


Concepts vs Typeclasses vs Traits vs Protocols

Conor Hoekstra

code_report





Concepts vs Typeclasses vs Traits vs Protocols vs Type Constraints

Conor Hoekstra


code_report


#include

https://github.com/codereport/Talks

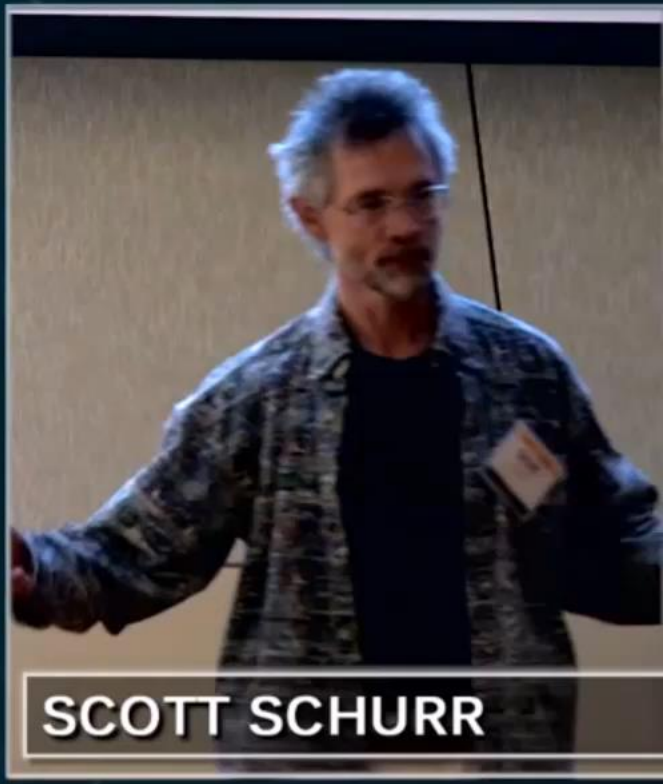
“I’m not an expert, I’m just a dude.”

- Scott Schurr, CppCon 2015

constexpr: Applications

By Scott Schurr for Ripple Labs at CppCon September 2015

cppcon the c++ conference



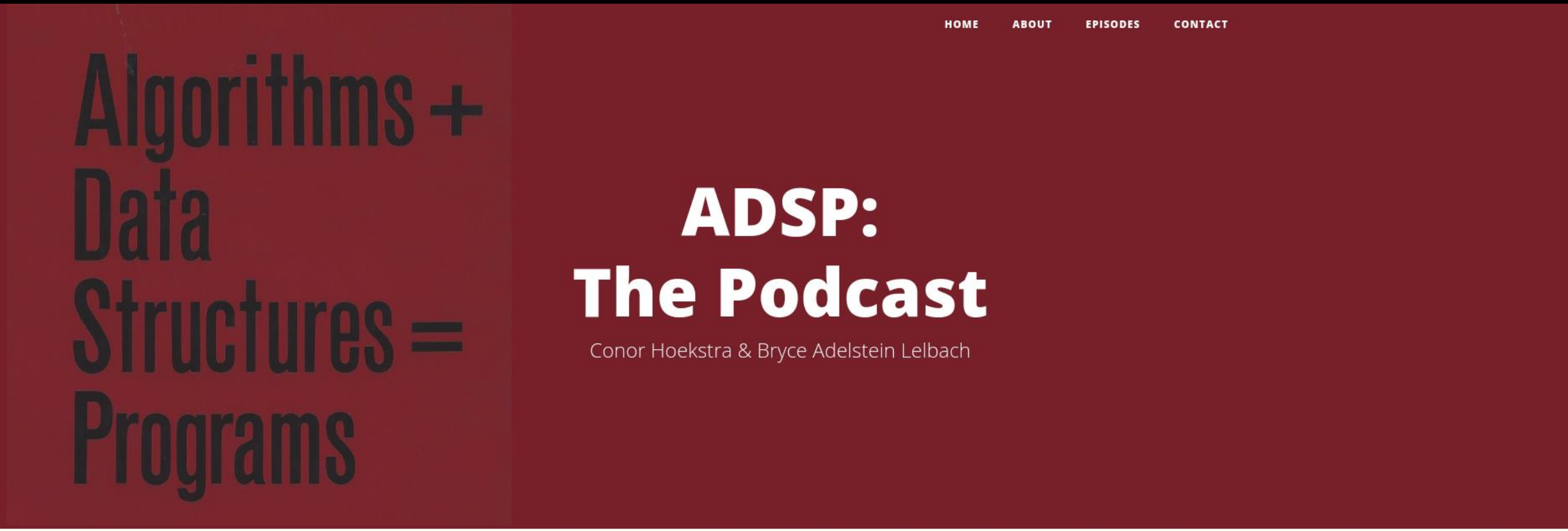
SCOTT SCHURR

constexpr:
Applications

www.CppCon.org

About Me

- I'm a Sr Library Software Engineer for NVIDIA
  - Working on the RAPIDS AI team (http://rapids.ai)
- I am a programming language enthusiast
- I love algorithms and beautiful code
- Organizer of Programming Languages Virtual Meetup
- I have a YouTube channel
- And …



Episode 0: Coming Soon ...

A informal podcast about algorithms and everything software related...

www.adspthepodcast.com

Nov 10, 2019
Sun, 12:44 PM GMT+00:00

Algorithms + Data Structures = Programs

Following

ADSP: The Podcast

@adspthepodcast

ADSP: The Podcast. Hosted by @blelbach & @code_report
A podcast about algorithms and everything software related...

Joined November 2020

0 Following **1** Follower

This is not a language war talk.

This is not a language war talk.

Conor Hoekstra
@code_report

1. First language: TI-BASIC
2. Had difficulties: Make
3. Most used: C++
4. Totally hate: I ❤ all PLs
5. Most loved: Haskell
6. For beginners: Python

@TI_BASIC #cplusplus #python #Haskell

> **Christopher Di Bella** @cjdb_ns · Oct 4, 2019
> 1. First language: GML
> 2. Had difficulties: C#
> 3. Most used: C++
> 4. Totally hate: CMake
> 5. Most loved: C++
> 6. For beginners: Unsure twitter.com/bstamour1/stat…

8:12 PM · Oct 4, 2019 from Sunnyvale, CA · Twitter for Android

This is not a language war talk.

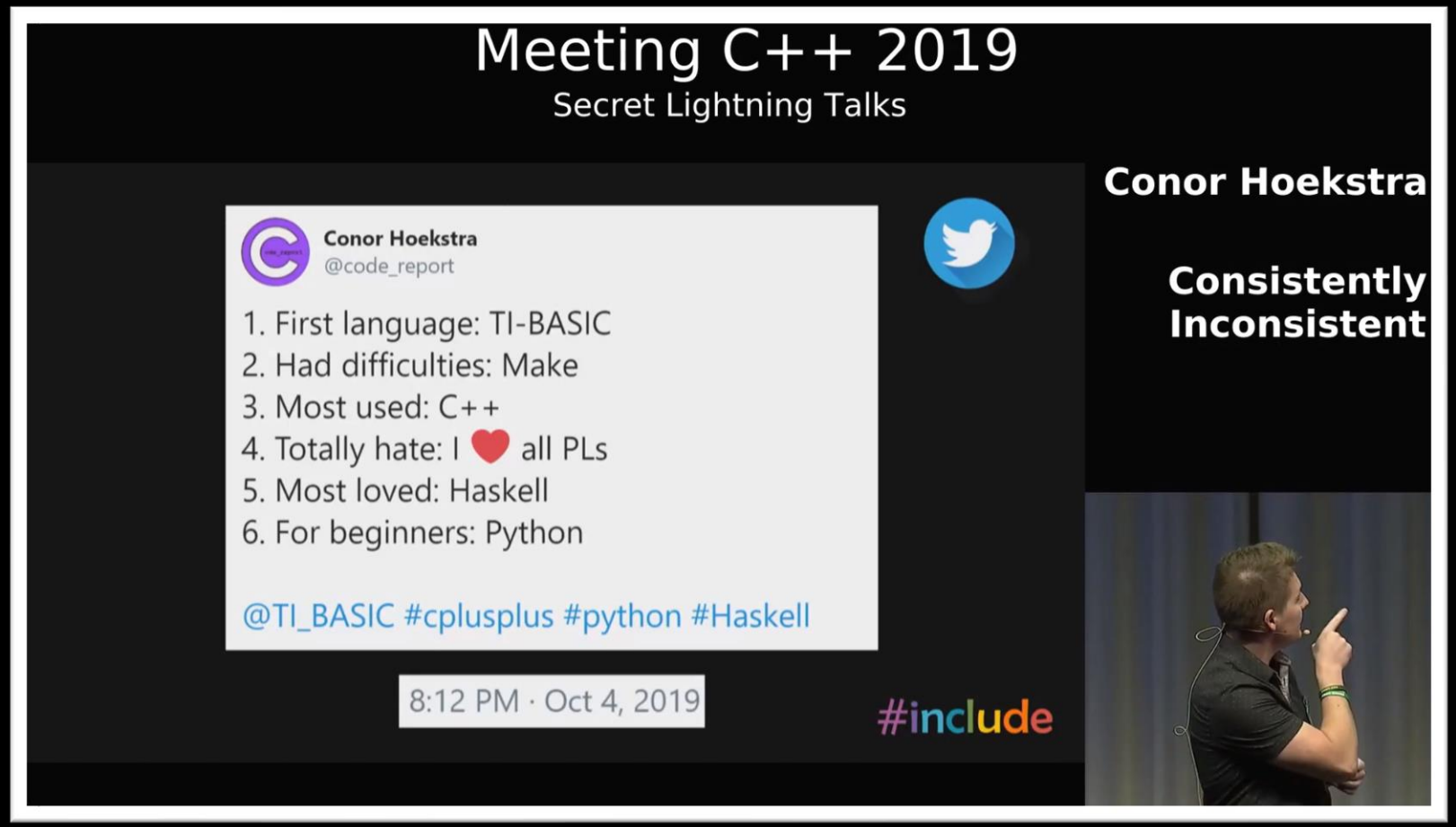
Conor Hoekstra
@code_report

1. First language: TI-BASIC
2. Had difficulties: Make
3. Most used: C++
4. Totally hate: I ❤️ all PLs
5. Most loved: Haskell
6. For beginners: Python

@TI_BASIC #cplusplus #python #Haskell

> **Christopher Di Bella** @cjdb_ns · Oct 4, 2019
> 1. First language: GML
> 2. Had difficulties: C#
> 3. Most used: C++
> 4. Totally hate: CMake
> 5. Most loved: C++
> 6. For beginners: Unsure twitter.com/bstamour1/stat...

