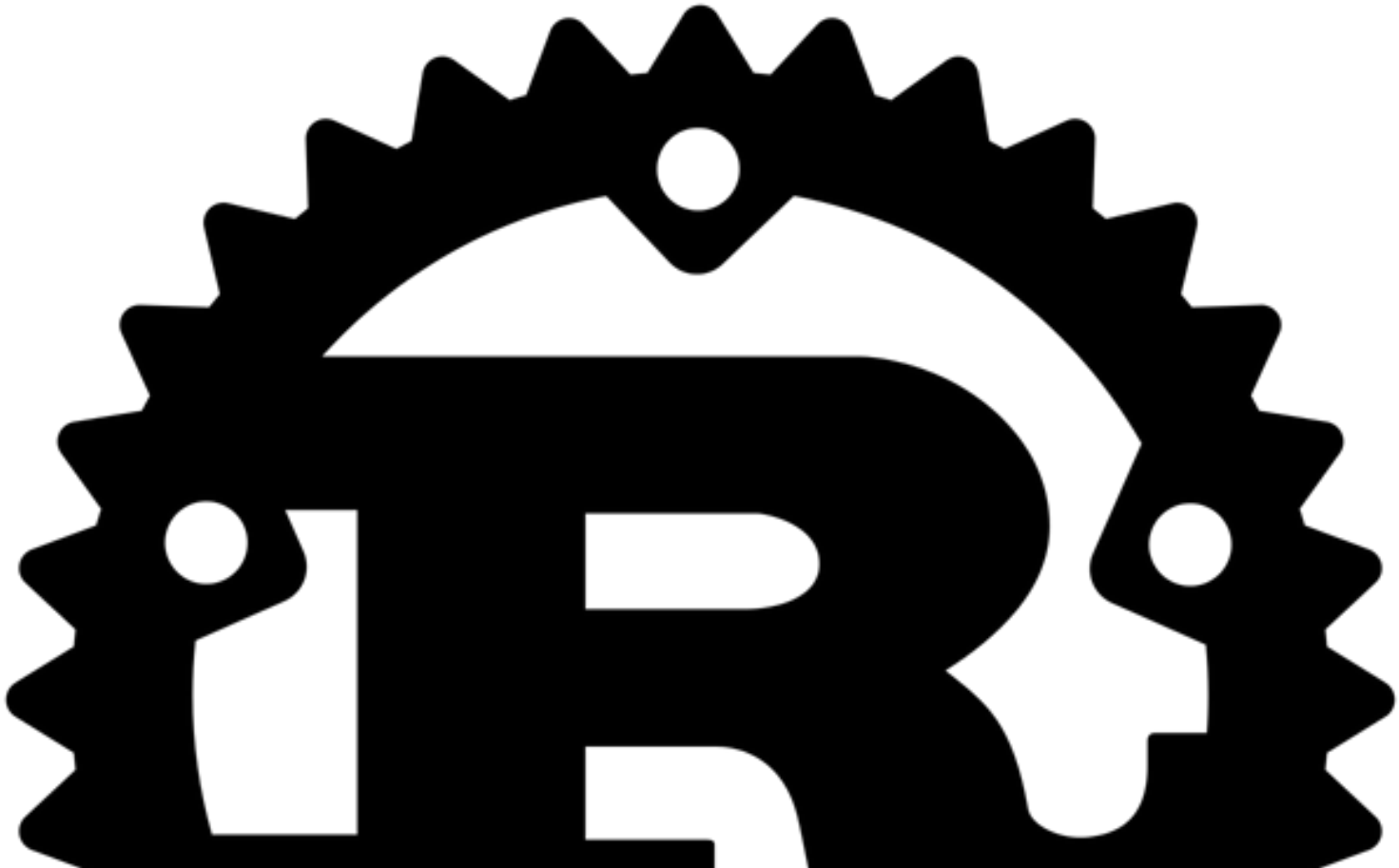
Použití programovacího jazyka Rust na produkčním systému

- Pavel Tišnovský, Red Hat
  - ptisnovs@redhat.com

- Prezentace:
  - https://tisnik.github.io/presentations/openalt2020/rust_in_production.html
- Zdrojový kód prezentace:
  - https://github.com/tisnik/presentations/blob/master/openalt2020/rust_in_production.md

Anotace

Jazyk Rust se stává mezi programátory stále populárnější alternativou k C++ na straně jedné a k jazykům vybavených automatickým správcem paměti (GC) na straně druhé. Na této přednášce si řekneme, které vlastnosti Rustu zjednodušují jeho použití v produkčních systémech, které knihovny se nejčastěji používají a jak se aplikace psané v Rustu zabezpečují.



Obsah přednášky

- Požadavky na produkční jazyk v současnosti
- Popularita a rozšířenost Rustu
- Charakteristické rysy Rustu
- Rust versus C/C++
- Rust versus Go
- Komunikace s překladačem
- Datové typy
- Zajímavé prvky jazyka
- Přístup k OO v Rustu
- Správa paměti
- Vlákna
- Testování
- Správce balíčků (Cargo)
- Vybrané balíčky
- Nasazení aplikací
- Web Assembly
- Rozhraní s Pythonem
- Dokumentace

Požadavky na produkční jazyk v současnosti

- Korektnost programů
- Udržovatelnost
- Bezpečnost
- Stabilita ekosystému
- Dostatek vývojářů
- Nároky na systémové zdroje
  - Více RAM -> větší náklady v kontejnerizovaném světě
- API a komunikace s dalšími (mikro)službami

