Středoškolská odborná činnost

Obor SOČ: 18. Informatika

Programování v jazyce Rust

Vypracoval: Marek Smolík

Škola: Střední škola spojů a informatiky Tábor, Bydlinského 2474

Kraj: Jihočeský

Konzultant: Ing. Dana Almášiová

Čestné prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v seznamu vloženém v práci SOČ. Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné. Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) v platném znění.

. . . .

datum podpis

Obsah

Anotace **4**

Klíčová slova **5**

Úvod **6**

Seznámení s Rustem **7**

Přehled a využití 7

Přednosti jazyka **8**

OOP 8

Kompozice místo dědičnosti 9

Generika místo polymorfismu 9

Modulový systém 10

Model vlastnictví 15

Cross-platform export aplikací **23**

Windows 23

GNU/Linux 24

WebAssembly 25

Hra života s GUI **29**

Vymezení cílů 29

Výběr frameworku 29

Struktura / rozvržení 31

Logická struktura 31

Závěr **36**

Seznam příloh **37**

Anotace

Tato práce si bere za cíl představit tento jazyk, jeho přednosti, výhody a nevýhody. Její součástí je program v jazyce vytvořený a přílohy obsahují příručku — zevrubného a rozsáhlého průvodce programováním v jazyce Rust.

Klíčová slova

Rust, nižší programovací jazyky, WebAssembly, WASM, Cross-platform, Hra života

Úvod

Rust je jedním z nejrychleji se rozšiřujících programovacích jazyků. Jeho kompilér je otevřený software vyvíjený firmou Mozilla. Dle statistik webového portálu Stack Overflow se jedná o „Programátory nejoblíbenější jazyk“ dle všech hlasování v rocích 2015-2021 konsekutivně.[?] Tomu je tak i přes jeho relativní mladost. Má všestranně se rozšiřující ekosystém — lze v něm programovat servery, vestavěné systémy, GUI, weby, ad. včetně systémového programování umožněnému díky jeho abstrakci s nulovou cenou.

Rust mě zaujal zejména díky velice dobrému hodnocení co se týče nižších programovacích jazyků, proto jsem si ho i zvolil jako první jazyk této kategorie, který jsem se naučil.

Seznámení s Rustem

Přehled a využití

Rust je nízkoúrovňový kompilační jazyk vyvíjený společností Mozilla. Jeho *de facto* cílem je být „modernější“ a „bezpečnější“ C++. S tímto jazykem má mnohé společné — má statickou typovou kontrolu, oba jsou objektově orientované, dovolují správu paměti na nízké úrovni, atd. Dokonce některé Rustové programy jsou pouhými wrappery pro stávající C++ aplikace. Čím se však snaží tento letitý standard předčít je svojí paměťovou a vláknovou bezpečností. Tu zaručuje jeho kompilér a velice efektivně je tak zabráněno runtime chybám (takovým, vytvořeným během doby, při níž je samotný program spuštěn).

Tato vlastnost je mezi programátory vysoce ceněna. Také dost stávajících projektů/standardů začalo integrovat Rust do svých produktů. Některé z těchto společností jsou DropBox,[?] Cloudflare,[?] npm,[?] Discord[?] a četné množství dalších. Nejvýznamnější entitou zakomponovávající Rust je Amazon,[?] který v Rustu napsal celý jejich Firecracker VMM a přispěl do ekosystému.

Přednosti jazyka

OOP

Podobně jako ve spoustě dalších jazyků nabízí Rust model pro objektově orientované programování. To je zprostředkováno pomocí struktur, nikoliv tříd. Výkonnostně jsou tyto metody velice podobné, liší se prakticky jen jejich programovacími rozhraními. Základním typem OOP je `enum`. Je to výčtový typ — tvoří ho libovolné množství pojmenovaných polí.

```
enum Jazyk {
    Python,
    Rust
    WebAssembly,
}
```

Následně lze instance těchto enumerací porovnávat pomocí vzorů řízení toku.

```
fn main() {
    let muj_jazyk = Jazyk::Rust;

    if let Jazyk::Rust =  muj_jazyk {
        // ...
    }

    match muj_jazyk {
        Jazyk::Rust => { /* ... */ },
        Jazyk::Python => { /* ... */ },
        Jazyk::WebAssembly => { /* ... */ },
    }

}
```

Avšak hlavní datastrukturou OOP v Rustu je struktura — `struct`. Ta, jak již bylo řečeno, je ekvivalentem tříd v jiných jazycích.

```
struct Clovek {
    jmeno: String,
    prijmeni: String,
    vek: u8,
}
```

Nadále lze použít `impl` pro tvorbu metod nad danou strukturou.

```
impl Clovek {
    fn rekni_ahoj(&self) {
        println!(
            "Ahoj, jsem {} {} a je mi {} let.",
            self.jmeno,
            self.prijmeni,
            self.vek
        );
    }
}
```

Kompozice místo dědičnosti

Rust nemá třídy, ale struktury. Tyto struktury používají místo dědičnosti kompozici. Při tomto paradigmatu je využíváno shlukování sdílených vlastností více struktur. Tyto množiny se nazývají `traits`. Identifukují sadu metod, které by měli `structy`, které ji implementují, mít.

```
trait Starnuti {
    fn zvysit_vek(&mut self);
}

struct Clovek {
    jmeno: String,
    prijmeni: String,
    vek: u8,
}

struct Zvire {
    jmeno: String,
    vek: String,
}

impl Starnuti for Clovek {
    fn zvysit_vek(&mut self) {
        self.vek += 1;
    }
}
impl Starnuti for Zvire {
    // ...
}
```

Generika místo polymorfismu

Polymorfismus je způsob použití stejného rozhraní pro různé datové struktury. Toho lze v Rustu dosáhnout pomocí využití generických funkcionalit.

```
fn vypsat<T: Display>(arg: T) {
    println!("{}", T);
}
```

Tento syntax značí, že funkce `vypsat()` využívá 1 generickou datovou strukturu, která musí implementovat `trait Display` (tato `trait` slouží pro vypsání hodnoty dané proměnné). Následně je poznamenáno, že argument `arg` je tohoto typu.