8:12 PM · Oct 4, 2019 from Sunnyvale, CA · Twitter for Android



This is not a language war talk.

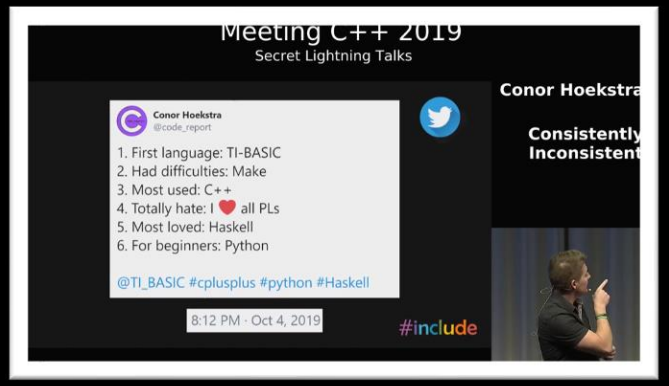
Conor Hoekstra
@code_report

1. First language: TI-BASIC
2. Had difficulties: Make
3. Most used: C++
4. Totally hate: I ❤ all PLs
5. Most loved: Haskell
6. For beginners: Python

@TI_BASIC #cplusplus #python #Haskell

> **Christopher Di Bella** @cjdb_ns · Oct 4, 2019
> 1. First language: GML
> 2. Had difficulties: C#
> 3. Most used: C++
> 4. Totally hate: CMake
> 5. Most loved: C++
> 6. For beginners: Unsure twitter.com/bstamour1/stat...

8:12 PM · Oct 4, 2019 from Sunnyvale, CA · Twitter for Android



Conor Hoekstra
@code_report

Highlight of @PLDI 2020 definitely has to be the AMA with Guy Steele who at one point said "I love ALL programming languages." I also ❤ heart all PLs: twitter.com/code_report/st... #pldi #programminglanguages

This is a language comparison talk.

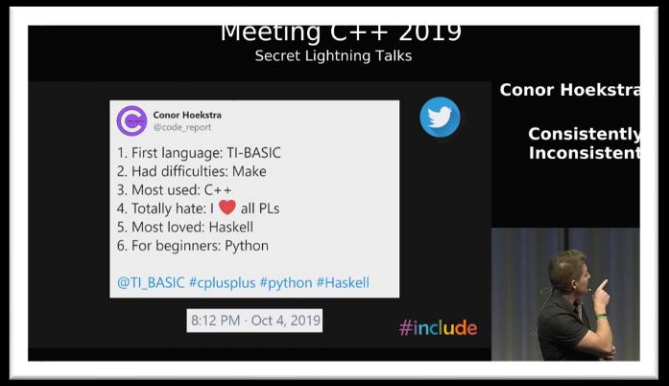
Conor Hoekstra
@code_report

1. First language: TI-BASIC
2. Had difficulties: Make
3. Most used: C++
4. Totally hate: I ❤ all PLs
5. Most loved: Haskell
6. For beginners: Python

@TI_BASIC #cplusplus #python #Haskell

> **Christopher Di Bella** @cjdb_ns · Oct 4, 2019
> 1. First language: GML
> 2. Had difficulties: C#
> 3. Most used: C++
> 4. Totally hate: CMake
> 5. Most loved: C++
> 6. For beginners: Unsure twitter.com/bstamour1/stat...

8:12 PM · Oct 4, 2019 from Sunnyvale, CA · Twitter for Android



Conor Hoekstra
@code_report

Highlight of @PLDI 2020 definitely has to be the AMA with Guy Steele who at one point said "I love ALL programming languages." I also ❤ heart all PLs: twitter.com/code_report/st... #pldi #programminglanguages

This is a language comparison talk.
This is part 1 of 2.

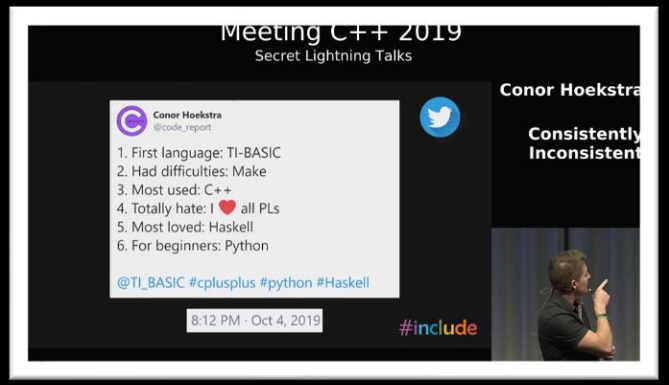
Conor Hoekstra
@code_report

1. First language: TI-BASIC
2. Had difficulties: Make
3. Most used: C++
4. Totally hate: I ❤ all PLs
5. Most loved: Haskell
6. For beginners: Python

@TI_BASIC #cplusplus #python #Haskell

> **Christopher Di Bella** @cjdb_ns · Oct 4, 2019
> 1. First language: GML
> 2. Had difficulties: C#
> 3. Most used: C++
> 4. Totally hate: CMake
> 5. Most loved: C++
> 6. For beginners: Unsure twitter.com/bstamour1/stat...

8:12 PM · Oct 4, 2019 from Sunnyvale, CA · Twitter for Android



Conor Hoekstra
@code_report

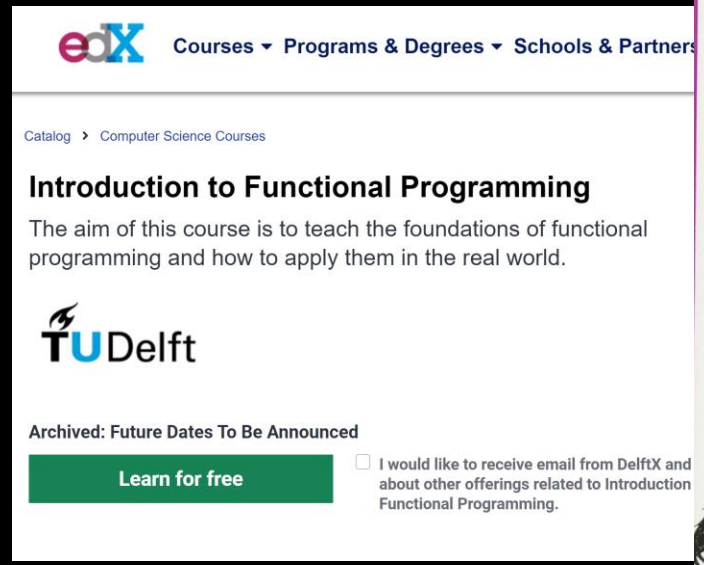
Highlight of @PLDI 2020 definitely has to be the AMA with Guy Steele who at one point said "I love ALL programming languages." I also ❤ heart all PLs: twitter.com/code_report/st... #pldi #programminglanguages