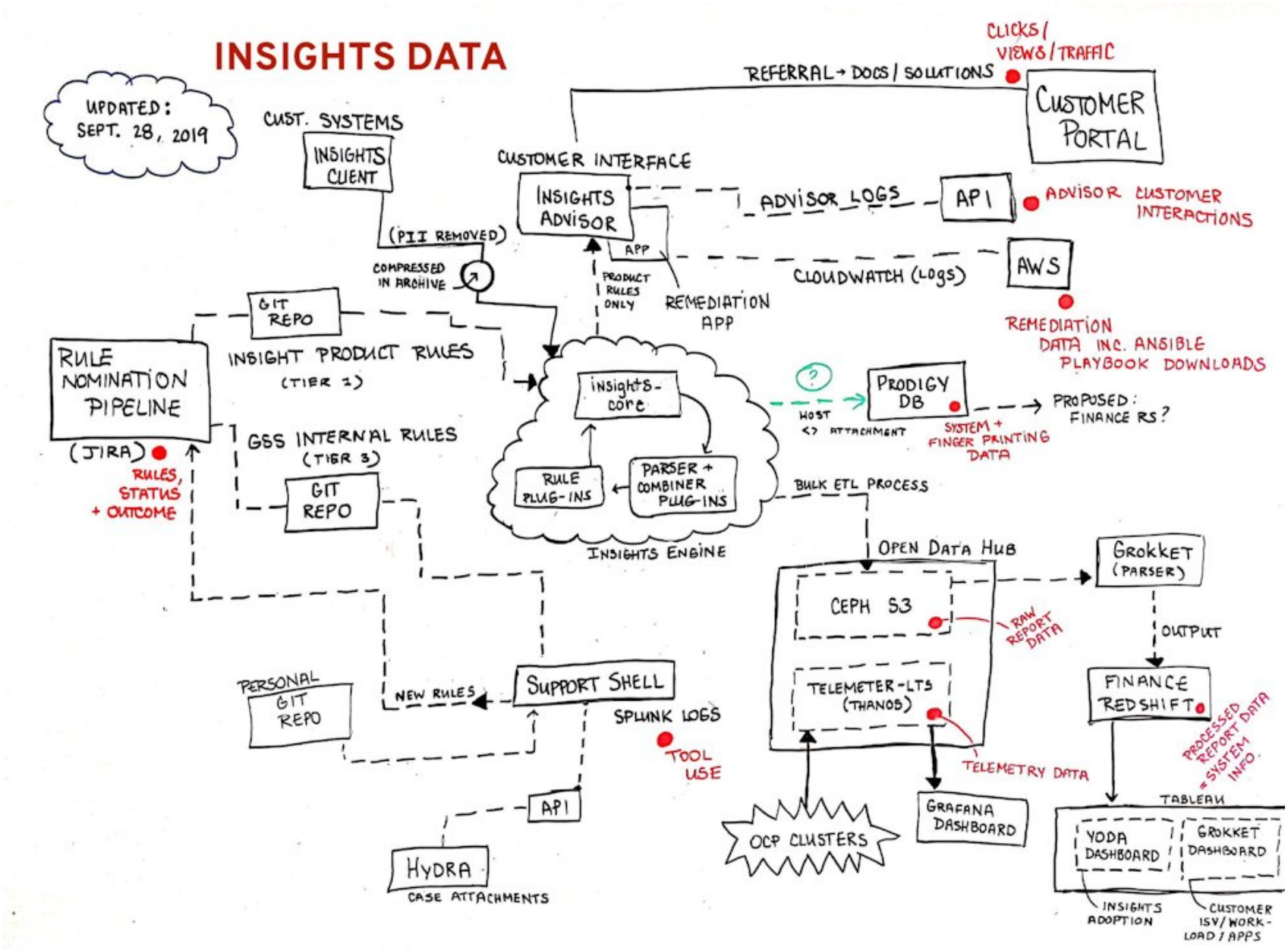


Figure 2: images/real_world.jpg

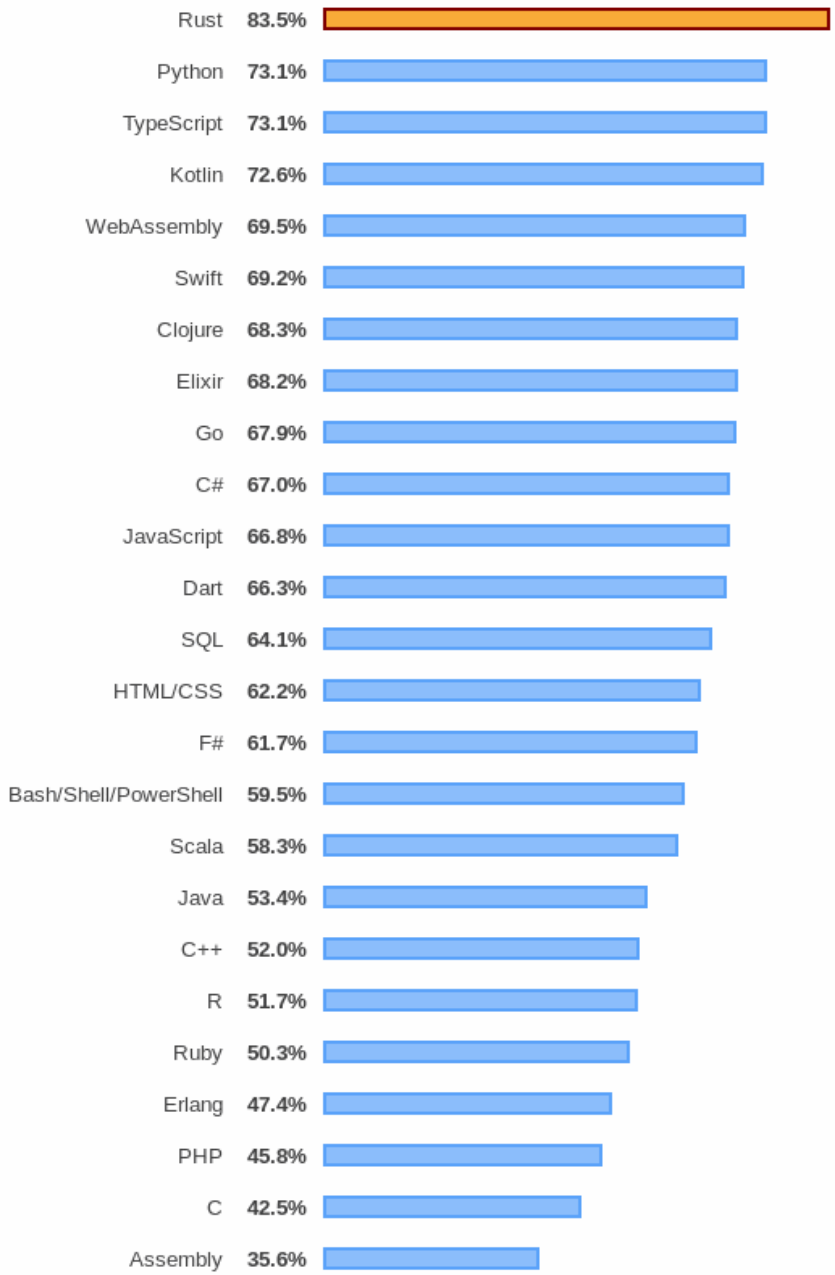
Popularita a rozšířenost Rustu

Popularita Rustu

- Několik různých metodik, jak popularitu měřit
  - Tiobe index
  - PYPL (PopularitY of Programming Languages)
  - OpenHub (pro zaregistrované repositáře)
  - StackOverflow (každoroční dotazníky)

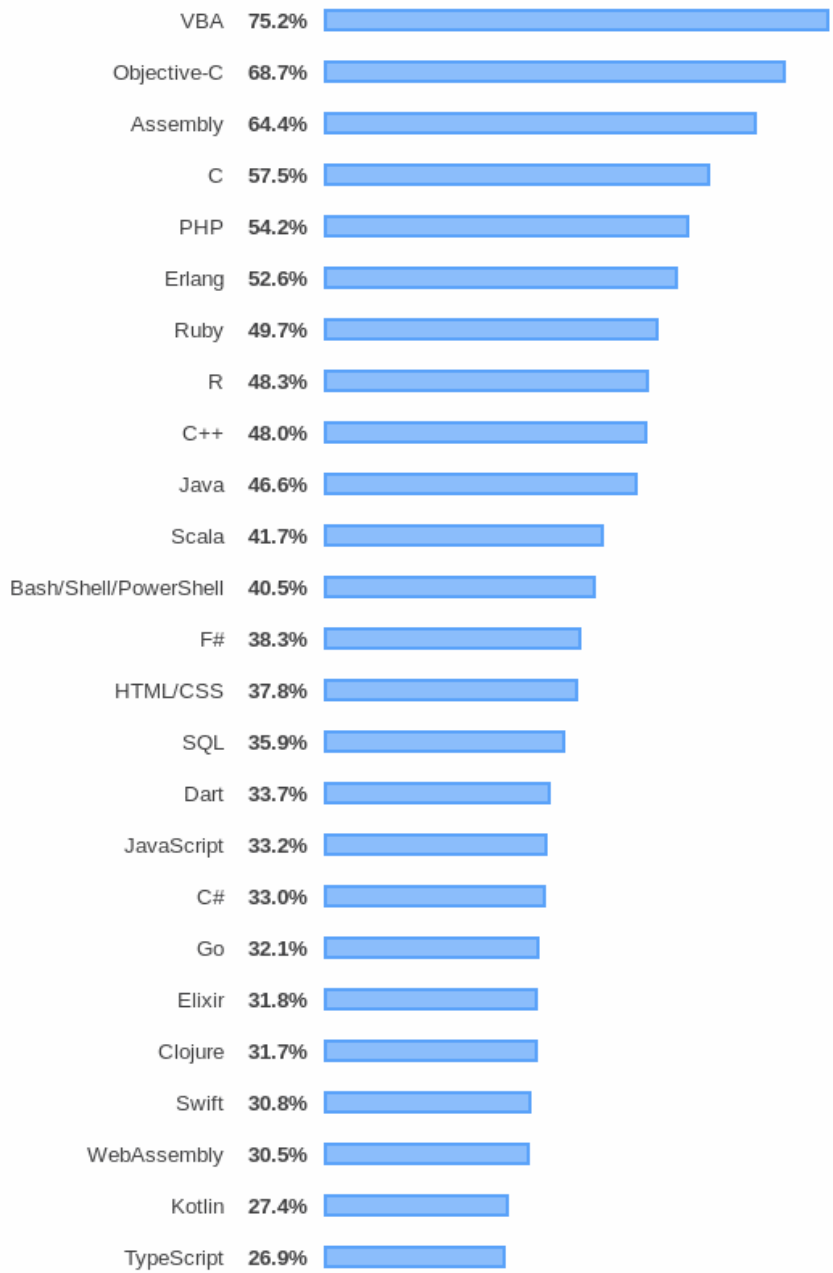
Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted



