VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE VIRTUÁLNÍHO STROJE SYSTÉMU STRONGTALK

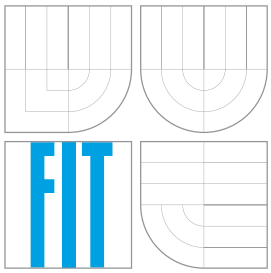
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ĽUBOŠ SITARČÍK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE VIRTUÁLNÍHO STROJE SYSTÉMU STRONGTALK

VIRTUAL MACHINE IMPLEMENTATION OF STRONGTALK SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ĽUBOŠ SITARČÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2013

Abstrakt

Cílem této bakalářské práce je vytvořit vlastní virtuální stroj na základě již existujícího virtuálního stroje systému Strongtalk, respektive upravit tento virtuální stroj napsaný v jazyce MS C++ tak, aby byl přenositelný mezi operačními systémy Windows a GNU-Linux. Práce popisuje programovací jazyk a systém Smalltalk-80, který implementuje systém Strongtalk, obsahuje informace o tom, co to je virtuální stroj a nakonec popisuje samotnou implementaci virtuálního stroje, jeho testování a jeho nedostatky, na které by bylo třeba se v budoucnu zaměřit.

Abstract

The goal of this bachelor's thesis is to develop own virtual machine based on existing system Strongtalk virtual machine, respectively modify the virtual machine written in the language MS C++ so that it would be portable between operating systems Windows and GNU-Linux. Thesis describes a programming language and a system Smalltalk-80, which is implement by Strongtalk, thesis contains informations about what is a virtual machine and then describes the actual implementation of virtual machine, its testing and its shortcomings, which should be focused in the future.

Klíčová slova

virtuální stroj, Smalltalk, Strongtalk, GNU-Linux, Windows

Keywords

virtual machine, Smalltalk, Strongtalk, GNU-Linux, Windows

Citace

Ľuboš Sitarčík: Implementace virtuálního stroje systému Strongtalk, bakalářská práce, Brno, FIT VUT v Brně, 2013

Implementace virtuálního stroje systému Strongtalk

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ľuboš Sitarčík
10. května 2013

Poděkování

Rád bych poděkoval panu Ing. Radku Kočímu Ph.D. za trpělivé vedení práce.

Obsah

Zoznam tabuliek

Kapitola 1

Úvod

Nasledujúci dokument popisuje bakalársku prácu „Implementácia virtuálneho stroja systému Strongtalk“ v akademickom roku 2012/2013 na Fakulte Informačných Technológií Vysokého Učenia Technického v Brne.

Dokument popisuje implementáciu virtuálneho stroja systému Strongtalk, ktorý je implementáciou programovacieho jazyka a systému Smalltalk-80. Úlohou práce bolo na základe už existujúceho virtuálneho stroja napísaného v MS C++ vytvoriť vlastný virtuálny stroj, respektíve prepísať pôvodný virtuálny stroj tak, aby bol prenositeľný medzi operačnými systémami Windows a GNU-Linux.

V kapitole 2 dokument popisuje základné znaky, vznik a históriu programovacieho jazyka Smalltalk-80. Systém Strongtalk prevzal syntax a sémantiku programovacieho jazyka Smalltalk-80 a mení pohľad na tento programovací jazyk. Implementuje ho iným, vlastným spôsobom za účelom zvýšenia výkonu a rýchlosti prevádzania kódu napísaného v tomto programocom jazyku. Samotný systém Strongtalk je napísaný pomocou jazyka Smalltalk a je z neho vytvorený *image*, teda binárny súbor, ktorý je načítaný a prevedený virtuálnym strojom. Potom je možné pracovať s jednotlivými objektmi systému. Popisom samotného systému Strongtalk a jeho virtuálneho stroja sa zaoberá kapitola 4.

V kapitole 3 dokument uvedie čitateľa do problematiky virtuálnych strojov. Objasní, čo to je virtuálny stroj, ukáže typy virtuánych strojov, popíše *garbage collector* a algoritmy, ktoré garbage collector používa.

Dokument v kapitole 5 ukazuje vlastnú implementáciu virtuálneho stroja. Postup práce, preklad pod operačným systémom Windows a GNU-Linux. Popisuje všetky utility potrebné k správnemu prekladu a chodu systému. Vysledný systém však nie je plne funkčný, obsahuje niekoľko nedostatkov, ktoré budú ďalej popísané.

Nakoniec sa v kapitole 6 rieši testovanie systému, ktoré bolo obmedzené vzhľadom na nedostatky systému.

Kapitola 2

Smalltalk

Smalltalk je objektovo orientovaný, dynamicky typovaný, reflektívny programovací jazyk. Smalltalk bol vytvorený ako jazyk pre podporu „nového sveta“ počítačov, z časti pre výukové účely a hlavne pre konstrukcionizmus[1] (*constructionist learning*) v *Learning Research Group* (LRG) v Xerox PARC. Na vývoji sa podieľali hlavne Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace a iný v sedemdesiatych rokoch dvadsiateho storočia[6].