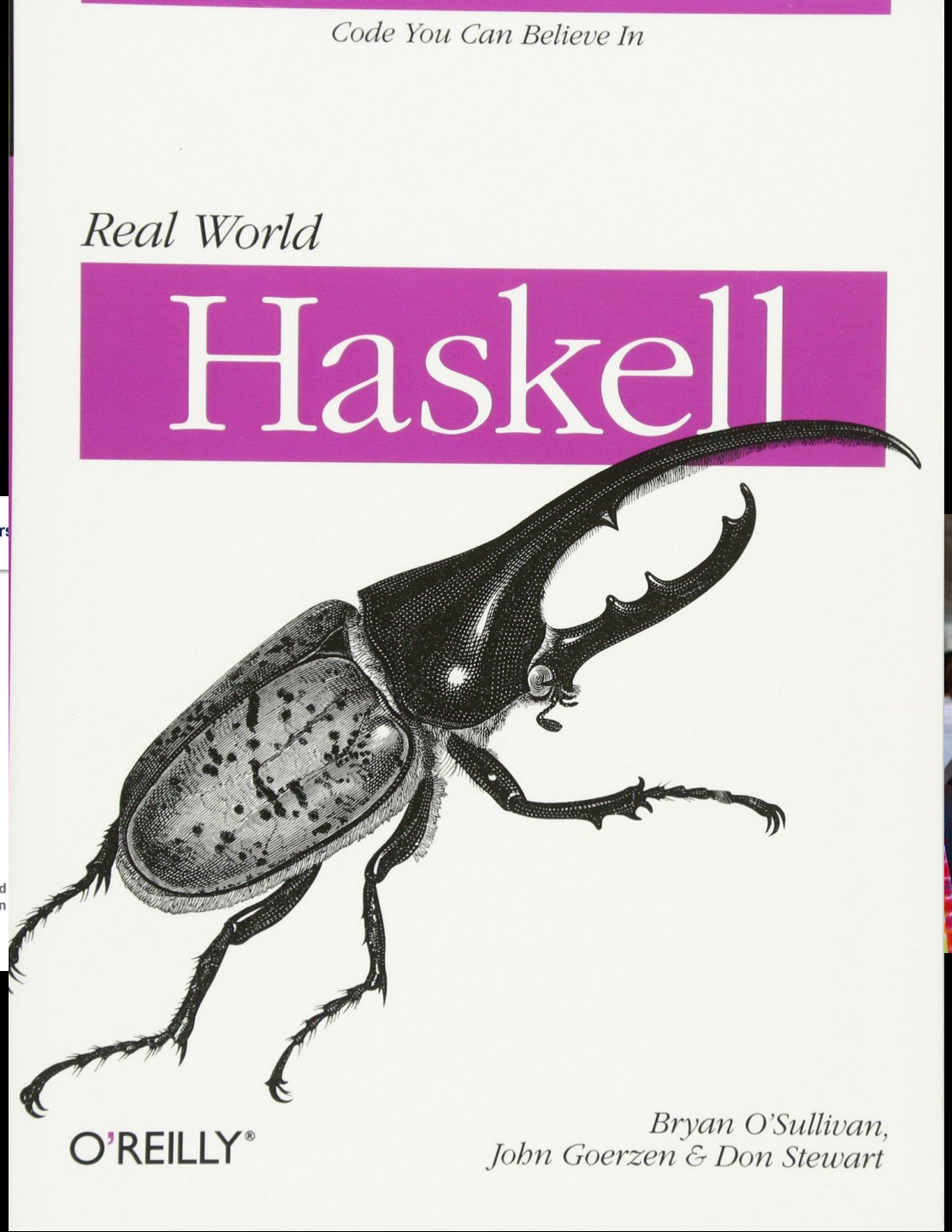
Agenda

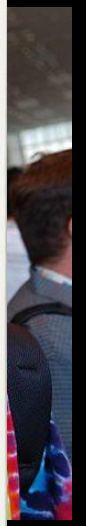
1. Introduction / History 🙃
2. Generics / Parametric Polymorphism
3. Example #1
4. Example #2
5. Final Thoughts

Introduction

2018-09: Haskell







Introduction

2018-09: Haskell

...

2019-12-08: Protocol Oriented Programming in Swift

Developer Tools

#WWDC15

Protocol-Oriented Programming in Swift

Session 408

Dave Abrahams Professor of Blowing-Your-Mind


Supports value types (and classes)
Supports static type relationships (and dynamic dispatch)
Non-monolithic
Supports retroactive modeling
Doesn't impose instance data on models
Doesn't impose initialization burdens on models
Makes clear what to implement

Introduction

2018-09: Haskell

...

2019-12-08: Protocol Oriented Programming in Swift

2020-01-09: Magic Read Along

“I watched a video today on **Swift** … about **protocol oriented programming** … and they basically just introduced **typeclasses** and they were like ‘We invented this, it’s amazing’”

Hardy Jones & Brian Lonsdorf

@st58 & @drboolean

Magic READ Along

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in `3*3`) integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a program using type classes to an equivalent program that does not use overloading. The translated programs are typable in the (ungeneralised) Hindley/Milner type system.

The body of this paper gives an informal introduction to type classes and the translation rules, while an appendix gives formal rules for typing and trans-

Introduction

2018-09: Haskell

...

2019-12-08: Protocol Oriented Programming in Swift

2020-01-09: Magic Read Along

2020-01-13: Reddit Article / Quora Answer

82

Influence of C++ on Swift (quora.com)

submitted 10 months ago by Austin_Aaron_Conlon

57 comments share save hide give award report crosspost

this post was submitted on 13 Jan 2020

82 points (92% upvoted)

shortlink: https://redd.it/eo10jo

[[digression]]

What are similarities and differences between C++ and Swift?

What are similarities and differences between C++ and Swift?

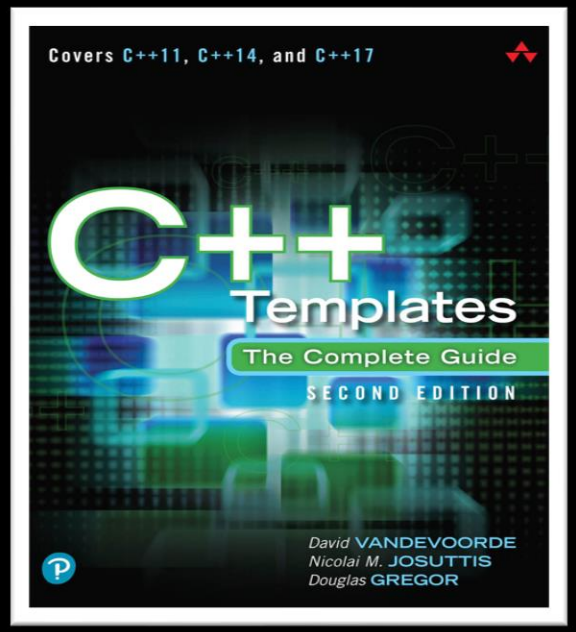
David Vandevoorde, C++ committee and direction group member

Updated January 13

What are similarities and differences between C++ and Swift?

David Vandevoorde, C++ committee and direction group member

Updated January 13



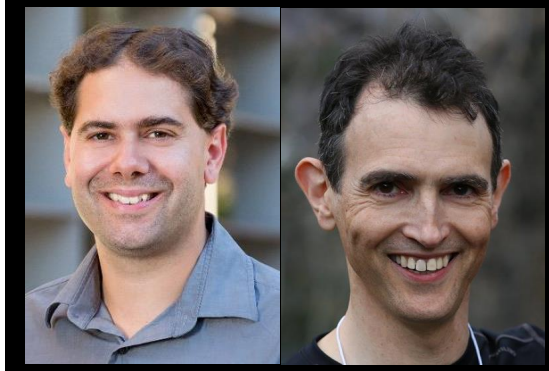
David Vandevoorde, C++ committee and direction group member

Updated January 13

Well, there are many… but I'll keep this relatively brief.

Remember that the original designer of Swift was Chris Lattner, who started and led the LLVM project. LLVM is written in C++ and the Clang C++ compiler is one of the primary drivers for its continued development. So Chris was very familiar with C++ and incorporated his experience with C++ to decide how to design Swift (including what *not* to do). But that's not all. When it came time to select a lead Swift compiler engineer and a lead Swift standard library designer, who did Apple turn to? Doug Gregor for the compiler and Dave Abrahams for the library. Both were some of the main contributors to the C++11 standard and widely recognized as world-class C++ experts. Doug is also a co-author for my "C++ Templates" book — I asked him to join that project because he is a friend, but also because he was behind some of the most fundamental new template work done during the C++11 standardization cycle (including variadic templates and the ill-fated C++0x concepts work).

All that to say that Swift was tremendously influenced by C++. (Apple does not acknowledge this. I've been told that it is because more senior Apple decision-makers dislike C++ at a personal level, in part because of the bitter rivalry between C++ and Objective-C in the 1980s.)




C++

Cppcon
The C++ Conference

C++ now

cpp.chat

[[end of digression]]

82

Influence of C++ on Swift (quora.com)

submitted 10 months ago by Austin_Aaron_Conlon

57 comments share save hide give award report crosspost

this post was submitted on 13 Jan 2020

82 points (92% upvoted)

shortlink: https://redd.it/eo10jo

82

Influence of C++ on Swift (quora.com)

submitted 10 months ago by Austin_Aaron_Conlon

57 comments share save hide give award report crosspost

this post was submitted on 13 Jan 2020

82 points (92% upvoted)

shortlink: https://redd.it/eo10jo

[–] **MrMobster** 2 points 4 hours ago

One of the key purposes of Swift was to replace Objective-C and that's why it has some OOP semantics and dynamism compatible with Obj-C. But protocols are a different thing altogether. They are not classes, but sets of type constraints which also serve as vtables for dynamic dispatch. Swift protocols and Rust traits are very similar. The only major difference that comes to my mind right now is that Swift can have optional protocol members, I don't think that Rust allows that. Both Rust and Swift have extensions, associated type constraints, custom trait implementation mappings etc.

Influence of C++ on Swift (quora.com)

82

submitted 10 months ago by Austin_Aaron_Conlon

57 comments share save hide give award report crosspost

this post was submitted on 13 Jan 2020

82 points (92% upvoted)

shortlink: https://redd.it/eo10jo

[–] **MrMobster** 2 points 4 hours ago

One of the key purposes of Swift was to replace Objective-C and that's why it has some OOP semantics and dynamism compatible with Obj-C. But protocols are a different thing altogether. They are not classes, but sets of type constraints which also serve as vtables for dynamic dispatch. Swift protocols and Rust traits are very similar. The only major difference that comes to my mind right now is that Swift can have optional protocol members, I don't think that Rust allows that. Both Rust and Swift have extensions, associated type constraints, custom trait implementation mappings etc.

[–] **MrMobster** 1 point 2 hours ago

In Obj-C (and it's spiritual ancestor Smalltalk) the notion of protocol is part of the class-based OOP system. In Swift, this notion is generalized to all kinds of types. Combine it with type constraints and you get something quite different from the original OOP construct, even if it looks similar on the surface. This is why I am saying that Swift protocols are more similar to Rust traits. Personally, I prefer the Rust approach (since I think it makes more sense conceptually), but Swift optional protocol members are nice to have as well.

