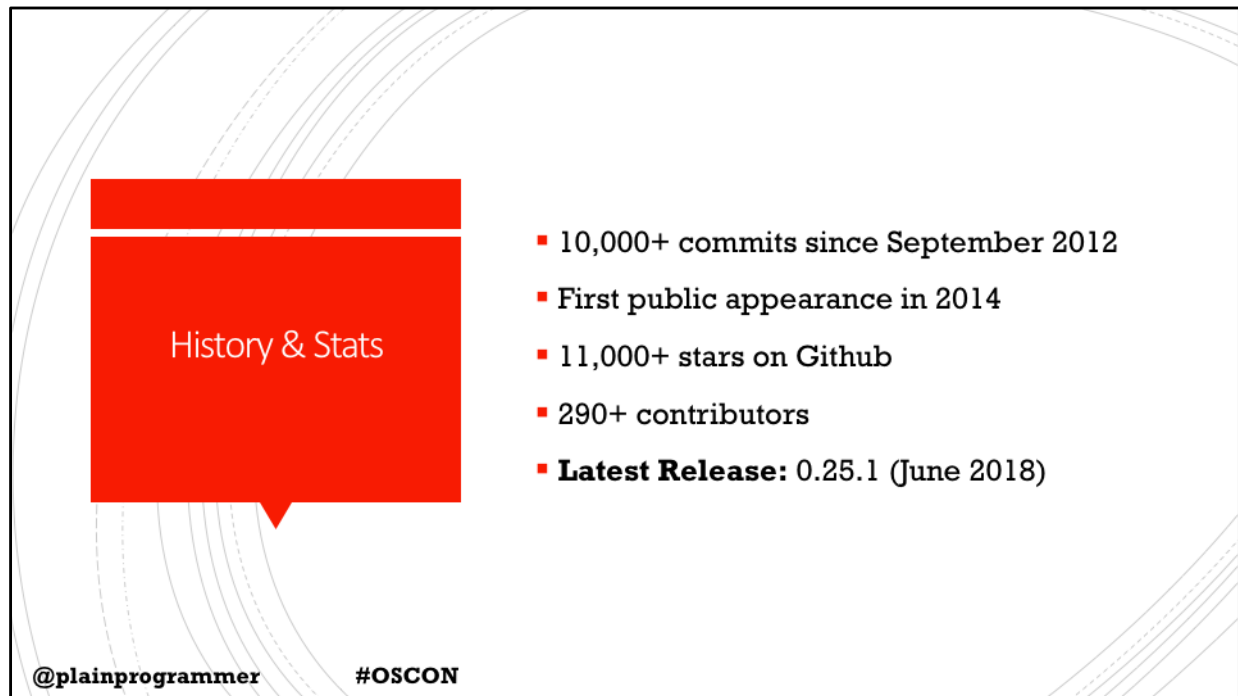


Thank you all for coming this morning. Lunch is next on all our agendas and so I promise not to go over my time so that all of us have a chance to find our way out at a reasonable time.

Today I'm going to be giving a sort of introduction to the Crystal programming language. I feel like I need to preface this talk by noting I am not currently doing any production work with Crystal, and I'm going to describe some of the reasons for that. I want all of you to leave here today feeling confident that you can decide whether Crystal is something you want to start using in production, on side projects, or otherwise. If you have really detailed questions I will do my best to answer them at the end of the presentation, but I make no guarantees that I will be able to.

So, let's jump in.