Programovací jazyk bol najprv vydaný ako Smalltalk-80. Programovacie jazyky vychádzajúce z jazyka Smalltalk pokračujú v aktívnom vývoji aj dnes a existujú okolo nich komunity užívateľov. ANSI Smalltalk bol ratifikovaný v roku 1998 a predstavuje štandard jazyku Smalltalk[1].

2.1 História

Existuje množstvo variácií jazyka Smalltalk. Slovo „Smalltalk“ sa najčastejšie spája s programovacím jazykom Smalltalk-80, ktorého prvá verzia bola zverejnená pre verejnosť v roku 1980.

Smalltalk bol produkt výskumu Alana Kaya vo výskumnom centre Xerox Palo Alto (*Xerox Palo Alto Research Center*, ďalej len PARC). Alan Kay vytvoril väčšinu prvej verzie jazyka Smalltalk, ktorú implementoval Dan Ingalls. Prvá verzia, známa ako Smalltalk-71, bola vytvorená Ingallsom počas niekoľkých dní na základe stávky, že programovací jazyk založený na myšlienke zasielania správ inšpirovanej programovacím jazykom Simula môže byť implementovaný v *„page of code“*. Neskoršia varianta, používané pre výskum, známa ako Smalltalk-72 ovplyvnila vývoj Aktorového modelu[2] (*Actor model*) [11].

Po výrazných revíziách, ktoré odstránili niektoré aspekty prevádzania kódu, k zvýšeniu výkonu (osvojením modelu dedičnosti tried na základe jazyka Simula), bol vytvorený Smalltalk-76. Systém mal vývojové prostredie obsahujúce veľké množstvo do dnes používaných nástrojov, vrátane prehliadača/editora kódu, knižnice tried. Smalltalk-80 pridal metatriedy ako pomoc pre udržanie paradigmatu „Všetko je objekt“ (okrem privátnych inštančných premenných) tým, že združil vlastnosti a správanie jednotlivých tried a dokonca

[1] Konstrukcionizmus vychádza z toho, že učenie je zvlášť efektivne, ak pri ňom tvoríme niečo pre ostatných. Môže to byť čokoľvek, od hovorenej vety alebo zdieľania informácie na internete až po složitějšie výtvory, akými sú obraz, dom alebo softwarový produkt.

[2] Aktorový model výpočtu je varianta objektovo orientovaného správania systému, kde dochádza k delegovaniu správ medzi objektami.

aj primitív ako celočíselné a logické hodnoty (napr. podpora rôznych spôsobov vytvárania inštancií) [6].

Smalltalk-80 bola prvá varianta jazyka Smalltalk, ktorá bola prístupná mimo PARC, najprv ako Smalltalk-80 Verzia 1, predaná malému počtu firiem (Hewlett-Packard, Apple Computer, Tektronix, DEC) a univerzít (UC Berkley) na „seberovné preskúmanie“ a implementáciu na ich platformách. Neskôr (v roku 1983), všeobecne dostupná implementácia, známa ako Smalltalk-80 Verzia 2, bola vydaná ako image (súbor s definíciou objektov nezávislí na platforme) a špecifikácia virtuálneho stroja[6].

Momentálne populárne Smalltalk implementácie sú potomkovia originálneho Smalltalk-80 imageu. *Squeak*[7] je open source implementácia odvodená zo Smalltalk-80 verzie 1 podľa Apple Smalltalku. *VisualWorks* je odvodená zo Smalltalk-80 verzie 2 podľa Smalltalk-80 2.5 a ObjectWorks (obidva sú produkty ParcPlace Systems, firmy pre dodanie Smalltalku na trh založenej organizáciou Xerox PARC). *Smalltalk/X* je produktom firmy *eXept Software AG*.

Koncom osemdesiatych rokov až do rokov deväťdesiatych, boli Smalltalk prostredia, včetne zákazníckej podpory, školení a doplnkov, predávané dvomi organizáciami: *ParcPlace Systems* a *Digitalk*, obidve z Kalifornie. ParcPlace Systems sa sústreďovala skôr na trh Unix/Sun Microsystems, zatiaľ čo Digitalk sa sústreďovala na Osobné počítače založené na platforme Intel s operačným systémom Microsoft Windows alebo OS/2 firmy IBM. Zatiaľ čo vysoká cena ParcPlace Smalltalku limitovala vniknutie na trh len do stredne veľkých a veľkých komerčných organizácií, produkty Digitalku sa snažili osloviť širšiu verejnosť s nízkou cenou. IBM spočiatku podporovala produkt Digitalku, ale potom sama vstúpila na trh v roku 1995 so Smalltalk produktom nazvaným *VisualAge/Smalltalk* [9].

