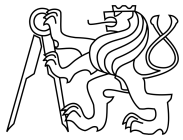
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Návrh a implementace jazyka s jednoduchou, bezpečnou a efektivní správou paměti
Student:	Miroslav Kravec
Vedoucí:	Ing. Jan Trávníček
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Analyzujte existující způsoby správy paměti založené na regionech v existujících programovacích jazycích.
Navrhnete sémantiku a syntaxi jednoduchého programovacího jazyka a pro něj vlastní model správy paměti založené na regionech.
Jazyk bude podporovat celočíselné datové typy, řetězce, pole, aritmetické operace, řídící struktury (if, if-else, while, for), třídy, metody, jednoduchou dědičnost.
Práce s regiony bude umožňovat vytvoření regionu, alokaci objektů a polí v regionu a automatickou destrukci regionu při opuštění jmenného prostoru regionu.
Implementujte prototyp Vašeho jazyka a prototyp Vašeho modelu správy paměti jako přední část překladače GCC nebo LLVM.
Navrhněte testovací programy pro změření paměťové efektivity Vašeho prototypu správy paměti.
Otestujte dosáhnutí požadovaných vlastností jazyka a prototypu správy paměti, změřte a porovnejte paměťové nároky vašeho jazyka a prototypu správy paměti s jinými programovacími jazyky.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 7. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalárska práca

Návrh a implementace jazyka s jednoduchou, bezpečnou a efektivní správou paměti

Miroslav Kravec

Vedúci práce: Ing. Jan Trávníček

9. mája 2017

Poďakovanie

Rád by som poďakoval vedúcemu práce, Ing. Janovi Trávničkovi, za možnosť pracovať na tejto téme, a taktiež za ochotu a pomoc pri vypracovaní tejto práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 9. mája 2017 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na túto prácu

Kravec, Miroslav. *Návrh a implementace jazyka s jednoduchou, bezpečnou a efektivní správou paměti.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Jednoduchosť použitia, efektívne využívanie prostriedkov, a bezpečnosť voči poškodeniu pamäte sú dôležité vlastnosti správ pamätí. Pre ich dosiahnutie sa práca zameriava na správu pamäte založenú na regiónoch . V práci boli analyzované existujúce jazyky, navrhnutý vlastný jazyk a k nemu implementovaný prototyp ako predná časť prekladača LLVM.

Klíčová slova Správa pamäte založená na regiónoch, návrh jazyka, LLVM, prekladač, programovací jazyk.

Abstract

Simplicity of use, efficient resources usage and safety against memory corruption are important characteristics of memory management. To achieve them, the thesis is focused on region based memory management. Existing languages were analysed. A new language was designed, and a prototype was implemented as a frontend for LLVM compiler.

Keywords Region based memory management, language design, LLVM, compiler, programming language.

Obsah

Zoznam obrázkov

Úvod

Webové aplikácie sú veľmi preferovanou voľbou nástroja na výmenu informácií a evidenciu údajov vo firmách, korporáciách, školstve a aj pre bežných užívateľov.

Podstatným kritériom pri voľbe jazyka pre implementáciu serverovej časti webovej aplikácie je jednoduchá a bezpečná správa pamäte, pretože zamedzí vzniku komplikovane riešiteľných chýb vzniknutých nesprávnou správou pamäte. Jednoduchosť správy znižuje nároky na vývojára, a tým zvyšuje rýchlosť vývoja.

Java, a podobné jazyky, sa preto stali preferovanou voľbou, mimo dynamických jazykov. Jazyk Java má automatickú správu pamäte založenú na zberaní odpadu (GC), ktorá zaručuje jednoduchú a bezpečnú prácu s pamäťou. Nevýhodou tejto správy pamäte je, že spotrebováva zbytočné vysoké množstvo pamäte, zvyčajne až 10-násobne viac, ako je potrebné.

Túto tému som si vybral, pretože efektivitu považujem za podstatnú zložku hodnotenia kvality implementovanej aplikácie. Správa pamäte založená iba na zberaní odpadu pre mňa predstavuje zbytočné plytvanie výpočtovými prostriedkami.

Cieľ práce

Hlavným cieľom bakalárskej práce je navrhnúť a implementovať jazyk s vlastným typom bezpečnej automatickej správy pamäte, ktorá bude efektívne využívať pamäť a zároveň bude čo najviac jednoduchá na použitie. Pri návrhu sa budem zameriavať na správu pamäte založenej na regiónoch.

V práci analyzujem existujúce programovacie jazyky implementujúce správu pamäte založenej na regiónoch. Následne navrhnem vlastnú správu pamäte, a

syntax a sémantiku jazyka. K navrhnutému jazyku implementujem kompilátor.

Pre overenie dosiahnutia vlastností navrhnem testovacie scenáre, a otestujem a porovnám svoje riešenie s inými jazykmi.

Kapitola 1

Analýza

1.1 Manuálna správa regiónov

Správa pamäte pomocou regiónov je starý koncept, ktorý existuje pod rôznymi názvami: zóny, arény, bazény (angl. pool), pamäťové kontexty. Explicitné regióny boli základom v návrhu veľkého množstva softvéru napísaného v C, vrátane Apache HTTP Server, a PostgreSQL.

V roku 1990 Hanson demonštroval, že explicitné použitie regiónov v C (ktoré nazýval arény) môže dosiahnuť lepšiu časový výkonnosť oproti najrýchlejšiemu známemu mechanizmu alokovania v halde.

Podobne ako alokácia na halde, tieto spôsoby nezaručujú pamäťovú bezpečnosť. Je povolené programátorom pristúpiť do regiónu po tom, ako bol dealokovaný, a to pomocou "dangling pointer", alebo je možné zabudnúť dealokovať región a spôsobiť únik pamäte.

1.2 Regióny v automatickej správe pamäte

V 1988, sa začal vývoj použitia regiónov pre bezpečnú alokáciu pamäte, a vznikol koncept region inference, kde kompilátor vloží do programu vytváranie a dealokáciu regiónov, taktiež priradenie jednotlivých statických alokačných výrazov do regiónov. Kompilátor to dokáže urobiť spôsobom, ktorý zaručí, že ”dangling pointer” a ”memory leaks” sa nevyskytnú.

V prvých prácach od Ruggieri a Murtagh, sa región vytvorí na začiatku každej funkcie a dealokuje na jej konci. Potom použijú analýzu dátového toku na určenie dĺžky života každého statického alokačného výrazu, a priradia ju do najmladšieho regiónu, ktorý obsiahne jeho celý čas života.

1.2.1 ParaSail

ParaSail je objektovo orientovaný programovací jazyk zameraný na bezpečné paralélne programovanie, a to vďaka jednoduchosti jazyka, a elimináciou vlastností brániacich bezpečnej paralelizácii.[1]

V ParaSail je odstránená práca s ukazateľmi, priradenia kopírujú hodnotu (sémantika hodnôt). Pre zvýšenie výkonu je možné "vymeniť" alebo "presunúť" objekty z jednej premennej do druhej. Jazyk nemá globálne premenné, spracovanie výnimiek za behu, automatickú správu pamäte so zbieraním odpadu, explicitné vlákna s blokovaním a signálmi.[2]

Keďže ParaSail nepodporuje aliasovanie objektov, tak pri každom priradení novej hodnoty je možné dealokovať pôvodnú hodnotu (objekt, strom objektov), pretože to je jediný ukazateľ na daný objekt.

1.2.2 Real time Java

Real time Java je pomenovanie pre technológie umožňujúce písanie programov v jazyku java tak, aby spĺňali potreby pre spracovanie reálnom čase.

Štandard RTJS (JSR) rozširuje Javu o mnoho vlastností, medzi nich patrí spôsob správy pamäte pomocou zón. Typy zón v RTJS:

Scoped memory poskytuje mechanizmus spravujúci objekty s dobou života definovanou "by scope", podobne ako alokovanie objektov na zásobníku.

Physical memory umožňuje alokovať objekty v špecifických oblastiach fyzickej pamäte, a tak využiť konkrétne špecifické vlastnosti danej pamäte, napríklad oveľa rýchlejší prístup (využiteľné v embedded vývoji so spracovaním v reálnom čase).

Immortal memory nesmrteľná pamäť, nepodlieha GC, a teda zdieľaná medzi všetkými plánovateľnými objektami, vrátane plánovaných objektov reálneho času, ktoré nesmú podliehať pauze spôsobenej GC.

Heap memory klasická alokácia na halde, podlieha GC.

V realtime java sú pamäťové zóny riadené pomocou inštancovania špeciálnych tried a volaní špeciálnych metód. Prevencia voči nesprávnemu priradeniu je zaručená JVM pre RT Java, ktorá je zvyčajne dynamická.

1.3 Bezpečná manuálna správa regiónov

1.3.1 Cyclone

Cyclone má byť bezpečný dialekt jazyka C. Cyclone je navrhnutý tak, aby predchádzal zraniteľnostiam jazyka C, a to bez straty možností jazyka C pre systémové programovanie.

Regióny v Cyclone

Pre dosiahnutie bezpečnej dealokácie Cyclone používa regióny. Každý objekt je alokovaný v regióne, a pre dealokkáciu objektu je potrebné dealokovať celý región (s výnimkou, nižšie). Cyclone má 4 typy regiónov[4]:

Stack regions región má fixnú veľkosť, a obsahuje lokálne premenné alokované na zásobníku.

Lexical regions život tohto regiónu je definovaný pomocou "scope", ktorá ho obklopuje. Objekty sú alokované na halde daného regiónu.

Heap region singleton, ktorý nie je nikdy dealokovaný. Nové objekty môžu byť v ňom vytvorené kedykoľvek. Dealokácia dát v tomto regióne je možná jedine pomocou GC.

Dynamic regions dynamický región je manuálne vytvorený a manuálne dealokovaný pomocou dekrementácie počítadla referencií. Používa sa pre dáta, ktorých doba života nie je staticky určitelná. Pre prácu s dátami regiónu je potrebné ho najprv otvoriť.

Aliasovateľnosť

Keďže objekty v regióne sú dealokované až pri dealokácií celého regiónu, tak môže dochádzať k zbytočnému použitiu pamäte, ktorá už nebude programom pristúpená. Preto Cyclone zaviedol mechanizmus pre umožnenie predčasnej bezpečnej dealokácie individuálnych objektov. Typ ukazateľa môže byt kvalifikovaný podľa jeho aliasovateľnosti[5]:

Aliasovateľný (angl. aliasable) predvolený typ ukazateľa, môžu byť aliasované. Nemôžu byť manuálne uvoľnené, pretože kompilátor nevie statickou analýzou určiť existenciu iného ukazateľa na rovnaký objekt.

Unikátny (angl. unique) ukazateľ ukazuje na objekt, pre ktorý nemôže existovať iný ukazateľ. Preto tento objekt môže byť kedykoľvek aj manuálne uvoľnený. Pre manuálnu dealokáciu, musí byť uvoľnený pomocou funkcie rufree, ktorá spôsobí konzumáciu ukazateľa.

