Getting Closure on Scientist

— Background
— A Toy Experiment
— Don't Panic!
— Tradeoffs
— What's Left?

Background: who am I?

James Dietrich (@jbdietrich)

Team Lead @ Zendesk

Background: what is `scientist`?

— A library for "carefully" refactoring[1]

— Where refactoring is a behavior-preserving change to the implementation of a system

— Developed by GitHub as a Ruby gem to replace permissions code[2]

— `tl;dr` a way to run two pieces of code and compare their output (e.g. timing, correctness, errors)

[1] https://github.com/github/scientist

[2] http://rick.github.io/long-refactorings-talk

Background: why (attempt) a port to Rust?

— Conceptually simple

— Simple 💎 implementation

— Has proven generally useful: versions written for PHP, C#, Python, Java, C++, Javascript, Clojure, Perl, Elixir, Go, Kotlin, Swift

— Don't get to write Rust @ work

Background: Ruby example

```ruby
require 'scientist/experiment'

class MyExperiment
  include Scientist::Experiment

  def initialize(name:)
    @name = name
  end

  def publish
    # store metadata about mismatched results
  end
end

def old_way
  5
end

def new_way
  10
end

result = MyExperiment.new("multiples of five") do |e|
  e.try { new_way }
  e.use { old_way }
  # e.compare { custom comparator code }
  # e.clean { transform the objects before publishing }
  e.run
end

result == 10 # true
# results are published (how long did each block take to run?, were there errors?, were the results mismatached?)
```

Background: why does this warrant a talk?

— Learned a lot about Rust's closures
— Illustrates some choices Rust forces
— Showcases benefits of Rust's type system

A Toy Experiment: initial requirements

— Create a `ToyExperiment` object with two 'behavior' fields: `control` and `candidate`

— Each field should contain some code to run

— Add a `.run` method to the that evaluates `control` and `candidate`, returning the result of evaluating `control`

— For now, ignore all the stuff that makes `scientist` useful: comparing the returned values, measuring performance, recording results. It's a toy!

A Toy Experiment: Ruby implementation

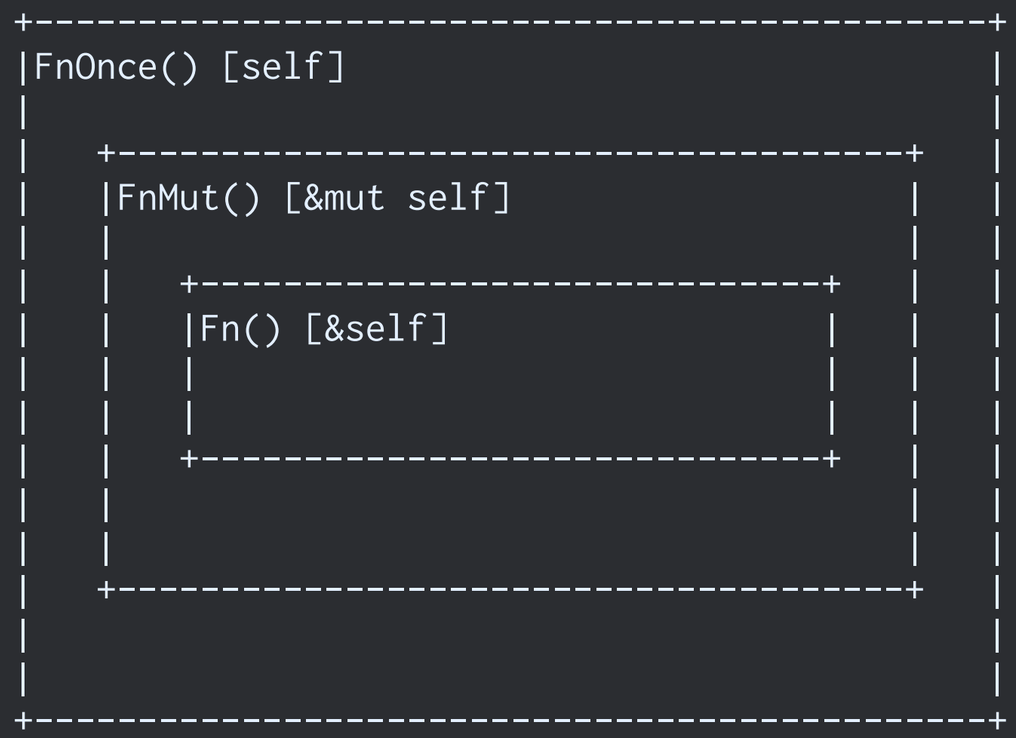
```ruby
class ToyExperiment
  attr_reader :control, :candidate

  def initialize(control:, candidate:)
    @control = control
    @candidate = candidate
  end

  def run
    control_res = control.call
    candidate_res = candidate.call

    control_res
  end
end

ex = ToyExperiment.new(control: -> { 5 }, candidate: -> { 4 })
raise "Oh no!" unless ex.run == 5
```

A Toy Experiment: Rust implementation - take one

- Use closures to contain the code that should be executed
- Parameterize our type over the `T` the behaviors return
- Use trait objects to contain the closures
- Playground

A Toy Experiment: closure trait interlude

Three closure traits



— Why Rust Closures are (Somewhat) Hard

A Toy Experiment: Rust take one, cont...

Problems:

- `Fn()` trait is inflexible
- `Fn()` doesn't encode a desirable invariant (each behavior probably should only be executed once)
- Trait objects (i.e. `Box<Fn()>`) can't be inlined by the compiler
- `FnOnce()` doesn't currently work as a trait object[3] `Box<FnOnce()>` not available in stable[4]

[3] https://www.reddit.com/r/rust/comments/4tae4l/howdoiboxfnonceclosureswith_hrtbs/d5fwauk

[4] https://github.com/rust-lang/rust/pull/54183

A Toy Experiment: Rust implementation - take two

— Use trait bounds to store the closure types in the struct

— Use `FnOnce() -> T` for additional flexibility

— Playground

A Toy Experiment: Rust take two, cont..

Problems:

— Nasty type signature (can be refactored a little with a trait + associated type)

— Unique closure types make it impossible to transform behaviors as a collection

A Toy Experiment: additional requirement

— Add a new `enabled` field to the object. This field stores a predicate to determine whether to evaluate the `candidate`

A Toy Experiment: additional requirement (Ruby)

```ruby
class ToyExperiment
  attr_reader :control, :candidate, :enabled

  def initialize(control:, candidate:, enabled:)
    @control = control
    @candidate = candidate
    @enabled = enabled
  end

  def run
    control_res = control.call
    return control_res unless enabled?

    candidate_res = candidate.call

    control_res
  end

  def enabled?
    enabled.call
  end
end

ex = ToyExperiment.new(control: -> { 5 }, candidate: -> { 4 }, enabled: -> { true })
raise "Oh no!" unless ex.run == 5
```

A Toy Experiment: additional requirement (Rust)

— Playground

— Can get the flexibility of `FnOnce()`, without consuming the outer struct by hiding information from the compiler (danger?)

Don't Panic!: an aside

— std::panic

```
pub fn catch_unwind<F: FnOnce() -> R + UnwindSafe, R>(f: F) -> Result<R>
```

— Playground

Tradeoffs:

What did we get?

— Guarantees about the behaviors returining the same `T`
— Guarantees about the experiment only being run once!
— Guarantees about the predicate actually behaving as a predicate

What did we give up?

— Readability?
— Flexibility (with monomorphized structs)

What's left?

— All sorts of callbacks (for e.g. transforming results)
— Crate code visibility
— Crate documentation
— Async publishing support