V roku 1995 sa ParcPlace a Digitalk spojili do *ParcPlace-Digitalk* a potom v roku 1997 sa premenovali na *ObjectShare*, nachádzajúcom sa v Irvine, Kalifornii. ObjectShare verejne obchodoval do roku 1999, kedy bol vyradený a rozpustený. Spojená firma nikdy neprišla na efektívnu odpoveď na Javu na trhu, a od roku 1997 hľadali majitelia spôsob, akým predať tento biznis. V roku 1999 Segaul Software získali vývojové laboratórium Java firmy ObjectShare (včetne vývojových tímov Smalltalk/V a Visual Smalltalk), a stále vlastní VisualSmalltalk, hoci svetové distribučné práva pre Smalltalk produkty ostali vo vlastníctve ObjectShare, ktorý ich predal firme Cincom. VisualWorks bol predaný firme Cincom a je súčasťou Cincom Smalltalku.

Cincom, Gemstone a Object Arts a ďalší predajcovia pokračujú v predaji Smalltalk prostredí. IBM ukončilo na konci deväťdesiatych rokov VisualAge a rozhodlo sa vrátiť k Jave. Open-source implementácia Squeak má aktívnu komunitu vývojárov, včetne mnohých z pôvodnej komunity Smalltalk vývojárov.

2.2 Vplyvy Smalltalk-u

Smalltalk ovplyvnil širší svet programovania v štyroch hlavných oblastiach. Smalltalk inšpiroval syntax a sémantiku iných počítačových programovacích jazykov. Po druhé, to bol prototyp výpočtového modelu známky ako zasielanie správ. Po tretie, jeho WIMP[3] GUI inšpirovalo prostredie s oknami osobných počítačov na prelome dvadsiateho a dvadsiateho prvého storočia, a to natoľko, že plocha prvého Macintosh-u vyzerá takmer totožne s MVC oknami Smalltalku-80. Nakoniec, integrované prostredie poslúžilo ako model pre generáciu

[3]V počítačovej terminológií skratka WIMP (**W**indow, **I**con, **M**enu, **P**ointing device) označuje ovládanie prostredia za pomoci týchto vymenovaných prvkov.

vizuálnych programovacích nástrojov, ktoré vyzerajú ako prehliadač a debugger Smalltalk kódu.

Programovacie jazyky Python a Ruby reimplementovali niektoré nápady Smalltalku v prostredí podobnom prostrediu AWK alebo Perl. Smalltalk „metamodel“ taktiež slúži ako inšpirácia pri návrhu objektového modelu *Perl 6*.

Syntax a beh programov programovacieho jazyka *Objective-C* je veľmi silne ovplyvnený Smalltalkom.

Niektoré programovacie jazyky, ako napríklad *self, JavaScript, Newpeak* prevzali mnohé myšlienky Smalltalku a rozšírili ich v nových smeroch.

2.3 Objektovo orientované programovanie

Tak ako v iných objektovo orientovaných programovacích jazykoch, základným konceptom v Smalltalku-80 (nie avšak v Smalltalku-72) je objekt. Objekt je vždy inštanciou triedy. Triedy sú návrhy, ktoré popisujú vlastnosti a správanie ich inštancií. Napríklad, trieda okna grafického prostredia môže hovoriť, že toto okno má vlastnosti ako názov okna, pozícia okna, a či je okno viditeľné alebo nie. Trieda môže ďalej popisovať, že inštancia tejto triedy podporuje operácie ako otváranie, zatváranie, presun a skrývanie okna. Každý objekt, inštancia tejto triedy, teda každé okno bude mať vlastné hodnoty týchto vlastností a každé z nich bude schopné previesť operácie definované ich triedou [11].

Smalltalk objekty dokážu robiť presne tri veci:

- Udržovať stav (referencie na iné objekty).
- Obdržať správu od seba alebo od iného objektu.
- Pri spracovaní správy, poslať správu sebe alebo inému objektu.

Stav objektu sa udržuje vždy v privátnej časti tohto objektu. Ostatné objekty môžu získať alebo zmeniť stav tohto objektu len zaslaním požiadavku (správy) tomuto objektu, aby takto učinil. Akákoľvek správa môže byť odoslaná akémukoľvek objektu: Keď je správa prijatá, prijímateľ určí, či je táto správa zodpovedajúca.

Smalltalk je čisto objektovo orientovaný programovací jazyk, to znamená, že narozdiel od napríklad Javy a C++, neexistuje rozdiel medzi hodnotami, ktoré sú objektami a hodnotami, ktoré sú primitívnymi typmi. V Smalltalku, primitívne typy, ako celočíselné typy, pravdivostné typy a znakové typy, sú tiež objektmi, v tomto zmysle sú tiež inštanciami zodpovedajúcich tried a operácie nad týmito typmi sú vyvolané zasielaním správ. Programátor môže zmeniť triedy, ktoré implementujú tieto primitívne typy, tak je možné definovať nové správanie ich inštancií.

Keďže všetky hodnoty sú objekty, triedy samotné sú tiež objekty. Každá trieda je inštancia metatriedy tejto triedy. Metatriedy sú v podstate tiež objekty a všetky sú inštanciami triedy nazvanej `Metaclass`, metatrieda.