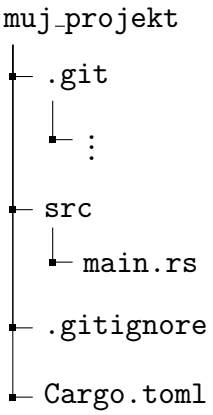
Specifikování `traits`, které daná datová struktura implementuje je nutné pro splnění statické kontroly typů.

Kompilérem je na základě analýzy kódu zjištěno, přesně kterými typy je ve skutečnosti generická funkce využívána (na základě statické typové kontroly) a následně funkci pro všechny tyto typy vygeneruje. Tomuto procesu se říká **monomorfizace**.

Modulový systém

Lokální soubory nejsou strukturovány manuálně. O tuto činnost se stará správce projektu `cargo`. Jde o systém podobný npm, což je správce projektů běžících pod systémem Node.js.

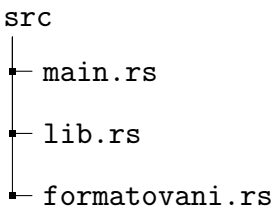
Struktura cargem vygenerovaná po inicializaci projektu je následující



Ve výchozím stavu, najde-li cargo při stavění projektu ve složce `src` soubor `main.rs`, považuje ho za root (kořen) projektu. Zároveň na jeho základě vytvoří automaticky binární tzv. crate (ekvivalent package, tj. „balíčky“ u jiných systémů). Pro rozšiřování projektu lze přidat do složky `src` soubor `lib.rs`. Ten

slouží jako knihovna — obsahuje kód pro využití ve více souborech, popřípadě odkazuje na logické části adresáře. Je-li tento soubor cargem nalezen, je automaticky vytvořená druhá crate — library crate. Pravidlem je, že každý projekt musí být alespoň 1 crate — buďto library crate, nebo binary crate. Zároveň binárních může být libovolné množství, ale library může být maximálně jedna.

Pokud je žádoucí více binárních crate (a tedy více vyprodukovaných binárních souborů), ve složce `src` potřeba vytvořit složku `bin`. Následně každý `.rs` soubor v této složce umístěný bude cargem zvlášť zpracován jako binární crate a každý z nich při stavění vyprodukuje binární soubor. Toto však není příliš časté. Většinou je ve složce `src` tvořeno více pomocných souborů, jejichž obsah je buďto implicitně, nebo explicitně volán přes kořenový soubor (`main.rs`). V praxi postup takové tvorby může být následovný.



Obsah souborů podléhající této souborové struktuře by mohl být takovýto.

```
// formatovani.rs

pub fn nahoru(s: String) -> String {
    return s.to_uppercase();
}

// lib.rs
pub mod formatovani;

// main.rs

mod formatovani;

fn main() {
    let a = "ahoj".to_string();

    println!("{}", formatovani::nahoru(s));
}
```

První poznatkem zde je, že funkce v odděleném souboru je označena klíčo-

vým slovem `pub` to je potřebné z toho důvodu, že by jinak nebyla dosažitelná z ostatních modulů. V Rustu jsou implicitně všechny datové struktury privátní a pro využití v jiné lokaci je nutno viditelnost explicitně označit.

Následně je `formatovani.rs` nutno označit jako validní modul v `lib.rs`. Poté ho již lze využít z `main.rs`.

std modul a přidružené

Holý Rust má 5 základních crates, které jsou součástí každé instalace a lze je využít i v prázdném projektu

- **alloc** - obsahuje většinu funkcionality pro alokaci a správu paměti
- **core** - obsahuje základní funkcionality, jako primitivní typy
- **proc_macro** - podpora pro definování vlastních (procedurálních) maker
- **std** - poskytuje abstrakci základní funkcionality pro cross-platform
- **test** - podpora testů a výkonnostního testování

Některé z nich, jako alloc a core, jsou reexportovány v std. Navíc část std je automaticky přidána do každého programu, jelikož by nebylo praktické importovat každý jeden komponent manuálně. Přesto některé prvky, které nejsou zbytné, jako `std::collections::HashMap` , je nutno importovat.

Externí crates

Stránka crates.io je oficiálním internetovým repozitářem pro komunitní crates. K 24.1.2022 jich hostuje stránka na 75251 a nejstahovanější crate je rand. V pravé liště stránky vybrané crate lze vyčíst, že pro import lze použít syntax typu `crate = "verze"`. V době citace je ekvivalentem pro tuto crate `rand = "0.8.4"`. Je-li ji chtěno v projektu použít, vloží se tento text do souboru `Cargo.toml` do sekce `[dependencies]`.

```
[dependencies]
rand = "0.8.4"
```

Alternativně lze importovat crates z git repozitářů. Např.: pro regex by mohl vypadat takto:

```
regex = { git = "https://github.com/rust-lang/regex", branch = "next" }
```