S počítaním referencií (angl. reference counted) ukazateľ ukazuje na objekt, pre ktorý je počítaný počet referencií na neho ukazujúcich. Počet referencií je sledovaný dynamicky pomocou skrytého počítania referencií uloženého s objektom. Pre vytvorenie ďalšieho ukazateľa je potrebné použiť funkciu alias_refptr. Pre odstránenie ukazateľa je potrebne zavolať rdrop_refptr. Cyclone nevolá rdrop_refptr automaticky.

Restricted ukazateľ je super-typ pre všetky ukazatele, nemôže byť nikdy uvoľnený, ani nemôžu byť vytvorené aliasy.

Pre zamedzenie chybám vzniknutých s nesprávnou manipuláciou s ukazateľmi, Cyclone kladie ďalšie obmedzenia:

- Automaticky vložené overenia na nulový obsah ukazateľov pre prevenciu voči segmentation fault,
- Limitovaná aritmetika s ukazateľmi, závislá aj na kvalifikátoroch ukazateľa (celkom rozsiahla kapitola, ktorá sa už netýka správy pamäte, ale formátov objektov),
- Ukazateľ musí byť inicializovaný pred jeho použitím,
- Polovenie len "bezpečných" pretypovaní.

Ukážka polymorfického kódu

Cyclone podporuje polymorfizmus pre štruktúry. Ich funkčnosť je podobná template systému v jazyku C++. V kombinácii s kvalifikátormi aliasovateľnosti je možné nadefinovať abstraktný typ štruktúry pre uzol zoznamu[5]:

```
struct List<`a::B,`r::R,`q::Q>{
  : RESTRICTED >= aquals(`a), RESTRICTED >= `q
  `a hd;
  struct List<`a,`r,`q> *@aqual(`q) `r tl;
};
typedef struct List<`a,`r,`q> *@aqual(`q) `r list_t<`a,`r,`q>;
```

Táto štruktúra je definovaná parametrami:

`a typ ukazateľa na obsah uzlu listu,

`r identifikátor regiónu pre uloženie ďalších uzlov (celého zoznamu),

`q kvalifikátor aliasovateľnosti pre uloženie ďalších uzlov (zoznamu).

S využitím argumentov, štruktúra obsahuje vlastnosti:

hd obsahuje hodnotu uzlu, typu `a,

tl ukazuje na ďalší uzol alokovaného na regióne `r s aliasovateľnosťou `q.

Deklarácia štruktúry definuje obmedzenie pre aliasovateľnosť typu `a ako podtyp typu RESTRICTED, a argument `q musí byť podtyp typu RESTRICTED. Toto kladie najvyššie obmedzenia pre písanie abstraktného kódu, ale umožnuje použiť danú štruktúru s argumentmi akejkoľvek aliasovateľnosti.

1.3.2 Rust

Rust je systémový programovací jazyk so zameraním na bezpečnosť, rýchlosť a paralélne/súbežne spracovanie.

Za nebezpečné správanie Rust považuje data races, dereferencovanie null ukazateľa, dereferncovanie dangling/raw ukazateľa, čítanie neinicializovanej pamäte, zmenu nezmeniteľnej hodnoty/referencie bez 'UnsafeCell', a špecifické low-level chyby.

Vlastníctvo dát

Systém 'vlastníctva dát' v Rust je zero-cost abstrakcia pre bezpečnú prácu s dátami. Všetka analýza prebieha počas kompilácie, aby sa dosiahla lepšia efektivita bez ceny run-time.

Premenné v Rust majú vlastnosť 'vlastníctva' dát, ktoré majú priradené. Rust zabezpečí, že existuje iba jedno priradenie k akýmkoľvek dátam. Preto, Rust používa 'move' sémantiku pre priradenia. Napríklad, následujúci kód zneplatní priradenie objektu do premennej 'v':

```
let v = vec![1, 2, 3];
let v2 = v;
```

Po priradení 'v' do 'v2', použitie 'v' dopadne chybou pri kompilácií. Keďže 'move' sémantika nie je žiadúca pri hodnotových typoch (bool, int, a iné vlastné), tak Rust podporuje 'copy' sémantiku priradenia pre typy implementujúce 'Copy' trait. Ten je implementovaný pre všetky základné typy hodnôt, a preto následujúci kód je platný aj pri systéme vlastníctva:

```
fn main() {
    let a = 5;
    let _y = double(a);
    println!("{}", a);
}

fn double(x: i32) -> i32 {
    x * 2
}
```

Požičiavanie dát

Keďže, do funkcií zvyčajne predávame aj dáta, ktoré budeme aj naďalej potrebovať, tak by sme pri 'move' sémantike museli písať kód takto':

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // Do stuff with `v1` and `v2`.
    // Hand back ownership, and the result of our function.
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Na zníženie komplikácií spôsobených systémom jediného vlastníctva dát, Rust implementuje systém požičiavania dát. A tak môžeme predať do funkcie požičiané dáta:

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // Do stuff with `v1` and `v2`.
    // Return the answer.
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);
```

Doba života

Každá premenná má vlasnosť 'doby života' dát, ktoré su k nej priradené. To slúži ako prevencia voči dangling ukazateľom spôsobených priradením do premennej s dlhšou dobou života, ako majú samotné dáta. Napríklad:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let f : Foo;
    {
        let y = &5;
        f = Foo { x: y };
    };
    println!("{}", f.x);
}
```

Tento kód neprejde kompiláciou. Premenná y je alokovaná vo vnútornom scope funkcie a do premennej f sa pokúšame priradiť štruktúru obsahujúcu referenciu na y. Keďže premenná f prežije y, tak je tento kód nebezpečný, a systém lifetime to ošetrí.

Regióny v Rust

Rust sam o sebe neimplementuje spravu pamäte založenej na regiónoch. S použitím unsafe operácií a lifetime je možné použiť arény. TypedArena[9]:

```
pub struct TypedArena<T> {
    /// internal structures
}

impl<T> TypedArena<T> {
    pub fn new() -> TypedArena<T> {
        // code
    }

    pub fn alloc(&self, object: T) -> &mut T {
```

```
        // code
    }
}
```

Parameter 'T' určuje typ objektu. Arena implementuje funkciu alloc, ktorá vráti mutable referenciu na nový objekt typu 'T'. Keďže v deklarácii typu Arena a funkcii alloc nie sú definované lifetime, tak sú použité defaultné, v tomto prípade lifetime objektu arena.

1.4 Porovnanie vlastností správy pamätí

Pri porovnaní sa zameriavam na vlastnosti jazyka vplývajúce na využitie pamäte, pretože cieľom práce je implementovať jednoduchý jazyk, ktorý by bezpečným spôsobom využíval pamäť efektívnejšie ako zberanie odpadu.

1.4.1 Regióny

Region inference

Region inference implementuje regióny na pozadí automatickej správy. Jeho použitie priamo nekladie požiadavky na programátora a tak nezvyšuje náročnosť programovania.

Regióny môžu obsahovať množstvo mŕtvych dát, teda dočasných únikov pamäte. Tento problém sa dá vyriešiť reštrukturovaním programu, a/alebo zavedením regiónov s kratšou dobou života vrámci veľkých výpočtov.

Nevýhodou region inference je, že tento problém sa ťažko rieši. Pretože, programátor musí chápať algoritmus na pozadí, a nesmie spraviť chybu, ktorá by algoritmu zabránila tvorbe regiónov s krátkou dobou života.

Region inference s explicitnou deklaráciou regiónov

Realtime Java implementuje region inference pri priradzovaní, a definovaní premenných a vlastností tried. Vytváranie regiónov ostáva na programátorovi.

Tuna sa problém s hromadením mŕtvych dát rieši ľahšie, pretože programátor môže manuálne definovať región so zaručene kratšou dobou života.

Nevýhodou je, že pri priradzovaní musia byť dodržané smery referencií. Chyba by dopadla runtime výnimkou, a teda mohla spôsobovať pád programu v reálnom použití, teda bezpečnú nestabilitu programu. Toto by sa dalo ošetriť dobrou veľkou sadou testov.

Keďže vzťahy sú overované runtime, tak to spôsobuje určitý overhead.

Explicitné regióny

Explicitné regióny vyžadujú asistenciu od programátora, kde programátor musí určiť v ktorom regióne sa dáta nachádzajú. To je nevýhodou oproti region inference v množstve písaného kódu, ale je to aj zároveň výhodou, pretože programátor má plnú kontrolu nad regiónmi, a zároveň má statickou analýzou (pri kompilácii) garantovanú správnosť kódu (priradzovaní).

Jazyk Cyclone implementuje robustný spôsob správy pamäte založenej na regiónoch. Implementuje scope, lexikálne a dynamické regióny. Jazyk Rust neimplementuje regióny, ale použitím lifetime a unsafe operácií je možné implementovať regióny jazyka Cyclone v Rust.

Pri návrhu jazyka sa budem odvíjať od Cyclone.

1.4.2 Aliasovateľnosť

Aliasovateľnosť je vlastnosť typu ukazateľov a referencií na objekty, ktorá vypovedá o spôsobe vytvárania aliasov pre rovnaké dáta/objekt. Pre určité druhy aliasovateľnosti je možné bezpečne dealokovať objekt predčasne, mimo hromadnej dealokácii regiónu, alebo pri zberaní odpadu. Napríklad, pri unikátnom ukazateli je možné objekt dealokovať okamžite po zániku jediného ukazateľa/referencie na objekt.

Wrapper definujúci aliasovateľnosť

Aliasovateľnosť môže byť definovaná zaobaľujúcim typom (angl. wrapper), ktorý si môže sledovať určité vlastnosti stavu (napr. počet referencií). Wrapper môže pri zániku dealokovať objekt na základe sledovaného stavu, a môže byť implementovaný:

- ako wrapper ukazateľa. V tomto prípade sa pre sledovanie stavu používa zvyčajne fat-pointer, teda štruktúru obsahujúcu ukazateľ na reálny objekt a na dáta pre sledovanie stavu aliasovateľnosti objektu.
- ako wrapper štruktúry. V tom prípade je štruktúra rozšírená na začiatku, alebo na konci, o dáta pre sledovanie aliasovateľnosti. Tento spôsob si vyžaduje aj rozšírenie spôsobu alokácii objektov.

Nevýhodou je nutnosť špecifikovania typu aliasovateľnosti pri každej deklaracii referencie na objekt. Pri reštrukturizácií programu a zmene typu aliasovateľnosti určitých dát.

Písanie abstraktného kódu zvyčajne vyžaduje viacnásobný preklad do strojového jazyka, pretože implementácia priradení je špecifická pre jednotlivé typy aliasovateľnosti. Počet prekladov narastá exponenciálne s počtom abstraktných typov funkcie/štruktúry.

Ďalší problém je kolízia s OOP, kde klasická deklarácia metód nemôže predávať ukazateľ na seba ďalej, pretože nevie akú má 'this' aliasovateľnosť. Vyriešené to môže byť deklaráciou metód špecifických pre jednotlivé aliasovateľnosti, alebo abstraktných metód.