2.4 Reflekcia

Smalltalk-80 je úplne reflektívny systém, implementovaný v samotnom Smalltalk-80. Smalltalk-80 poskytuje štrukturálnu aj výpočtovú reflekciu. Smalltalk je štrukturálne reflektívny systém, ktorého štruktúra je definovaná v samotnými Smalltalk-80 objektmi. Triedy a metódy, ktoré definujú systém sú samy objektmi a plne súčasťou systému, ktorý pomáhajú

definovať. Smalltalk prekladač prekladá textový zdrojový kód na metódy objektov. Tie sú pridané do tried ich uložením do slovníka metód triedy. Časť hierarchie tried, ktorá definuje triedy, môže pridať nové triedy do systému. Systém je rozšírený počas behu Smalltalk-80 kódu, ktorý vytvára a definuje triedy a metódy. V tomto smere je Smalltalk-80 „živý“ systém so schopnosťou rozšíriť sám seba počas behu [9].

Keďže triedy sú tiež samotnými objektmi, je možné im zasielať správy, ako napríklad „Aké metódy implementuješ?“ alebo „Aké definuješ polia/inštancie“. Takže objekty môžu byť jednoducho preskúmané, skopírované, (de)serializované a pod.

Smalltalk-80 taktiež obsahuje výpočtovú reflekciu, schopnosť pozorovať stav výpočtu v systéme. V jazykoch odvodených z jazyka Smalltalk-80 je aktuálna aktivácia metódy dostupná ako objekt pomenovaný pomocou pseudo-premennej (jedno zo 6tich rezervovaných slov), `thisContext`. Zaslaním správy objektu `thisContext` môže aktivácia metódy položiť otázku ako "Kto mi zaslal túto správu?". Toto správanie umožňuje implementovať Koprogrami[4] (*co-routine*) alebo backtracking podobne ako v *Prologu*. Ďalej je takto implementovaný systém výnimiek.

[4]Koprogramy sú programové komponenty, ktoré umožňujú na rozdiel od podprogramov (procedúr, funkcií, metód) viac vstupných bodov, pozastavenie a obnovenie výpočtu na ich rôznych miestach.

Kapitola 3

Virtuálny stroj

Virtuálny stroj je v informatike softvér, ktorý vytvára virtualizované prostredie medzi platformou počítača a operačným systémom, v ktorom môže koncový užívateľ prevádzať softvér na abstraktnom stroji [3]. Samotný termín „virtuálny stroj“ má niekoľko odlišných významov.

3.1 Hardvérový virtuálny stroj

Pôvodný význam pre virtuálny stroj, tiež nazývaný hardvérový virtuálny stroj, označuje niekoľko totožných pracovných prostredí na jednom počítači, z nich na každom beží operačný systém. Vďaka tomu môže byť aplikácia písaná pre jeden operačný systém na stroji, na ktorom beží OS alebo zaisťuje vykonanie *sandboxu*[1], ktorý poskytuje väčšiu úroveň izolácie medzi procesmi než je dosiahnuté pri vykonávaní niekoľkých procesov naraz (*multitasking*) na tom istom operačnom systéme. Jedným využitím môže byť taktiež poskytnúť ilúziu viacerým užívateľom, že používajú celý počítač, ktorý je ich „súkromným”strojom, izolovaným od ostatných užívateľov aj napriek tomu, že všetci používajú jeden fyzický stroj. Ďalšou výhodou môže byť to, že *bootovanie* a reštart virtuálneho počítača môže by oveľa rýchlejší, než u fyzického stroja, pretože môžu byť preskočené mnohé úlohy, ako je napríklad inicializácia hardvéru.

Podobný softvér je často označovaný ako virtualizácia a virtuálne servery. Hostiteľský softvér, ktorý poskytuje túto schopnosť je označovaný ako *hypervisor* alebo virtuálny strojový monitor (*virtual machine monitor*).

Softvérové virtualizácie môžu byť prevádzané na troch hlavných úrovniach:

- **Emulácia** je plná systémová simulácia alebo „plná virtualizácia“ s dynamickým prestavením (*recompilation*) – virtuálny stroj simuluje kompletný hardvér dovoľujúci prevádzku nemodifikovaného operačného systému na úplne inom procesore.
- **Paravirtualizácia** – virtuálny stroj nesimuluje hardvér, ale namiesto toho ponúkne špeciálne rozhranie API, ktoré vyžaduje modifikáciu operačného systému.
- **Natívna virtualizácia**[2] a „plná virtualizácia“ – virtuálny stroj je čiastočne simuluje

[1] *Sandbox* je označenie pre bezpečnostný mechanizmus v rámci počítačovej bezpečnosti, ktorý slúži k oddeľovaniu procesov bežiacich s rovnakým oprávnením

[2] Pojem *natívna virtualizácia* sa niekedy používa k zdôrazneniu, že je využitá hardvérová podpora pre virtualizáciu.

hardvér aby mohol nemodifikovaný operačný systém bežať samostatne, ale hostiteľský operačný systém musí byť určený pre rovnaký druh procesoru.