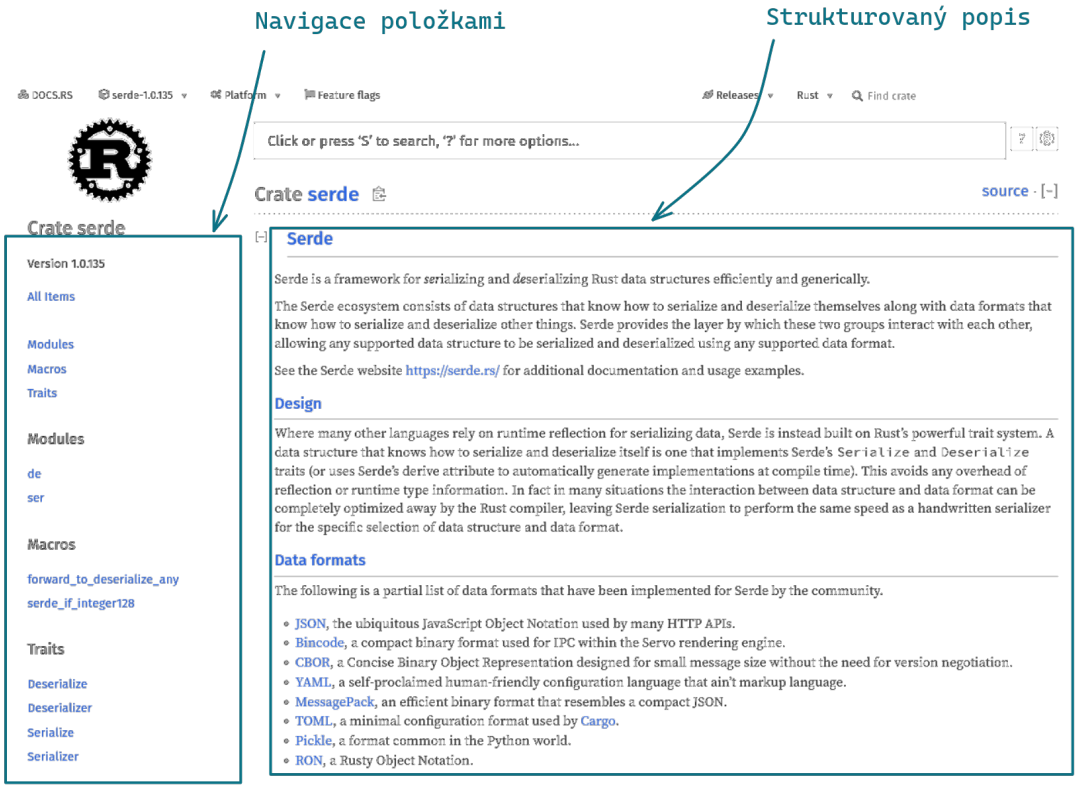
Poté lze s těmito moduly pracovat. Při prvním stavění projektu po přidání se tyto crates (a všechny ostatní crates, na kterých projekt závisí) nejprve zkompilují (při dalších stavěních již toto není nutné).

Obrázek 1: Ukázka a popis crate serde na crates.io

Avšak ještě je potřeba vědět, jak s danou crate zacházet. Na její stránce na crates.io lze přečíst její popis, který většinou poskytuje minimalistickou ukázku kódu znázorňující zacházení s ní. Avšak (pokud zde vůbec je) obsahuje minimální informace. Popřípadě, je-li crate komplexnější, může obsahovat jakéhosi průvodce, označeného v pravé liště odrážkou `Homepage`.

Dokumentace

Pro hlubší obeznámení s jednotlivými crates je interakce s přímou dokumentací nevyhnutelná. Není však nutné číst přímo zdrojový kód. Ekosystém poskytuje stránku docs.rs, která autogeneruje nízkoúrovňovou dokumentaci přímo z repozitářů. Lze zde najít i dokumentaci pro crate `std`. Například pro crate `serde` vypadá takto

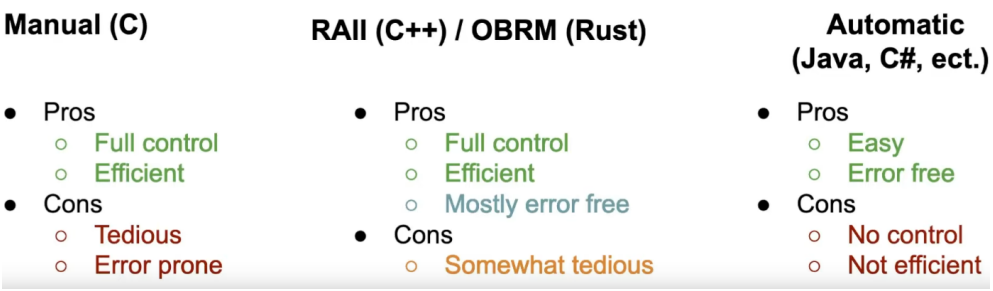


Obrázek 2: Ukázka a popis crate serde na docs.rs

Model vlastnictví

Pravidla pro manipulaci paměti

Některé programovací jazyky jako Java, Haskell nebo Go mají garbage collector (odvoz odpadu). Ten uvolňuje pamět zabranou proměnnými právě, když vyjdou z rámce, v kterém jsou definovány a proměnné jsou takto zinvalidněny. Jiné jazyky nic takového neposkytují a paměť musí být spravována přímo programátorem. Takovým příkladem může být C. Rust se liší od těchto dvou skupin tím, že vnutí přesná pravidla a postupy dané modelem vlastnictví, díky čemuž kompilér ví přesně, kdy proměnná má zaniknout, nebo skončí-li ji životnost (tzv. lifetime) a na základě toho použije adekvátní instrukce LLVM nebo jazyka symbolických adres (tento systém se označuje jako OBRM - ownership based resource management).



Obrázek 3: Různé strategie správy paměti[?]

Pro zachování bezpečnosti ohledně proměnných má Rust velice sofistikovaný a specifický model vlastnictví. Je tomu tak, jelikož jeho filozofií je napsat o něco více kódu, což se však záhy vyplatí redukcí runtime chyb.

Tento model má několik pravidel, které musí být vždy dodržena.

- v jakýkoliv daný moment smí mít proměnná buďto 1 mutable referenci, nebo libovolný počet immutable (neměnitelných) referencí
- reference musí vždy být validní (tj. nesmí odkazovat na neexistující proměnné)

(Ve skutečnosti se tato pravidla avšak nevztahují na tzv. chytré ukazatele.) Dodržení těchto pravidel je při kompilaci zaručeno pomocí borrow checkeru (kontroléru půjčení).

```
fn main() {
    let a = "a".to_string();

    let b = a;

    println!("{}", a);
}
```

Tato ukázka kódu zámerně nezkompiluje. Borrow checker to nedovolí a vrátí následující chybu

```
error[E0382]: borrow of moved value: `a`
    --> src/main.rs:6:20
     |
2 |     let a = "a".to_string();
     |         - move occurs because `a` has type `String`, which does not
     ↪  implement the `Copy` trait
3 |
4 |     let b = a;
     |             - value moved here
5 |
6 |     println!("{}", a);
     |                    ^ value borrowed here after move
     |
     = note: this error originates in the macro `$crate::format_args_nl` (in
     ↪  Nightly builds, run with -Z macro-backtrace for more info)
```

Ukazuje se, že ačkoliv kompilér Rustu (`rustc`) je velice striktní. Při chybě vrací přesná hlášení o tom, co je špatně.