And then of course we have C++ concepts, which are very interesting as well. They are somewhat like traits/protocols but sans the vtable part and with more ways to describe constraints. I am not sure yet however whether concepts can be considered a proper higher-order type system for C++ or whether they are another language within the language for checking types (just like templates are a language within a language for generating types).

permalink embed save parent report give award reply

Introduction

2018-09: Haskell

...

2019-12-08: Protocol Oriented Programming in Swift

2020-01-09: Magic Read Along

2020-01-13: Reddit Article / Quora Answer

Interfaces

Introduction

One of TypeScript's core principles is that type checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

```
function printLabel(labeledObj: { label: string }) {
    console.log(labeledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

The type checker checks the call to printLabel. The printLabel function has a single parameter that requires that the

LLVM DEVELOPERS' MEETING

2017 • SAN JOSE, CA

SLAVA PESTOV
JOHN MCCALL

Implementing Swift Generics

LLVM.org

Constrained Generics

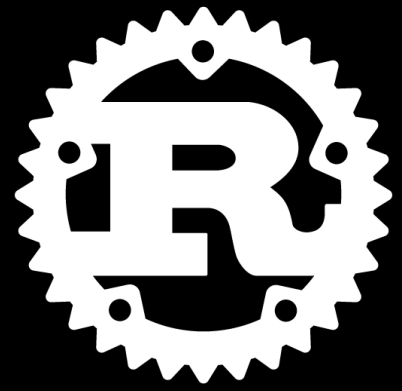
John McCall, Apple

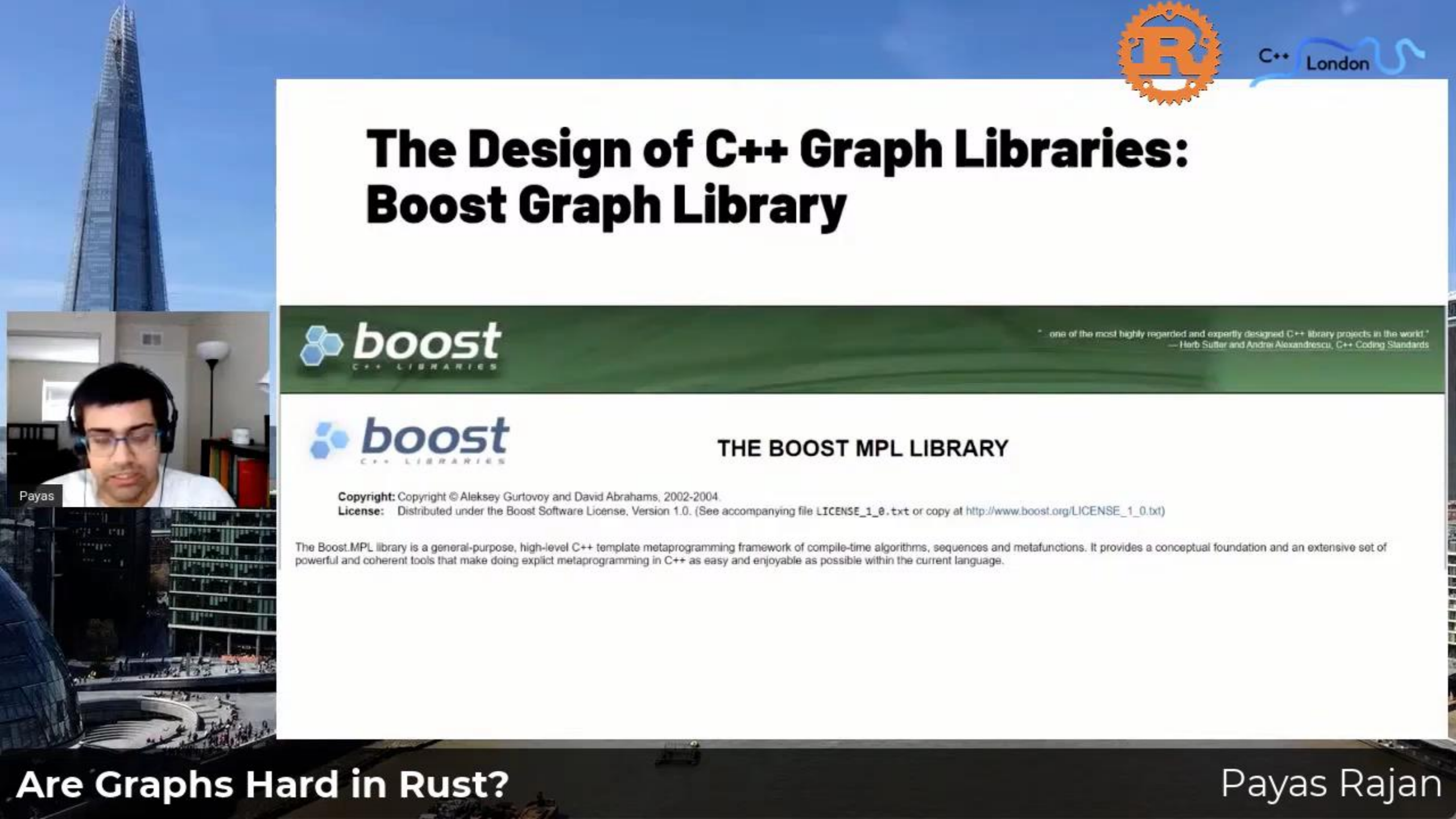

C++
London

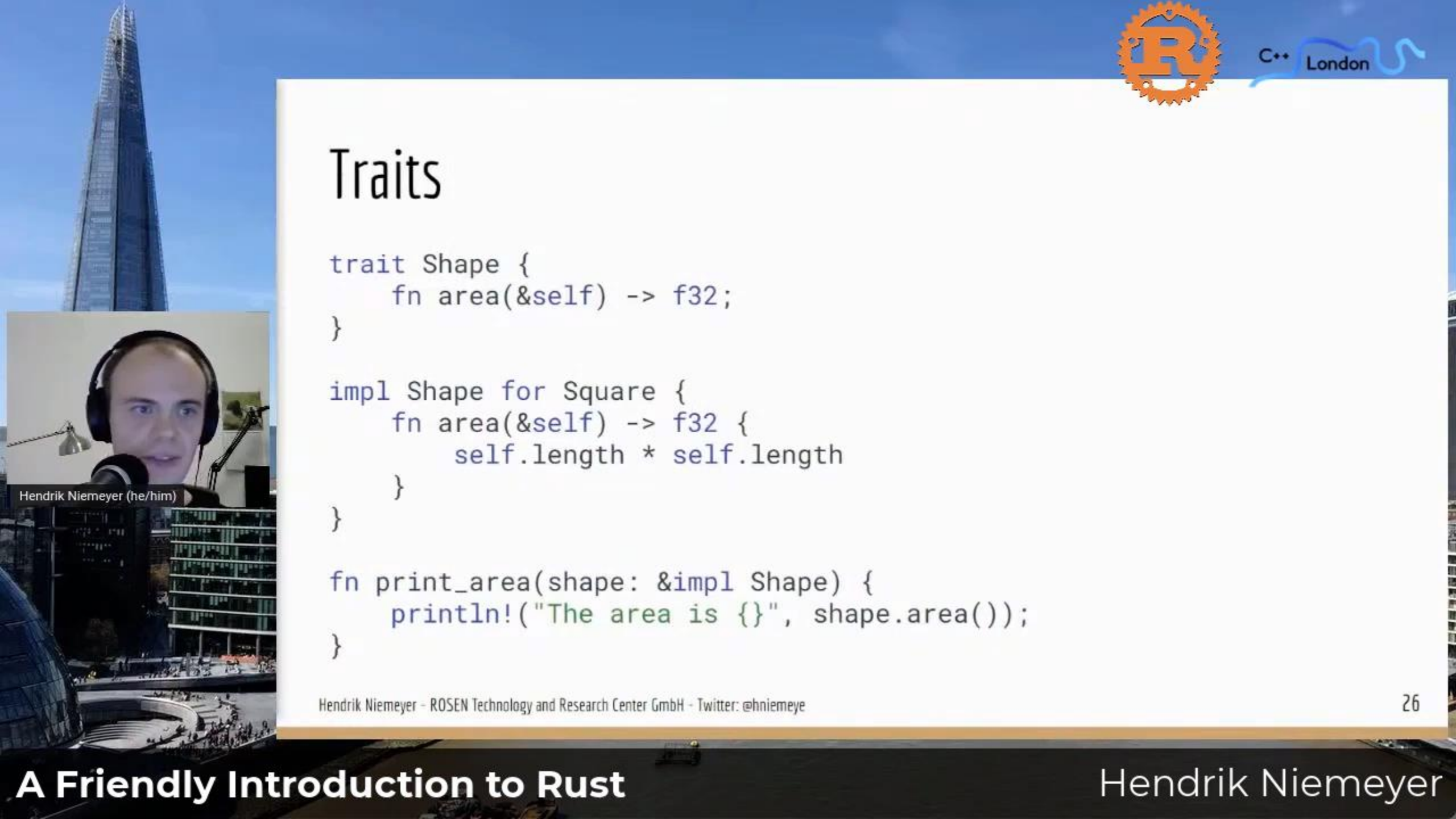

meetup


C++

&




C++ London
The Design of C++ Graph Libraries:
Boost Graph Library
boost
C++ LIBRARIES
" one of the most highly regarded and expertly designed C++ library projects in the world."
— Herb Sutter and Andrei Alexandrescu, C++ Coding Standards
boost
C++ LIBRARIES
THE BOOST MPL LIBRARY
Copyright: Copyright © Aleksey Gurtovoy and David Abrahams, 2002-2004.
License: Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)
The Boost.MPL library is a general-purpose, high-level C++ template metaprogramming framework of compile-time algorithms, sequences and metafunctions. It provides a conceptual foundation and an extensive set of powerful and coherent tools that make doing explict metaprogramming in C++ as easy and enjoyable as possible within the current language.
Payas
Are Graphs Hard in Rust?
Payas Rajan


C++ London
Traits
trait Shape {
fn area(&self) -> f32;
}
impl Shape for Square {
fn area(&self) -> f32 {
self.length * self.length
}
}
fn print_area(shape: &impl Shape) {
println!("The area is {}", shape.area());
}
Hendrik Niemeyer - ROSEN Technology and Research Center GmbH - Twitter: @hniemeye
26
Hendrik Niemeyer (he/him)
A Friendly Introduction to Rust
Hendrik Niemeyer

C++ London

Rust threw away a lot of things.

WEIRD SYNTAX, GREEN THREADS, GARBAGE COLLECTION, TYPE STATES, AND MORE.

The Rust That Could Have Been

Marijn Haverbeke

RustFest Berlin - 2016

ferrous systems