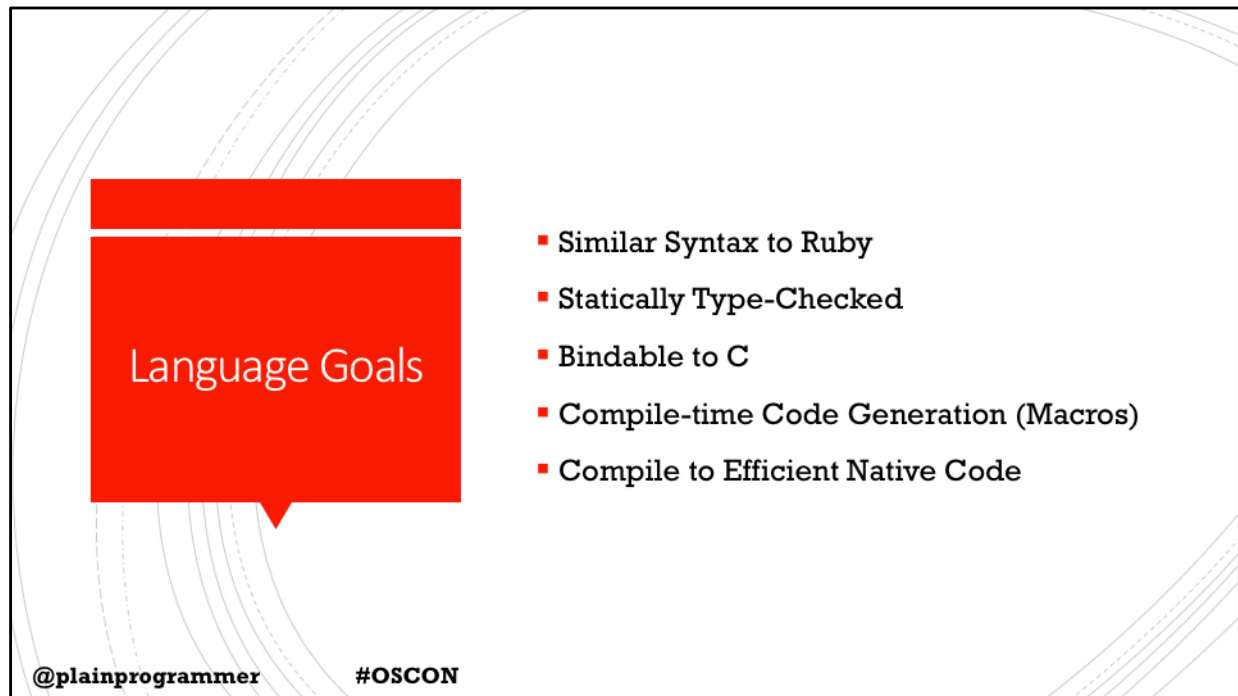
## History & Stats

- 10,000+ commits since September 2012
- First public appearance in 2014
- 11,000+ stars on Github
- 290+ contributors
- **Latest Release:** 0.25.1 (June 2018)

@plainprogrammer #OSCON

Crystal as a language is not a terribly new project, it has been in development for almost six years now. To provide some context: Go and Elixir each appeared only a handful of years prior. But, adoption of Crystal is still very limited and it still has some notable feature gaps. It is still a promising language that brings together an interesting collection of features.

The latest release comes from June of this year, and there are releases coming every few months. There are lots opportunities to get involved with numerous facets of the Crystal community, and so I think there is a lot of potential for growth.



Language Goals

- Similar Syntax to Ruby
- Statically Type-Checked
- Bindable to C
- Compile-time Code Generation (Macros)
- Compile to Efficient Native Code

@plainprogrammer #OSCON

The goals for the Crystal programming language are pretty straightforward. It's basic syntax is heavily inspired by Ruby and we will see that in a little bit. Crystal has taken a more opinionated stance in regard to certain aspects of Ruby's semantics, but a lot of Ruby code can be compiled, unmodified as Crystal code. One example of this semantic difference is with regards to arrays and how you determine the size of them. Ruby provides multiple methods that are identical in behavior, which Crystal opts for a single method. And this tendency is reflected consistently in the language's standard library.

Crystal adds to a fairly friendly syntax static type-checking and support for type inference. So, Crystal provides the same compile time safety of other type checked languages, but without the necessity to always declare types explicitly.

