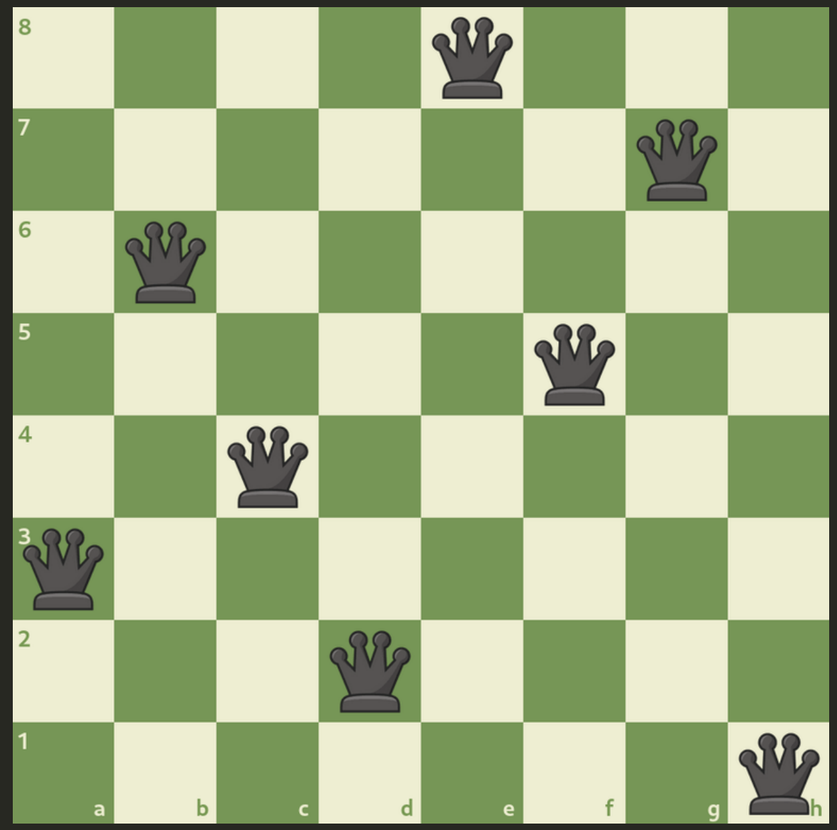


How far can this go?