Tento exaktní problém vyvstává z toho, že vlastnictví hodnoty původně vlastněné proměnnou `a` bylo přesunuto proměnné `b`. V tomto bodě je proměnná `a` zinvalidněna a nelze ji nadále použít (pokud jí není přiřazena nová hodnota, kterou by byla schopna vlastnit). Tomuto lze předejít použitím referencí.

```
fn main() {
    let a = "a".to_string();

    let b = &a;

    println!("{}", a);
}
```

Tento případ se již zkompiluje, jelikož proměnná `b` pouze odkazuje na hodnotu proměnné `a`. Toto má jeden limit — hodnotu proměnné `b` takto nelze měnit. Pro tuto funkcionalitu by bylo nutno získat měnitelnou referenci, například takto

```
fn main() {
    let mut a = "ahoj".to_string();

    let b = &mut a;
    println!("{}", b.to_uppercase()); // AHOJ

    // `a` je stále validní
    println!("{}", a); // ahoj
}
```

Každá proměnná v Rustu je ve výchozím stavu neměnitelná, proto ji je nutno definovat jako `mut` (mutable). Následně lze vytvořit měnitelnou referenci.

S referencemi však může někdy dojít k problémům, je-li referováno na hodnotu, která již neexistuje (byla uvolněná z paměti)

```
fn visici_ref() -> &String {
    let s = String::from("Jovan");

    return &s;
}

fn main() {
    let reference_na_nic = visici_ref();
}
```

```
?
```

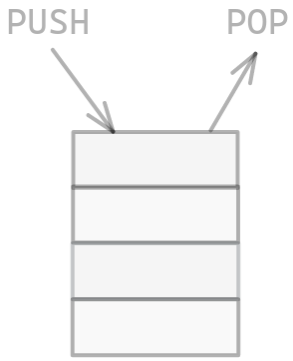
Tento příklad se nezkompiluje. Při spuštění funkce je vytvořena proměnná typu `String`. Posléze se tato funkce snaží vrátit referenci na hodnotu této proměnné, avšak při ukončení funkce je proměnná `s` uvolněna z paměti, a tedy odkazovat na ní nemá smysl (adresa v paměti bude přepsána něčím jiným).

Stack a heap

Fundementálním konceptem je zde skutečnost, že při referování na proměnnou se nejedná o typ `String`, kdežto se jedná o `&str` (string slice). Tento typ je

jakýmsi pouhým náhledem do `Stringu`. Jedná se o to, že `String` je alokován na heap (haldu), kdežto `&str` je na alokován na stack (zásobník).

Stack je způsob ukládání dat do paměti, při kterém je daná hodnota nabita (push) na místo v paměti s konstantní velikostí a posléze na něj může být nabita další hodnota, nebo může být hodnota ze stacku vytlačena (pop). Avšak vždy lze operovat pouze s hodnotou naposledy přidanou (tzv. LIFO model - Last In First Out).

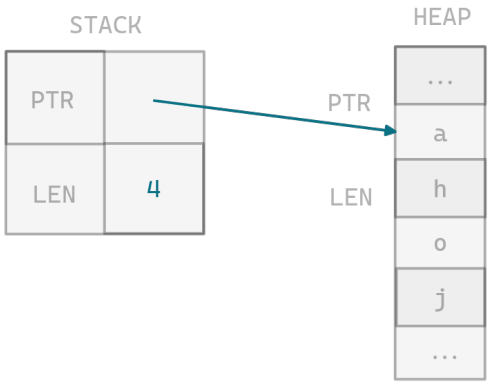


Obrázek 4: Vizualizace správy paměti stack

Druhým způsobem ukládání dat je heap. Ta je naopak od stacku používána pro hodnoty dynamicky měnící jako je `String` (a datové struktury, jejichž celková velikost není známa při kompilaci). Je to jakýsi region paměti, ke kterému může být postupně přialokována (nebo odalokována) další paměť.

Obecně se tyto dvě struktury liší rychlostí. Stack je znatelně rychlejší než heap, avšak nelze na něj alokovat hodnoty neznámé velikosti.

Kdykoliv je vytvořen řetězec, na heap je přidán samotný obsah řetězce a zároveň je na stack přidán pointer (reference) odkazující na adresu v paměti, kde začíná řetězec) a jeho délka (len)



Obrázek 5: Vizualizace vytvořeného řetězce znaků

Část tohoto diagramu zobrazující stack je právě `&str`. Toto je přesně typ, který je vracen inicializací jako `let s = "ahoj"`. U tohoto partikulárního příkladu je `"ahoj"` pevně zakódovaný do binárního souboru vyprodukovaného po kompilaci a při spuštění je načten do operační paměti.

Dereferenční operátor

Občas může nastat situace, kdy je žádáno získat hodnotu proměnné přímo, ač může být schována za referencí. K tomu lze využít dereferenční ho operátoru `*`

```
let jovan = 7;
let navoj = &jovan;


// `navoj` s `jovan` nelze porovnat přímo,
// jelikož jsou jiných typů | i32 a &i32
if jovan == *navoj {
    /* ... */
}
```

Lifetime

Lifetime (životnost) je vyjádřením doby, kterou daná proměnná žije. Buď demonstrací tento příklad s visící referencí.

```
fn main() {
    let t;

    {
        let u = 42;
        t = &u;
    }
    // pokud nejsou složené závorky {} nijak označené,
    // představují vlastní scope (rámec)

    // nic, co je vně těchto závorek, nemá přístup
    // k ničemu inicializovanému v tomto scope

    println!("{}", t);
}
```

Tento příklad se nezkompiluje, jelikož by `t` byla visící reference. Lifetimes proměnných `t`, `u` by vypadaly takto

```
fn main() {
    let t;               // ---------+ 'a
                         //          |
    {                    //          |
        let u = 42;      // -+ 'b    |
        t = &u;          //  |       |
    }                    // -+       |
                         //          |
    println!("{}", t);   //          |
}                        // ---------+
```

Lifetime `t` je zde anotován jako `'a` a lifetime `u` je anotován jako `'b`. Diagram značí kde dané lifetimes začínají a kde končí (tj. kde jsou proměnné zinvalidněny). Pomocí tohoto lze dovysvětlit visící referenci podrobněji — díky `t = &u;` odkazuje `t` na `u`, avšak `u` nežije dostatečně dlouho na to, aby mohla být využita její reference v `println!("{}", t);`.