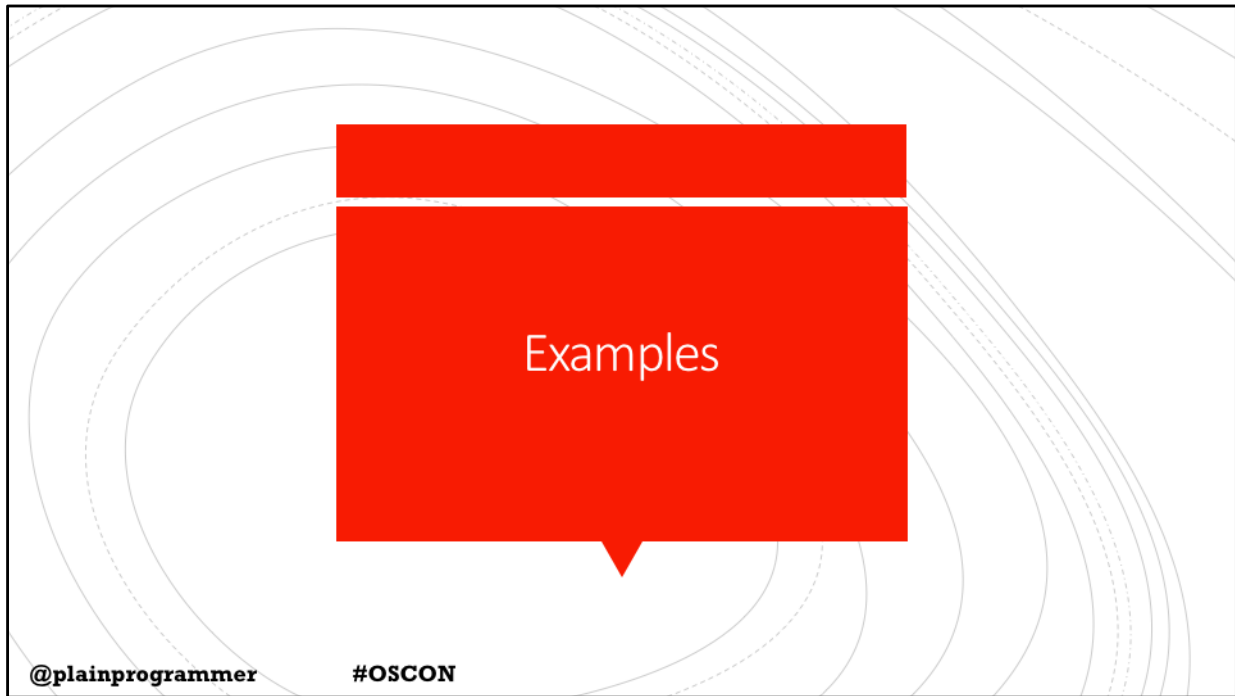
It also provides language features to make binding to C libraries straightforward. This has allowed the language to be much more useful to those willing to invest some effort in using the language than it might have been otherwise.

In place of the kind of runtime reflection and metaprogramming capabilities of Ruby, Crystal provides a robust macro system to enable compile-time code generation that

can help eliminate a lot of boilerplate code that even incorporates compile-time code reflection to enable things typically reserved for runtime-style metaprogramming.

Finally, and unsurprisingly, Crystal aims to produce efficient native code. This goal is aided by the fact that Crystal relies on the LLVM toolchain.

Now, it is also worth mentioning that like Ruby, Crystal is also thoroughly object-oriented.



Up next we're going to look at a few examples of Crystal code. If you are familiar with Ruby you will spot the similarities very easily. I will make reference to a few differences as they come up, but my goal is to give only a passing glimpse of Crystal's syntax and semantics. I will provide resources you can follow-up on at the end of my examples if you want to get more in depth insights.

```
require "spec"

describe Array do
  describe "#size" do
    it "reports the right number of elements in the array" do
      [1, 2, 3].size.should eq 3
    end
  end
end
```

@plainprogrammer

#OSCON

I'm gonna start with testing in Crystal because I like to test first, and so should you. I also really like Rspec in Ruby.