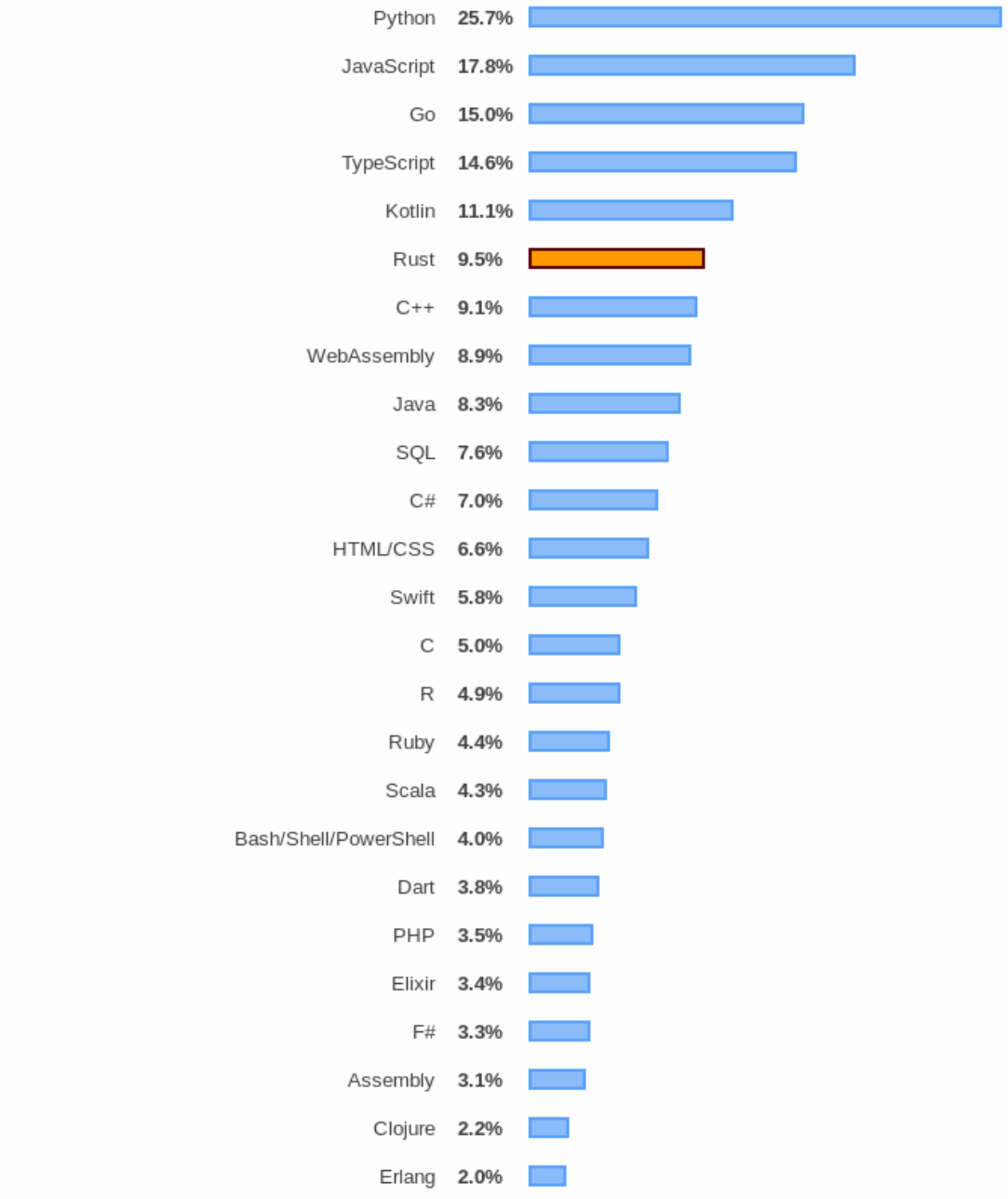
Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted



Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted



Rozšířenost Rustu

- Dostupných mnoho statistických informací
  - Můžeme jim věřit?

Worldwide, Nov 2020 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	30.8 %	+1.8 %
2		Java	16.79 %	-2.3 %
3		JavaScript	8.37 %	+0.3 %
4		C#	6.42 %	-0.9 %
5		PHP	5.92 %	-0.2 %
6		C/C++	5.78 %	-0.2 %
7		R	4.16 %	+0.4 %
8		Objective-C	3.57 %	+1.0 %
9		Swift	2.29 %	-0.2 %
10		TypeScript	1.84 %	-0.0 %
11		Matlab	1.65 %	-0.1 %
12		Kotlin	1.64 %	-0.0 %
13	↑↑	Go	1.43 %	+0.2 %
14	↓	Ruby	1.2 %	-0.2 %
15	↓	VBA	1.11 %	-0.2 %
16	↑↑	Rust	0.97 %	+0.3 %
17	↓	Scala	0.87 %	-0.2 %
18	↓	Visual Basic	0.78 %	-0.2 %
19	↑↑↑↑	Ada	0.62 %	+0.3 %
20	↑↑↑↑	Lua	0.58 %	+0.2 %
21	↑	Dart	0.57 %	+0.2 %
22	↓↓↓	Perl	0.47 %	-0.1 %
23	↓↓↓	Abap	0.45 %	-0.1 %
24	↓↓↓	Groovy	0.43 %	-0.0 %
25	↑↑	Julia	0.41 %	+0.1 %

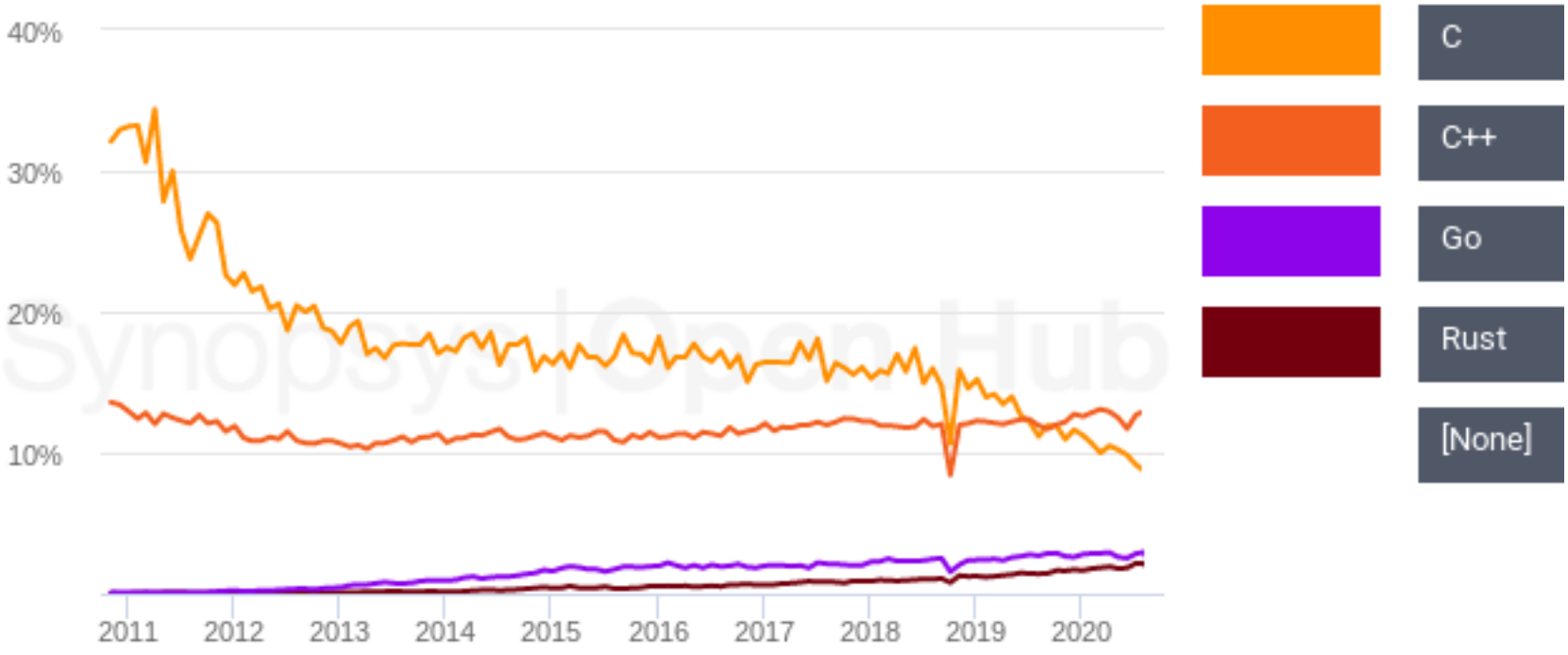


Figure 7: images/stat_openhub.png

The Rust Programming Language

Some information about Rust:

Highest Position (since 2011): #18 in Sep 2020

Lowest Position (since 2011): #211 in Dec 2012

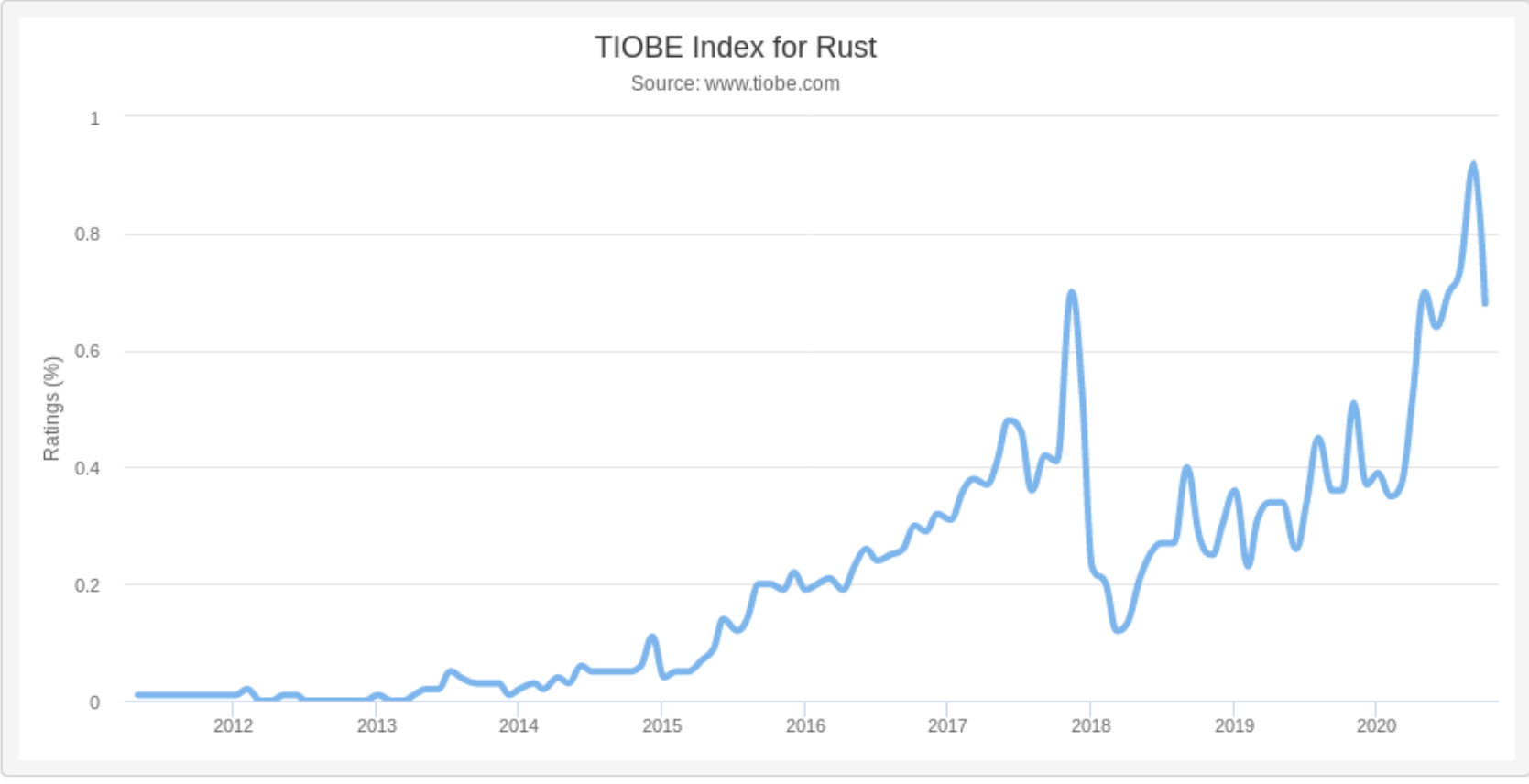
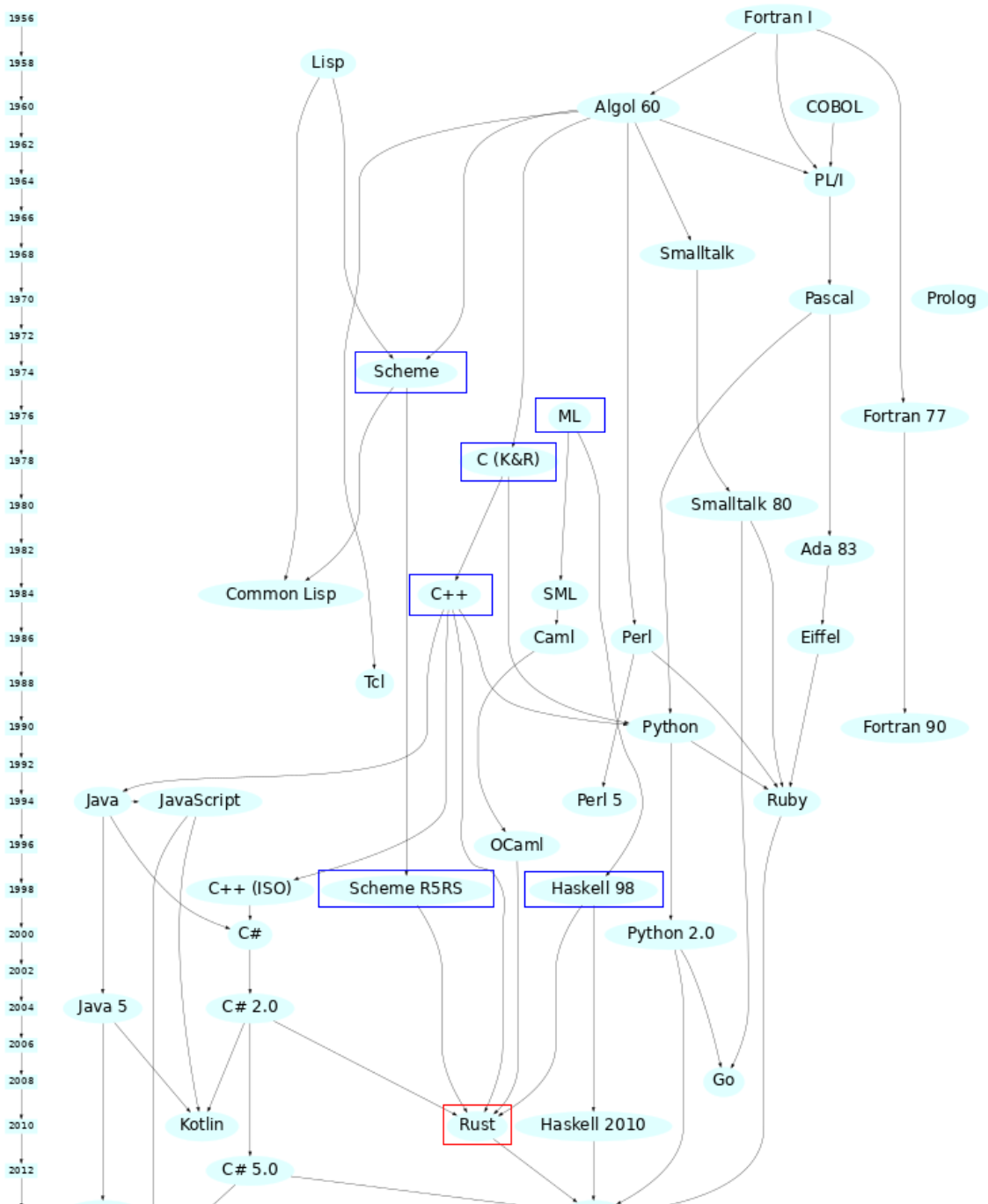