Konsekvencí lifetimes proměnných je nutnost jejich specifikace v některých případech. Buď uvažován příklad

```
fn nejvetsi(cislo1: &i32, cislo2: &i32) -> &i32 {
    if (cislo1 > cislo2) {
        cislo1
    } else {
        cislo2
    }
}
```

```
fn main() {
    let t = 7;
    let u = 42;

    println!("{}", nejvetsi(&t, &u));
}
```

Toto se nezkompiluje, důvod je následující: výstupem funkce `nejvetsi()` je odkaz na jednu z proměnných, které jsou referencovány v jejích parametrech. Avšak z pohledu borrow checkeru, který kontroluje lifetimes, není jasné, jaké mají tyto vstupní parametry lifetimes. Co kdyby vracel odkaz na proměnnou, která však nebude mít v kontextu, ve kterém bude funkce použita, dostatečnou životnost?

Toto lze obejít anotací lifetimes v hlavičce funkce `nejvetsi` .

```
fn nejvetsi<'a>(cislo1: &'a i32, cislo2: &'a i32) -> &'a i32
```

Tato syntaxe říká, že lifetime výstupní hodnoty bude stejný jako nejmenší lifetime proměnných, na něž je odkazováno pomocí `cislo1` a `cislo2` .

Toto není samozřejmě jediný postup anotace. Ten se upraví dle způsobu, jakým má být funkce použita. Například pokud by bylo žádoucí svázat lifetime výstupu funkce s lifetime jedného z parametrů, šlo by to následovně.

```
fn jovan<'a, 'b>(i: &'a mut str, j: &'b i32) -> &'b i32
```

Lifetime anotace dovoluje také tvorbu **structů**, které nevlastní hodnoty svých polí

```
struct Jovan {
    navoj: &str,
}

fn main() {
    let s = "abc";

    let i = Jovan {
        navoj: s
    };
```

```
    println!("{}", i.navoj);
}
```

Toto by se nezkompilovalo, avšak po následné modifikaci ano.

```
struct Jovan<'a> {
    navoj: &'a str,
}
```

Cross-platform export aplikací

Jedna ze zásad Rustu je být lehce použitelný na kterékoliv běžné platformě, včetně vestavěných systémů.

Dostupnost možných kompilovacích cílů lze zjistit následovně

```
rustc --print target-list
```

V současné době tento list vrátí asi 175 možných cílů. Ty zahrnují architektury jako x86_64, aarch64, arm, i686, wasm32, a četné množství dalších pro systémy jako Windows, GNU/Linux, BSD, Android, MacOS a další. Pro export do kterékoliv z těchto cílů je však nejprve nutno stáhnout sadu pro danou architekturu. Například pro x86_64-pc-windows-gnu by příkaz vypadal následovně

```
rustup target add x86_64-pc-windows-gnu
```

Dále pro přehled všech instalovaných cílů lze použít

```
rustup target list
```

Windows

Windows obecně nevyužívá žádné kontejnerizace aplikací, díky čemuž je export poměrně jednoduchý.

Přídání ikony binárního souboru

Pro přidání ikony lze využít crate `winres`. Nejprve je nutno ji přidat do `Cargo.toml`

```
[target.'cfg(windows)'.build-dependencies]
winres = "0.1"
```

Význam pole `[target.'cfg(windows)'.build-dependencies]` spočívá v přidávání crates, které mají být importovány pokud je kompilováno s cílovou plat-

formou Windows.

Dále je potřeba přidat do stejného adresáře speciální soubour `build.rs`. Tento soubor slouží pro pokročilejší manipulaci při kompilaci. Jeho obsah bude následující

```
fn main() -> std::io::Result<()> {
    #[cfg(windows)] {
        winres::WindowsResource::new()
            .set_icon("resources/icon.ico")
            .compile()?;
    }
    Ok(())
}
```

Zde je využito podmíněné kompilace — pro rozhodnutí, zda-li má daná sekce být zkompilována, je využito nějakého logického výrazu, v tomto případě `#[cfg(windows)]`. Nadále v tomto úryvku `"resources/icon.ico"` je cesta k ikoně relativní vůči adresáři se souborem `Cargo.toml`.?

Následně, pokud aplikace využívá GUI, lze do souboru `main.rs` přidat následující

```
#![windows_subsystem = "windows"]
```

Toto zaručí, že se společně s GUI neotevře okno s příkazovým řádkem. Následuje

```
cargo build --release --target=x86_64-pc-windows-msvc
```

A tím je program postaven, výsledkem je binární `.exe` soubor.?

GNU/Linux

Linux je poněkud složitější co se týče exportu, to je dáno tím, že preferabilně jsou aplikace distribuovány kontejnerizované. V následujícím příkladu je popsána tvorba aplikace kontejnerizované dle standardu AppImage. Tento formát je široce rozšířený, má oficiální podporu na distribucích jako Arch Linux, Ubuntu, Debian, ad.?

Nejprve se aplikace zkompiluje, např. pomocí

```
cargo build --release --target x86_64-unknown-linux-gnu
```

Poté je nutno vytvořit dočasnou složku

```
mkdir temp.AppDir
```

Následně se do této složky nakopíruje binární soubor a jeho ikona

```
cp target/x86_64-unknown-linux-gnu/release/muj_program temp.AppDir/AppRun
cp resources/icon.png temp.AppDir/
```

V neposlední řadě je nutno v této složce vytvořit soubor s manifestem obsahující metadata

```
echo "
[Desktop Entry]
Name=Muj_program
Exec=muj_program
Icon=icon.png
Type=Application
Categories=Game;
" > temp.AppDir/Muj_program.desktop
```

