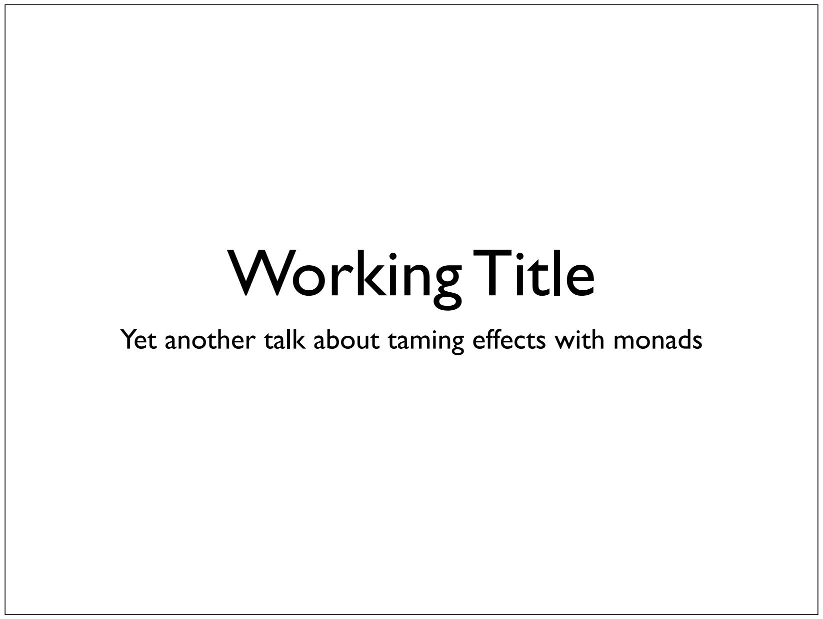
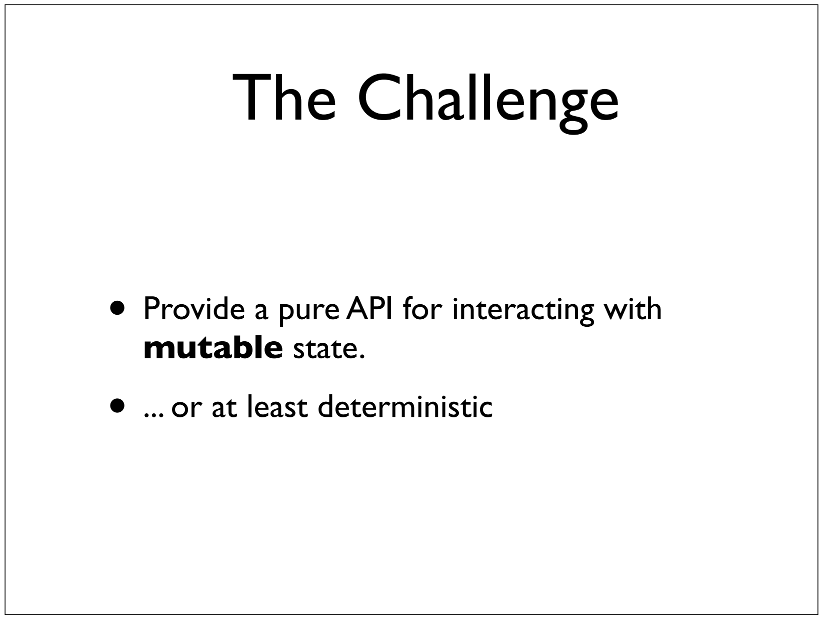


Working Title

Yet another talk about taming effects with monads

So, I don't even know what to call this talk. I said that I was going to talk about the Free monad, and we'll get to it at the end but it turns out to be an implementation detail here ... you can 90% of the way there with some really basic stuff that hopefully everyone can follow without too much trouble. So I want to focus on a problem that I spend a lot of time dealing with.