Figure 8: images/rust_tiobe_index.png

Charakteristické rysy Rustu



- Cíle
  - Bezpečné aplikace
  - Paralelní běh částí aplikace
  - Výkon srovnatelný s C a C++ (i pro nové prvky jazyka)
  - https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html
  - Překladač s rozumným chybovým hlášením
  - Nízkoúrovňový a současně vysokoúrovňový jazyk
- Poučení z chyb, které najdeme například v C/C++ nebo v Javě
  - (=, string, ptr, makrosystém)
  - NPE


ONE DOES NOT SIMPLY PROGRAM
JAVA
WITHOUT
NULL.POINTER.EXCEPTION

Charakteristické rysy Rustu

- Multiparadigmatický jazyk
  - Funcionální rysy
  - Imperativní
  - Má některé OO rysy (ne však systém tříd)
- Dostupný pro všechny „zajímavé“ systémy
  - Linux, (Free)BSD, OS X, Windows

- Používaný na velkém množství architektur procesorů
  - i686, x86-64, ARMv6/v7 (32), AArch64, MIPS, PowerPC, S390
  - RISC-V
  - Bare Cortex-M0, M0+, M1, M4(F), M7(F) bare = bez OS, jen core library
  - (dokonce i pro MSP430 - 16bit MCU!)
  - Platform Support (1)
  - Platform Support (2)
- Současná verze používá LLVM backend
  - Možnosti pro další vylepšování překladu (dovoluje i WebAssembly přes Emscripten i přímo)
  - https://www.rust-lang.org/what/wasm
- Další info
  - Object-Orientation in Rust

Charakteristické rysy Rustu

- Unicode řetězce (UTF-8)
- Odvození typů proměnných (type inference)
- Striktní typová kontrola
- OOP založené strukturách (struct) a traitech
  - × třídy, objekty a rozhraní
- Životní cyklus hodnot (zejména referencí)
  - 

- Bezpečná práce s objekty uloženými na zásobníku i haldě
  - NPE? co to znamená? :-)
- Sémantiky „copy“ a „move“
- Generické parametry funkcí, prvky struktur, ...
- Pattern matching
- Funkce jsou taktéž datovým typem
  - lambdy atd.

Rust versus C/C++

- Syntaxe Rustu jen částečně odvozena od C/C++
- Využití existujícího „ekosystému“
  - Použití již hotových C knihoven
    - Foreign Function Interface (FFI)
  - C++ knihovny
    - Stále ještě v některých případech problematické
- C Rust
  - Project Corrode

Rust versus Go

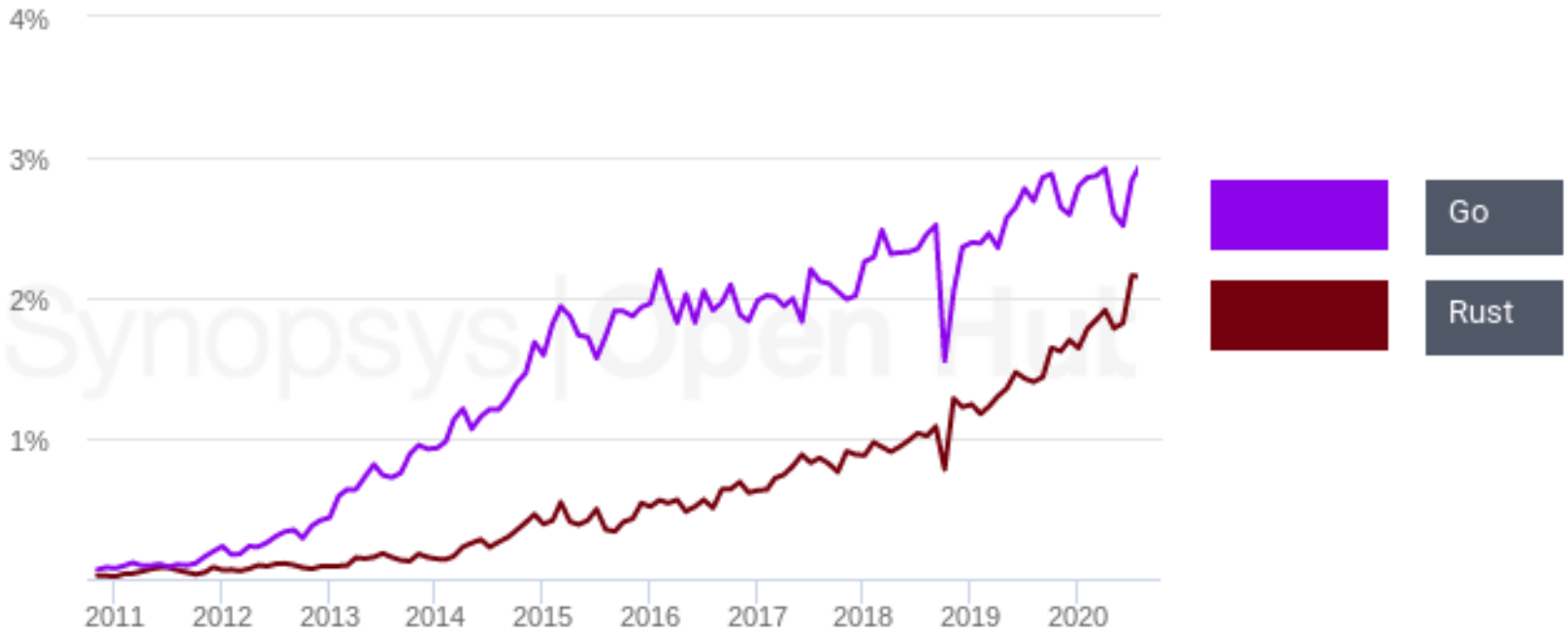


Figure 11: images/rust_go.png

- Jedná se skutečně o konkurenty?
  - Z pohledu mnoha vývojářů ano...
- Vznik zhruba ve stejný čas
  - Go veřejně představeno 2009
  - Rust 2010 (teoretické práce jsou starší)

- Společná snaha o vyřešení některých problémů céčka
  - bezpečná práce s paměti
    - nutnost
    - nikdo dnes nemůže přijít s jazykem s manuální správou paměti
  - vícevláknové aplikace
  - řetězce
  - výjimečné stavy a jejich detekce/řešení/řízení
  - problémy s textovými makry

- Překlad do nativního kódu
- Jazyky s podporovaným ekosystémem
  - Dnes prakticky nutnost
    - pypi
    - Ruby Gems
    - Rust: Cargo
    - Go: zpočátku přístup “jedno repo, jeden master”
    - dnes postupně vytvářený ekosystém okolo Go mod

Dobře, ale jedná se SKUTEČNĚ o konkurenty?

- Rust míří na vývojáře používající C++ či D
  - Pravděpodobné směřování i do oblasti výkonnějších MCU (tudíž složitějších aplikací)
- Go směřuje spíše do oblasti, kde se používá Node.js, Python, Ruby
  - Webové služby
  - Síťové nástroje

A samozřejmě oblíbené téma pro debaty...

- Formát zápisu programů
  - autoři Go: lepší je se soustředit na vlastní vývoj
  - Definován kanonický formát
    - gofmt
    - taby atd.:)

Vybrané společné rysy jazyků Go a Rust

- Podporovány společnostmi soutěžícími na poli browserů
- Výsledkem překladu jsou nativní knihovny nebo spustitelné aplikace
- Dobré (nekryptické) chybové zprávy překladačů
  - $\times$ chyba v šabloně v C++
- Syntaxe částečně připomínající “lepší” C

Porovnání Rustu a Go z hlediska vývojáře

Jazyk	Rust	Go
Přístup	moderní	konzervativní
Syntaxe	komplikovaná	jednoduchá, minimalistická
Učící křivka	menší sklon	větší sklon
Učící křivka	větší ampli.	menší amplituda
Rychlost překladu	pomalejší	rychlejší
Backend	LLVM	vlastní
Linkování	static/dynamic	přes -buildmode (//export
Rychlost kódu	rychlejší	pomalejší
Typový systém	rozsáhlý	bez generik
Neměnnost	explicitní	string, další přes rozhran
Práce s pamětí	vlastnictví	GC
Detekce souběhu	ano	jen nepřímo
Závislosti	cargo	Go moduly