Aliasovateľnosť definovaná typom

Aliasovateľnosť môže byť definovaná v deklarácii typu objektu. Každý ukazateľ na daný typ objektu by používal rovnakú aliasovateľnosť. Zmena aliasovateľnosti by bola jednoduchá, a nie je potrebné prepisovať množstvo kódu.

Problémom je kolízia s rozhraniami v OOP. Keďže aj deklarácie na referencie rozhrania musia obsahovať informáciu o aliasovateľnosti typu, tak by aj deklarácia rozhrania musela definovať aliasovateľnosť. A to spôsobí nekompatibilitu rozhraní rôznych aliasovateľnosti. Tým zníženia možností abstrakcie kódu pomocou rozhraní v OOP.

1.5 Ostatné vlastnosti jazykov

Jazyk definujú mnohé ďalšie vlastnosti, nielen spôsob správy pamäte. Keďže je práca zameraná na správu pamäte v jazykoch, preto je popísaná iba jedna vlasnosť, ktorá bude do určitej miery zakomponovaná do jazyka.

1.5.1 OOP

Low-level jazyky rozlišujú medzi štruktúrami ako dátovými typmi a dynamicky alokovanými štruktúrami ako "objekty". To síce pri správnom používaní znižuje využitie pamäte ukazateľmi a hlavičkami objektov, ale môže aj spôsobiť zvýšené používanie pamäte vďaka kopírovaniu. Každopádne však zvyšuje komplexnosť používania, pretože programátor musí rozlišovať čo a kedy použiť.

Navrhnutý jazyk bude preto plne objektovo orientovaný bez explicitného používania ukazateľov. Dynamická alokácia nezvyšuje asymptoticky spotrebu pamäte, iba konštantne o pár bajtov.

Kapitola 2

Návrh jazyka

Objektovo orientované programovanie som si zvolil ako základné paradigmum pri návrhu jazyka, a rozšírim ho automatickú správu založenú na regiónoch.

Primárnym zameraním práce je správa pamäte, a preto sa návrh a implementácia prototypu jazyka zameriava hlavne na správu pamäte.

Programovací jazyk som nazval **elism**, podľa zamerania jazyka v angličtine: **e**asy, but **li**ght and **s**afe **m**emory management.

2.1 Dátové typy

Dátove typy definujú druh hodnôt, ktoré môže premenná alebo hodnota nadobúdať. Jazyk podporuje primitívne dátove typy, polia, reťazce, a referencie na objekty.

2.1.1 Primitívne typy

Podporovanými primitívnymi typmi sú celé čísla, čísla s plávajúcou desatinnou čiarkou a boolean.

Návrh dátových typov sa odvíja od jazyka Rust, pretože má jednoznačne definované číselné typy, ktoré majú veľkosť definovanú v názve ako počet bitov:

- celé čísla so znamienkom: i8, i16, i32, i64.
- celé čísla bez znamienka: u8, u16, u32, u64.
- s plávajúcou desatinnou čiarkou: f32, f64.

Implementácia prototypu pokrýva iba i32, i64 a boolean.

2.1.2 Referencie na objekty

Referencie ukazujú na dynamicky alokované dáta, a to sú inštancie tried a polia. Rovnako ako v ostatných jazykoch vyššej úrovne, funkčnosť referencií je podobná ukazateľom na triedu v C++, ale s určitými obmedzeniami.

Pre zaistenie bezpečného prístupu do pamäte, nie je povolená aritmetika ukazateľov (referencií), a programátor nemá možnosť prístupu k surovej forme štruktúry reprezentujúcej alokovaný objekt.

Mnoho jazykov podporuje nulovú hodnotu referencií, ktorá reprezentuje nepriradenie objektu do referencie. Dôsledkom je nutnosť overovať referencie na nulový obsah pri každom ich použití (volania metód, prístup k vlastnostiam). V prípade výskytu nulovej hodnoty dôjde k vzniku výnimky za behu aplikácie.

Jazyk nebude implementovať nulové referencie, a ako náhradu bude podporovať optional typy. Bez nulových referencií sa zamedzí propagácia nulových hodnôt do oblasti kódu, kde nie sú vôbec očakávané. A ošetrenie výskytu nulovej referencie je vynútené pred vstupom do časti kódu, ktorá neočakáva nulový obsah.

2.1.3 Polia

Pole je dátový typ, ktorý uchováva sériu hodnôt rovnakého dátového typu. Prvky v poli sú indexované od nuly po veľkosť poľa mínus jedna.

Samotný dátový typ má charakter referencie ukazujúcej na dynamicky alokovanú štruktúru, ktorá obsahuje veľkosť poľa a prvky poľa.

Pomocou špeciálnych operátorov je možné z poľa vybrať prvok, alebo priradiť mu novú hodnotu. Pri použití poľa sú kompilátorom automaticky pridané overenia na veľkosť poľa. Na rozdiel od C++, aritmetické operácie nie sú povolené.

Keďže neexistuje null, tak prvky poľa musia byť inicializované.

2.1.4 Znak a reťazec

Keďže v modernom IT je bežné používanie znakov mimo rozsah ASCII, tak znakový typ má veľkosť 4B a uchováva znak vo formáte Unicode.

Reťazec je klasický UTF-8 kódovaný reťazec, a nie je možné pristupovať k jednotlivým znakom. Pre prístup po znakoch je možné reťazec skonvertovať do poľa znakov.

Implementácia prototypu obsahuje iba jednoduché reťazce.

2.2 Základné štruktúry jazyka

2.2.1 Funkcie

Funkcia je postupnosť príkazov, a vykonáva určitú úlohu. Každý program obsahuje hlavnú funkciu main, ktorá definuje funkčnosť celého programu.

Najkratší validný program je možné napísať takto ako prázdnu funkciu:

```
fn main() {}
```

Funkcia môže mať parametre, a ohraničuje scope príkazov. Príkazy môžu v scope definovať nové premenné, priradiť určité hodnoty k názvom, a využívať definované hodnoty menného priestoru.

```
fn foo(x: i32) -> i32 {
    return x * 2;
}
fn main() {
    let x = foo(x);
    println("x = %d", x);
}
```

2.2.2 Príkazy (statements)

Let

Príkaz let v aktuálnom mennom priestore priradí výsledok výrazu k názvu:

```
let NAZOV = VYRAZ;
let NAZOV : TYP = VYRAZ;
```

Toto priradenie neslúži na vytvorenie premennej, hodnota je priradená k názvu iba raz. Následne je možné opakovane využívať nadefinovanú hodnotu.

Var

Príkaz var vytvorí novú premennú uloženú na zásobníku a vloží do nej výsledok výrazu.

```
var NAZOV = VYRAZ;
var NAZOV : TYP = VYRAZ;
```

Po tvorení je možné premennú používať ako hodnotu, alebo je možné ju používať ako cieľ priradenia a tým do nej vložiť novú hodnotu.

Priradenie

Príkaz priradenie priradí výsledok výrazu do adresy získanej pomocou iného výrazu:

```
VYRAZ_ADRESY = VYRAZ_HODNOTY;
```

Výsledok výrazu adresy musí byť hodnota umožňujúca priradenie. Napríklad premenná, alebo vlastnosť triedy(popísané v ďalšej sekcii):

```
nazov_premennej = 5 * 3 * 2;
instancia_triedy.vlastnost = 5 * 3 * 2;
```

2.2.3 Výrazy (expressions)

Výraz spracováva jednu alebo viacero vstupných hodnôt, a výsledkom je nová hodnota. Výrazy majú zhodnú funkčnosť ako v jazykoch C, C++, Java,...

Aritmetické operácie

V prototype sú implementované aritmetické operácie na celo-číselných dátových typoch: +, –, *, /. Zložené výrazy sa vyhodnocujú v matematickom poradí - násobenie a delenie má prednosť pred sčítaním a odčítaním.

Volanie funkcie

Volanie funkcie je výraz pozostávajúci z názvu funkcie a jedného, žiadneho, alebo viacerých vstupných argumentov.

Volanie predá argumenty funkcii a výslednou hodnotou volania je návratová hodnota z funkcie.

Získanie vlastnosti

Získanie vlastnosti je výraz, ktorý pozostáva zo základného výrazu a názvu vlastnosti. Výraz umožňuje získať vlastnosť s daným z hodnoty základného výrazu, ktorá má komplexný typ.

Výsledkom výrazu je získaná vlastnosť vzhľadom na hodnotu základného výrazu. Dostupné vlastnosti sa líšia vzhľadom na typ základnej hodnoty.

Polia majú vlastnosť length, ktorá definuje ich dĺžku:

```
let pole = [];
instancia_triedy.vlastnost = 5 * 3 * 2;
```

Objekty, inštancie tried, majú vlastnosti definované svojou triedou.

2.2.4 Podmienky

Podmienené príkazy umožňujú podmienené vykonanie určitého príkazu alebo skupiny príkazov, ak je určitý výraz vyhodnotený ako pravdivý. Prípadne, podmienený príkaz môže obsahovať aj príkaz alebo skupinu príkazov, ktorá sa má vykonať, ak je výraz nepravdivý.

Funkčnosť a syntax je zhodná s jazykom C:

```
if ( expression ) {
    ... statements executed if expression is true ...
}
```

```
if ( expression ) {
    ... statements executed if expression is true ...
} else {
    ... statements executed if expression is false ...
}
```

2.2.5 Cykly

Cykly sú bežným konštruktom umožňujúcim opakovanie vykonávania rovnakého kódu za určitých podmienok.

Jazyk podporuje cyklus for a while.

While

Cyklus while pozostáva z podmienky a príkazu. While sa používa na predom neurčitý počet opakovaní tohto príkazu, pokiaľ daný výraz je vyhodnotený ako pravdivý.

Syntax a sémantika je rovnaká ako v jazyku C. Zápis while:

```
while(vyraz_podmienka)
    prikaz;

while(vyraz_podmienka) {
    blok_prikazov;
}
```

For

Cyklus for pozostáva z generátora hodnôt, deklarácie premennej, a príkazu. Generátorom hodnôt môže byť iterátor poľa, alebo generátor určitej postupnosti. Cyklus for sa zvyčajne na prechádzanie prvkov daného generátora, pri každom opakovaní sa hodnota prvku priradí do danej premennej a vykoná sa príkaz.

Syntax a sémantika je podobná cyklu for v Python, respektíve ako foreach v jazyku Java. Zápis for:

```
for(name: expr)
    prikaz;

for(name: expr) {
    blok_prikazov;
}
```

2.3 Triedy

Hlavnou charakteristikou OOP je používanie objektov. Keďže navrhovaný jazyk má statické typy, tak aj objekty musia mať statický typ, teda ich vlastnosti a metódy musia byť definované pred kompiláciou konštruktov, ktoré ich využívajú. Triedy slúžia na kompletnú definíciu typu objektu.

2.3.1 Deklarácia vlastnej triedy

Deklarácia triedy pozostáva z názvu triedy, a obsahuje zoznam vlastností a metód:

```
class Foo {
    var value: i32 = 0;

    fn getValue() -> i32 {
        return value;
    }
}
```

Metódy sú funkcie so skrytým parametrom, ktorý ukazuje na aktuálnu inštanciu triedy, teda objekt, na ktorom sa daná metóda vykonáva.