So, Crystal's built in testing framework delights me because it is inspired by Rspec.

If you're used to xUnit style assertion frameworks, this may look a little odd. But the goal is to produce an executable specification that incorporates a certain level of documentation alongside the test cases.

```

def greet(name, age)
  "I'm #{name}, and I'm #{age} years old."
end

greet "OSCON", 20
# I'm OSCON, and I'm 20 years old.

def add(x : Number, y : Number)
  x + y
end

add 1, 2
# 3

add true, false
# Error: no overload matches 'add' with types Bool, Bool

```

@plainprogrammer

#OSCON

Here is some more typical sample code from Crystal. The first example is also completely valid Ruby code. Crystal uses a type inference system that is similar to Ruby's behavior. In the first case, as long as the two values can be cast to strings anything is acceptable as arguments. This means that in many cases the type-checking stays out of the developers way.

But, in the second example we see that we can also specify argument types explicitly and gain additional compile-time safety checks.

But, unlike Ruby it is also possible to perform method overloading and specify method implementations that vary based on argument types provided.

Methods can also specify return types using the same kind of type annotation we see here for arguments; and class instance variables also rely on the same kind of type annotations when such specificity is needed or desired.

```
require "http/server"

server = HTTP::Server.new do |context|
  context.response.content_type = "text/plain"
  context.response.print "Hello world!"
end

address = server.bind_tcp 8080
puts "Listening on http://#{address}"
server.listen
```

@plainprogrammer

#OSCON

Here is a trivial web server example, a Hello World microservice really.

Except for the fact that Ruby doesn't have a ready-to-go HTTP server library, this code looks indistinguishable from Ruby.

In Crystal the require call can handle relative path lookups , whereas Ruby has a special require relative call. Block syntax is identical, parens around method arguments are optional, and object traversal is the same.

This kind of friendly syntax is why I think Crystal has so much promise when combined with the fact that the programs it produces are often, at a minimum, an order of magnitude faster than Ruby. And in some use cases can rival C.



```

METHODS = %w(foo bar baz)

{% for name, index in METHODS %}
  def {{name.id}}
    {{index}}
  end
{% end %}

-----

methods.each_with_index do |name, index|
  define_method(name.to_sym) do
    index
  end
end

```

@plainprogrammer

#OSCON

Here is a real simple example of Macros in Crystal.

Below I've replicated similar code in Ruby.

Macros in Crystal are evaluated during compilation, meaning the methods are fully defined at runtime, rather than them being evaluated when the context first enters scope. This provides similar capabilities to what a lot of useful metaprogramming in Ruby accomplishes but without the same level of risk. Macros get the context of the Abstract Syntax Tree, so they have fairly rich context for performing reflection. In my experience, about 80% of what I have achieved via metaprogramming in Ruby could likely be achieved via Crystal macros.

```
channel = Channel(Nil).new

spawn do
  puts "Before send"
  channel.send(nil)
  puts "After send"
end

puts "Before receive"
channel.receive
puts "After receive"

# Before receive
# Before send
# After receive
```