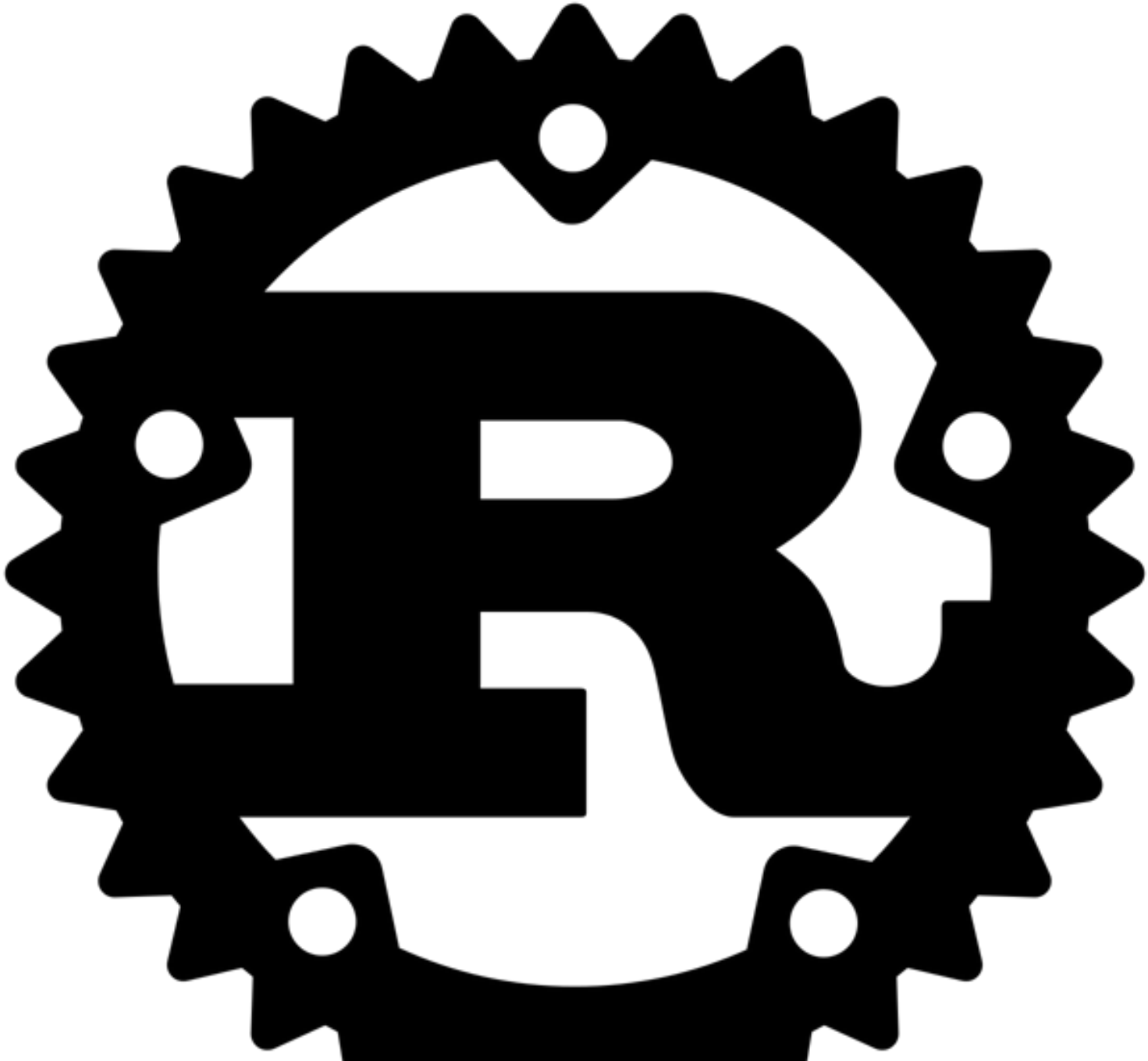
Rychlost přeložených aplikací

- Go - vlastní překladač
  - Self hosting
  - (bootstraping problem)
  - Rychlejší překlad
  - Méně optimalizovaný strojový kód
  - (projekt llgo - Go frontend pro LLVM)
- Rust - založen na LLVM
  - Pomalejší překlad
  - Optimalizace na úrovni dalších LLVM jazyků

Více info

- Rust vs Go in 2020
- Go vs Rust: Which is Better and Why?

Některé význačné rysy Rustu



Komunikace s překladačem

- Chybová hlášení musí být přesná a ideálně obsahovat nápovědu
- Ne všechny programovací jazyky toto dodržují
  - Generate the longest error message in C++
    - http://tinyurl.com/longest-error-message

Chybová hlášení překladače Rustu

```
error[E0382]: use of moved value: `c`
  --> an_example.rs:40:8
   |
39 |     funkce1(c);
   |         - value moved here
40 |     funkce2(c);
   |         ^ value used here after move
   |
   = note: move occurs because `c` has type `std::rc::Rc<Co
```

Další příklad

- Několik začátečnických chyb v programovém kódu:

```
fn foo(a int32) {
  println!(a)
}

fn main() {
  let x:i32 = 10
  foo(x);
}
```

- Výsledek běhu překladače Rustu:

```
> rustc -o main main.rs
error: expected one of `:`, `@`, or `|`, found `int32`
 --> main.rs:1:10
  |
1 | fn foo(a int32) {
  |         --^^^^^
  |         | |
  |         | expected one of `:`, `@`, or `|`
  |         help: declare the type after the parameter binding: `<identifier>: <type>`

error: expected `;`, found ``foo``
 --> main.rs:5:17
  |
5 |   let x:i32 = 10
  |                 ^ help: add `;` here
6 |   foo(x);
  |   --- unexpected token

error: format argument must be a string literal
 --> main.rs:2:12
  |
2 |   println!(a)
  |            ^
  |
help: you might be missing a string literal to format with
  |
2 |   println!("{}", a)
  |            ^^^^^

error: aborting due to 3 previous errors
```

Datové typy

- Skalární
  - celá čísla `i8`, `u128` atd.
  - s plovoucí řádovou čárkou
  - pravdivostní
  - znak
- Složené
  - n-tice
  - pole
  - struktury
  - enum
- Odvozené
  - vektory
  - řetězce
  - ...
- Další info
  - Rust: struktury, n-tice a vlastnictví objektů

Odvození typů

```
// odvození typu všech proměnných (i z)
// použití makra println!
fn main() {
    let x = 6;
    let y = 7;
    let z;
    // překladač až nyní získá informace o typu
    z = x * y;
    println!("{} * {} = {}", x, y, z);
}
```

Typ `Option`

- Nahrazuje koncept `null`
- Lze použít pattern matching
- Další info
  - Datový typ Option v programovacím jazyku Rust

```rust
fn div(x: i32, y: i32) -> Option<i32> {
    if y != 0 {
        Some(x / y)
    } else {
        None
    }
}

fn main() {
    let z1 = div(2, 1);
    println!("{:?}", z1);

    let z2 = div(2, 0);
    println!("{:?}", z2);
}
```

Typ `Result`

- Nahrazuje koncept `null` popř. speciálních chybových hodnot
- Lze použít pattern matching
- Další info
  - Reakce na chyby v programovacím jazyku Rust

```rust
fn div(x: i32, y: i32) -> Result<i32, &'static str> {
    if y != 0 {
        Ok(x / y)
    } else {
        Err("Divide by zero!")
    }
}

fn main() {
    let z1 = div(2, 1);
    println!("{:?}", z1);

    let z2 = div(2, 0);
    println!("{:?}", z2);
}
```

Typ Result a pattern matching

```
fn div(x: i32, y: i32) -> Result<i32, &'static str> {
    if y != 0 {
        Ok(x / y)
    } else {
        Err("Divide by zero!")
    }
}

fn print_div_result(result: Result<i32, &'static str>) {
    match result {
        Ok(value)  => println!("value: {}", value),
        Err(error) => println!("error: {}", error),
    }
}

fn main() {
    let z1 = div(2, 1);
    print_div_result(z1);
```

- Další info
  - Rust: funkce, lambda výrazy a rozhodovací konstrukce match

Použití typu Result při výpočtech

```
fn div(x: i32, y: i32) -> Result<i32, &'static str> {
    if y != 0 {
        Ok(x / y)
    } else {
        Err("Divide by zero!")
    }
}

fn print_div_result(result: Result<i32, &'static str>) {
    match result {
        Ok(value)  => println!("value: {}", value),
        Err(error) => println!("error: {}", error),
    }
}

--

fn inc(x: i32) -> i32 {
    x + 1
```