```
type collection<T> =
  obj { fn length() -> int
      ; fn item(int) -> T }

fn is_big(c: obj { fn length() -> int })
  -> bool { ... }

log(is_big(my_collection))
```

Conor Hoekstra
@code_report

"I've been referring to it [Rust] 🦀 as a love child between Haskell and C++."

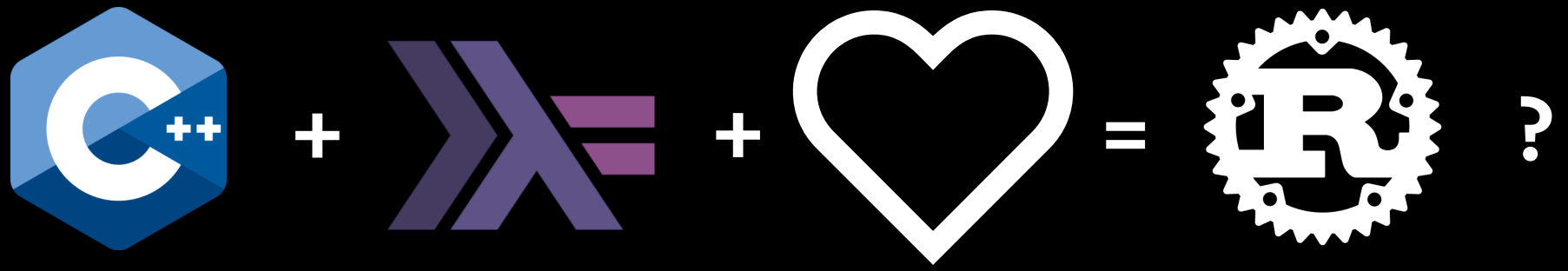
- quote from @roeschinc on Episode 77 of @fngeekery Another awesome episode! #Haskell #cplusplus @rustlang

12:45 PM · Aug 26, 2019 from Sunnyvale, CA · Twitter for Android


Conor Hoekstra
@code_report
"I've been referring to it [Rust] as a love child between Haskell and C++."
- quote from @roeschinc on Episode 77 of @fngeekery Another awesome episode! #Haskell #cplusplus @rustlang
12:45 PM · Aug 26, 2019 from Sunnyvale, CA · Twitter for Android


C++
+
=
?


Conor Hoekstra
@code_report
"I've been referring to it [Rust] as a love child between Haskell and C++."
- quote from @roeschinc on Episode 77 of @fngeekery Another awesome episode! #Haskell #cplusplus @rustlang
12:45 PM · Aug 26, 2019 from Sunnyvale, CA · Twitter for Android


C++
+
+
=
?

C++ Concepts
Rust Traits
Swift Protocols
Haskell Typeclasses
D Type Constraints
TypeScript Structural Interfaces
Go Interfaces
Standard ML Modules
Standard ML Signatures
Java Interfaces
C# Interfaces










C++ Concepts
Rust Traits
Swift Protocols
Haskell Typeclasses
D Type Constraints
TypeScript Structural Interfaces
Go Interfaces
Standard ML Modules
Standard ML Signatures
Java Interfaces
C# Interfaces



Agenda

1. Introduction / History 🙃
2. Generics / Parametric Polymorphism
3. Example #1
4. Example #2
5. Final Thoughts

6. Bonus Question

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in `3*3`) and multiplication of floating point values (as in `3.14*3.14`).

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. A typical example is the `length` function, which acts in the same way on a list of integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a program using type classes to an equivalent program that does not use overloading. The translated programs are typable in the (ungeneralised) Hindley/Milner type system.

The body of this paper gives an informal introduction to type classes and the translation rules, while an appendix gives formal rules for typing and translation, in the form of inference rules (as in [DM82]). The translation rules provide a semantics for type classes. They also provide one possible implementation technique: if desired, the new system could be added to an existing language with Hindley/Milner types simply by writing a pre-processor.

introduction to type classes, and defines them formally by means of type inference rules.

1 Introduction

Strachey chose the adjectives *ad-hoc* and *parametric* to distinguish two varieties of *polymorphism* [Str67].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in `3*3`) and multiplication of floating point values (as in `3.14*3.14`).

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same

to distinguish two varieties of *polymorphism* [Strachey].

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in `3*3`) and multiplication of floating point values (as in `3.14*3.14`).

Parametric polymorphism occurs when a function is defined over a range of types, acting in the same way for each type. A typical example is the `length` function, which acts in the same way on a list of

```go
func main() {
    var a, b int = 1, 2
    var c = math.Min(a, b)
    fmt.Println(a + b)
}
```

```
function IsLeapYear(Year: Integer): Boolean;
begin
    if Year mod 400 = 0 then
        IsLeapYear := True
    else if Year mod 100 = 0 then
        IsLeapYear := False
    else if Year mod 4 = 0 then
        IsLeapYear := True
    else
        IsLeapYear := False
end;
```

```go
func main() {
    var a, b int = 1, 2
    var c = math.Min(a, b)
    fmt.Println(a + b)
}
```

```
// FAIL: cannot use a (type int) as type
// float64 in argument to math.Min

func main() {
    var a, b int = 1, 2
    var c = math.Min(a, b)
    fmt.Println(a + b)
}
```

Would you be interested in helping us get polymorphism right (and/or figuring out what “right” means) for some future version of Go? — Rob Pike

Lightweight Parametric Polymorphism for Oberon

Paul Roe and Clemens Szyperski

Queensland University of Technology, Brisbane QLD 4001, Australia

Abstract. Strongly typed polymorphism is necessary for expressing safe reusable code. Two orthogonal forms of polymorphism exist: inclusion and parametric, the Oberon language only supports the former. We describe a simple extension to Oberon to support parametric polymorphism. The extension is in keeping with the Oberon language: it is simple and has an explicit cost. In the paper we motivate the need for parametric polymorphism and describe an implementation in terms of translating extended Oberon to standard Oberon.

1 Introduction

A key goal of Software Engineering is to support the production and use of reusable code. Reusable code, by definition, is "generic" i.e. applicable in a number of different contexts. To guarantee that code is reused correctly strong typing is desirable. Genericity in code can best be expressed by polymorphic types. Two different forms of polymorphism have been identified: inclusion and parametric [2]. In theory inclusion and parametric polymorphism are orthogonal concepts and neither can be used to satisfactorily replace the other.

3 Parametric Polymorphism for Oberon

We introduce parametric polymorphism via our previous example. A type may be parametrised on types in much the same way as a procedure may be parametrised on values.

3 Parametric Polymorphism for Oberon

We introduce parametric polymorphism via our previous example. A type may be parametrised on types in much the same way as a procedure may be parametrised on values.

Type Constraints are to Types as Types are to Values

Agenda

1. Introduction / History 🙃
2. Generics / Parametric Polymorphism
3. Example #1
4. Example #2
5. Final Thoughts

Example #1

Adding Two Integers

```
auto add(int a, int b) -> int {
    return a + b;
}
```

```cpp
auto add(int a, int b) -> int { return a + b; }
```

```d
int add(int a, int b) { return a + b; }
```

```rust
fn add(a: i32, b: i32) -> i32 { a + b }
```

```swift
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```haskell
add :: Int -> Int -> Int
add a b = a + b
```

```cpp
auto add(int a, int b) -> int { return a + b; }
```

```d
int add(int a, int b) { return a + b; }
```

```rust
fn add(a: i32, b: i32) -> i32 { a + b }
```

```swift
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```haskell
add :: Int -> Int -> Int
add a b = a + b
```

```
auto add(int a, int b)        -> int { return a + b; }
```

```
int  add(int a, int b)              { return a + b; }
```

```
fn   add(a: i32, b: i32)      -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

	Keyword Before Function	Integer	Trailing Return Type	Return Necessary
				
				
				
				
				

```
auto add(int a, int b)        -> int { return a + b; }
```

```
int  add(int a, int b)              { return a + b; }
```

```
fn   add(a: i32, b: i32)      -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

	Keyword Before Function	Integer	Trailing Return Type	Return Necessary
	auto			
	*type*			
	fn			
	func			
	-			

```
auto add(int a, int b)        -> int { return a + b; }
```

```
int  add(int a, int b)              { return a + b; }
```

```
fn   add(a: i32, b: i32)      -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