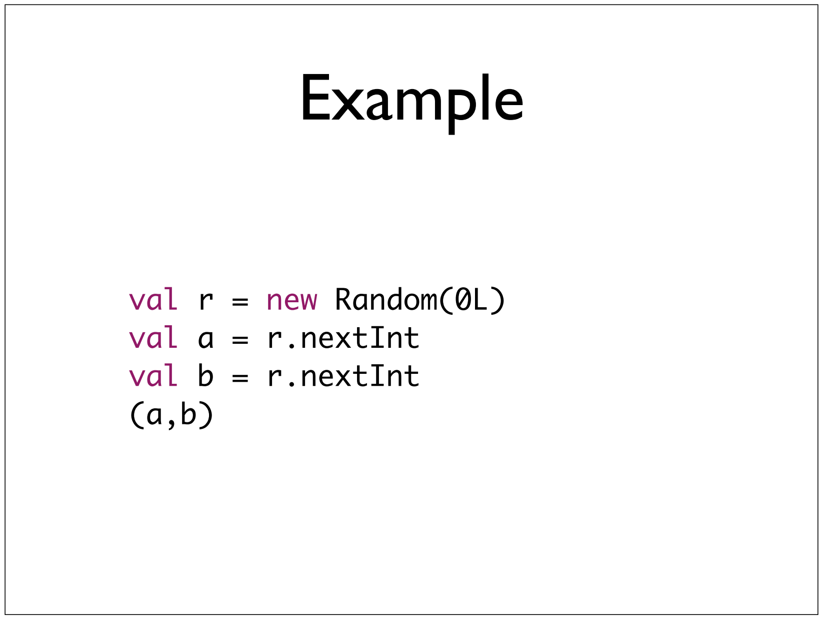
The Challenge

- Provide a pure API for interacting with **mutable** state.
- ... or at least deterministic

And I really do mean mutable state; I'm talking about the kind of stuff you end up doing whenever you have to interact with a Java API where you have JavaBeans and builders and all this other crap where you have an object with internal state that you're pushing and prodding from the outside, and any computation you perform with this thing is tainted.

So I have a solution to this, and it turns out to be pretty easy which makes me think it might be obvious and/or stupid. I don't know, I just want to run through it and see what people think. So let me set up the problem so hopefully my reasoning makes sense.

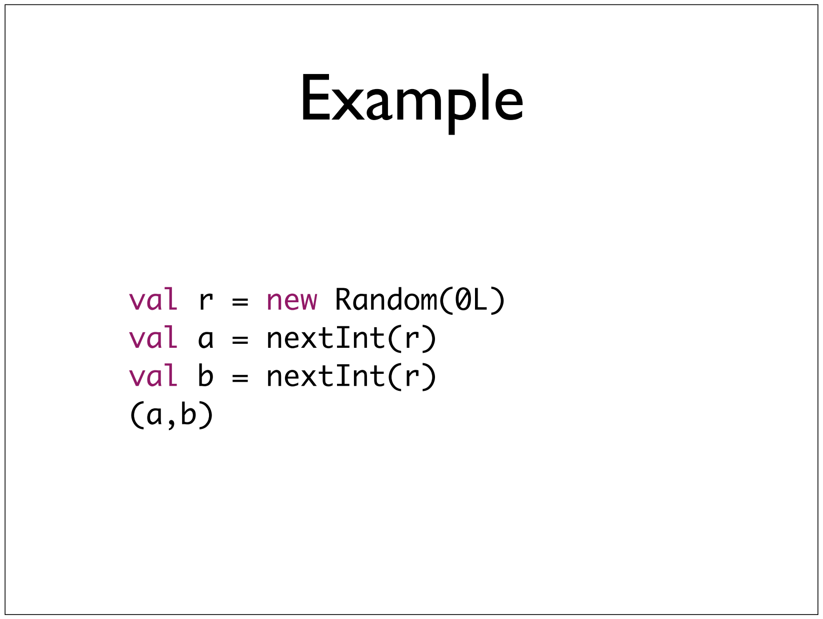
So here's a really simple example of what I'm talking about:



Example

```scala
val r = new Random(0L)
val a = r.nextInt
val b = r.nextInt
(a,b)
```