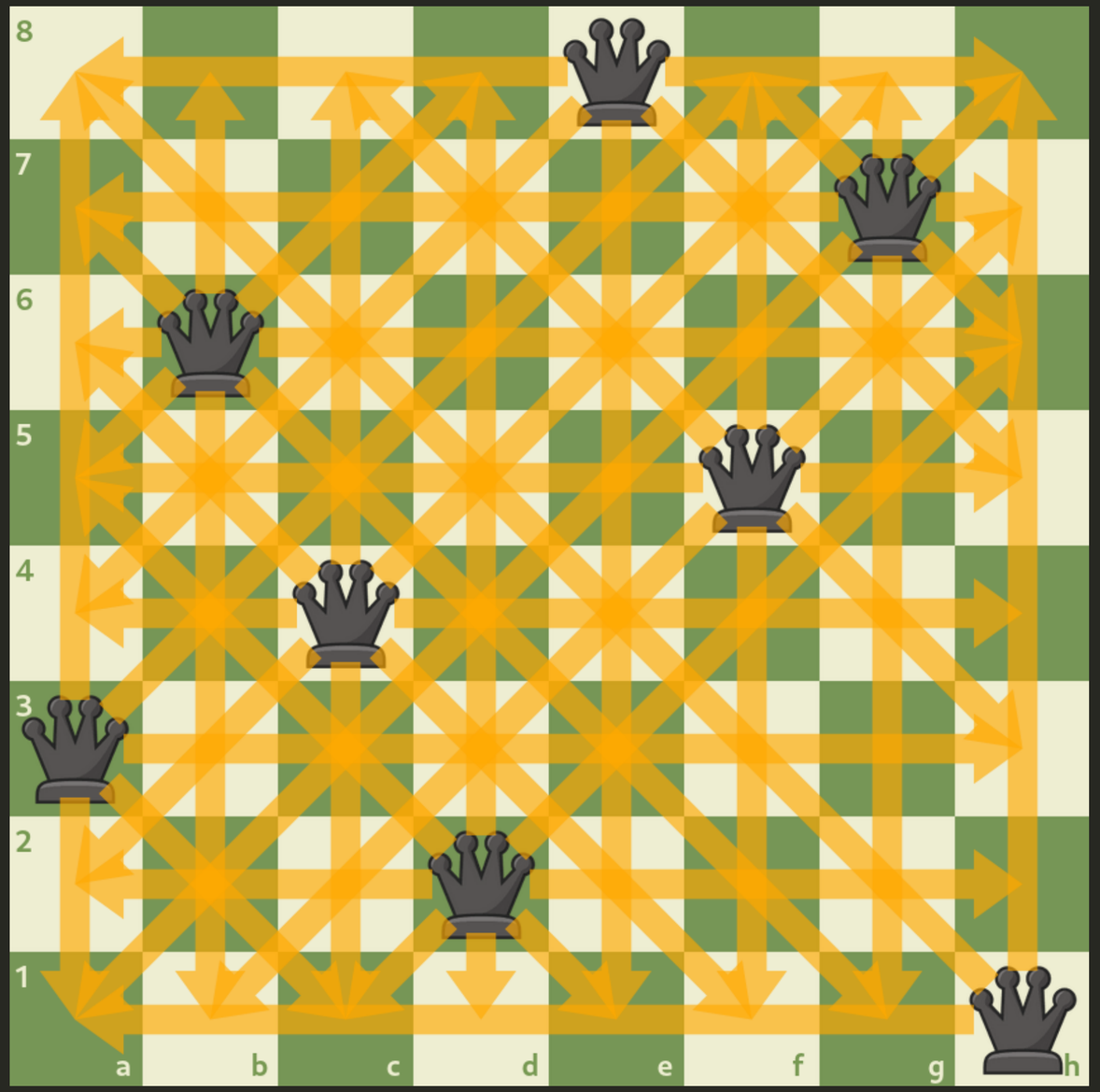
```
trait Threatens {
    type Output;
}

impl<Ax, Ay, Bx, By> Threatens for (Queen<Ax, Ay>, Queen<Bx, By>)
where
    (Ax, Bx): PeanoEqual,
    (Ay, By): PeanoEqual,
    (Ax, Bx): PeanoAbsDiff,
    (Ay, By): PeanoAbsDiff,
    (<(Ax, Bx) as PeanoEqual>::Output,   <(Ay, By) as PeanoEqual  >::Output): Or,
    (<(Ax, Bx) as PeanoAbsDiff>::Output, <(Ay, By) as PeanoAbsDiff>::Output): PeanoEqual,
    (<(<(Ax, Bx) as PeanoEqual>::Output, <(Ay, By) as PeanoEqual  >::Output) as Or>::Output, <(<(Ax,
Bx) as PeanoAbsDiff>::Output, <(Ay, By) as PeanoAbsDiff>::Output) as PeanoEqual>::Output): Or,
{
    type Output = <
        (
            <(
                <(Ax, Bx) as PeanoEqual>::Output,
                <(Ay, By) as PeanoEqual>::Output,
            ) as Or>::Output,
            <(
                <(Ax, Bx) as PeanoAbsDiff>::Output,
                <(Ay, By) as PeanoAbsDiff>::Output,
            ) as PeanoEqual>::Output,
        ) as Or>::Output;
}
```

How far can this go?

```
❯ cargo run
   Compiling ttti-rs v0.1.0 (/home/zac/dev/ttti-rs)
    Finished dev [unoptimized + debuginfo] target(s) in 1m 14s
     Running `target/debug/ttti-rs`
Cons<Queen<S<S<S<S<S<S<S<Z>>>>>>>>, S<S<S<S<Z>>>>>, Cons<Queen<S<S<S<S<
S<S<Z>>>>>>>, S<S<S<S<S<S<S<Z>>>>>>>, Cons<Queen<S<S<S<S<S<Z>>>>>, S<Z>>,
 Cons<Queen<S<S<S<S<Z>>>>, S<S<S<S<S<Z>>>>>>, Cons<Queen<S<S<S<Z>>>>, S
<S<Z>>>, Cons<Queen<S<S<Z>>>, Z>, Cons<Queen<S<Z>, S<S<S<Z>>>>, Cons<Qu
een<Z, S<S<S<S<S<S<S<Z>>>>>>>>, Nil>>>>>>>>
```



How far can this go?



What really is type-level programming?

- Rust is comprised of two meta-languages: code and types.
- Rust code describes a statically-typed, compiled, multi-paradigm language.
- Rust types describe a dynamically-typed, interpreted, functional language.
- Rust types are turing complete, but resolution is extremely slow!
- Beware of type resolution recursion limits!

Is this actually useful?

- In very specific scenarios, maybe!
- For example, using typestates you can form type-safe builders, which validate the builder state at compile-time!
- Be careful not to overdo it!

Thanks to...

Kyle Kingsbury, a.k.a "Aphyr", creator of the brilliant blog post "Typing the technical interview".

Credit:

- Original inspiration.
- N-queens functional implementation.
- "To birth an algebra into the world is a beautiful thing."
- "Haskell is a dynamically-typed, interpreted language."

https://aphyr.com/posts/342-typing-the-technical-interview

Finally, questions!?

Thanks everyone!