	Keyword Before Function	Integer	Trailing Return Type	Return Necessary
	auto	int (int32_t)		
	*type*	int		
	fn	i32		
	func	Int		
	-	Int		

```
auto add(int a, int b)        -> int { return a + b; }
```

```
int  add(int a, int b)              { return a + b; }
```

```
fn   add(a: i32, b: i32)      -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

	Keyword Before Function	Integer	Trailing Return Type	Return Necessary
	auto	int (int32_t)	YES	
	*type*	int	NO	
	fn	i32	YES	
	func	Int	YES	
	-	Int	YES	

```
auto add(int a, int b)        -> int { return a + b; }
```

```
int  add(int a, int b)             { return a + b; }
```

```
fn   add(a: i32, b: i32)      -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

	Keyword Before Function	Integer	Trailing Return Type	Return Necessary
	auto	int (int32_t)	YES	YES
	*type*	int	NO	YES
	fn	i32	YES	NO
	func	Int	YES	NO
	-	Int	YES	NO

[[digression]]

	Groovy	**def**	Doc
	Elixir	**def**	Doc
	Crystal	**def**	Doc
	Python	**def**	Doc
	Ruby	**def**	Doc
	Scala	**def**	Doc
	Fortran	**function**	Doc
	JavaScript	**function**	Doc
	Julia	**function**	Doc
	Lua	**function**	Doc
	Go	**func**	Doc
	Nim	**func**	Doc
	Swift	**func**	Doc
	Rust	**fn**	Doc
	Zig	**fn**	Doc
	Clojure	**defn**	Doc
	Kotlin	**fun**	Doc
	Racket	**define**	Doc
	C++	**auto**	Doc
	LISP	**defun**	Doc

	Groovy	`def`	Doc
	Elixir	`def`	Doc
	Crystal	`def`	Doc
	Python	`def`	Doc
	Ruby	`def`	Doc
	Scala	`def`	Doc
	Fortran	`function`	Doc

	JavaScript	`function`	Doc
	Julia	`function`	Doc
	Lua	`function`	Doc
	Go	`func`	Doc
	Nim	`func`	Doc
	Swift	`func`	Doc
	Rust	`fn`	Doc

	Rust	**fn**	Doc
	Zig	**fn**	Doc
	Clojure	**defn**	Doc
	Kotlin	**fun**	Doc
	Racket	**define**	Doc
	C++	**auto**	Doc
	LISP	**defun**	Doc

[[end of digression]]

```cpp
auto add(int a, int b) -> int { return a + b; }
```

```d
int add(int a, int b) { return a + b; }
```

```rust
fn add(a: i32, b: i32) -> i32 { a + b }
```

```swift
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```haskell
add :: Int -> Int -> Int
add a b = a + b
```

```cpp
template <typename T>
auto add(T a, T b) -> T { return a + b; }
```

```d
int add(int a, int b) { return a + b; }
```

```rust
fn add(a: i32, b: i32) -> i32 { a + b }
```

```swift
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```haskell
add :: Int -> Int -> Int
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
int add(int a, int b) { return a + b; }
```

```
fn add(a: i32, b: i32) -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add(a: i32, b: i32) -> i32 { a + b }
```

```
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

```cpp
auto add(auto a, auto b) { return a + b; }
```

```d
T add(T)(T a, T b) { return a + b; }
```

```rust
fn add<T>(a: T, b: T) -> T { a + b }
```

```swift
func add(_ a: Int, _ b: Int) -> Int { a + b }
```

```haskell
add :: Int -> Int -> Int
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T>(a: T, b: T) -> T { a + b }
```

```
func add<T>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: Int -> Int -> Int
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T>(a: T, b: T) -> T { a + b }
```

```
func add<T>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: t -> t -> t
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
```

```
func add<T>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: t -> t -> t
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
```

```
func add<T: Numeric>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: t -> t -> t
add a b = a + b
```

```
auto add(auto a, auto b) { return a + b; }
```

```
T add(T)(T a, T b) { return a + b; }
```

```
fn add<T: std::ops::Add<Output = T>>(a: T, b: T) -> T { a + b }
```

```
func add<T: Numeric>(_ a: T, _ b: T) -> T { a + b }
```

```
add :: Num t => t -> t -> t
add a b = a + b
```

Type Constraints

"constrain"

vs

Type Classes

"consent"

```
template< typename T>
auto f(T t) -> T
```

∞

```
f :: t -> t
```

1

```
template< typename T>
auto f(T t) -> T
```

∞

```
id :: t -> t
```

1

std::identity

Defined in header <functional>

```
struct identity;
```
(since C++20)

LLVM DEVELOPERS' MEETING

2017 · SAN JOSE, CA

SLAVA PESTOV
JOHN MCCALL

Implementing Swift Generics

LLVM.org

Constrained Generics

John McCall, Apple

LLVM DEVELOPERS' MEETING

2017 • SAN JOSE, CA

SLAVA PESTOV
JOHN MCCALL

Implementing Swift Generics

LLVM.org

Swift Generics

- Bounded parametric polymorphism
  - Similar to Java, C#, Haskell, ML…
  - Constraints described in terms of "protocols"

- Bounded parametric polymorphism

Type Constraints

"constrain"

vs

Type Classes

"consent"

Agenda

1. Introduction / History 🙃
2. Generics / Parametric Polymorphism
3. Example #1
4. Example #2
5. Final Thoughts

Example #2

Shapes: Circle & Rectangle

```cpp
class circle {
    float r;
public:
    explicit circle(float radius) : r{radius} {}
    auto name()      const -> std::string { return "Circle"; }
    auto area()      const -> float       { return pi * r * r; }
    auto perimeter() const -> float       { return 2 * pi * r; }
};

class rectangle {
    float w, h;
public:
    explicit rectangle(float height, float width) : h{height}, w{width} {}
    auto name()      const -> std::string { return "Rectangle"; }
    auto area()      const -> float       { return w * h; }
    auto perimeter() const -> float       { return 2 * w + 2 * h; }
};
```

```
class Circle {
    float r;
    this(float radius) { r = radius; }
    string name()      const { return "Circle"; }
    float  area()      const { return PI * r * r; }
    float  perimeter() const { return 2 * PI * r; }
}

class Rectangle {
    float w, h;
    this(float width, float height) { w = width; h = height; }
    string name()      const { return "Rectangle"; }
    float  area()      const { return w * h; }
    float  perimeter() const { return 2 * w + 2 * h; }
}
```





```rust
struct Circle    { r: f32 }
struct Rectangle { w: f32, h: f32 }

impl Circle {
    fn name(&self)      -> String { "Circle".to_string() }
    fn area(&self)      -> f32    { PI * self.r * self.r }
    fn perimeter(&self) -> f32    { 2.0 * PI * self.r }
}

impl Rectangle {
    fn name(&self)      -> String { "Rectangle".to_string() }
    fn area(&self)      -> f32    { self.w * self.h }
    fn perimeter(&self) -> f32    { 2.0 * self.w + 2.0 * self.h }
}
```

```
class Rectangle {
    let w, h: Float
    init(w: Float, h: Float) { self.w = w; self.h = h }
    func name()      -> String { "Rectangle" }
    func area()      -> Float  { w * h }
    func perimeter() -> Float  { 2 * w + 2 * h }
}

class Circle {
    let r: Float
    init(r: Float) { self.r = r }
    func name()      -> String { "Circle" }
    func area()      -> Float  { Float.pi * r * r }
    func perimeter() -> Float  { 2 * Float.pi * r }
}
```

```haskell
data Circle    = Circle {r :: Float}
data Rectangle = Rectangle {w :: Float, h :: Float}

name :: Circle -> String
name (Circle _) = "Circle"

area :: Circle -> Float
area (Circle r) = pi * r ^ 2

perimeter :: Circle -> Float
perimeter (Circle r) = 2 * pi * r

name :: Rectangle -> String
name (Rectangle _ _) = "Rectangle"

area :: Rectangle -> Float
area (Rectangle w h) = w * h

perimeter :: Rectangle -> Float
perimeter (Rectangle w h) = 2 * w + 2 * h
```

```haskell
data Circle    = Circle {r :: Float}
data Rectangle = Rectangle {w :: Float, h :: Float}

circleName :: Circle -> String
circleName (Circle _) = "Circle"

circleArea :: Circle -> Float
circleArea (Circle r) = pi * r ^ 2

circlePerimeter :: Circle -> Float
circlePerimeter (Circle r) = 2 * pi * r

rectangleName :: Rectangle -> String
rectangleName (Rectangle _ _) = "Rectangle"

rectangleArea :: Rectangle -> Float
rectangleArea (Rectangle w h) = w * h

rectanglePerimeter :: Rectangle -> Float
rectanglePerimeter (Rectangle w h) = 2 * w + 2 * h
```





```
void print_shape_info(auto s) {
    fmt::print("Shape: {}\nArea:  {}\nPerim: {}\n\n",
        s.name(), s.area(), s.perimeter());
}
```





```d
void printShapeInfo(T)(T s) {
   writeln("Shape: ",   s.name(),
           "\nArea:  ", s.area(),
           "\nPerim: ", s.perimeter(), "\n");

}
```





```
fn print_shape_info<T>(s: T) {
    println!("Shape: {}\nArea:  {}\nPerim: {}\n",
        s.name(), s.area(), s.perimeter());
}
```





```
// 27 |          s.name(), s.area(), s.perimeter());
//    |            ^^^^ method not found in `T`
```

```
fn print_shape_info<T>(s: T) {
    println!("Shape: {}\nArea:  {}\nPerim: {}\n",
        s.name(), s.area(), s.perimeter());
}
```





```
// 27 |          s.name(), s.area(), s.perimeter());
//    |            ^^^^ method not found in `T`
//    |
//    = help: items from traits can only be used if the type
//            parameter is bounded by the trait
//      help: the following trait defines an item `name`,
//            perhaps you need to restrict type parameter `T` with it:
//    |
// 25 | fn print_shape_info<T: Shape>(s: T) {
//    |                      ^^^^^^^^
```