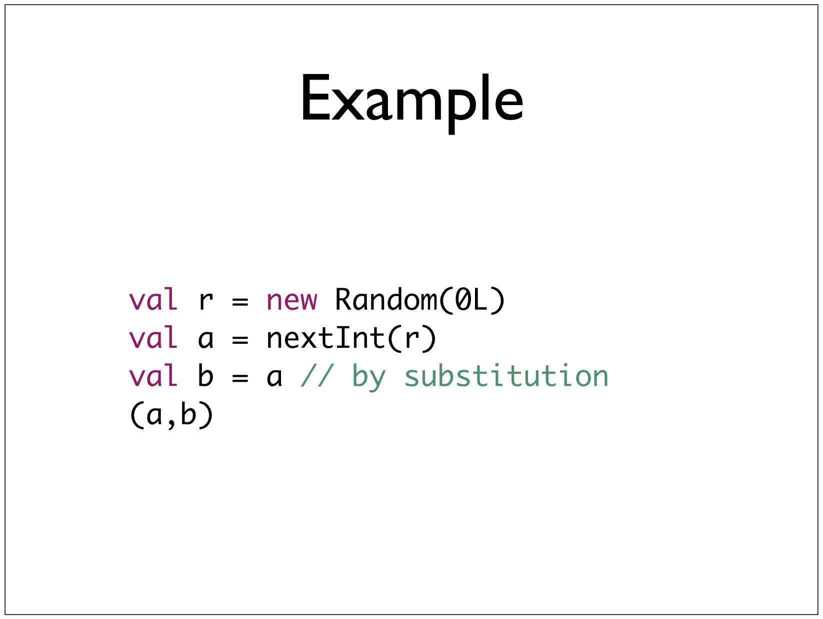
This code is deterministic but impure, because r.nextInt is not referentially transparent. And we can show this easily. Let's rewrite this to make it clear what the parameters are.



Example

```scala
val r = new Random(0L)
val a = nextInt(r)
val b = nextInt(r)
(a,b)
```

Right? The thing you call the method on is really a parameter. So if nextInt is a pure function we should be able to do this...

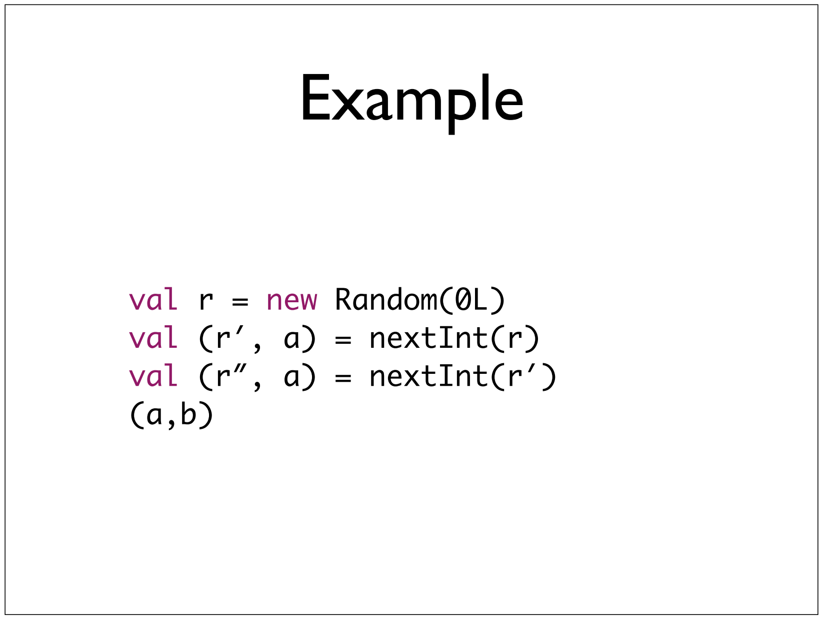


Example

```scala
val r = new Random(0L)
val a = nextInt(r)
val b = a // by substitution
(a,b)
```

but we can't do this because it changes the meaning of the program. We get a different answer. So that's our little existence proof that nextInt isn't a pure function, ... probably because it's referencing something on `r` that's not referentially transparent, and that poisons the whole computation.

Ok. I'm afraid this connection may be overreaching a little, but I'll go through it anyway because I think we end up in the right place. So one way we can deal with this is uniqueness typing, which is what the Clean language does. The idea is that some values and parameters are tagged as unique, which means you can only use them once, and when you do you get a new value back. There's syntax for it so you don't see it looking like this, but this is the basic idea:

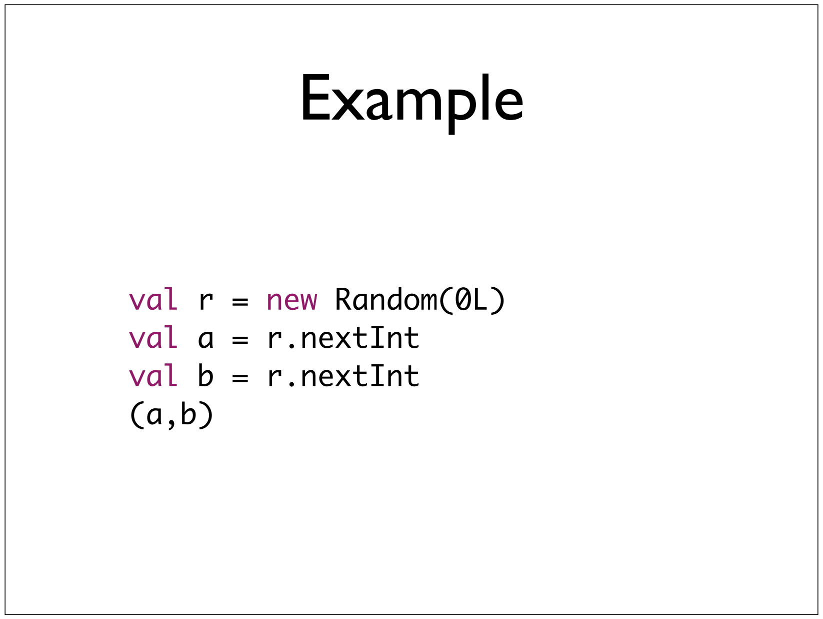


Example

```scala
val r = new Random(0L)
val (r′, a) = nextInt(r)
val (r″, a) = nextInt(r′)
(a,b)
```

So for instance if we tried to pass `r` again we'd get a compile error. So by enforcing this rule you get your referential transparency back because there is never a legal substitution you can do; you can't substitute nextInt(r) for nextInt(r') because they're not the same computation anymore (they have different arguments). So it's a type system trick that gets you purity just by getting rid of potential substitutions. I think it's pretty clever.

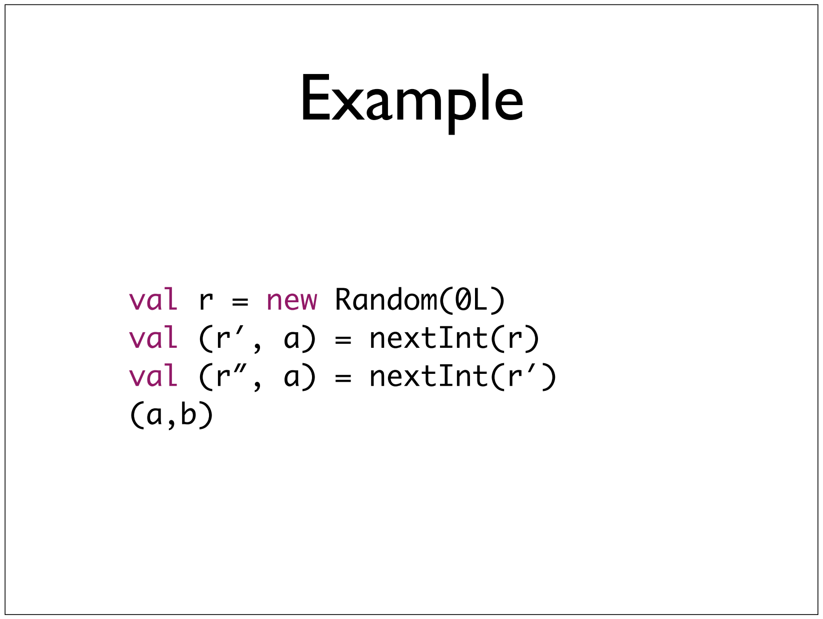
So what I want to do is figure out how to do this in Scala, so we can turn our original program into a pure program.



Example

```
val r = new Random(0L)
val a = r.nextInt
val b = r.nextInt
(a,b)
```