@plainprogrammer

#OSCON

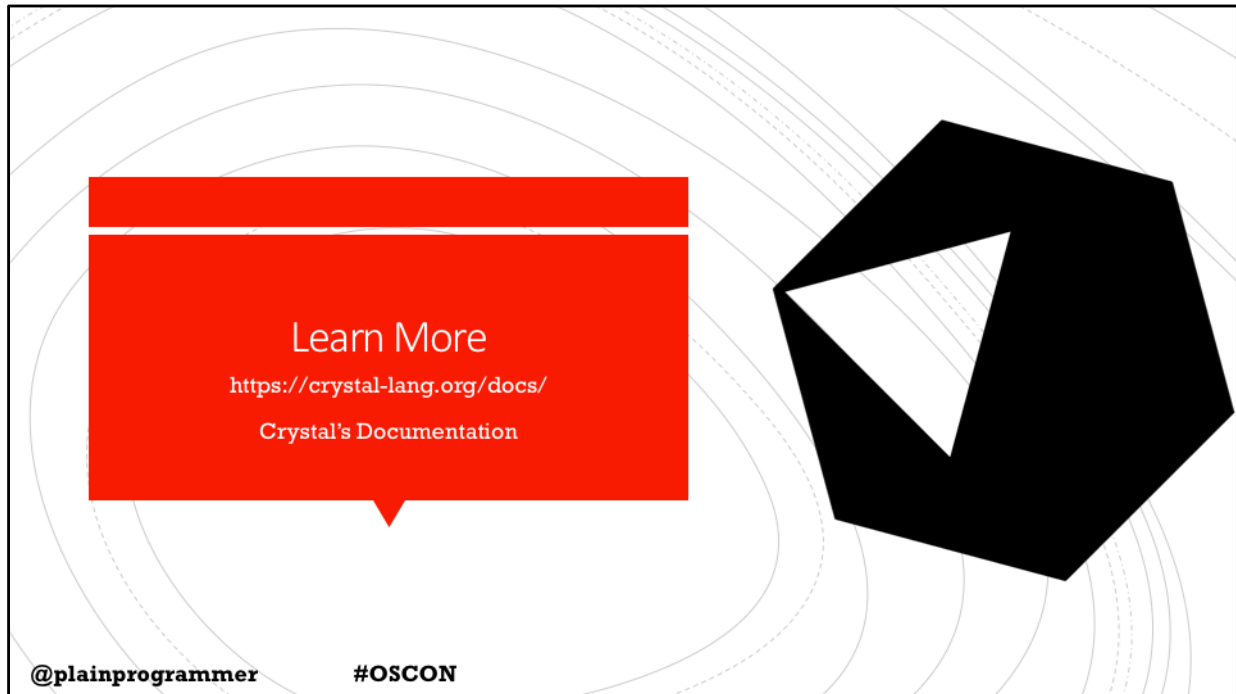
Concurrency is one of those things about Crystal that disappoints me.

Channels allow for communication across Fibers, which is what Crystal calls its green threads implementation. Here we can see how a Channel can be setup and how data can be sent from the Fiber to the main thread of execution. Since the Fiber does not start execution immediately nothing happens until the main thread calls channel receive, which triggers the Fiber to execute and as soon as it sends something the main thread resumes and the program exits.