```rust
fn print_shape_info<T>(s: T) {
    println!("Shape: {}\nArea:  {}\nPerim: {}\n",
        s.name(), s.area(), s.perimeter());
}
```





```
func printShapeInfo<T>(_ s: T) {
    print("Shape: \(s.name())\n" +
          "Area:  \(s.area())\n" +
          "Perim: \(s.perimeter())\n")
}
```





```
// error: value of type 'T' has no member 'name'

func printShapeInfo<T>(_ s: T) {
    print("Shape: \(s.name())\n" +
          "Area:  \(s.area())\n" +
          "Perim: \(s.perimeter())\n")
}
```









```
void print_shape_info(auto s) {
    fmt::print("Shape: {}\nArea:  {}\nPerim: {}\n\n",
        s.name(), s.area(), s.perimeter());
}
```





```cpp
template <typename S>
concept string = is_string<char, S>::value;

template <typename S>
concept shape = requires(S s) {
    { s.name() }      -> string;
    { s.area() }      -> std::floating_point;
    { s.perimeter() } -> std::floating_point;
};


void print_shape_info(auto s) {
    fmt::print("Shape: {}\nArea:  {}\nPerim: {}\n\n",
        s.name(), s.area(), s.perimeter());
}
```





```cpp
template <typename S>
concept string = is_string<char, S>::value;

template <typename S>
concept shape = requires(S s) {
    { s.name() }      -> string;
    { s.area() }      -> std::floating_point;
    { s.perimeter() } -> std::floating_point;
};


void print_shape_info(shape auto s) {
    fmt::print("Shape: {}\nArea:  {}\nPerim: {}\n\n",
        s.name(), s.area(), s.perimeter());
}
```





```d
void printShapeInfo(T)(T s) {
   writeln("Shape: ",   s.name(),
           "\nArea:  ", s.area(),
           "\nPerim: ", s.perimeter(), "\n");

}
```





```d
template shape(T) {
    const shape = __traits(compiles, (T t) {
        t.name();
        t.area();
        t.perimeter();
    });
}

void printShapeInfo(T)(T s) {
   writeln("Shape: ",   s.name(),
           "\nArea:  ", s.area(),
           "\nPerim: ", s.perimeter(), "\n");
}
```





```
template shape(T) {
    const shape = __traits(compiles, (T t) {
        t.name();
        t.area();
        t.perimeter();
    });
}

void printShapeInfo(T)(T s)
    if (shape!(T))
{
   writeln("Shape: ",   s.name(),
           "\nArea:  ", s.area(),
           "\nPerim: ", s.perimeter(), "\n");
}
```





```
impl Circle    { ... }
impl Rectangle { ... }

fn print_shape_info<T>(s: T) {
    println!("Shape: {}\nArea:  {}\nPerim: {}\n",
        s.name(), s.area(), s.perimeter());
}
```





```rust
trait Shape {
    fn name(&self)      -> String;
    fn area(&self)      -> f32;
    fn perimeter(&self) -> f32;
}

impl Shape for Circle    { ... }
impl Shape for Rectangle { ... }

fn print_shape_info<T>(s: T) {
    println!("Shape: {}\nArea:  {}\nPerim: {}\n",
        s.name(), s.area(), s.perimeter());
}
```





```rust
trait Shape {
    fn name(&self)      -> String;
    fn area(&self)      -> f32;
    fn perimeter(&self) -> f32;
}

impl Shape for Circle    { ... }
impl Shape for Rectangle { ... }

fn print_shape_info<T: Shape>(s: T) {
    println!("Shape: {}\nArea:  {}\nPerim: {}\n",
        s.name(), s.area(), s.perimeter());
}
```





```
class Rectangle { ... }
class Circle    { ... }

func printShapeInfo<T>(_ s: T) {
    print("Shape: \(s.name())\n" +
          "Area:  \(s.area())\n" +
          "Perim: \(s.perimeter())\n")
}
```





```
protocol Shape {
    func name()      -> String
    func area()      -> Float
    func perimeter() -> Float
}

class Rectangle : Shape { ... }
class Circle    : Shape { ... }

func printShapeInfo<T>(_ s: T) {
    print("Shape: \(s.name())\n" +
          "Area:  \(s.area())\n" +
          "Perim: \(s.perimeter())\n")
}
```





```
protocol Shape {
    func name()      -> String
    func area()      -> Float
    func perimeter() -> Float
}

class Rectangle : Shape { ... }
class Circle    : Shape { ... }

func printShapeInfo<T: Shape>(_ s: T) {
    print("Shape: \(s.name())\n" +
          "Area:  \(s.area())\n" +
          "Perim: \(s.perimeter())\n")
}
```





```haskell
data Circle    = Circle {r :: Float}
data Rectangle = Rectangle {w :: Float, h :: Float}

name :: Circle -> String
name (Circle _) = "Circle"

area :: Circle -> Float
area (Circle r) = pi * r ^ 2

perimeter :: Circle -> Float
perimeter (Circle r) = 2 * pi * r

name :: Rectangle -> String
name (Rectangle _ _) = "Rectangle"

area :: Rectangle -> Float
area (Rectangle w h) = w * h

perimeter :: Rectangle -> Float
perimeter (Rectangle w h) = 2 * w + 2 * h
```

```haskell
class Shape a where
    name      :: a -> String
    area      :: a -> Float
    perimeter :: a -> Float

data Circle    = Circle {r :: Float}
data Rectangle = Rectangle {w :: Float, h :: Float}

instance Shape Circle where
    name      (Circle _) = "Circle"
    area      (Circle r) = pi * r ^ 2
    perimeter (Circle r) = 2 * pi * r

instance Shape Rectangle where
    name      (Rectangle _ _) = "Rectangle"
    area      (Rectangle w h) = w * h
    perimeter (Rectangle w h) = 2 * w + 2 * h

printShapeInfo :: Shape a => a -> IO()
printShapeInfo s = putStrLn ("Shape: " ++ (name s)           ++ "\n" ++
                             "Area:  " ++ show (area s)      ++ "\n" ++
                             "Perim: " ++ show (perimeter s) ++ "\n")
```





```haskell
class Shape a where
    name      :: a -> String
    area      :: a -> Float
    perimeter :: a -> Float

instance Shape Circle    where ...
instance Shape Rectangle where ...