Anonymní funkce jsou hodnotami

```
fn main() {
    let is_odd = |x: i32| x & 1 == 1;
    //let is_even = |x: i32| !is_odd(x);
    let square = |x: i32| x*x;
    for x in 0..10 {
        println!("{}*{}={}, {} is {} number",
                 x, x, square(x), x, if is_odd(x) {"odd"} e
    }
}
```

- Další info
  - Rust: funkce, lambda výrazy a rozhodovací konstrukce match

(Anonymní) funkce jsou hodnotami

```
fn main() {
    let is_odd = |x: i32| x & 1 == 1;
    //let is_even = |x: i32| !is_odd(x);
    let square = |x: i32| x*x;
    for x in 0..10 {
        println!("{}*{}={}, {} is {} number",
                 x, x, square(x), x, if is_odd(x) {"odd"} e
    }
}
```

Zajímavé prvky jazyka

- Neměnitelné hodnoty
  - Výchozí modifikátor
  - Lze změnit pomocí `mut`
- Rozsah (`range`)
- Řídicí struktury
  - Vrací hodnotu
- Anonymní funkce
- Funkce vyššího řádu
  - `map`
  - `filter`
  - `take`
  - `take_while`
  - `fold`
  - Nekonečné sekvence
- Pattern matching
- Makra
- Unsafe bloky

Tabulka faktoriálů

- Použití anonymních funkcí a funkcí vyššího řádu

```
fn main() {
    for n in 1..10 {
        let fact = (1..n + 1).fold(1, |prod, x| prod * x);
        println!("{}! = {}", n, fact);
    }
}
```

Nekonečné sekvence

```
fn main() {
    let iter1 = 1..;
    let iter2 = iter1.filter(|x| x % 2 == 0);
    let iter3 = iter2.take(10);
    let suma  = iter3.fold(0, |sum, x| sum + x);
    println!("sum = {}", suma);
}
```

Pattern matching

- Je nutné uvést všechny možné větve
  - hlídáno překladačem

```
// matching (nejjednodušší varianta)
fn main() {
    let x: i32 = 1;
    match x {
        0 => println!("zero"),
        1 => println!("one"),
        2 => println!("two"),
        3 => println!("three"),
        _ => println!("something else"),
    }
}
```

Pattern matching: složitější konstrukce

```
// matching, složitější ukázka
fn classify(x:i32) -> &'static str {
    match x {
        0          => "zero",
        1 | 2      => "one or two",
        3 | 4 | 5 => "from three to five",
        10 ... 20 => "from ten to twenty",
        _          => "something else",
    }
}

fn main() {
    for x in 0..10 {
        println!("{}:{}", x, classify(x))
    }
}
```

Přístup k OO v Rustu

- Vlastnictví objektů
  - Reference
  - Sémantika „move“
  - Sémantika „copy“
- Traity
  - Kombinace trait+struktura+metody
- Konstruktory a destruktory
  - Trait „Drop“
  - Přetěžování operátorů
    - Přetížení = implementace traitu
- Generické funkce
- Další info
  - Programovací jazyk Rust: metody a traity
  - Generické typy v programovacím jazyku Rust

Správa paměti

- Zásobník versus halda
- Box
- Rc
- Arc
- Pole a vektory
  - slice
- Další info
  - Správa paměti v programovacím jazyku Rust s počítáním referencí
  - Práce s poli v programovacím jazyku Rust
  - Práce s vektory v programovacím jazyku Rust

Box

- Alokace objektu na haldě
- „Obaluje“ vlastní objekt (číslo, strukturu, pole)
- Trait Deref - snadný přístup k obalenému objektu
- Hlídání životnosti objektu i ukazatele
- Nemůže být `NULL`/`nil`

```
fn main() {
    let x = Box::new(42);
    println!("{}", x);
}
```

```
let c = Box::new(Complex::new(1.0, 2.0));

// deref
fn print_complex(c: Box<Complex>) {
    println!("Complex number: {:}+{:}i", c.real, c.imag);
}
```

Rc

- Počítání referencí
- `Rc::clone()`
- Pokud čítač dosáhne nuly, je Rc i objekt jím vlastněný zrušen
- Automatická dereference (`Deref` trait)

```rust
fn main() {
    println!("main begin");
    let c = Rc::new(Complex::new(0.0, 0.0));
    c.print();
    {
        println!("inner block begin");
        let c2 = Rc::new(Complex::new(0.0, 0.0));
        c2.print();
        {
            println!("inmost block begin");
            let c3 = Rc::new(Complex::new(0.0, 0.0));
            c3.print();
            println!("inmost block end");
        }
        println!("inner block end");
    }
    println!("main end");
}

// jeden sdílený objekt referencovaný třikrát
```

Arc

- Taktéž počítání referencí, ovšem atomické
  - obecně pomalejší
  - možnost přístupu z více vláken
- `Arc::clone()`
- `Deref` trait

```rust
fn start_threads() {
    let c = Arc::new(Complex::new(1.0, 1.0));
    for id in 0..10 {
        let owner = ComplexNumberOwner { id: id, value: c.c
        // move protože objekt může přežít toto vlákno
        thread::spawn(move || {
                owner.print();
                delay(400);
        });
    }
}
```

Pole

- V Rustu považováno za primitivní datový typ
- Dva typy konstruktorů
- Zjištění délky pole za běhu programu
- Přístup k prvkům přes indexy
- Indexy začínají od nuly (C-like $\times$ Fortran, Lua)
- „Slice polí“ (efektivní operace)

```rust
fn main() {
    let array = [10, 20, 30, 40];
    // délka pole
    println!("array has {} items", array.len());
    // range + délka pole
    for i in 0..array.len() {
        println!("item #{} = {}", i + 1, array[i]);
    }
    // for-each
    for i in array.iter() {
        println!("{}", i);
    }
}
```

```rust
fn main() {
    let array = [1; 10];
    // délka pole
    println!("array has {} items", array.len());
    // range + délka pole
    for i in 0..array.len() {
        println!("item #{} = {}", i + 1, array[i]);
    }
    // for-each
    for i in array.iter() {
        println!("{}", i);
    }
}
```

Vektory

```rust
fn main() {
    let vector = vec![1, 2, 3, 4, 5];
    // délka vektoru
    println!("vector has {} items", vector.len());
    // range + délka pole
    for i in 0..vector.len() {
        println!("item #{} = {}", i + 1, vector[i]);
    }
    // for-each
    for item in vector.iter() {
        println!("{}", item);
    }
    // také funguje
    for item in &vector {
        println!("{}", item);
    }
}
```