3.2 Virtuálne prostredie

Virtuálne prostredie (tiež uvádzané ako virtuálny súkromný server) je iný druh virtuálneho stroja.

V skutočnosti to je virtualizované prostredie pre beh programov na úrovni užívateľa (tj. nie jadra operačného systému a ovládačov, ale aplikácií). Virtuálne prostredie je vytvorené použitím softvéru zavádzajúceho virtualizáciu na úrovni operačného systému ako napríklad *Virtuozzo, FreeBSD Jails, Linux-VServer.*

3.3 Združovanie virtuálnych strojov

Taktiež menej bežný termín „počítačový cluster“, čo je mnoho počítačov združených do veľkého a výkonnejšieho virtuálneho stroja. V tomto prípade softvér vytvára jednotné prostredie fyzicky nachádzajúce sa na viacerých počítačoch tak, že sa koncovému uživateľovi javí, že používa jediný počítač.

3.4 Aplikačný virtuálny stroj

Ďalším významom termínu virtuálny stroj je počítačový softvér, ktorý izoluje aplikácie používané užívateľom na počítači od operačného systému. Pretože virtuálne stroje sú písané pre rôzne počítačové platformy, akákoľvek aplikácia napísaná pre virtuálny stroj môže byť prevádzaná na ktorejkoľvek z platforiem namiesto toho, aby sa museli vytvárať oddelené verzie aplikácii pre každý počítač a operačný systém. Aplikácia bežiaca na počítači používa interpreta alebo *Just in time* kompiláciu.

3.5 Garbage collector

Garbage collector je často súčasťou aplikačného virtuálneho stroja, ktorý ma za úlohu automaticky určiť, ktorá časť pamäte programu je už nepoužívaná a pripraviť ju pre ďalšie znovupoužitie [5].

3.5.1 Vznik garbage collectingu

Garbage collecting je označenie pre metódu automatickej správy pamäte programu. Garbage collecting vymyslel v roku 1959 John McCarthy pre riešenie problému manuálnej správy pamäte v Lispe[3]. Je najčastejšie popisovaná ako opak manuálnej správy pamäte, ktorá vyžaduje špecifikovať program tak, aby bolo zrejmé, ktoré objekty sa môžu uvoľniť a ktoré sa majú vrátiť späť do pamäte.

Miesta v pamäti, ktoré už program použil a ďalej ich program nepoužije sa nazývajú *memory leaks* a Garbage collector tieto miesta hľadá a odstraňuje ich. Ďalším problémom je tzv. *dangling pointer*. Je to ukazateľ na prázdnu pamäť, alebo pamäť, ktorá bola znovu alokovaná inde v programe a prepísaná inými dátami.

Tieto chyby sú ťažko odhaliteľné a zlá podmienka väčsinou spôsobí nesprávne chovanie programu. Takže vyvarovanie sa chýb spôsobených správou pamäte na halde bolo jedným z dôvodov vzniku automatickej správy pamäte.

3.5.2 Základný princíp garbage collectingu

1. Vyhľadajú sa v programe také dátové objekty, ktoré nebudú v budúcnosti použité
2. Vrátia sa pamäťové zdroje, kde sa vyskytovali nájdené objekty.

Uvoľňovanie pamäte pomocou garbage collectoru oslobodzuje programátora od uvoľňovania objektov, ktoré už ďalej nie sú potrebné, čo ho väčšinou stojí značné úsilie. Je to vlastne pomôcka pre stabilnejší program, pretože zabraňuje niektorým prevádzkovým chybám. Napríklad zabraňuje chybám ukazateľov, ktoré ukazujú na už nepoužívaný objekt alebo na objekt, ktorý je už zrušený. [2]

3.5.3 Algoritmus počítania referencií

Vôbec prvý algoritmus pre garbage collector sa nazýval *reference counting* (počítanie referencií). Funguje tak, že ku každému objektu je priradený čítač referencií. Keď je objekt vytvorený, jeho čítač je nastavený na hodnotu 1. v okamžiku, keď si nejaký iný objekt alebo koreň programu (korene sú hľadané v programových registroch, v lokálnych premenných uložených na zásobníkoch jednotlivých vlákien a v statických premenných) uloží referenciu na tento objekt, hodnota čítača je inkrementovaná o 1. Vo chvíli, keď je referencia mimo rozsah platnosti (napr. po opustení funkcie, ktorá si referenciu uložila), alebo keď je referencii priradená nová hodnota, čítač je dekrementovaný o 1. Ak je hodnota čítača u niektorého objektu nulová, môže byť tento objekt uvoľnený z pamäte. Keď je uvoľňovaný z pamäte tak všetkým objektom, na ne má tento objekt referenciu, sa zníži hodnota čítača o 1, to znamená, že uvoľnenie jedného objektu môže viesť k uvoľneniu ďalších objektov.