printShapeInfo :: Shape a => a -> IO()
printShapeInfo s = putStrLn ("Shape: " ++ (name s)           ++ "\n" ++
                             "Area:  " ++ show (area s)      ++ "\n" ++
                             "Perim: " ++ show (perimeter s) ++ "\n")
```





Agenda

1. Introduction / History 🙃
2. Generics / Parametric Polymorphism
3. Example #1
4. Example #2
5. Final Thoughts

Final Thoughts 😀

1. 
2. 
3. 
4. 
5. 

Time To Implement

Least to Greatest

Final Thoughts 😀

1.  25
2.  33
3.  36
4.  44
5.  60

LOC:

Lines
Of
Code

Final Thoughts 😀

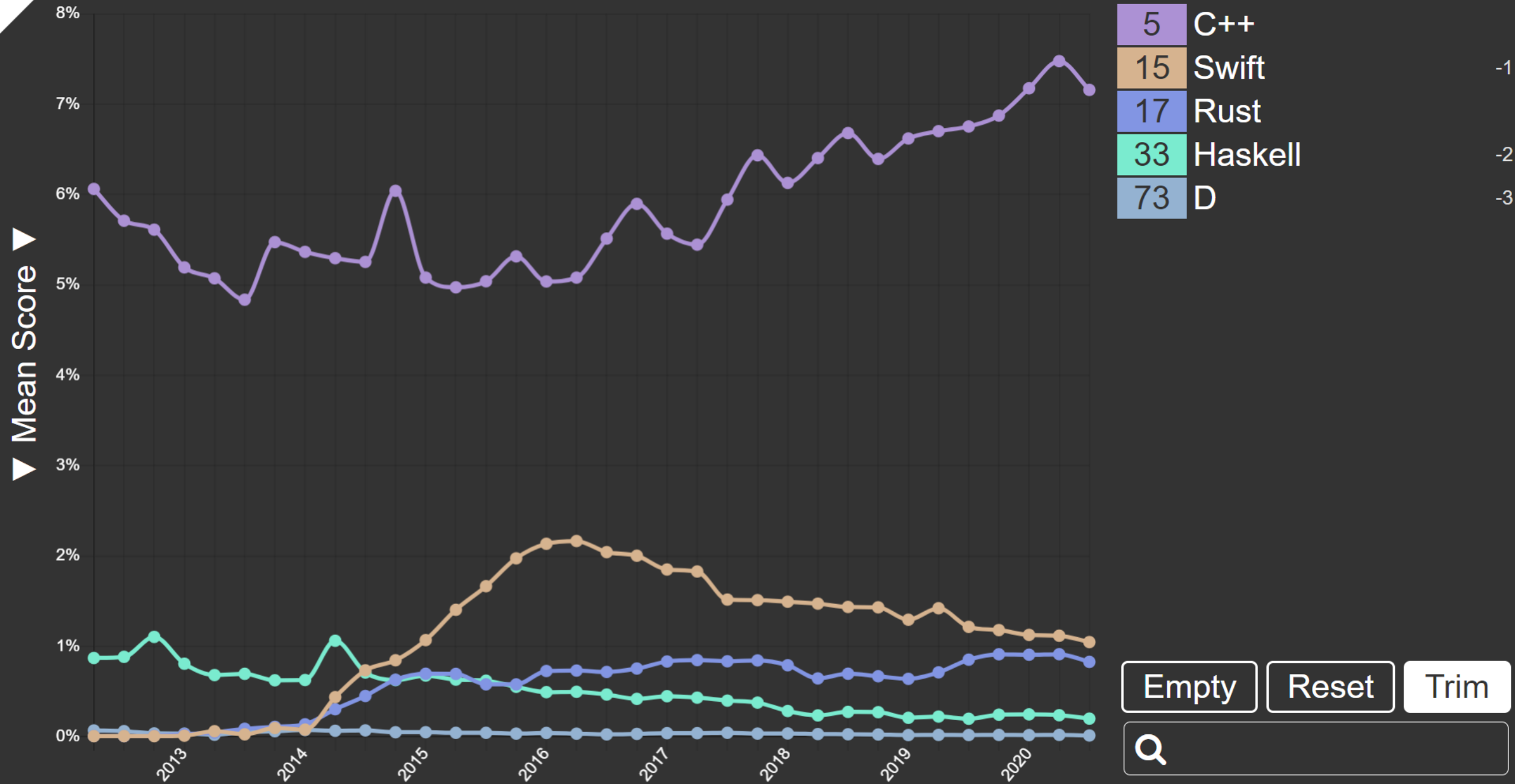
1. 
   - Half the time I just "guessed right"
   - PDoC: Progressive Disclosure of Complexity
   - Defaults are all correct
2. 
   - Compiler messages are amazing
   - Defaults are all correct
3. 
   - Seems too similar to C++
   - Added complexity in some places

😊-ness

4. 
   - Steep learning curve
   - Compiler messages are bad
5. 
   - 40 years of history = less elegance
   - Most defaults are wrong
   - C++20 is a work in progress

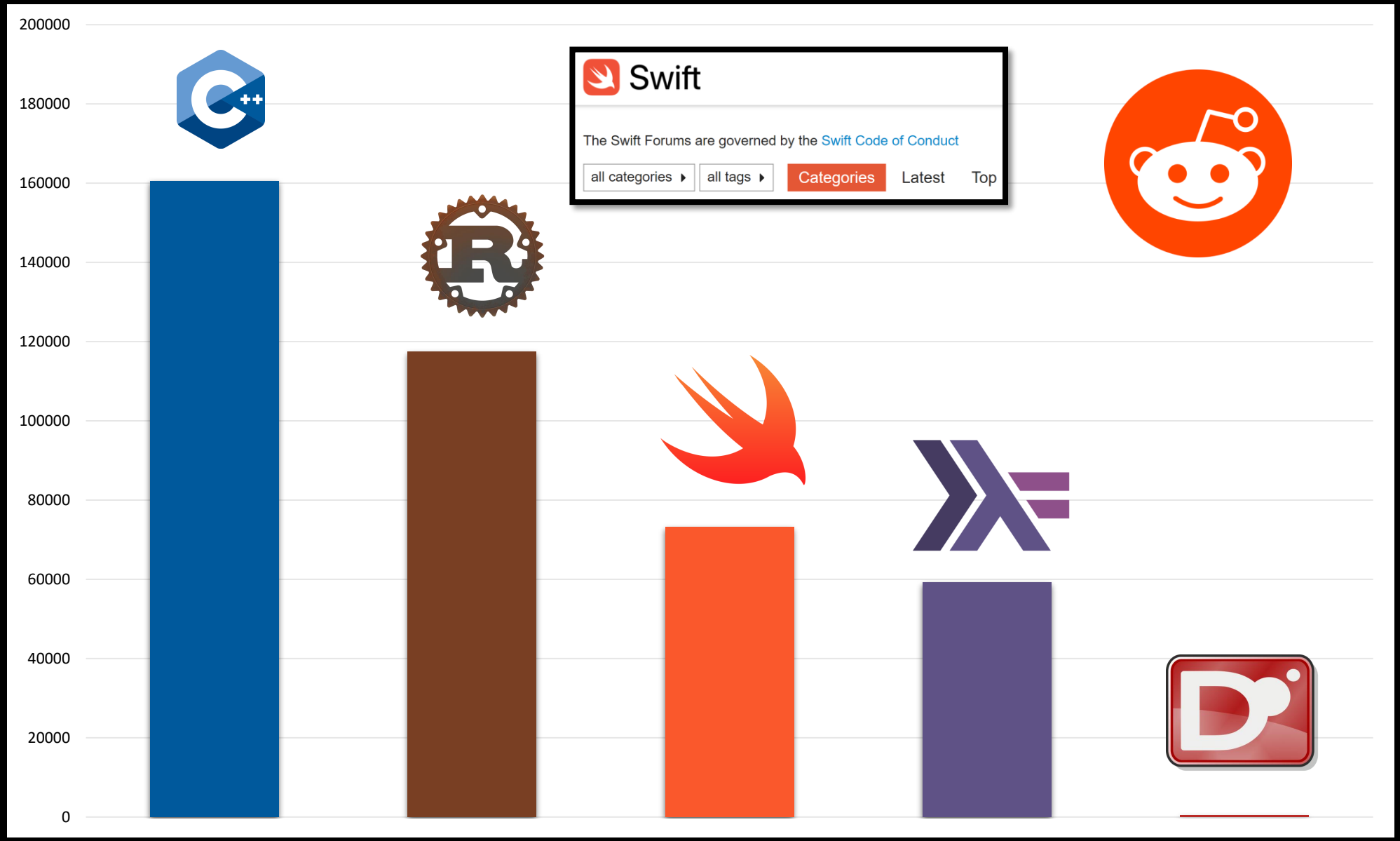
Languish

Programming Language Trends
... for more, subscribe to Context Free



Empty | Reset | Trim

	Languish	TIOBE	PYPL	RedMonk	Google Trends
	5	4	6	5	1
	15	14	9	11	2
	17	25	16	20	3
	33	*41*	27	-	4
	73	*32*	-	-	-


200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
Swift
The Swift Forums are governed by the Swift Code of Conduct
all categories
all tags
Categories
Latest
Top

Final Thoughts 😃

😊-ness

1. 
   - Half the time I just “guessed right”
   - PDoC: Progressive Disclosure of Complexity
   - Defaults are all correct
2. 
   - Compiler messages are amazing
   - Defaults are all correct
3. 
   - Seems too similar to C++
   - Added complexity in some places
4. 
   - Steep learning curve
   - Compiler messages are bad
5. 
   - 40 years of history = less elegance
   - Most defaults are wrong
   - C++20 is a work in progress

#include

https://github.com/codereport/Talks

Podcast Links:

Podcast	Guest	Date	Link
Magic Read Along	-	2016-12-01	Episode 28: I Am Not Full of Beans!
The Swift Community Podcast	-	2019 - 2020	All Episodes (1 - 8)
Swift by Sundell	Dave Abrahams	2020-04-23	Polymorphic Interfaces
Swiftly Speaking	Chris Lattner	2020-06-18	Episode 11
cpp.chat	Conor Hoekstra	2020-10-08	Episode 75: I Really Like Sugar
Lex Fridman Podcast	Chris Lattner	2020-10-18	Episode 131: The Future of Computing and Programming Languages
cpp.chat	Panel	2020-10-20	Episode 78: The C++ and Rust Round Table

YouTube Video Links:

Speaker	Conference/Meetup	Year	Talk
Panel	LangNext	2014	C++ vs Rust vs D vs Go
Chris Lattner	WWDC	2014	Swift Introduction
Dave Abrahams	WWDC	2015	Protocol-Oriented Programming in Swift
Scott Schurr	CppCon	2015	constexpr: Applications
Marijn Haverbeke	RustFest	2016	The Rust That Could Have Been
Slava Pestov John McCall	LLVM Developers' Meeting	2017	Implementing Swift Generics
Bryan Cantrill	Systems We Run Meetup	2018	The Summer of RUST
Sean Allen	YouTube Video	2019	Swift Programming Language Introduction - A Brief History
Daniel Steinberg	GOTO	2019	What's New in Swift

Philip Wadler	Chalmers FP Seminar Series	2020	Featherweight Go
Context Free (Tom Palmer)	YouTube Video	2020	Demo: C++20 Concepts Feature
Payas Rajan	C++ London Meetup	2020	Are Graphs Hard in Rust?
Henrik Niemeyer	C++ London Meetup	2020	A Friendly Introduction to Rust
James Munns	C++ London Meetup	2020	Access All Arenas

Paper Links:

Author	Date	Link
Philip Wadler Stephen Blott	1988	How to make ad hoc polymorphism less ad hoc
Paul Roe Clemens Szyperski	1997	Lightweight Parametric Polymorphism for Oberon
Jeremy G. Siek Andrew Lumsdaine	2008	A language for generic programming in the large
Yizhou Zhang Andrew C. Myers	2020	Unifying Interfaces, Type Classes, and Family Polymorphism

Article/Other Links:

Author	Site	Date	Link
Philip Fong	URegina	2008-04-02	CS 115: Parametric Polymorphism: Template Functions
Zuu	StackOverflow	2016-04-16	Why is C++ said not to support parametric polymorphism?
matt_d	HackerNews	2016-12-16	Concepts: The Future of Generic Programming
Austin_Aaron_Conlon	reddit/cpp	2020-01-13	Influence of C++ on Swift
David Vandevoorde	Quora	2020-01-13	What are similarities and differences between C++ and Swift?
-	Wikipedia	-	Parametric Polymorphism

Thank You!

Conor Hoekstra

code_report

#include <C++>

Questions / Feedback?

Conor Hoekstra

code_report

#include <C++>