2.3.2 Používanie tried a objektov

Používanie tried nezavádza nové konštrukty, ale využíva už existujúce konštrukty jazyka. Deklarácia triedy vytvorí novú funkciu a typ, ktoré priradí k názvu triedy.

Konštruktor

Konštruktor je funkcia, ktorá vytvára nový objekt. Konštruktor je automaticky vygenerovaný kompilátorom.

Konštruktor je možné použiť ako funkcium a jeho návratovou hodnotou je nová inštancia objektu:

```
let foo = Foo();
```

Dátový typ

Dátový typ vlastnej triedy má charakter referencie na dáta uložené na halde. Tento typ umožňuje pristupovať k vlastnostiam a metódam tried pomocou použitia existujúcich konštrukov s inštanciami tried.

```
class FooPair {
    var a : Foo = Foo();
    var b : Foo = Foo();

    fn calculate() {
        return a.value + b.value;
    }
}

let foo1 = Foo();
let foo2 = Foo();

foo1.value = 3;
foo2.value = 7;

let pair = FooPair();
pair.a = foo1;
pair.b = foo2;

let result = pair.calculate();
```

2.3.3 Dedičnosť

Deklarácia triedy umožňuje dediť vlastnosti od inej triedy:

```
class Foo {
    var n1: i32 = 1002;
    var str1 = "_1001_";

    fn printSelf() {
        printf("self [n1: %d, str1: %s]\n", n1, str1);
    }

}

class Bar: Foo {
    var nBar = 2001;
    var strBar = "_2001_";

    fn printSelf() {
        printf( "self [n1: %d, nBar: %d, str1: %s, strBar: %s]\n",
                n1, nBar, str1, strBar );
    }
}

fn main () -> i32 {
    var foo = Foo();
```

```
    foo.printSelf();

    var bar = Bar();
    bar.printSelf();

    return 0;
}
```

V ukážke dedí trieda Bar vlastnosti od triedy Foo.

2.3.4 Obmedzenia návrhu

Každá referencia na objekt musí byť inicializovaná, keďže jeden z cieľov pri návrhu bolo aj odstránenie výskytu nulových referencií v miestach, kde nie sú očakávané.

V ukážke v predchádzajúcej sekcii je vidieť, že typ FooPair má automatický priradenú novú inštanciu do vlastnosti a, a do vlastnosti b.

Parametre konštruktora

Jedným spôsobom prevencie voči vytváraniu zbytočných objektov je možnosť vytvoriť a parametrizovať vlastný konštruktor, ktorý by nastavil dané vlastnosti.

Takýto konštruktor by pracoval s inštanciou triedy, ktorá nemá plne inicializované vlastnosti. Preto by bolo potrebné pridať komplexné overenia pri preklade konštruktora, aby nedošlo k použitiu vlastnosti pred jej inicializáciou.

Parametrizovaný a vlastný konštruktor nebol navrhnutý, ani implementovaný, v prototype jazyka.

Nepovinné typy

Konštruktor nevyrieši nemožnosť vytvoriť nulovú referenciu v miestach, kde očakávame prípadnú neprítomnosť referencie.

Toto obmedzenie je možné vyriešiť pomocou nového 'nepovinného' typu referencie, ktorý by mohol uchovávať nulovú referenciu.

Nepovinné typy neboli navrhnuté, ani implementované, v prototype jazyka.

2.4 Správa pamäte založená na regiónoch

Správa pamäte založená na regiónoch rozširuje správu možnosti jazyka o alokáciu objektov v regiónoch a tvorbu vlastných regiónov.

Regióny umožňujú bezpečnú a efektívnu hromadnú dealokáciu dát. A ich použitie je vhodné pre aplikácie s dávkovým spracovaním. Napríklad spracovania viacerých sád dát, alebo spracovanie požiadavkov.

2.4.1 Úvodný koncept

Cieľom pridania regiónov je, okrem iného, aj sekvenčného spracovanie požiadavkov s odstraňovaním dočasných dát ihneď po dokončení spracovania požiadavku. Následujúci kód popisuje koncept hlavnej funkcie, ktorá:

1. vytvorí riadiace objekty pre spracovanie požiadavkov (konfigurácia).
2. spracováva požiadavky v cykle, tak že:
   a) vytvorí nový región pre alokáciu dočasných dát pri spracovaní požiadavku;
   b) čaká a prijme požiadavok;
   c) spracuje požiadavok na aktuálnom regióne, a s prístupom do globálneho regiónu pre používanie riadiacich objektov;
   d) ukončí prácu s dočasným regiónom a dôjde k dealokácii jeho dát.

Koncept funkcie:

```
fn main () -> i32 {
    let global_stuff = ...;

    while(...) {
        // new region is created
        on REQUEST_REGION = NewRegion() {
            let request = ...;

            // process request
        }
        // data in REQUEST_REGION are deallocated
    }

    return 0;
}
```

Funkčnosť tohto príkladu bude vysvetlená v ďalších sekciách.

2.4.2 Význam regiónov v sémantike jazyka

Aplikácie budú typicky pracovať s dátami viacerých regiónov. Napríklad, spracovanie požiadavku pracuje s dočasnými dátami v samostatnom regióne, ale pri spracovaní sa využívajú aj dáta v globálnom regióne.

Cieľom návrhu je aj efektivita správy pamäte a práce s touto pamäťou. Preto som si zvolil zakomponovanie regiónov do sémantiky jazyka podobne, ako

to má jazyk Cyclone. To umožní implementovať prekladač tak, aby počas prekladu overil bezpečnosť pomocou statickej analýzy.

Zakomponovanie práce s viacerými regiónmi rozširuje určité konštrukty o prácu s regiónmi. Sémantický význam rozšírení môže byť iba lexikálny alebo aj fyzický.

Fyzický región

Fyzický región je dátová štruktúra, ktorá slúži na alokovanie dát do daného regiónu, a taktiež umožňuje vykonať svoju dealokáciu aj so všetkými alokoványmi dátami.

Priame použitie štruktúry a prístup k fyzickému regiónu nie je možné. Fyzicky region je priamo využívaný iba pri preklade určitých konštruktov.

Reálna implementácia alokátora a celej štruktúry je závislá na implementácii. Sémantika jazyka požaduje iba, aby dana štruktúra mohla byť použitá ako alokátor, a aby podporovala svoje kompletné dealokovanie.

Lexikálne označenie regiónu

Lexikálne označenie regiónu jednoznačne identifikuje región v mennom priestore.

Lexikálne označovanie regiónov sa používa pri preklade konštruktov na vykonanie statickej analýzy, ktorá kontroluje správnosť manipulácie s dátami s ohľadom na ich umiestnenie v regiónoch.

2.4.3 Rozšírenie návrhu jazyka o regióny

Rozšírenie jazyka o regióny spočíva v zakomponovaní regiónov do konštruktov, teda do syntaxe a sémantiky celého jazyka.

Špecifikácia regiónov v typoch

Referencie objektov a polí sú rozšírené o vlastnosť označujúcu región, v ktorom sú alokované odkazované dáta (objekt, pole).

Zápis typov s regiónmi:

```
i64[] @A
i64[] is equal to i64[] @CURRENT_DEFAULT

X @A
X is equal to X @CURRENT_DEFAULT

X@A[] @B
X[] @B is equal to X@CURRENT_DEFAULT[] @B
```

```
X@A[] is equal to X@A[] @CURRENT_DEFAULT
X[] is equal to X@CURRENT_DEFAULT[] @CURRENT_DEFAULT
```

Označenie regiónu v typoch poskytuje informácie o umiestnení referovaných dát, a má lexikálny význam. Tieto informácie sú využité iba pre statickú analýzu na kontrolu zhody typov pri predávaní hodnôt, napríklad hodnota priradenia, argumenty volanie. Do natívného kódu sa neprenesú informácie o regiónoch v typoch.

Vytvorenie nového regiónu

Vytvorenie nového regiónu je možné pomocou príkazu 'on', ktorý pozostáva z identifikátora pre názov nového regiónu, z výrazu pre konštrukciu nového regiónu, a z bloku príkazov.

Momentálne jediný validný výraz na konštrukciu regiónu je volanie funkcie 'NewRegion'.

Blok príkazov sa vykoná v novom mennom priestore, ktorý je potomkom aktuálneho menného priestoru, a umožňuje lexikálny aj fyzický prístup k danému regiónu.

Príklad vytvorenia nového regiónu:

```
fn main() {
    on X = NewRegion() {
        let foo : Foo@X = ... ;
    }
}
```

Špecifikácia regiónov vo funkciách

Funkcie budú pravdepodobne často pracovať s dátami minimálne dvoch regiónov. Zvyčajne budú využívať región so vstupnými dátami, a výstup vygenerujú do iného, výstupného, regiónu.

Deklarácia funkcie s regiónmi sa skladá z názvu, a:

regiónov - zoznam regiónov, pre prvý región je automatický predaná handle na fyzický región, ostatné regióny majú lexikálny význam.

parametrov - zoznam parametrov, ich typy môžu používať iba názvy regiónov špecifikované v deklarácii funkcie.

výstupu - výstupný typ môže používať názvy regiónov špecifikované v deklarácii funkcie.

Prvý špecifikovaný región v rozhraní funkcie má význam lexikálny a aj fyzický, pretože je k nemu predané handle na región ako skrytý parameter. Zvyšné špecifikované regióny majú iba lexikálny význam.

Zápis funkcie s viacerými regiónmi vyzerá následovne:

```
fn foo @TMP, @B (i64[] @TMP a, i64[] @B b) -> Result@TMP {
    // ...
    return something;
}
```

Funkcia uvedená v príklade pracuje s regiónmi TMP a B, preberá parametre referujúce na dáta v obidvoch regiónoch a jej výstupom je referencia na objekt typu Result v regióne TMP.

Volanie funkcie vyzerá následovne:

```
fn main() {
    on X = NewRegion() {
        on Y = NewRegion() {
            // ... vytvorenie dat: aaa na regione X ...
            // ... vytvorenie dat: bbb na regione X ...

            let result : Result@X = foo @X,@Y ( aaa, bbb );
        }
    }
}
```

Vytvorený menný priestor musí mať prístup k fyzickému regiónu X, pretože je fukncia foo volaná s prvým regiónom ako X. Vzhľadom na to, že pre volanie funkcie foo vytvára funkcia main menný priestor obsahujúci nové regióny X a Y, a teda počas volania má fyzický prístup k obidvom regiónom.

2.4.4 Definícia viacerých regiónov pre objekty

Objekty môžu byť naviazané na dáta z viacerých regiónov. Napríklad, obdobne ako pri funkciách, objekty môžu referovať na globálne dáta a dáta pre spracovávaný požiadavok.

Použitie viacerých regiónov vyžaduje rozšírenie deklarácie triedy o zoznam regiónov, a prípadne pravidlá pre vzťahy medzi regiónmi.