Nevýhoda tejto metódy spočíva v tom, že nedokáže detekovať cykly. Cyklus nastáva v okamžiku, keď dva a viacej objektov ukazujú samy na seba, napríklad keď rodičovská trieda ukazuje na svojho potomka a ten má referenciu späť na rodiča. tieto dva objekty nebudú mať nikdy čítač rovný nule, hoci sú nedosiahnuteľné z koreňa programu. Ďalšia nevýhoda spočíva v réžii, ktorá je nutná pre inkrementáciu a dekrementáciu čítača u každého objektu. Kvôli nedostatkom sa *reference counting* sa v dnešnej dobe prestal používať.

3.5.4 Sledovacie algoritmy

Sledovacie algoritmy (*tracking algorithms*) zastaví svet (v tomto zmysle beh programu) a začnú vyhľadávať objekty. Začínajú v koreňovej množine programu a pokračujú po referenciách, pokiaľ nepreskúmajú všetky dosiahnuteľné objekty. Algoritmy, založené na tomto princípe sa používajú takmer výlučne pre implementáciu garbage collectorov v dnešných programovacích jazykoch.

Mark & Sweep Algoritmus Mark & Sweep najprv nastaví všetkým objektom, ktoré sú v pamäti, špeciálny príznak *navštívený* na hodnotu *nie*. Potom prejde všetky objekty, ku ktorým sa je možné dostať, tým ktoré navštívil, nastaví príznak na hodnotu *áno*. V okamžiku, keď sa už nemôže dostať k žiadnemu ďalšiemu objektu, znamená to, že všetky objekty s príznakom *navštívený* nastaveným na hodnotu *nie* sú odpad.

Táto metóda má niekoľko nevýhod. Najväčšou je, že pri garbage collectingu je prerušený beh programu. To znamená, že sa programy pravidelne zmrazí.

Kopírovací collector Algoritmus kopírovací collector (*Copying collector*) najprv rozdelí priestor na halde na dve časti, kde jedna je aktívna a s druhou sa nepracuje. Vždy môžeme alokovať objekty v celkovej veľkosti, ktorá je polovičná veľkosť haldy. Pokiaľ sa pri alokácii nevojde do miesta na časti haldy, je potreba previesť garbage collecting. ten spočíva v prehodení aktívnej a neaktívnej časti. Do novo aktívnej časti sa prekopírujú živé objekty zo starej, už neaktívnej, časti. Mítve objekty sa nekopírujú, ale pri ďalšom prehodení aktívnej a neaktívnej časti sa jednoducho prepíšu.

Kopírovací algoritmus prebieha zdĺhavo, pretože objekty sa musia presúvať. Kvôli náročnosti celého presunu môžu teda vzniknúť značné oneskorenia pri behu programu. Výhodou je, že nenastáva žiadna fragmentácia.

3.5.5 Generačný algoritmus

Pri použití garbage collectorov sa dajú empiricky vypozorovať dva dôležité fakty. Prvým faktom je, že mnoho objektov sa stane odpadom krátko po svojom vzniku. Tým druhým je skutočnosť, že len malé percento referencií v „starších“ objektoch ukazuje na objekty mladšie.

U sledovacieho collectoru, kde sa používa celá halda, musel collector pri každom čistení prechádzať medzi objektami a všetky živé objekty buďto prekopírovať do inej časti haldy alebo ich označiť a ďalej prejsť celou haldou a uvoľniť mítve objekty. A práve z dôvodu rýchleho úmrtia väčšiny objektov je táto metóda neefektívna.

Generačný garbage collector využíva týchto skutočností a rozdeľuje pamäť programu do niekoľkých častí, tzv. *generácií*. Obejkty sú vytvárané v spodnej (najmladšej) generácii a po splnení určitej podmienky, obvykle vekom, sú preradené do generácie staršej. Pre každú generáciu môže byť garbage collecting prevádzaný v rozdielnych časových intervaloch (obvykle najkratšie intervaly budú pre najmladšie generácie) a dokonca pre rozdielne generácie môžu byť použité rozdielne algoritmy. V okamžiku, keď sa priestor v spodnej generácii zaplní, všetky dosiahnuteľné objekty v najmladšej generácii sú skopírované do staršej generácie. I tak bude množstvo kopírovaných objektov iba zlomkom z celkového množstva mladších objektov, pretože väčšina z nich sa stane odpadom.

Kapitola 4

Systém Strongtalk

Systém Strongtalk je implementácia založená na pôvodnom systéme Smalltalk-80 a programovacom jazyku Smalltalk, ktorá sa na tento programovací jazyk a systém pozerá novým spôsobom. Popri tom ako zachováva pôvodnú Smalltalk syntax a sémantiku, prináša Strongtalk množstvo zlepšení.