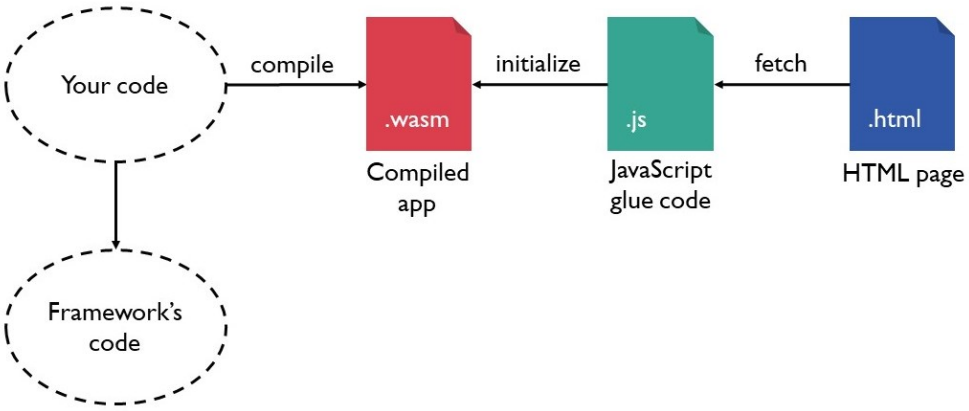
Takto je již aplikace připravená a lze použít např. nástroje appimagetool[?]

```
appimagetool temp.Appdir
```

Tímto dojde k vytvoření souboru `Muj_program.AppImage`, který lze distribuovat.

WebAssembly

WebAssembly (WASM) je technologie umožňující spuštění výpočetně náročných aplikací ve webovém rozhraní. Toho je dosáhnuto tak, že kód nějakého jazyka (preferabilně nízkoúrovňového) je přeložen do `.wasm` bajtkódu. Pokud tedy aplikace potřebuje vysoký výkon pro nějaký úkon, může ho JavaScriptové API přenechat tomuto WASM bajtkódu.[?]



Obrázek 6: Proces využití WebAssembly pro webové aplikace[?]

Postup pro získání bajtkódu i lepícího kódu (který bude komunikovat mezi JS a bajtkódem) lze postupovat následovně.

Nejprve je potřeba přidat dependencies do `Cargo.toml`

```
[target.'cfg(target_arch = "wasm32")'.dependencies]
tracing = "0.1.29"
tracing-subscriber = "0.2.25"
tracing-wasm = "0.2.0"
```

Užitý syntax má význam v přidávání crates specifických pro cílovou architekturu `wasm32`. Crate `tracing` není povinná, ale nastane-li chyba ze strany wasm bajtkódu, je schopna tuto chybu vypsat do vývojářské konzole webového rozhraní (DevTools). Posléze je nutno poskytnou speciální funkci, která bude vstupním bodem pro wasm

```
#[cfg(target_arch = "wasm32")]
#[wasm_bindgen]
pub fn main_web(canvas_id: &str) {
    tracing_wasm::set_as_global_default();

    // *kód pro spuštění na webu*
}
```

Nyní jen zbývá přidat následující do `Cargo.toml`

```
[lib]
crate-type = [ "cdylib", "rlib" ]
```

A nyní už je hotovo po stránce editování původního kódu a lze přejít ke kompilování.

```
cargo build --release --lib \
            -p muj_program \
            --target wasm32-unknown-unknown \
```

Následně je potřeba vygenerovat glue code. K tomu lze použít programu `wasm-bindgen`. Ten se dá nainstalovat přes

```
cargo install -f wasm-bindgen-cli
cargo update -p wasm-bindgen
```

Poté ho lze použít na vygenerovaný WASM soubor

```
wasm-bindgen target/wasm32-unknown-unknown/release/muj_program.wasm \
            --no-modules \
            --no-typescript \
```

Toto v adresáři WASM souboru vygeneruje soubor `muj_program_bg.js`. Nyní se vezme tento soubor společně s `muj_program.wasm` a oba jsou přesunuty do nové složky. V této samé složce je nutno vytvořit HTML soubor, např. `index.html`, s následujícím obsahem

```
<!DOCTYPE html>
<html>
<head>
    <title>Hra života</title>
</head>
<body>
    <canvas id="app"></canvas>

    <script src="muj_program_bg.js"></script>
    <script>
        wasm_bindgen("./muj_program.wasm")
            .then(on_wasm_loaded)
            .catch(console.error);

        function on_wasm_loaded() {
            wasm_bindgen.main_web("app");
```

```
        }
    </script>
</body>
</html>
```

Díky minimálnímu kódu je načten bajtkód a spuštěna dříve definované funkce `main_web()` odkazující na `canvas`, do kterého je GUI vykreslováno.

Nyní však ještě není hotovo. Po otevření `index.html` ve webovém prohlížeči nebude aplikace fungovat. Při otevření konzole lze identifikovat vadu - CORS (Cross-origin resource sharing). Toto dává smysl — při pohledu na soubor `index.html` lze zjistit, že funkce `wasm_bindgen()` potřebuje načíst soubor `muj_program_bg.wasm` toto však nelze, nejsou-li soubory získány přes webový server. Jeden způsob by byl instalací webového serveru, například `basic-http-server`

```
cargo install basic-http-server
```

Ten lze využít (v adresáři s webovými soubory) příkazem

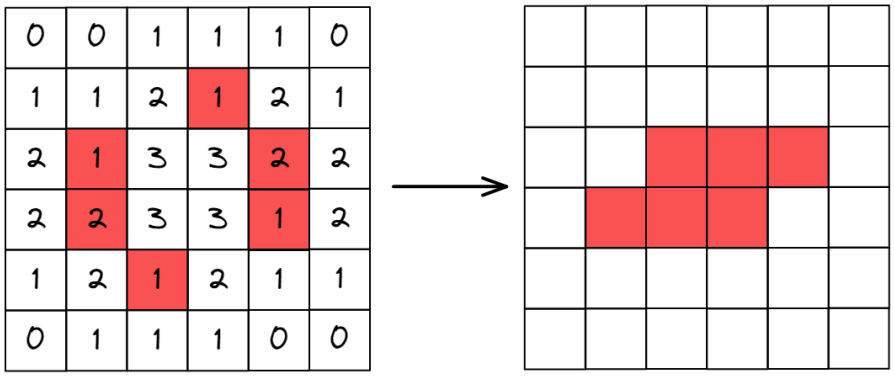
```
basic-http-server --addr 127.0.0.1:3000 .
```

Nyní je celý proces hotov a webovou aplikaci lze spustit z webového rozhraní na zpětné adrese `127.0.0.1:3000`.[?,?]