Vzhľadom na aktuálny rozsah návrhu a implementácie, táto časť jazyka nebola presne navrhnutá ani implementovaná. Návrh si vyžaduje definovanie sémantiky, a implementácia overenie závislosti, a rôzne premapovávania regiónov pri používaní tried s regiónmi.

Momentálne, referencie v triede ukazujú na rovnaký región, ako samotná trieda.

2.4.5 Metódy s regiónmi

Regióny je predávať aj do metód. Regióny sa deklarujú ako zoznam identifikátorov pred zoznamom argumentov:

```
class Foo {
    var length = 0;

    fn onThisLength() -> i32 {
        return length;
    }

    fn onOtherLength @other, @this () -> i32 {
        return length;
    }
}
```

Rovnako ako pri funkciách, prve prvý región je predaná handle na fyzický región, ostatné regióny majú lexikálny význam.

Región this

Metódy pracujú so skrytým parametrom ukazujúcim na aktuálnu inštanciu objektu. Preto deklarácia musí obsahovať región s označením this, v ktorom sa nachádza aktuálna inštancia triedy.

V prípade, že regióny niesu špecifikované, tak this je automaticky prvý a jediný región.

Volanie metód

Volanie metód je rovnaké, ako volanie funkcií, ale predáva sa naviac skrytý parameter ukazujúci na inštanciu triedy. Ak nie sú pri volaní špecifikované regióny, tak sa použije štandardný región v aktuálnom kontexte.

Obmedzením je, že metódy deklarované bez regiónov majú štandardne definovaný región this, a preto handle k regiónu inštancie musí byť dostupná v mieste volania takej metódy. Obísť sa to dá alternatívnym špecifikovaním regiónov, napríklad ako v metóde onOtherLength.

Ukážky rôznych volaní metód:

```
fn test_default (A a, A b) -> i32 {
    return a.onThisLength() + b.onThisLength();
}

fn test_named @FOO (A @FOO a, A @FOO b) -> i32 {
    return a.onThisLength() + b.onThisLength();
}

fn test_two @FOO_A,@FOO_B (A @FOO_A a, A @FOO_B b) -> i32 {
    return a.onThisLength() + b.onOtherLength @FOO_A,@FOO_B ();
}
```

2.4.6 Bezpečnosť práce s regiónmi

Pre dosiahnutie bezpečnosti, je potrebné zabezpečiť, aby žiadna referencia neodkazovala na dáta, ktoré môžu byť dealokované skôr, ako zanikne daná referencia.

Referencie sa môžu vyskytovať ako premenné, vo vlastnosti objektov, a argumenty funkcie.

Validný obsah premenných

Premenná nebude nikdy referovať do regiónu, ktorý nie je dostupný v aktuálnom mennom priestore. Pretože, typ premennej môže používať iba regióny definované v aktuálnom mennom priestore.

Premenná bude obsahovať referenciu do správneho lexikálneho regiónu, pretože pri priradeniach do premennej je vykonaná kontrola typu, pri ktorej sa kontroluje aj zhoda regiónov

O dostupnosť regiónu v konkrétnych menných priestoroch sa postarajú príslušné konštrukty.

Validný obsah vlastností

Keďže triedy nepodporujú viaceré regióny, tak ich typy nemôžu obsahovať iný región, teda referencia ako vlastnosť objektu nebude nikdy ukazovať mimo svoj región.

Validný obsah argumentov volania

Volanie nebude nikdy vykonané na regiónoch, ktoré nie sú dostupné v aktuálnom mennom priestore. Pretože do volania funkcie môžu vstupovať iba regióny, ku ktorým je prístup v danom mennom priestore.

Argumenty budú obsahovať referencie do správnych lexikálnych regiónov. Pretože, pri volaní sa kontrolujú typy predaných argumentov oproti typom parametrov funkcie. Pri tejto kontrole sa premapujú regióny vzhľadom na špecifikované regióny volania. Napríklad, ak definovaná funkcia pracuje s regiónmi A a B, a volá sa s regiónmi X a Y, tak pri kontrole typov argumentov sa v typoch parametrov substitujú regióny A a B za X a Y, a následne sa porovnajú typy premapovaných parametrov funkcie oproti typom predaných argumentov.

Volanie nikdy nebude vykonané na regiónoch, ku ktorým nie je prístup v aktuálnom mennom priestore. Argumenty volania budú obsahovať referencie v správnom lexikálnom regióne.

Rovnako ako pri premenných, bezpečný prístup k regiónom dostupné v mennom priestore definujú príslušné konštrukty.

Dostupnosť regiónov pri vykonávaní funkcie

Menný priestor funkcie obsahuje iba validné regióny, ktoré su dostupné počas celého vykonávania. Pretože, tieto regióny sú predané funkcii pri volaní, a volanie predáva iba validné regióny.

Toto pravidlo sa rekurzívne viaže na pravidlo pre bezpečnosť volania, ktoré sa odvoláva na aktuálny menný priestor. Počiatkom tejto rekurzie je volanie funkcie main, ktorá je volaná pomocou runtime jazyka a je jej predaný globálny región.

Dostupnosť regiónov pri vykonávaní funkcie

Pre vykonávanie metód platia rovnaké pravidlá ako pre vykonávanie funkcií. Rozdiel je len v tom, že metóda povinne obsahuje región this.

Dostupnosť nového regiónu v príkaze 'on'

Príkaz 'on' pridáva vytvorí nový menný priestor ako potom aktuálneho, a bude obsahovať novo vytvorený región.

Novo pridaný región je dostupný počas celého vykonávania bloku príkazov pri konštrukte 'on'. Pretože, tento región bol práve vytvorený, a konštrukt ho dealokuje až pri opustení bloku príkazov. Región nie je možné dealokovať iným spôsobom, ani manuálne, ani automaticky.

Kapitola 3

Implementácia jazyka

3.1 Architektúra implementácie

3.1.1 Komponenty

Implementácia riešenia pozostáva z:

Prekladača pre preklad zdrojového kódu do natívneho kódu,

Runtime knižnice implementujucej funkcie používané pre sémantické akcie,

Základnej knižnice implementujucej funkcie a triedy použitelné v programe.

Prekladač

Prekladač implementujeme ako frontend pre existujúci backend. Úlohou frontendu je:

- spracovanie zdrojovového kódu z textovej podoby do syntaktického stromu,
- preklad syntaktického stromu do medzikódu pre backend (LLVM IR).

Backend (LLVM) preloží medzikód do natívneho kódu.

3.1.2 Knižnice a nástroje pre implementáciu

Pre implementáciu riešenia sú použité existujúce riešenia 3-tích strán. Je využitý prekladač LLVM, build systém cmake, parser generator lemon, a lexer generator re2c.

LLVM Backend

Ako backend som si zvolil LLVM, pretože je ho možné použiť ako knižnicu, a má menej restriktívnu knižnicu.

Implementáciu frontendu pre GCC spočíva v zakomponovaní frontendu do samotných zdrojov GCC, a to výrazne komplikuje vývoj a správu zdrojových kódov.

Lexer a Parser

Keďže spracovanie textového kódu nepredstavuje veľkú časť práce, a nástroje sa výrazne nelíšia v množstve potrebného kódu na dosiahnutie účelu, tak som nástroje na implementáciu parsovania textu nevolil na základe výhod v používaní.

Zvolil som si lemon a re2c, pretože som chcel vyskúšať nové, menej používané, nástroje.

CMake

Nástroj CMake som si zvolil kvôli jednoduchosti použitia a pre jednoduchosť tvorby konfigurácie projektu - 'CmakeLists.txt'.

Použitie CMake vyžaduje jeden príkaz, ktorý vygeneruje klasické Makefile, a následne je možné projekt skompilovať s nástrojom 'make'.

3.2 Vývoj

3.2.1 Kompilácia v Ubuntu 16.04

Na vývoj jazyka som používal Ubuntu 16.10 a neskôr Ubuntu 17.04. Tieto vydania majú sú podporavné iba 9 mesiacov. Preto, pre popísanie dokumentácie som si zvolil vydanie Ubuntu 16.04, ktoré ma 3 ročnú podporu.

Projekt využíva LLVM verzie 3.9, do Ubuntu 16.04 pridáme repozitáre pomocou následujúcich príkazov:

```
wget -O - http://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add -
sudo add-apt-repository \
     'deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-3.9 main'
```

Následne aktualizujeme informácie o baličkoch z internetových zdrojov, a nainštalujeme potrebné balíčky:

```
sudo apt update
sudo apt install -y gcc g++ git cmake clang-3.9 llvm-3.9-dev \
    llvm-3.9-runtime re2c lemon zlib1g-dev
```

Následne stiahneme projekt:

```
git clone https://github.com/kravemir/elism.git && cd elism
```

Ubuntu 16.04 ponúka starú verziu nástroja lemon, preto je potrebné nainštalovať novšiu verziu. Lemon z SVN je možné lokálne nainštalovať pomocou predpripraveného skriptu:

```
bash scripts/install_local_lemon.sh
export PATH="${HOME}/.local/bin:$PATH"
```

Následne by už malo byť celé prostredie pripravené, a projekt je možné skompilovať a spustiť pomocou:

```
mkdir -p build
sh -c 'cd build; cmake ..; make -j4'
sh scripts/tests/test_fine_errors.sh
```

3.2.2 Automatické testovanie

Pri vývoji boli využité viaceré formy automatického testovania. Využitie automatického testovania si žiada určitý čas na prípravu materiálu a konfigurácii pre testovacie nástroje.

V dlhodobom meradle, automatické testovanie prináša zrýchlenie testovania kódu, pomáha zachovaniu spätnej kompatibility, pomáha taktiež stálosti verejného rozhrania, a zároveň zaručuje určitú kvalitu implementácie.

Skripty, testovacie sady

Automatické testovanie je založené na skompilovaní a spustení programu, a následne overení návratovej hodnoty procesu a porovnaní výstupu s očakávaným výstupom.

Testovanie obsahuje 2 sady programov s výstupmi:

- sada ukážkami a testami, ktorá obsahuje validné programy
- kontrolná sada, ktorá obsahuje chybné programy pre kontrolu overovania sémantických chýb

Automatické testovanie je implementované pomocou shell skriptu, ktorý prechádza vstupy v testovacích sadách, a overuje správnosť výstupu programov, respektíve správnosť chybovej hlášky kompilátora pri kontrolnej sade.

TravisCI

TravisCI je online nástroj pre priebežnú integráciu. TravisCI je integrovaný s GitHub-om a pre jeho využitie je potrebné vytvoriť konfiguračný skript a repozitár s projektom pridať do TravisCI.

Konfigurácia je uložená v '.travis.yml', a využíva novú infraštruktúru založenú na kontajneroch, ktorá sa vyznačuje veľmi krátkou dobou štartu, ale vykonávanie nemá root práva.

Konfigurácia sa skladá z následujúcich častí:

addons so zoznamom potrebných repozitárov a balíkov

install so zoznamom príkazov pre inštaláciu potrebných nástrojov

before_script so zoznamom zoznam príkazov pre prípravu prostredia na kompiláciu

script so zoznamom príkazov pre kompiláciu a automatické testovanie projektu

after_success so zoznamom príkazov pre ďalšie info po úspešnom skompilovaní a otestovaní projektu

Konfigurácia tohto projektu pre TravisCI obsahuje podobné príkazy, ako by sme použili na lokálny build.