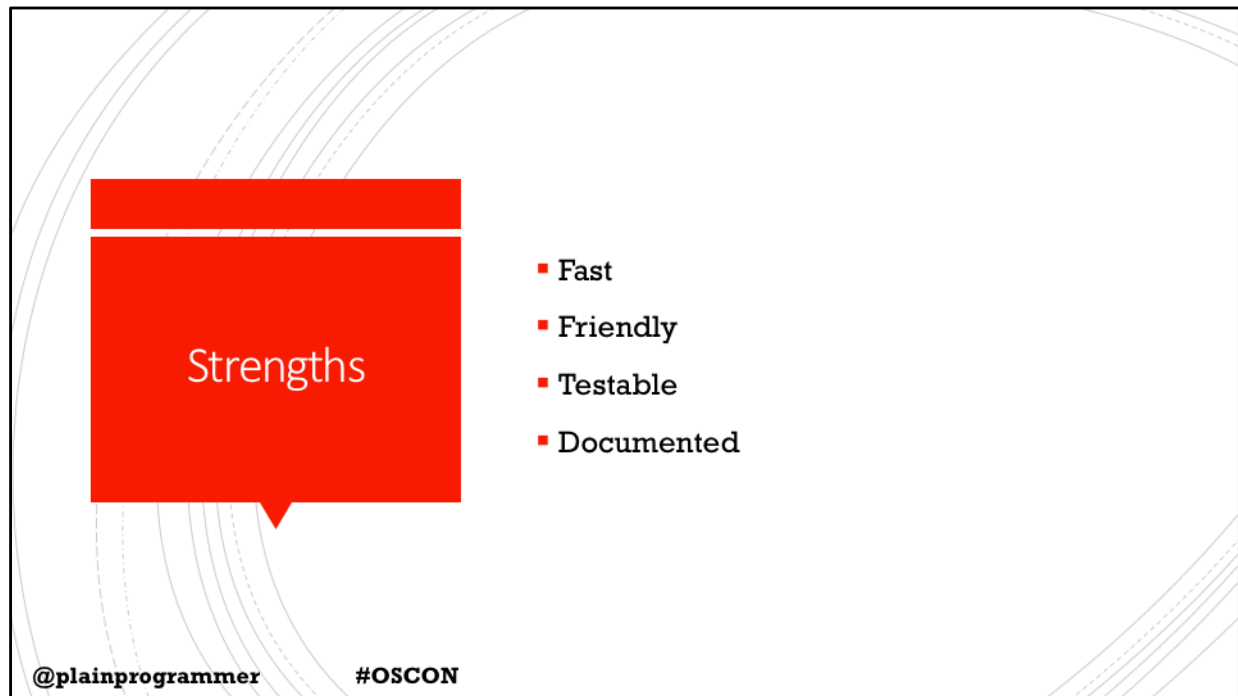
The issue I have with Crystal is that this is concurrency, but not parallelism. Crystal does not map Fibers onto operating system threads, and so you can only achieve a cooperative form of multitasking in your programs. There is experimental support for true threading. Now, coming from Ruby this is not a big deal since C Ruby does not support full parallelism either because of its Global Interpreter Lock. But I see this as one of Crystal's key issues right now.



Hopefully those very limited examples give you a hint of the similarities between Ruby's syntax and Crystal's. If you want a more extensive exploration of Crystal's syntax and language features, I encourage you to look up this video on YouTube. Jon goes into more detail about specific syntax and semantic patterns in Crystal that will help you get a better introduction to the language itself.



Crystal's official documentation is really good and covers all the key features in great detail, including addressing performance, concurrency, testing, and writing libraries for Crystal (called Shards). So, if you want to get into the gritty details I definitely recommend spending some time with the language documents, there much more accessible than many other language docs I've come across.



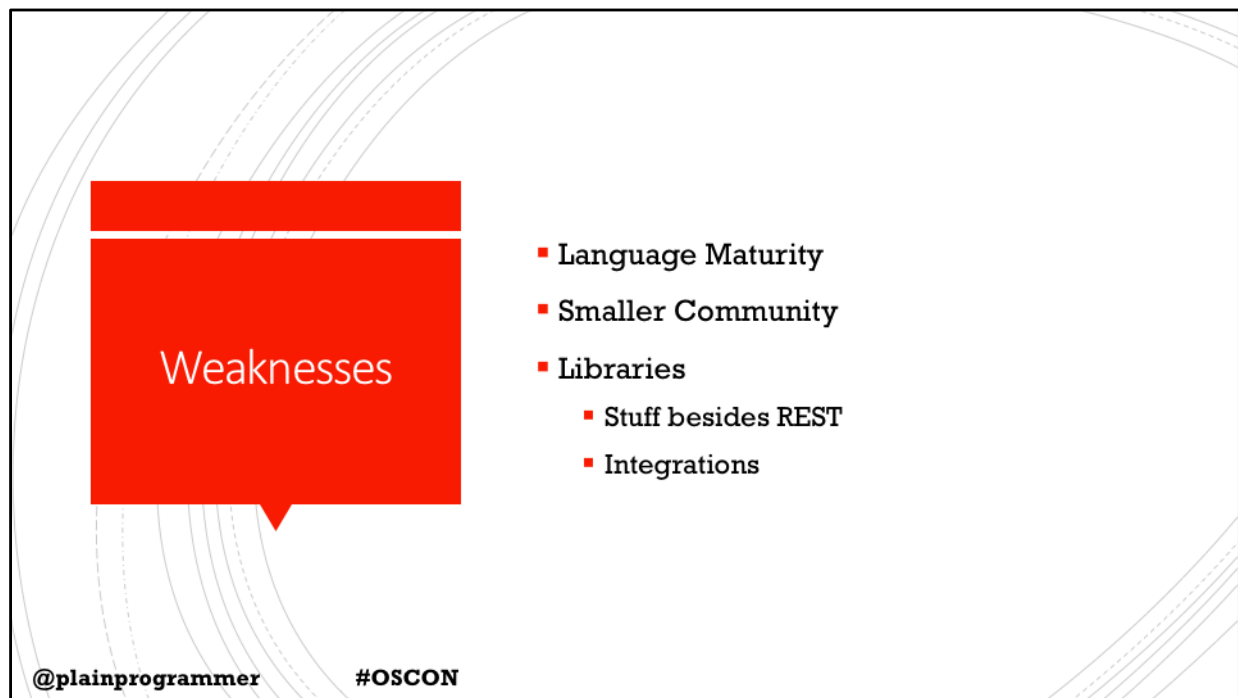
So, what is good about Crystal?