Hra života s GUI

Vymezení cílů

Cílem této aplikace je naprogramovat cross-platform řešení pro uživatelem přizpůsobitelnou simulaci Hry života Johna Conwaye. Jedná se o „hru jednoho hráče“, která je vizualizována jako dvojrozměrné pole buněk. Každá tato buňka je v jakémkoliv čase právě v jednom ze dvou stavů — mrtvá nebo živá. Čas v simulaci je kvantifikován na tzv. generace. To, zda mrtvá buňka v další generaci oživne, nebo to, zda-li živá buňka zemře, podléhá nastaveným pravidlům hry, které se vážou na počet živých sousedních buněk. Tradiční konfigurace se označuje jako S23/B3. To znamená, že aby živá buňka přežila, musí mít 2 nebo 3 živé sousedy a aby mrtvá buňka oživla, musí mít právě 3 živé sousedy (nastane-li cokoliv jiného, poté buňka umře, respektive zůstává mrtvá).[?]



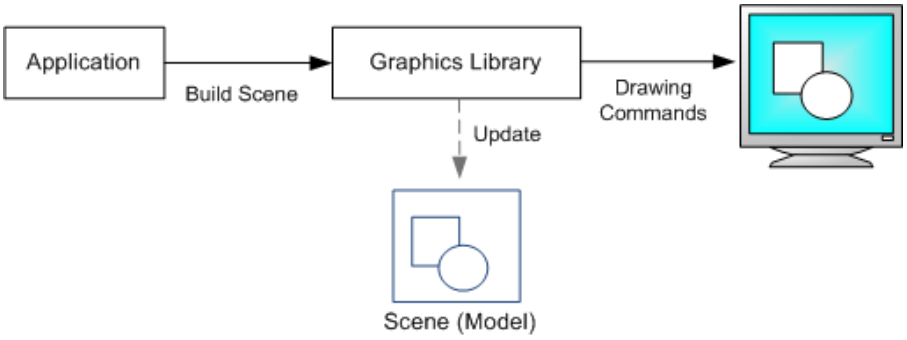
Obrázek 7: Příklad vytvoření nové generace v Conwayově Hře života, S23/B3

Aplikace by měla běžet na všech signifikantních platformách a měla by mít minimální systémové požadavky.

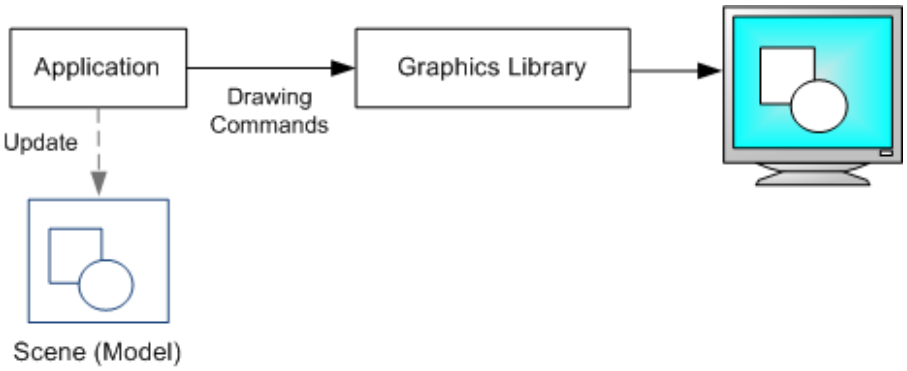
Výběr frameworku

Bohužel co se GUI frameworků týče, ekosystém Rustu ještě není dostatečně vybaven. Obecně není příliš velký výběr, všechny jsou immediate mode. Respektive existují ve vývoji i retained mode, avšak ty jsou ve velice útlém a experimentálním stádiu.

Rozdíl mezi těmito dvěma je ten, že retained mode GUI si uchovává informace o rozvržení okna aplikace, a tedy je toto okno přerenderováno pouze tehdy, je-li v jeho vzhledu detekována nějaká změna. Kdežto immediate mode GUI si neuchovává nic a každý jeden snímek je znovu renderován. Na základě těchto poznatků lze vyvodit, že pokud se vzhled okna má často měnit (jako třeba nějaká hra), poté je vhodnější immediate mode, kdežto u desktopových aplikací, jejichž vzhled se příliš nemění (jako třeba nějaké formuláře), je vhodnější retained mode. Tato aplikace má být něco mezi těmito dvěma paradigmaty, a tedy limit daný omezeným výběrem není tak veliký problém.



Obrázek 8: Diagram procesu renderování immediate mode[?]

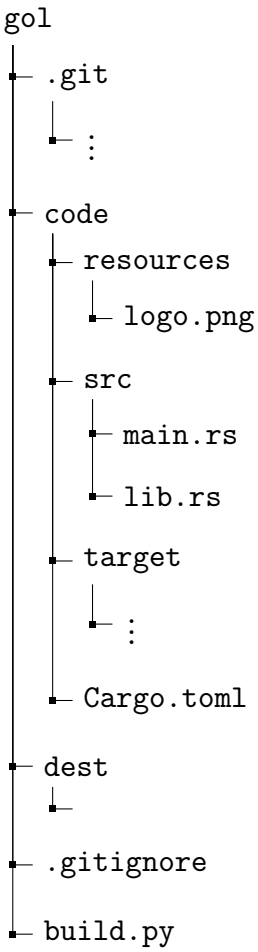


Obrázek 9: Diagram procesu renderování retained mode[?]

Frameworkem, který jsem nakonec zvolil je egui, a to zejména kvůli tomu, že disponuje všemi funkcionality, které potřebuji - vykreslováním všech widgetů, které chci a má excelentní podporu pro WebAssembly.[?]