- Systém Strongtalk vykonáva Smalltalk kód oveľa rýchlejšie než akákoľvek iná Smalltalk implementácia. Využíva na to technológiu *type-feedback* pôvodne navrhnutú v *Sun Microsystem Labs*. Smalltalk kód je je dynamicky kompilovaný a dekompilovaný podľa potreby, dokonca aj počas behu. Kód môže byť transparentne debugovaný a menený počas jeho behu (*on-the-fly*).
- Systém Strongtalk obsahuje prvý silný, statický typový systém pre jazyk Smalltalk (odtiaľ meno Strongtalk).

Systém Strongtalk bol pôvodne vyvinutý v utajení v deväťdesiatych rokoch malou začínajúcou firmou. Ale pred tým než mohol byť systém Strongtalk vydaný, firmu získala firma *Sun Microsystems, Inc.* na prácu na *Java® virtual machine*. Vývoj systému Strongtalk bol v tomto bode zastavený. Keďže vývoj prebiehal v utajení, len veľmi málo ľudí malo možnosť vidieť systém Strongtalk v akcii. To je však škoda, pretože Smalltalk je stále veľmi elegantný a pokročilý programovací jazyk.

Našťastie *Sun Microsystems* vydala Strongtalk ako open-source softvér. Prvé vydanie od Sunu v roku 2002 neobsahovalo zdrojové kódy virtuálneho stroja. Bol vydaný len ako binárny súbor, čím sa systém stal prakticky nepoužiteľný mimo výskum. V septembri 2006 sa to zmenilo, keď firma Sun prišla s novým vydaním systému Strongtalk, ktoré už obsahovalo aj zdrojové kódy virtuálneho stroja.

Systém Strongtalk je avšak vyvinutý pre platformu Windows a je neprenositeľný. Ďalej nepodporuje 64 bitové platformy. Dokonca ho nie je možné preložiť ani v prostredí Windows, lebo používa konštrukcie jazyka C++, ktoré sú v rozpore s novými prekladačmi.

4.1 Typový systém

Nasledujúca podkapitola je prebraná z [8] a [4].

Výhody statickej kontroly typov v oblasti vývoja softvéru sú všeobecne uznávané. Systém Strongtalk zavádza statickú kontrolu typov do jazyka Smalltalk spôsobom, ktorý neohrozuje flexibilitu jazyka alebo programovacie prostredie. Strongtalk systém je navrhnutý

tak, aby uľahčil proces vývoja, zlepšil spoľahlivosť a čitateľnosť počas vývoja, údržby a používania.

Kľúčové charakteristiky Strongtalk typového systému sú:

- Typový systém stačí na to, aby bolo umožnené kontrolovať typ prirodzených Smalltalk idiómov. K tomu Strongtalk:
  1. Oddeľuje väzby medzi podtriedou a podtypom.
  2. Vkladá parametrizované triedy a typy.
  3. Podporuje polymorfné správy s flexibilným mechanizmom pre automatické odvodzovanie aktuálnych parametrov typu.
  4. Podporuje zhodu typu aj podtypu.
  5. Zachováva podtypy vzťahov medzi triedami definovanými v hierarchii metatried Smalltalku a vzťahuje tieto podtypy na typy ich inštancií.
  6. Poskytuje zázemie pre dynamické typovanie.
- Typový systém zachováva zapuzdrenie. To znamená, že vnútorné zmeny v triedach nemajú vplyv na ich klientov. To je samo o sebe žiadúca vlastnosť. Ďalšie výhody zahŕňajú:
  - Systém nevyžaduje prístup do vnútra triedy alebo metódy, aby skontroloval ich typ (to sa týka aj dedičnosti). To znamená:
    - Systém môže byť použitý, aj keď je zdrojový kód neprístupný
    - Systém môže byť použitý aj keď kód nemôže byť staticky typovo skontrolovaný.
  - Je oveľa ľahšie vytvoriť reagujúce inkrementálne implementácie, čo je v podstate požiadavka na všetky programovacie prostredia Smalltalk.
- Typový systém je voliteľný. To znamená, že vývojár sa môže rozhodnúť medzi:
  - Ktorý kód chce typovo skontrolovať alebo kód typovo nekontrolovať vôbec.
  - Ak typovo skontrolovať kód, či ho kontrolovať vždy.
  - Kde pridať typové anotácie alebo ich nepridať nikde.

Tu sa nachádza množstvo možností predstavujúcich rôzne náklady a prínosy spojené s rôznymi úrovňami typovej kontroly. Jeden extrém je nikdy typovo nekontrolovať kód a nepoužiť žiadne typové anotácie. Tak ako to je použité v tradičnom Smalltalku-80. Ďalším extrémom je použiť typové anotácie na celý kód a typovo kontrolovať celý kód. To je cesta, ktorú využívajú tradičné staticky typované jazyky. Ani jeden extrém nie je povinný, hlavnou myšlienkou je, že si každý vývojár môže vybrať. Typová kontrola musí byť skutočne voliteľná, dynamické správanie programu Smalltalk nesmie byť ovplyvnené prípadnými typovými anotáciami. To nie je triviálna požiadavka. Mnoho staticky typovaných jazykov má konštrukcie, ktoré porušujú toto obmedzenie. Príklady zahŕňajú implicitné dynamické typové kontroly (Beta, Eiffel) alebo preťažovanie (C++, Java).