[CLICK] It is fast. In some cases it matches C's performance. I have seen some benchmarks relying on specific libraries that showed Crystal outperforming C, but I'm skeptical that anything like that would be consistently the case. But, Crystal is definitely at least an order of magnitude faster than Ruby, and often much more than that.

[CLICK] The speed of Crystal along with the friendly syntax is the big draw for me. I have loved working with Ruby since 2006, and am happy to see another language like Ruby available that does well with regards to performance in addition to JRuby.

[CLICK] Another big win in my view is testability. I like the Rspec inspired test syntax and the availability of that as part of the standard library.

[CLICK] Finally, Crystal is well documented. The language reference available on the main web site is accessible and goes into a good bit of detail about not just the language and its capabilities, but also matters such as performance and concurrency.



In my opinion, the three biggest weaknesses for Crystal are maturity, community and ecosystem.

[CLICK] The language has some issues, like no support for true multi-threading. Fibers in Crystal are not that dissimilar from Ruby's green threads just with no need for the protections of the Global Interpreter Lock. This makes it hard for me to want to adopt Crystal for some of my production projects since at least with Ruby I can switch to using JRuby without a lot of hassle and get true thread support.

[CLICK] The community is also fairly small, but it is growing in some areas. I know a user group is forming around Crystal in Utah that I look forward to being a part of. Hopefully that will continue to be the case since I think a lot of what Crystal needs will be driven by a growing community of practitioners.

[CLICK] But, that smaller community also means the library ecosystem is not as well established. There are lots of gaps for things like message queue producer and consumer libraries, implementations of communication models other than REST, and the like. The good support for C bindings in Crystal means that some of these issues can be addressed with a modest amount of effort, but the community needs to exist

that is willing to put in that effort.

Readiness

**USE IT TODAY**

- Stuff That Is Easy to Rewrite
- Command-Line Tools
- Some REST Microservices

**WAIT AND SEE**

- Almost Everything Else
- Monolithic Applications
- Large Codebases

@plainprogrammer #OSCON

So, what is Crystal ready to do today. I wish I could say use it for everything. But, really it's still best to use Crystal for things you could easily rewrite [CLICK]. If you have a command-line tool that you want to get a more performance from, Crystal may be a reasonable option [CLICK]. Some REST microservices could also be good fits [CLICK].

If you're using gRPC, or something else for synchronous inter-service communication you are likely going to have a fair bit of work ahead of you.

So, that leaves pretty much everything else as a wait and see, in my opinion [CLICK]. Unless you are wanting to bet on where the bleeding edge is going, Crystal is not a good choice for monolithic applications [CLICK], or projects expected to have large codebases [CLICK].