Struktura / rozvržení

Po vytvoření projektu pomocí `cargo new gol` je vytvořeno několik dalších složek a souborů tak, že struktura souborů bude vypadat následovně:



- `code` slouží pro zdrojový kód a související zdroje
  - `resources` slouží pro obrázky, které program využívá
- `dest` slouží pro exportované aplikace
- `build.py` je skript pro export aplikací

Logická struktura

Jak je u větších projektů zvykem, samotný soubor `main.rs` je jakousi pouhou „vstupní bránou“, která pouze inicializuje program.

```rust
use eframe::{run_native, NativeOptions};

// Game struct v lib.rs
use gol::Game;

fn main() {
    // import GUI iniciátorů
    use eframe::{run_native, NativeOptions};
    // import z knihovny
    use gol::Game;

    // `tracing` slouží pro hlášení informací
    tracing_subscriber::fmt::init();

    let game = Game::new();

    let win_option = NativeOptions::default();
    // spuštění GUI
    run_native(Box::new(game), win_option);
}
```

Čiže se jen odkáže na položky v jiných modulech, v tomto případě v `lib.rs`. V něm bude struktura následující

```rust
// import lokálních souborů
mod config;
use config::Config;

// import komunitních beden
use eframe::{ /* ... */ };

// objekt buňky
#[derive(Default, Clone)]
struct Cell {
    pub is_alive: bool,
}

// herní objekt
pub struct Game {
    /* ... */
}

impl Game {
    // ---------------------------------------
    //      Tvorba
    // ---------------------------------------

    // pomocná funkce pro generování reprezentace pole s buňkami
    fn generate_board(size: usize) -> Vec<Vec<Cell>> { /* ... */ }

    // vytvoření instance hry
    pub fn new(size: usize) -> Game { /* ... */ }

    // ---------------------------------------
```

```
    //      Manipulace
    // ---------------------------------------

    // umožní změnit velikost hrací desky do spec. velikosti
    pub fn resize_to(&mut self, size: usize) { /* ... */ }

    // nastaví náhodný stav buněk
    pub fn randomize_cells(&mut self, factor: u8) { /* ... */ }

    // změní stav specifikované buňky
    pub fn change_cell_state(&mut self, cell: (usize, usize)) { /* ... */ }

    // vrátí pozici (ne)kliknuté buňky
    fn get_clicked_cell(&self, pos: Pos2, rect: Rect) -> Option<(usize,
↪   usize)> { /* ... */ }

    // ---------------------------------------
    //      Herní logika
    // ---------------------------------------

    // získá počet živých sousedů buňky
    fn get_neighbor_count(&self, pos: (i32, i32)) -> u8 { /* ... */ }

    // vytvoří novou desku na základě pravidel
    pub fn iterate(&mut self) { /* ... */ }

    // ---------------------------------------
    //      Kontroléry, komunikace s GUI
    // ---------------------------------------

    // rozhodování o možnosti další iterace v závislosti na konf.
    pub fn handle_iterate(&mut self, ui: &mut Ui) { /* ... */ }

    // přidělí místo pro pole s buňkami
    fn allocate_space_for_cells(&self, ui: &mut Ui) -> (Id, Rect) { /* ... */
↪   }

    // vykresluje pole s buňkami
    fn show_cells(&mut self, ui: &mut Ui, rect: Rect) { /* ... */ }
}

// egui vyžadouje implementaci `App` pro daný herní objekt
impl App for Game {
    // vykreslování
    fn update(&mut self,
                ctx: &eframe::egui::CtxRef,
                _frame: &mut eframe::epi::Frame<'_>){
        egui::CentralPanel::default().show(ctx, |ui| { /* ... */ });
    }

    // název okna
    fn name(&self) -> &str { "Hra života" }
}
```

Pro herní `struct` je tedy implementována požadovaná `trait App`, které

vyžaduje dvě metody. Zbylé implementované metody souvisejí s logikou hry a její funkčností. Zde však není vše, je také zapotřebí pár úprav pro WebAssembly. Konkrétně je nejprve nutno nahradit 2 funkcionality - sledování času a generování náhodných čísel. Pro tyto dva úkony jsou ve výchozím stavu užívány crates `std::time` a externí crate `rand` v tomto pořadí. Ty je nutno nahradit WebAssembly podporujícími alternativami.

```
#[cfg(not(target_arch = "wasm32"))]
use std::time::{SystemTime, UNIX_EPOCH};
#[cfg(target_arch = "wasm32")]
use wasm_timer::{SystemTime, UNIX_EPOCH};
```

Ještě je nutno označit v `Cargo.toml` `lib` pro podporu WASM

```
[lib]
crate-type = [ "cdylib", "rlib"]
```

Tato část by byla hotova a již stačí vytvořit vstupní bod pro WebAssembly (tj. funkci, na kterou se JavaScript gluecode napojí)

```
#[wasm_bindgen]
pub fn main_web(canvas_id: &str) {
    extern crate console_error_panic_hook;
    use std::panic;
    tracing_wasm::set_as_global_default();
    panic::set_hook(Box::new(console_error_panic_hook::hook));

    let game = Game::new();

    eframe::start_web(canvas_id, Box::new(game));
}
```

Již zbývá poslední krok — nastavit ikonu pro Windows pomocí `winres` v souboru `build.rs`

```
fn main() -> std::io::Result<()> {
    #[cfg(windows)] {
    winres::WindowsResource::new()
        .set_icon("resources/logo.ico")
        .compile()?;
    }

    Ok(())
}
```

A nastavit, aby se na Windows neotevíralo okno příkazového řádku, tedy přidat do hlavičky.

```
#![windows_subsystem = "windows"]
```

Závěr

Rust je velice dobrým programovacím jazykem a osobně jsem si ho oblíbil. Ačkoliv má poměrně strmou učící křivku, má velice přívětivou komunitu vývojářu, kteří jsou ochotni pomoci. Dalším nedostatkem je relativní nevyspělost ekosystému a nejspíše bude trvat ještě přinejmenším několik let, než bude mít Rust takové množství kvalitiních frameworků a nástrojů, které mají některé jiné programovací jazyky. Toto je však dle mého názoru vše, co lze vytknout.

V průběhu této práce jsem si upevnil svoji znalost jazyka Rust a naučil jsem se pracovat s novým grafickým frameworkem. Největší obtíže mi působilo vymyšlení podléhajících algoritmů.

Jakožto přílohu přikládám vypracovanou příručku — zevrubného průvodce pro započetí programování v Rustu.

Seznam příloh

- Dokument práce ve formátu PDF
- Dokument průvodce jazykem Rust ve formátu PDF
- Spustitelná aplikace pro platformu Linux, x86_64
- Spustitelná aplikace pro platformu Windows, x86_64
- Kolekce webových souborů (HTML, CSS, WASM, JS) pro spuštění aplikace webovým serverem
- Zdrojový kód aplikace