FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Formální jazyky a překladače
Méně známé jazyky: Rust

KIV/FJP

Pavel Třeštík
A22N0137P

29. října 2022

1 Úvod

V této eseji, budou prozkoumány základy jazyka Rust a na konci zhodnoceno, zda-li je jazyk vhodný pro začátečníky. Musím podotknout, že osobně s jazykem nemám zkušenosti, kromě příkladů, které jsem zkoušel pro tuto esej. O jazyk mám ale zájem, protože stále více lidí věří, že by Rust mohl (zvětšiny) nahradit C/ C++ a navíc má být jazyk podporován přímo v Linuxovým kernelu od verze 6.

2 Jazyk Rust

Rust začal vznikat v roce 2006 jako osobní projekt a lze ho tedy považovat za relativně nový jazyk. Obecně Rust nabízí více způsobů programování - tedy není to ani funkcionální, či objektový jazyk, ale trochu od všeho. Hlavní účel Rustu, je poskytnout vysokoúrovňové koncepty, ochrany a zároveň zachovat efektivitu výsledných programů. Rust je možné využít k programování široké škály aplikací od operačních systémů po webové prohlížeče a podobné aplikace. Rust je překládaný jazyk, což tvoří požadavky na některé vlastnosti.

2.1 Datové typy a deklarace

V moderním jazyku je nezbytné mít možnost deklarovat proměnné, se kterými se dále pracuje. V Rustu se proměnné deklarují klíčovým slovem `let`. Definovat datový při deklaraci není vždy nutné. Pokud ovšem typ není definován tak musí být jednoznačně definována hodnota, podle které lze typ určit. Typ musí být znám při překladu programu, pokud typ není deklarován, nebo nezle určit, tak se program nepřeloží. Konstanty musí mít vždy definovaný typ.

Listing 1: Deklarace int proměnné

```
let a = 5;
```

Velkou změnou oproti většině používaných jazyků je, že všechny proměnné jsou při běžné `immutable`. Pokud má být hodnota proměnné měněna, tak je třeba, aby deklarace proměnné obsahovala klíčové slovo `mut`.

Listing 2: Deklarace mutable proměné

```
let mut a: u32 = 5; // variable a is mutable and its
```

```
//type is set to be unsigned 32-bit integer
```

Druhý způsob, jak je možné dosáhnout změny hodnoty proměnné je tzv. `shadowing`.`Shadowing` je opětovné deklarování stejné proměnné ve stejném scope.

Listing 3: Příklad shadowingu

```
let a = 5;
let a = a + 1;
// a now contains values 6
```

2.2 Řízení toku programu

Rust nabízí standardně známou funkcionalitu pro řízení programu. Tedy poměrně klasické `if - else`, `for`, `while` a navíc `loop`, což je pouze nekonečná smyčka, kterou je třeba přerušit pomocí `break`. Rust nemá standardní `switch - case` syntaxi, ale funkcionalitu přepínače poskytuje pomocí konstrukce `match`.

2.3 Správa paměti - Ownership

Jedna z nejvýznačnějších věcí Rustu je jeho způsob správy paměti. Rust nepoužívá ani manuální správu paměti ani garbage collection. Místo toho používá koncept, kterému říká `ownership (vlastnictví)`. Ve zkratce vlastnictví funguje tak, že paměť proměnných je uvolněna na konci `scope`. Koncept je ale mnohem složitější. Například pokud funkce vrátí proměnou, která je ve funkci definována, tak vlastnictví této proměnné je přesunuto na volajícího této funkce.

Pokud je proměnná použita mimo svůj scope, tak dochází ke kopírování nebo přesunutí. To ale nemusí být chtěné chování a proto jazyk používá reference. Ve správě paměti jsou poté další složitější mechanismy, které slouží k ochraně paměti. Například - na proměnou smí existovat pouze jedna mutable reference. Nebo - pokud na proměnou již existují immutable reference, tak není možné vytvořit mutable referenci, dokud jsou immutable reference používané.

Podrobnější zkoumání mechanismu vlastnictví a správy paměti v Rustu je mimo tuto esej. Nejdůležitějším bodem, co je třeba si zde odnést je, že Rust má vysokou bezpečnost správy paměti a programátor je vždy informován o

nebezpečné práci s pamětí ve formě chyby (error) při překladu a je nucen chybu řešit.