This risk of the big rewrite is too substantial, in my opinion. But, there are lots of opportunities to invest in Crystal. The community needs library authors to produce good, idiomatic ways to do a lot of different things. And, if there is already a good C library for what you need, the lift is reasonable.

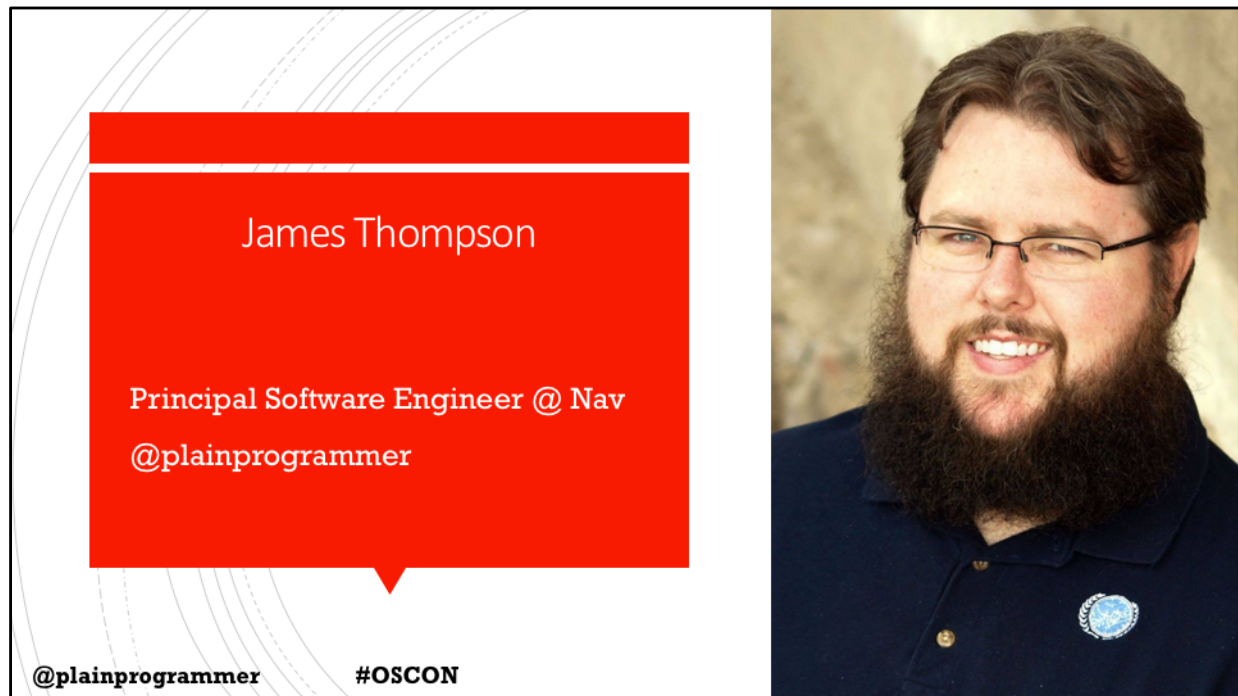


If you want to write bindings for Machine Learning libraries, or implement producers and consumers for message queues, or develop support for things like gRPC, or GraphQL, there is a need for that in the Crystal community. There is a lot of potential for Crystal to be a fast and friendly language option, but it needs to establish a community.

So, that is the biggest thing I hope some of you will be interested in. I hope some of you will join me in trying to do some work to make Crystal an inviting community and a more reasonable choice in the near future for production projects of a much broader sort.



So, what questions can I attempt to answer?



Thank you for coming today, I'm James Thompson, a principal software engineer at Nav, where we're working to decrease the death rate of America's small businesses. You can find me online as plainprogrammer. And, be sure to chat me up about anything from Ruby to barbecue.

Also, please go online or in the O'Reilly event app and leave feedback so I can improve this talk and others I give in the future. Again, thank you!