Rozdielom pri TravisCI je, že ten beží na Ubuntu 14.04 LTS Server Edition 64 bit, ktorý obsahuje zastaralú verziu lemon generátora. Preto, lemon je skompilovaný a inštalovaný lokálne pod aktuálnym užívateľom v sekcii install.

3.3 Syntaktický strom, spracovanie zdrojového kódu

3.3.1 Syntaktický strom

Syntaktický strom reprezentuje program vo forme stromu. Každý uzol syntaktického reprezentuje určitý konštrukt jazyka, a uchováva presné hodnoty konštánt.

Syntaktický strom je implementovaný ako množstvo tried, kde každá trieda odráža nejaký konštrukt jazyka.

Základné typy konštruktov (základy polymorfických tried):

výraz (ExpressionNode) - reprezentuje výraz pre získanie alebo výpočet hodnoty,

príkaz (StatementNode) - reprezentuje príkaz, operáciu,

funkcia (FunctionNode) - reprezentuje funkcie so všetkým jej obsahom,

trieda (ClassNode) - reprezentuje triedu so všetkým jej obsahom.

3.3.2 Lexikálna analýza

Lexikálna analýza spracuje program v textovej forme, a jej výstupom sú tokeny reprezentujúce jednotlivé prvky kódu. Tokeny môžu mať iba identifikačný význam (priradenie, trieda, zátvorka,...), alebo môžu obsahovať aj hodnotu (text identifikátora, text reťazca,...).

Lexer je implementovaný v súbore src/lexer.re. Pravidlá pre lexikálnu analýzu su uložené v špeciálne označenom komentári, ktorý re2c spracuje a nahradí vygenerovaným kódom.

Zoznam možných typov tokenov je vygenerovaný automaticky z nástroja na tvorbu parsera, ktorý je popísaný v ďalšej sekcii.

3.3.3 Syntaktická analýza - parsovanie

Syntaktická analýza je vykonávaná parserom, a jej úlohou je spracovať tok tokenov do syntaktického stromu.

Parser je implementovaný v súbore src/parser.yy, a jeho kód je vygenerovaný nástrojom lemon. Parser pozostáva z hlavičky pre include, a zoznamu pravidiel.

3.4 Preklad syntaktického stromu do LLVM IR

3.4.1 Základné info o LLVM IR

V krátkosti by som LLVM IR nazval ako high-level jazyk s low-level sémantikou. LLVM IR obsahuje:

základné dátové typy celočíselné, číselné s desatinnou čiarkou, ukazatele.

konštrukty typov pre definovanie vlastných štruktúrovaných typov.

inštrukcie podobné bežným CPU inštrukciám, ale majú prevažne charakter operácie so vstom a výstupom (mimo skokov), ich poskládanie tvorí orientovaný dataflow graf.

bloky zlučujúce inštrukcie do súvislých sekvencií pre vykonávanie, zakončených na skok, alebo return.

funkcie skladajúce sa z blokov, a parametrov.

Tvorba LLVM IR sa mierne podobá písaniu v jazyku C, kde programátor pracujeme priamo s dátami, a programovací jazyk automaticky nepridáva žiadne operácie. V LLVM IR neexistujú ani zložené výrazy, a konštrukty pre cykly. Všetko je potrebné "rozbiť" do jednoduchých inštrukcií, a blokov s inštrukciami so skokmi medzi danými blokmi.

3.4.2 Informácie počas prekladu do LLVM IR

Konštrukty vytvárajú nové funkcie, triedy, premenné a hodnoty, a môžu pracovať aj s aktuálne vytvorenými prvkami.

Výstupom prekladu konštruktov je LLVM IR kód, teda inštrukcie, dátové typy a funkcie. Inštrukcie majú správny význam, iba ak sú uložené v správnom poradí a na správnom mieste.

Kontext generácie LLVM IR kódu

Pri preklade je preto dostupná referencia na aktuálny kontext generácie IR kódu, ktorý uchováva informácie potrebné pre preklad konštruktu do LLVM IR, a umožňuje konštruktom pridávať nové informácie do kontextu.

Kontext generácie kódu je implementovaný ako polymorfická C++ trieda 'CodegenContext', ktorá uchováva tieto informácie vo forme 'std::map', alebo ako jednoduché vlastnosti, alebo ako ukazateľov na určité objekty.

Dcérsky kontext

Určité konštrukty obsahujú vnorené konštrukty, ktoré môžu deklarovať nové veci do kontextu, ale deklarácie vnorených konštruktov majú iba rozsah platnosti. Napríklad, premenné deklarované v tele cyklov môžu byť využívané iba v danom cykle.

Dcérsky kontext generácie je preto špecifický 'CodegenContext' s obdnobnou funkcionalitou, ale pri prístupe k informáciám poskytuje prístup aj informáciám materského kontextu.

Po zániku dcérskeho kontextu zanikne prístup k informáciám deklarovaných v dcérskom kontexte, a informácie v materskom kontexte ostanú neporušené.

3.4.3 Preklad výrazov

Výrazy sú konštrukty, ktoré vytvárajú novú hodnotu. Ich prekladom je postupnosť LLVM IR inštrukcií, ktorá reprezentuje jednotlivé operácie na vyhodnotenie výrazu.

Preklad výrazov nemení informácie v aktuálnom kontexte, ale môže ho využívať pre získanie vstupných hodnôt operácie. Vstupnou hodnotou môže byť aj výsledok vnoreného výrazu.

Samotné výrazy nemenia dáta priamo, ale môžu ich meniť nepriamo, napríklad volanie funkcií.

Aritmetické výrazy

Aritmetický výraz sa skladá z dvoch vnorených výrazoch a operandu. Pri preklade aritmetického výrazu sa najprv preložia vnorené výrazy, a potom sa vygenerujú inštrukcie pre danú aritmetickú operáciu.

Aritmetické výrazy pracujú nad číselným dátovým typom, a preto obsahujú overenie, či vnorené výrazy sú aritmetického dátového typu.

Výstupná hodnota je číselná hodnota rovnakého typu ako typ vnorených výrazov.

"Premenné" výrazy

"Premenný" výraz obsahuje názov premennej, a získava hodnotu podľa informácií daného kontextu. Výstupom tohto výrazu je hodnota, ktorá prislúcha názvu premennej v aktuálnom kontexte.

Volanie funkcie

Volanie funkcie je navrhnuté univerzálnejšie tak, že sa skladá z výrazu pre získanie hodnoty reprezentujúcej funkciu pre volanie. Ďalej, obsahuje zoznam výrazov reprezentujúcich argumenty funkcie.

Najprv preloží výraz pre získanie funkcie, následne sa preložia výrazy pre argumenty. Následne, sa o vytvorenie LLVM IR inštrukcií pre volanie funkcie sa postará polymorfická metóda typu získanej hodnoty s funkciou, do ktorého sa predajú hodnoty argumentov.

Pri preklade je vykonané statické overenie zhody typov predaných argumentov s typmi parametrov funkcie. Pri kontrole typov sú regióny porovnávané na základe mapovaciej tabuľky medzi názvami regiónov volania a názvami regiónov pri deklarácii.

Výsledkom výrazu je hodnota, ktorej typ je definovaný návratovým typom z typu volanej funkcie/hodnoty.

Získanie vlastnosti

Získanie vlastnosti hodnoty sa skladá z výrazu pre získanie hodnoty, a názvu vlastnosti. Konštrukt je univerzálny, a umožňuje vracať vlastnosti tried, metódy tried, a vlastnosti vstavaného typu pre pole.

Najprv sa preloží základný výraz, a o získanie vlastnosti z danej hodnoty sa postará polymorfická metóda typu hodnoty základného výrazu.

Výsledkom výrazu je hodnota, ktorej typ je závislý na type základnej hodnoty a názvu vlastnosti.

3.4.4 Preklad jednoduchých príkazov

Výrazy samostatne nemajú zmysel, pretože ich výsledkom je hodnota. Jedine volanie funkcie, respektíve jej vykonávanie, môže zmeniť stav dát.

Príkazy sú konštrukty, ktoré slúžia na zmenu dát. Príkazy využívajú výrazy, ktorým pripisujú rôzne významy pre vyhodnotenia, napríklad hodnota na uloženie do premennej, hodnota riadiaca opakovanie cyklu,...

"Výrazový" príkaz

Tento príkaz obsahuje vnorený výraz. A jeho preklad je tvorený iba prekladom výrazu.

Použitie tohto príkazu je typické v kombinácii s výrazom volania funkcie, ktorá mení dáta.

Priradenie

Priradenie je obsahuje názov premennej na uloženie hodnoty, a výraz pre výpočet hodnoty na uloženie.

Preklad tohto príkazu je tvorený prekladom výrazu, získaním adresy cieľového priradenia, a inštrukciou pre uloženie výsledku výrazu do danej pamäte. Preklad priradenia nemení informácie v kontexte prekladu.

Pri preklade sa kontroluje, či cieľová premenná, do ktorej sa má hodnota priradiť, má rovnaký typ, ako je typ výsledku výrazu. Kontrola typu zároveň kontroluje aj regióny, keďže regióny sú vlastnosťou typu premennej.

Definícia hodnoty

Definícia hodnoty obsahuje lexikálny názov pre hodnotu, a výraz pre výpočet hodnoty.

Pri preklade sa preloží výraz, a výsledok výrazu je priradený k danému názvu v aktuálnom kontexte.

Vytvorená hodnota je hodnota, ktorú je možné ďalej použiť. Nie je to však premenná, pretože nemá alokovaný priestor na zásobníku.

Deklarácia premennej

Definícia premennej obsahuje lexikálny názov pre premennú, typ premennej a výraz pre výpočet hodnoty.

Pri preklade sa na začiatku funkcie vytvorí LLVM IR inštrukcia pre alokáciu priestoru na zásobníku. Následne sa preloží výraz, a vytvorí sa inštrukcia pre vloženie výslednej hodnoty do alokovanej pamäte.

Pri preklade je vykonané statické overenie, či typ výsledku výrazu je zhodný s typom premennej.

V kontexte prekladu sa vytvorí priradenie názvu k danej premennej.

3.4.5 Preklad zložených príkazov

Zložené príkazy pozostávajú z viacerých vnorených príkazov, a iných výrazov, alebo ďalších prvkov.

Zložené výrazý zvyčajne definujú nový menný priestor pre vnorené príkazy. A preto ich preklad využíva dcérsky kontext.

Podmienky

Podmienka je definovaná výrazom pre riadenie toku, príkazom na vykonanie pri pravdivosti výrazu, a nepovinne príkazom na vykonanie pri nepravdivosti výrazu.