Ok? So the trick Clean plays on you

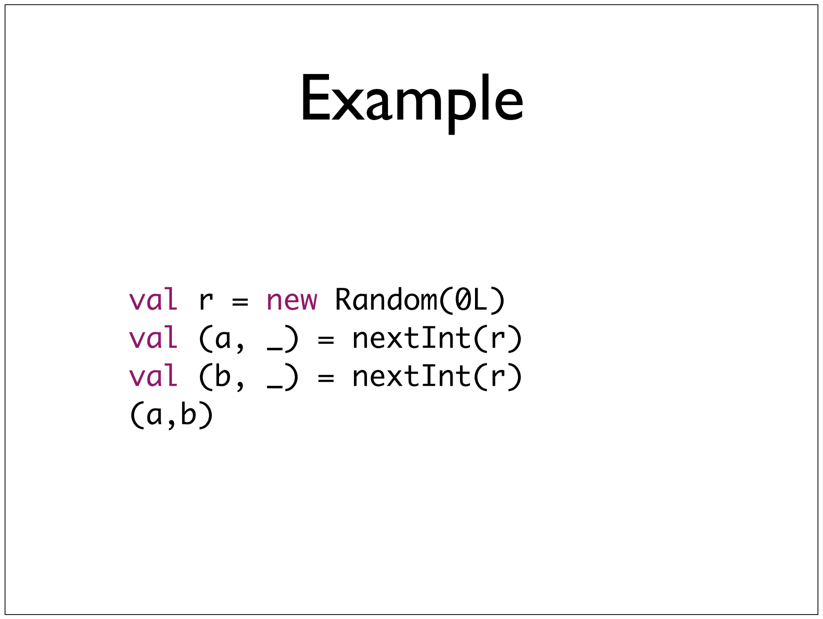


Example

```
val r = new Random(0L)
val (r′, a) = nextInt(r)
val (r″, a) = nextInt(r′)
(a,b)
```

is that it claims you get a different `r` back each time, but really you don’t. It just prevents you from ever having more than one reference to `r`, so whenever you use it in an expression you've given up the one and only reference and you get something new back. This may remind you of the IO monad ... we'll come back to that.

Of course, this doesn't work in Scala because you can just do this



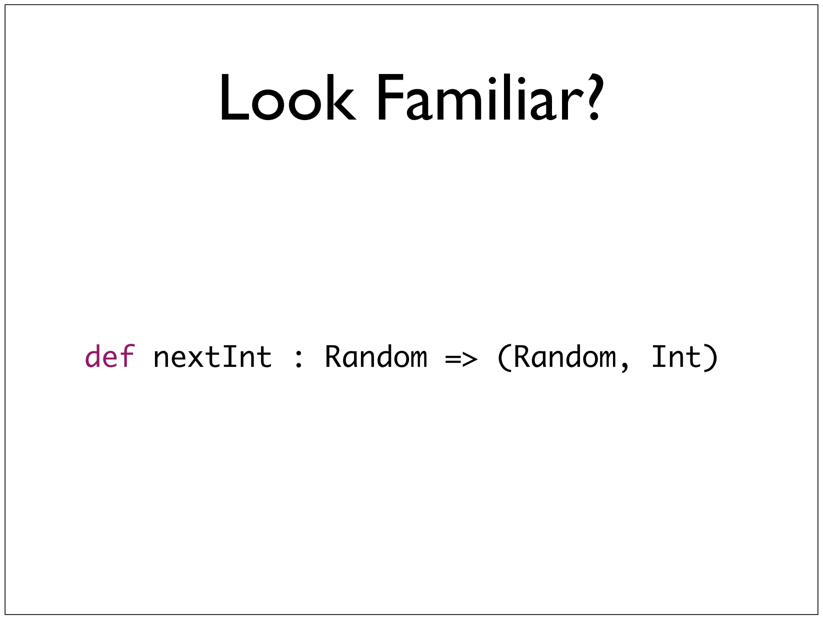
Example

```scala
val r = new Random(0L)
val (a, _) = nextInt(r)
val (b, _) = nextInt(r)
(a,b)
```

which makes impurity visible again. So the problem is that there's no control over the number of references you can have to `r`. If we can somehow guarantee that no references can escape (so you can't use it again or make a copy) then we can turn this into a pure program. So, that's the setup. That's what I'd like to talk about.

Everyone with me?

So this style probably looks familiar.



Look Familiar?

```scala
def nextInt : Random => (Random, Int)
```

is just an instance of the `State` monad, which lets you thread an evolving state through a computation.

Look Familiar?

```scala
type State[S,A] = S => (S,A)

def nextInt : State[Random, Int]
```

is just an instance of the `State` monad, which lets you thread an evolving state through a computation.

Ok. We already know how to write the `State` monad in Scala (and if not we'll see it in a minute). The challenge is to somehow prevent you from ever duplicating a reference to `S`. If you can do this my contention is that you can build pure computations with impure state, as long as you start off in a known state that evolves deterministically.

So in other words, if I have this thing (with a deterministically evolving internal state), if I can control all the references to it, then I can write pure computations that pass this thing around.

The Plan

- Model this program using `State[S,A]`
- Make S disappear.

If you can never even see the state, you can't possibly leak a reference, nor can you do anything with the state that we don't want you to do. Ok so let's look at some code.

Code

Blech

- More examples on github/tpolecat/tiny-world