2.4 Obsluha chyb

Jak již bylo zmíněno, tak Rust se snaží o to, aby výsledné programy byly robustní a bezpečné (proti chybám programátora). Z tohoto důvodu Rust nutí programátora ošetřovat potenciálně nebezpečné operace. Příkladem může být parsování řetězce na int (viz následující kód).

```
let number: u32 = super_string.trim().parse()
                    .expect("Not a number!");
```

V tomto příkladu je podstatná část `.expect("Not a number!");`, která shodí program s uvedenou hláškou, pokud parsování selže. Pokud by `expect` nebyl uveden, tak by kód nešel ani zkompilovat!

Způsob, jakým se pozná, že dojde k chybě je, že funkce jako `parse` vrací speciální strukturu `Result`, která obsahuje hodnotu `OK(operation_result)` a `ERR(specific_error)`. Místo volání `expect` je možné vytvořit vlastní obsluhu těchto chyb, která nemusí způsobit konec programu.

2.5 Rust paradigm

Rust nelze popsat jedním programátorským stylem. Kód lze psát funkcionálně nebo pokud programátor opravdu chce, tak i objektově. Návrhem se ale Rust více podobá jazyku C, kdy má funkce a struktury (`struct` pro uživatelské datové typy).

Objektové programování v Rustu je možné, ale je odlišné od běžného OOP. Jedním rozdílem je, že neexistují objekty, ale pouze struktury, ve kterých není možné specifikovat funkce. Funkce "nad objektem"jsou deklarovány a implementovány ve vlastním bloku `impl Nazev_Struktury)`. Dalším význačným rozdílem je, že v Rustu neexistuje dědičnost. Polymorfismu je ale možné dosáhnout pomocí tzv. `trait`, což je obdoba rozhraní.

2.6 Další vlastnosti

Rust také nabízí možnost generického programování. Podpora vláken je samozřejmostí a navíc díky celkovému konceptu Rustu, by měla být práce s vlákny bezpečná. Rust by měl programátora navést správným směrem, aby

byla práce s více vlákny bezpečná (hlavně nedocházelo k souběhům). Dále je zde možnost práce se smart pointery či třeba iterátory. V poslední řadě Rust také nabízí tzv. `unsafe` Rust, což je blok kódu, ve kterém je možné dělat nebezpečné operace jako například práce s raw pointery, která v Rustu jinak není možná.

Rust také obsahuje framework pro testování a psaní testů by mělo být velmi jednoduché.

3 Závěr

V této eseji je velmi povrchově prozkoumán jazyk Rust. K vypracování eseje byla použita především oficiální dokumentace, která je velmi podrobná a propracovaná (`https://doc.rust-lang.org/book/title-page.html`).

Osobně s jazykem nemám žádné zkušenosti a při vytváření této eseje jsem s jazykem pracoval poprvé. Můj názor na jazyk je převážně pozitivní. Jazyk nutí programátora psát bezpečný kód a nabízí velkou škálu moderních konstrukcí a postupů. Jednou z výtek, kterou bych k jazyku měl, je trochu "otravná"syntaxe v některých věcech. Například vracení hodnoty z funkce. Ačkoliv Rust má klíčové slovo `return`, které funguje, jak se předpokládá, tak ve všech příkladech je pro vrácení hodnoty použita syntaxe `expression_that_is_returned` bez středníku. Tedy např. `fn pokus() 3` je funkce, která vrací hodnotu 3. Pokud by ale funkce byla rozsáhlá, tak vrácení hodnoty tímto způsobem může být snadno přehlédnuto.

Celkově bych Rust určitě nedoporučil pro nováčky. Jazyk obsahuje řadu konceptů, které je vhodné znát pro práci s ním a nováčci jednoduše nemůžou mít tyto znalosti. Příkladem může být například to, že všechny proměnné jsou immutable a aby byly mutable, tak je třeba to extra deklarovat. Dalším příkladem je třeba to, že Rust poměrně hodně pracuje s referencemi, ale nedělá to automaticky (uživatel musí používat `&` na místě reference). Pokud se navíc tyto příklady zkombinují, tak vzniká nový příklad a to je správné použití mutable referencí. Jazyk bych doporučil pro programátory, co mají aspoň základní nebo radši pokročilé znalosti. Pro zájemce je dobré, aby znal pojmy stack, heap, pointer, reference a další.