Preklad vytvorí dve, alebo tri, nové bloky:

- blok s obsahom, do ktorého sa preloží príkaz. Blok končí skokom do bloku pre pokračovanie.
- blok "inak", do ktorého sa preloží príkaz pri nepravdivosti podmienky. Blok je vytvorený iba, ak má podmienka definovaný daný príkaz. Blok končí skokom do bloku pre pokračovanie.
- blok pre pokračovanie, je to blok pre pokračovanie generovania ostatných príkazov.

Výstupná sekvencia LLVM IR inštrukcií začína výrazom a zakončí aktuálny blok skokom do príslušného bloku. Ak je podmienka vyhodnotená ako pravdivá, tak skok skočí do bloku s obsahom. Ak je nepravdivá, tak do bloku "inak", alebo do bloku s pokračovaním, ak blok "inak" nie je definovaný. Následne sú uložené bloky v uvedenom poradí. Po preklade je v kontexte nastavený blok pre pokračovanie ako výstupný blok prekladu.

Cyklus while

Cyklus while pozostáva z výrazu pre opakovanie a príkazu na opakované vykonávanie, pripadne zloženého príkazu. Preklad vytvorí tri nové bloky:

- blok pre podmienku, do ktorého sa preloží výraz podmienky, blok je zakončený skokom do bloku s obsahom, ak je podmienka vyhodnotená ako pravdivá, inak skočí do bloku s pokračovaním.
- blok s obsahom, do ktorého sa preloží príkaz, blok končím skokom do bloku podmienky pre prípadné opakovanie cyklu.
- blok pre pokračovanie, je to blok pre pokračovanie generácie ostatných príkazov.

Výstupná sekvencia LLVM IR inštrukcií začína skokom do podmienky, následne sú uložené bloky v uvedenom poradí. Po preklade je v kontexte nastavený blok pre pokračovanie ako výstupný blok prekladu.

Cyklus for

Cyklus for pozostáva z názvu premennej pre aktuálny prvok, výrazu pre získanie iterátora, a príkazu na opakované vykonávanie. Preklad vytvorí tri nové bloky:

- blok pre iteráciu, do ktorého sa vygeneruje overenie prítomnosti ďalšieho prvku, blok je zakončený skokom do bloku s obsahom, ak je ďalší prvok prítomný, inak skočí do bloku s pokračovaním.
- blok s obsahom, do ktorého sa vygeneruje kód pre získanie ďalšieho prvku, a preloží sa do neho príkaz, blok končím skokom do bloku iterácie pre prípadné opakovanie cyklu.
- blok pre pokračovanie, je to blok pre pokračovanie generácie ostatných príkazov.

Výstupná sekvencia LLVM IR inštrukcií začína vyhodnotením výrazu pre získanie iterátora, následne pokračuje skokom do bloku s iteráciou, následne sú uložené bloky v uvedenom poradí. Po preklade je v kontexte nastavený blok pre pokračovanie ako výstupný blok prekladu.

On region

Príkaz pozostáva z názvu nového regiónu, výrazu pre vytvorenie nového regiónu, a príkazu (bloku príkazov). Blok príkazu má lexikálny charakter, a nevytvára nové LLVM IR bloky.

Pri preklade sa vyhodnotí výraz na získanie z regiónu. Vytvorí sa dcérsky kontext prekladu, a výsledok výrazu sa priradí k názvu v dcérskom kontexte. Následne sa s dcérskym kontextom preloží obsah vnorený príkaz.

3.4.6 Preklad deklarácií tried a funkcií

Ideálna implementácia prekladu by pozostávala z viacerých iterácii prekladu konštruktov, a to:

1. prejsť typy, nastaviť ich priradenie názvom v kontexte prekladu.
2. prejsť funkcie, nastaviť ich priradenie k názvom v kontexte prekladu.
3. prejsť deklarácie typov, definovať rozhrania typov - metódy a vlastnosti.
4. prejsť implementácie typov a funkcií, vygenerovať spustiteľný kód.

Aktuálna implementácia prototypu prejde prejde všetky konštrukty postupne, a to iba raz. Nevýhodou je nemožnosť použitia funkcií a typov, ktoré neboli deklarované nad aktuálnou pozíciou prekladu.

Funkcie

Funkcie sú definované názvom funkcie, názvami a typmi parametrov, výstupným typom, a blokom príkazov.

Preklad funkcie vytvorí novú LLVM IR funkciu, preloží typy parametrov, a definuje typ funkcie. V aktuálnom kontexte priradí funkciu k názvu. Následne sa preloží celý obsah funkcie.

Triedy

Trieda je definovaná názvom, vlastnostiami a metódami.

Pri preklade sa najprv vytvorí LLVM IR kód pre konstruktor a typ štruktúry. Následne sa preložia metódy.

Metódy tried

Metódy sú definované názvom metódy, názvami a typmi parametrov, výstupným typom, a blokom príkazov.

Preklad metódy vytvorí novú LLVM IR funkciu, preloží typy parametrov, a definuje typ funkcie. V aktuálnom kontexte priradí funkciu k názvu. Následne sa preloží celý obsah funkcie.

Pri preklade metódy sa pridáva skrytý parameter obsahujúci ukazateľ na aktuálny objekt.

3.4.7 Typy, operácie nad typmi

Pri preklade do LLVM IR je potrebné sledovať viac vlastností typov, než sú dostupné v LLVM IR typoch.

Typy obsahujú polymorfickú funkciu pre získanie hodnoty z inštancie typu (triedy, poľa) pomocou názvu vlastnosti. Definícia typu štruktúry v LLVM IR obsahuje iba zoznam prvkov.

Typy taktiež uchovávajú informáciu o regióne, v ktorom majú alokované dáta.

3.5 Runtime knižnica

Runtime knižnica obsahuje implementácie vnútorných funkcií jazyka. Tieto funkcie sú používané na vykonávanie zložitejších operácií rôznych sémantických akcií.

Runtime knižnica je implementovaná statická knižnica, v jazyku C. Momentálne obsahuje iba funkcie na prácu s regiónmi, a alokáciu objektov.

Knižnica je prilinkovaná k kompilátoru pre dostupnosť funkcií pri JIT spustení programu.

Pri kompilácii programu ako samostatného objektu, je potrebné k nemu nalinkovať túto knižnicu.

3.6 Štandardná knižnica

Štandardná knižnica obsahuje bežne používané triedy, a je implementovaná vo vlastnom jazyku.

Pri implementácii prototypu je zdrojový kód štandardnej knižnice zakomponovaný do kompilátora. Tento zdrojový kód je automatický preložený pred prekladom vstupného súboru.

Momentálne štandardná knižnica obsahuje iba implementáciu iterátora, ktorý generuje konečnú jednoduchú postupnosť čísel.

Vzhľadom na nízku veľkosť knižnice, opakované kompilovanie štandardnej knižnice nemá viditeľný vplyv na dobu kompilácie. V prípade rozsiahlejšej knižnice by knižnica mala byť oddelená od kompilátora, a to by zároveň vyžadovalo výraznejšie rozšírenie funkčnosti kompilátora a jazyka o podporu rozdelenia kódu do viacerých súborov a knižníc.

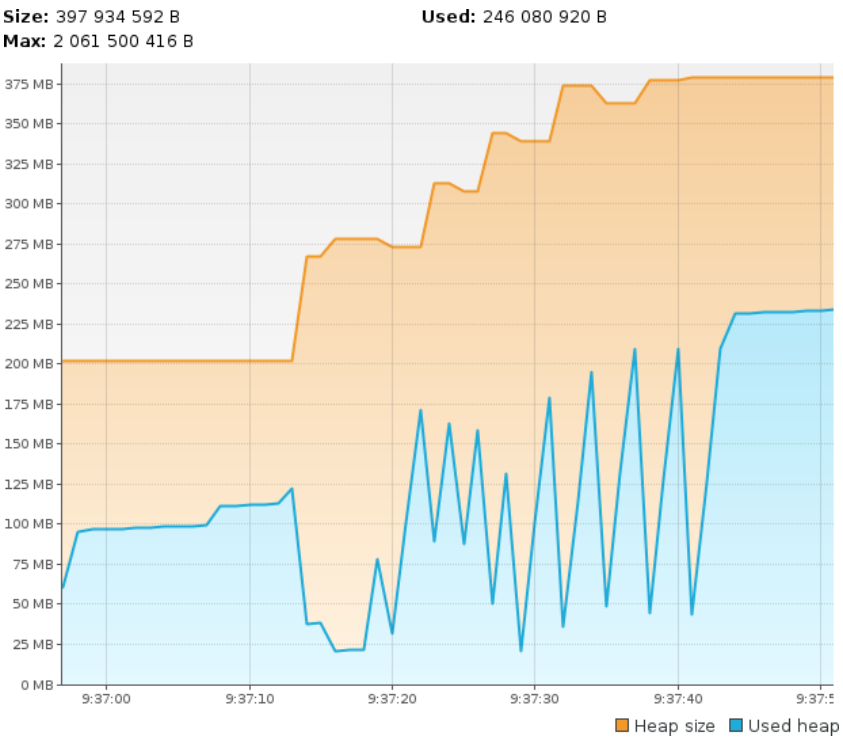
Kapitola 4

Testovanie

4.1 Spotreba pamäte Spring aplikácie

Primárnou oblastou využitia jazyka sú webové aplikácie. Typickou činnosťou je jednorázové spracovanie dát vo forme požiadavkov.

Následujúci graf zobrazuje využitie pamäte malej aplikácie založenej na Springu. Graf obsahuje štart aplikácie, 2000 sériových požiadavok s rovnakým vstupom na vygenerovanie SVG obrázku, a nečinnú aplikáciu po dobehnutí testu:



Pre štart aplikácie bolo alokovaných 200MB heap pamäte a využitých bolo najviac 125 MB pamäte. Po štarte bol spustený GC, a ostalo využitých 25MB. Reálna potreba pamäte pre štart je niekde medzi 25MB až 125MB.

Počas sériového spracovania cca 200 požiadavok došlo iba 9x k spusteniu zberu pamäte. Čoskoro by asi došlo k ďalšiemu zberu pamäte, takže uvažujeme 10 zberov pamäte pre 2000 požiadavok, teda 1 zber každých 200 požiadavkov.

Všetky požiadavky generovali rovnaký SVG obrázok a mali rovnaký vstup. Vygenerovanie odpovede záležalo iba zadanom vstupe, a nebola pri spracovaní sa nepoužívala databáza, ani neboli prítomné žiadne cache. Takže, všetky požiadavky pre svoj beh spotrebovali rovnaké množstvo pamäte.

Rozdiel medzi minimom a maximom využitej pamäte počas spracovania požiadavkov je cca 150 MB. Ak by pri zbere po každých cca 200 požiadavkov došlo k uvoľneniu všetkej nepotrebnej pamäte, tak na spracovanie jednej požiadavky bolo potrebných menej ako 1MB pamäte naviac. Ak by bol zber pamäte neefektívny, a zanechával by v pamäti viac dát, než je potrebných, tak by reálna spotreba pamäte na požiadavku bola menšia, pretože rozdiel využitia by sme delili väčším číslom, ktoré by reprezentovalo dáta väčšieho počtu požiadavkov.