„Slice“ polí a vektorů

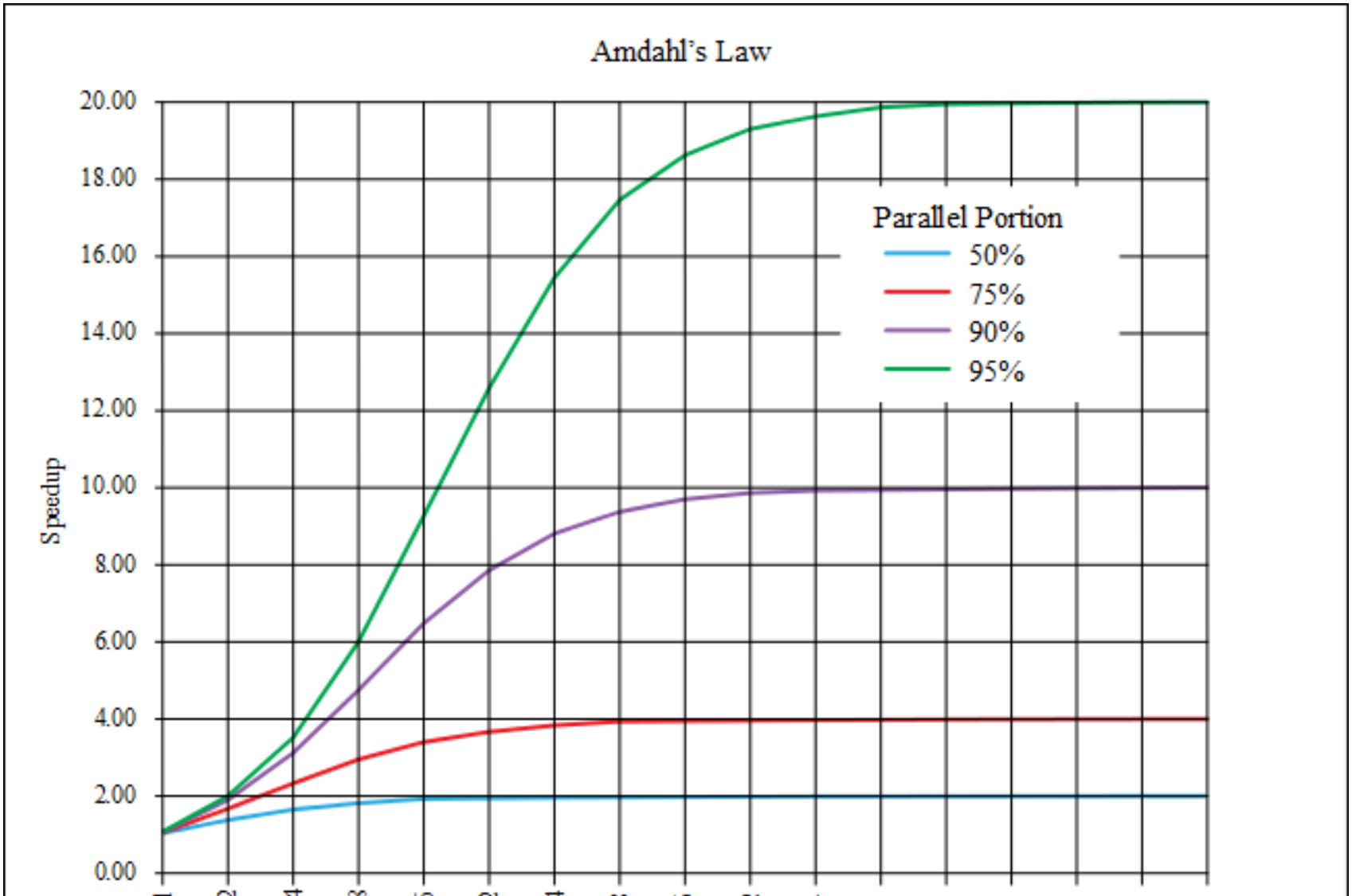
```
fn main() {
    let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let array2 = &array[2..6];
    for i in array2.iter() {
        println!("{}", i);
    }
}
```

```rust
fn main() {
    let array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let array2 = &array[5..];
    for i in array2.iter() {
        println!("{}", i);
    }
}
```

```rust
fn main() {
    let vector = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    println!("vector has {} items", vector.len());
    let slice = &vector[3..7];
    println!("slice has {} items", slice.len());
}
```

Vlákna

- Konformní práce s vlákny velmi důležitá, zejména v současnosti



- Více info
  - Programovací jazyk Rust: vlákna a sdílení objektů mezi nimi

```rust
use std::thread;

fn main() {
    println!("Starting");
    for i in 1..10 {
        thread::spawn(move || {
            println!("Hello from a thread #{}", i);
        });
    }
    println!("Stopping");
}
```

Testování

- Problematika testování stále složitějších aplikací a systémů
- CI/CD
- Základní problém
  - čím později je chyba odhalena, tím dražší je její oprava
  - z jiného oboru:
    - triviální úprava ventilu při návrhu motoru
    - vs svolávání aut do servisu
    - vs případné žaloby v případě, že chyba způsobí nehody
- Další časté problémy dnešních aplikací
  - velký vývojářský tým
  - používá se větší množství jazyků (jak se domluvit?)
  - zákazník a jeho role při vývoji
  - někdy nejasné role (vývojář či tester?)

“Pyramida” s různými typy testů

- Business část
  - Beta testy
  - Alfa testy
  - Akceptační testy
- Technologická část
  - UI testy
  - API testy
  - Integrační testy
  - Testy komponent
  - Unit testy
- Další typy testů
  - Benchmarky

Jednotkové testy v Rustu

- Spouštěné přes `cargo test`

```
#[test]
fn ok_test() {
}

#[test]
fn failure() {
    assert!(false);
}
```

Další testy

- `mockiato` - mocking
- `mockito` - HTTP mocking
- `rust-fuzz/afl.rs` - fuzzer postavený nad AFL

Správce balíčků (Cargo)

Základní funkce

- Vytvoření kostry nového projektu
- Nový projekt obsahuje i adresáře a soubory umožňující podporu SCM
  - Git
  - Mercurial
- Automatická kontrola, které soubory je zapotřebí přeložit
- Automatické stažení všech knihoven a jejich závislostí
- Spuštění projektu s možností předání parametrů příkazového řádku.
- Spuštění jednotkových testů
- Spuštění benchmarků
- Vyhledání knihovny v centrálním registru zaregistrovaných knihoven
- Publikování vlastního balíčku v centrálním registru (crates.io)
- Instalace aplikace
- Další info
  - Cargo: správce projektů a balíčků pro programovací jazyk Rust

Statistika (tento týden)

- https://crates.io/
- 4,444,104,010 downloads
- 49,047 Crates in stock

Cargo.toml

- De facto popis projektu/modulu
- https://doc.rust-lang.org/cargo/reference/manifest.html

```
[package]
name = "projectXYZ"
version = "0.1.0"
authors = ["Pepa z depa <pepa@openalt.cz>"]

[dependencies]
rand = "0.3.14"
```

Použití nástroje Cargo

- První překlad a sestavení

```
$ cargo build
    Compiling project1 v0.1.0 (file:///home/tester/temp/pro
    Finished debug [unoptimized + debuginfo] target(s) in (
```

- Další překlad a sestavení

```
$ cargo build
    Finished debug [unoptimized + debuginfo] target(s) in (
```

- Spuštění

```
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in (
     Running `target/debug/project1`
Hello, world!
```

Použití nástroje Cargo

- Jednotkové testy

```
$ cargo test
   Compiling project1 v0.1.0 (file:///home/tester/temp/proj
    Finished debug [unoptimized + debuginfo] target(s) in (
     Running target/debug/project1-b888664ab405e319

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Použití nástroje Cargo

- Instalace modulu

```
$ cargo install
   Compiling libc v0.2.17
   Compiling rand v0.3.14
   Compiling projectXYZ v0.1.0 (file:///home/tester/temp/pr
    Finished release [optimized] target(s) in 5.88 secs
  Installing /home/tester/.cargo/bin/projectXYZ
warning: be sure to add `/home/tester/.cargo/bin` to your P
```

Vybrané balíčky

- Jak vybírat?
  - Awesome Rust
  - https://awesome-rust.com/

Databáze

- `mysql-proxy-rs`
- `rust-postgres`
- `redis-rs`
- `mongo-rust-driver`

ORM

- `diesel`

Data processing

- `ndarray`
- Více info
  - Rust: knihovna ndarray pro práci s n-rozměrnými poli
  - Programovací jazyk Rust: knihovna ndarray pro práci s n-rozměrnými poli (2)
  - Programovací jazyk Rust: knihovna ndarray pro práci s n-rozměrnými poli (dokončení)

Web

- `cargo-web`
- `hyper` (HTTP client)
- `Gotham` (web framework)
- `tiny-http`
- `Iron` (založeno na konceptu middleware)
  - https://github.com/iron/iron

API a message brokery

- `futures-jsonrpc`
- `nanomsg-rs`
- `stomp-rs`
- `rust-zmq`

Nasazení aplikací

- Kontejnerizace
- Statické linkování
- Více info
  - Linkage
  - Packaging a Rust web service using Docker

Web Assembly

Rozhraní s Pythonem

- Name mangling lze zakázat
- `ctypes` nebo `CFFI`
- Více info
  - Programovací jazyk Rust: rozhraní mezi Rustem a Pythonem

```rust
#[no_mangle]
pub extern fn add_integers(x: i32, y: i32) -> i32 {
    x + y
}
```

```python
#!/usr/bin/env python3
import ctypes

testlib1 = ctypes.CDLL("target/debug/libtest1.so")

result = testlib1.add_integers(1, 2)
print("1 + 2 = {}".format(result))

result = testlib1.add_integers(1.5, 2)
print("1.5 + 2 = {}".format(result))
```

Dokumentace

- Generovaná ze zdrojových kódů
- Proč?
  - Source of truth
- Markdown

Odkazy

- For Complex Applications, Rust is as Productive as Kotlin https://ferrous-systems.com/blog/rust-as-productive-as-kotlin/