Minimálna potrebná pamäť pre riadiace štruktúry aplikácie je 25MB, spracovania jednej požiadavky vyžaduje 1MB pamäte (s rezervou, 5MB), teda reálne aplikácia potrebuje 30MB pamäte pre spracovanie sériových požiadavkov, a ak by spracovávala 15 požiadavkov paralelne, tak potrebuje 40MB (bez rezervy) až 100MB (s rezervou, 5MB/požiadavok).

V operačnom systéme má Java alokovaných 375MB pamäte, čo je 4 až 10 krát viac, než je potrebné.

4.2 Spôsob testovania

4.2.1 Meranie spotreby pamäte natívnej aplikácie

Na meranie spotreby pamäte natívnych aplikácií som použil profilovací nástroj massif z frameworku valgrind.

Pre vytvorenie čo najpodrobnejšieho grafu je potrebné zvýšiť maximálny počet snapshotov nástroja massif pomocou parametra –max-snapshots.

Celý príkaz pre spustenie aplikácie s profilovaním vyzerá následovne:

```
valgrind --tool=massif --max-snapshots=1000 ./build/knapsack-test
```

Nástroj massif vytvorí výstup s množstvom informácií. Z tohto výstupu som vytvoril graf pomocou nástroja massif-visualizer.

4.2.2 Meranie spotreby pamäte Java aplikácie

Java aplikácie bežia vo virtuálnom stroji - Java Virtual Machine. Pri testovaní som sa zameral na spotrebu pamäte dátami aplikácie, a na to som zvolil profilovaciu aplikáciu pre JVM s názvom VisualVM.

Pri profilovaní pomocou JVM ma zaujímal graf využitia pamäte v sekcii monitorovania aplikácie.

Keďže profilovacie nástroje JVM sa zvyčajne pripájajú až po štarte aplikácie, tak bolo potrebné spustiť aplikácie pomocou pluginu Startup Profiler, ktorý umožní spustiť profilovanie aplikácie okamžite pri štarte aplikácie.

4.3 Knapsack tester

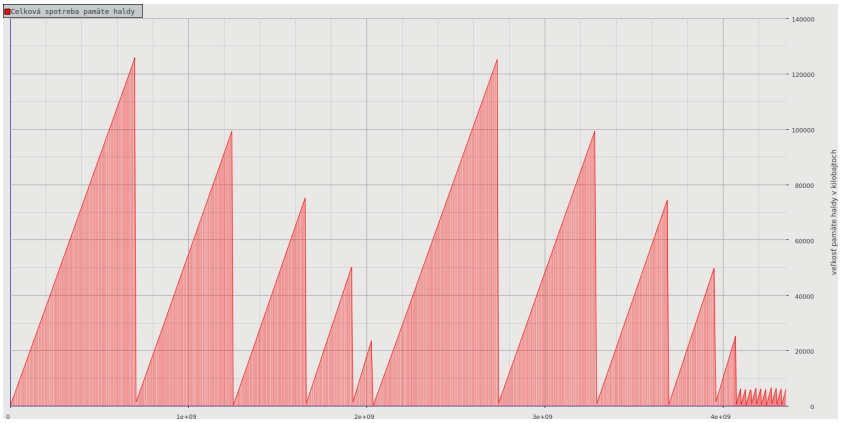
Pri tvorbe testovacej aplikácie som chcel napodobniť beh aplikácie, kde sú nezávisle spracovávané vstupy. Zvolil som si algoritmus Knapsack, pretože je jednoduchý na implementáciu a jeho zložitosť narastá kvadraticky so vstupom, resp. ako násobok počtu prvkov a maximálnej váhy v batohu.

Pri implementácii algoritmu som sa nezameriaval na kvalitu implementácie pre efektivitu, pretože webové aplikácie sa zameriavajú na kvalitu kódu po stránke čitateľnosti a na nízku dobu implementácie.

Vstupné dáta boli pseudo-náhodne generované s počtom prvkov 4000. Na začiatku bolo dva-krát päť vstupov s maximálnou váhou 4000, 3200, 2400, 1600, 800. Po nich nasledovalo 10 vstupov s maximálnou váhou 200.

4.3.1 Knapsack v elism

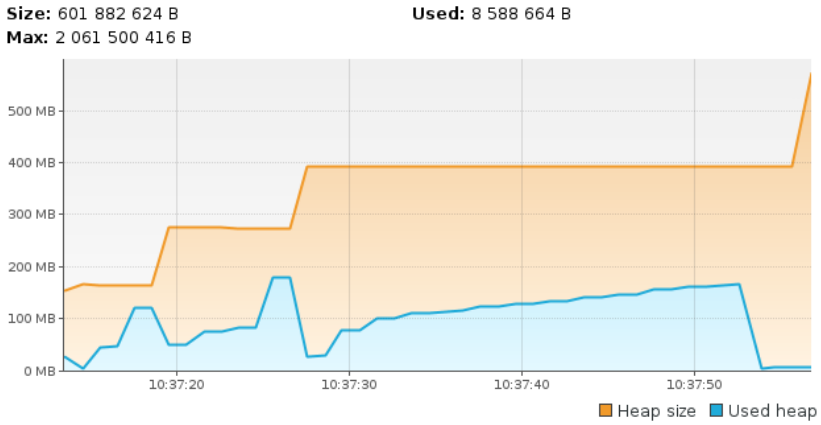
Spotreba pamäte aplikácie sa správala podľa očakávania. Po dokončení spracovania každého vstupu sa uvoľnila využitá pamäť:



Maximálna spotreba pamäte bola približne 126MB (pravdepodobne JEDEC jednotky), čo odpovedá potrebnej pamäte pre spracovanie 4000 predmetov s maximálnou váhou 4000 a veľkosťou dátového typu 8B (64bit), teda 4000*4000*8B = cca 122MiB (128MB).

4.3.2 Knapsack v Java

Keďže VisualVM sledoval spotrebu pamäte v dosť dlhých intervaloch. Tak medzi každým testom bola pridaná 2s pauza. Spotreba pamäte algoritmu v Java sa správala následovne:



Maximálne využitie pamäte bolo cca 175MB, čo je približne o 40% viac, ako je potrebné. Taktiež, spotreba pamäte narastala aj pri spracovaní malých vstupov, na úroveň oveľa vyššie, než je potrebná.

4.3.3 Porovnanie

Maximálna potrebná pamäť pre beh algoritmu je približne 125 MiB. Maximálne využitie pamäte v aplikácii elism bolo rovné maximálnemu potrebnému. Maximálne využitie pamäte v Java bolo o 40% vyššie, ako potrebné.

Spotreba pamäte pri malých testovacích vstupoch bola v elism maximálna potrebná. Využitie pamäte pri malých vstupoch v Java bolo niekoľko násobne vyššie.

Java pre dáta využívala oveľa väčšie množstvo pamäte, ako bolo potrebné. Do porovnania nebolo zahrnuté využitie pamäte pre virtuálny stroj, respektíve runtime knižnicu.

Záver

Zhrnutie

V rámci práce bola vykonaná analýza existujúcich jazykov využívajúcich správu pamäte založenú na regiónoch.

Po analýze bol navrhnutý vlastný jazyk s jednoduchším modelom správy pamäte založenej na regiónoch. K navrhnutému jazyku bol implementovaný prototyp prekladača ako predná časť prekladača LLVM.

Správanie využitia pamäte implementovaného jazyka bolo otestované na testovacom programe. Následne bolo porovnané využívanie pamäte oproti obdobnému programu v jazyku Java.

Výsledky testu využívania pamäte implementovaného jazyka dopadli podľa očakávania.

Dosiahnuté vlastnosti

Bezpečnosť správy pamäte bola dosiahnutá zakomponovaním regiónov do sémantiky jazyka, a správnosť použitia je overená statickou analýzou počas kompilácie programu.

Použitie navrhnutej správy pamäte je jednoduchšie v porovnaní s manuálnou správou pamäte, pretože nevyžaduje od programátora presné určenie stavu, pri ktorom je možne dealokovať dáta. Narozdiel od plne automatickej správy pamäte je použitie zložitejšie, pretože využitie regiónov je potrebné definovať v programe pri typoch a funkciách.

Oproti plne automatickej správe pamäte dosahuje navrhnutá správa pamäte lepšie výsledky pri nezávislom spracovaní sérii požiadavkov, respektíve vstupných dát.

Literatúra

[1] Wikipedia: *ParaSail (programming language) — Wikipedia, The Free Encyclopedia [online]*. 2016, [cit. 09. 05. 2017]. Dostupné z: https://en.wikipedia.org/wiki/ParaSail_(programming_language)

[2] Taft, T.: A Pointer-Free Path to Object-Oriented Parallel Programming. *Designing ParaSail, a new programming language [online]*, 2012, [cit. 09. 05. 2017]. Dostupné z: http://parasail-programming-language.blogspot.cz/2012/08/a-pointer-free-path-to-object-oriented.html

[3] *RTSJ 1.0.2 [online]*. [cit. 09. 05. 2017]. Dostupné z: https://www.aicas.com/rtsj/Version_1_0_2/

[4] Rust Project Developers: *Cyclone: Pointers with Restricted Aliasing [online]*. [cit. 09. 05. 2017]. Dostupné z: http://cyclone.thelanguage.org/wiki/Introduction%20to%20Regions/

[5] Rust Project Developers: *Cyclone: Pointers with Restricted Aliasing [online]*. [cit. 09. 05. 2017]. Dostupné z: http://cyclone.thelanguage.org/wiki/Pointers%20with%20Restricted%20Aliasing/

[6] Rust Project Developers: *References and Borrowing [online]*. [cit. 09. 05. 2017]. Dostupné z: https://doc.rust-lang.org/book/references-and-borrowing.html

[7] Rust Project Developers: *Lifetimes [online]*. [cit. 09. 05. 2017]. Dostupné z: https://doc.rust-lang.org/book/lifetimes.html

[8] Rust Project Developers: *arena::TypedArena - Rust [online]*. [cit. 09. 05. 2017]. Dostupné z: https://doc.rust-lang.org/1.1.0/arena/struct.TypedArena.html

[9] Rust Project Developers: *lib.rs.html – source [online]*. [cit. 09. 05. 2017]. Dostupné z: `https://doc.rust-lang.org/1.1.0/src/arena/lib.rs.html`

Dodatok **A**

Zoznam použitých skratiek

ASCII – American standard code for information interchange

GC – Garbage collection, garbage collector

IR – Intermediate representation

JSR – Java specification request

OOP – Object oriented programming

RTJS – Real time Java specification

Dodatok B

Obsah priloženého CD

```
README.md ..... stručný popis obsahu CD
bin ..... adresár so spustiteľnou formou implementácie
impl ..... adresár s projektom implementácie
    examples ..... adresár s ukážkovými programami
    scripts ..... adresár so skriptami pre vývoj a testovanie
    src ..... adresár so zdrojovými kódmi implementácie
text ..... text práce
    praca.pdf ..... text práce vo formáte PDF
    praca.ps ..... text práce vo formáte PS
    src ..... zdrojová forma práce vo formáte LaTeX
```