4.2 Dizajn virtuálneho stroja

Strongtalk virtuálny stroj je veľký komplexný C++ program. Pre zvládnutie jeho komplexnosti je dizajn virtuálneho stroja založený na vysoko štruktúrovanom použití množstva bázových C++ tried a techník programovania, ktoré je možné vidieť v kóde opakovať sa stále dookola. K porozumeniu dizajnu virtuálneho stroja je nevyhnutné najprv porozumieť bázovým triedam tvoriacich jadro virtuálneho stroja.

Zásadným problémom pri vytváraní vysoko výkonného virtuálneho stroja je konflikt medzi potrebnou abstrakciou, vysokou úrovňou štýlu programovania pre zvládnutie komplexnosti virtuálneho stroja s požiadavkou na rýchly výsledný kód. Virtuálny stroj systému Strongtalk toto splňuje pomocou rozsiahleho využívania C++ inline funkcií, makier pre pre-procesor a optimalizovanej alokácie pre objekty virtuálneho stroja (Vyhýbanie sa použitia haldy vždy, keď je to možné). Zatiaľ, čo inline funkcie a makrá robia ladenie oveľa ťažším a kód menej čitateľným, umožňujú virtuálnemu stroju byť na vysokej úrovni dizajnu a umožňujú použitie abstrakcie bez straty výkonu.

4.2.1 Smalltalk a C++ objekty

Najprv je potrebné vysporiadať sa so základnou dvojznačnosťou: Keď sa povie objekt alebo trieda, hovorí sa o C++ objektoch/triedach v kóde virtuálneho stroja alebo sa hovorí o Smalltalk objektoch, ktoré virtuálny stroj simuluje? Je teda potrebné spraviť si jasno v rozdieloch. Vo virtuálnom stroji systému Strongtalk je vynaložené veľké úsilie na mapovanie Smalltalk objektov do C++ objektov, takým spôsobom, aby potom mohol virtuálny stroj pracovať so simulovanými Smalltalk objektmi prirodzene, objektovo orientovaným spôsobom.

Pri snahe porozumieť štruktúre tried virtuálneho stroja si treba uvedomiť množstvo základných problémov:

Alokácia Smalltalk objekty sú vždy alokované na halde, kde pracuje garbage collector, zatiaľ čo C++ objekty môžu byť alokované kdekoľvek, na C halde, na zásobníku alebo na množstve špeciálnych oblastí používaných vo virtuálnom stroji. Dve najdôležitejšie z týchto oblastí sú:

- **Resource Area (oblasť pre prostriedky):** Dočasné C++ objekty, ktoré nemôžu byť alokované na zásobníku, ale sú potrebné len do konca aktuálnej operácie virtuálneho stroja sú alokované v tejto oblasti, ktorá je kus súvislej pamäte alokovanej pre veľmi rýchlu alokáciu. Keď sa aktuálna operácia virtuálneho stroja dokončí, ukazateľ na koniec alokovanej oblasti je jednoducho obnovený na začiatok, instantne sa uvoľnia všetky prostriedky pre objekty. Avšak je potrebné si pamätať, že pri uvoľnení objektu z tejto oblasti sa nevolá deštruktor tohto objektu. Teda deštruktory nie sú použité pre triedy, ktoré budú alokované v Resource Area.
- **Zone (zóna):** Zóna sa používa na uchovanie preloženého kódu objektov.

Garbage collector Pretože Smalltalk objekty sú riadené pomocou garbage collectoru, ktorý používa algoritmus *Mark & Sweep*, 3.5.4, všetky Smalltalk referencie na Smalltalk objekty musia byť zaznamenané pre garbage collector.

Premiestnenie a Garbage collector Je potrebné dbať na to, aby referencie vo virtuálnom stroji na objekty na halde garbage collectoru neboli spravované naprieč zberu smetí, pretože objekty môžu byť premiestnené a referencie sa môžu stať neplatnými.

Štruktúra Smalltalk objekty majú inú štruktúru a iný formát hlavičky než C++ objekty. Keďže Smalltalk objekty nemajú tabuľku virtuálnych metód (vtable) vo svojich hlavičkách, C++ objekty, ktoré sú mapované na Smalltalk objekty nemôžu obsahovať virtuálne funkcie. Tento problém je priblížený v kapitole 4.3.2.

4.3 Bázové triedy virtuálneho stroja

Problémy popísané v kapitole 4.2.1 hovoria, že C++ bázové triedy vo virtuálnom stroji musia byť rozdelené do dvoch hlavných skupín:

- C++ objekty určené pre prístup do objektov na Smalltalk halde z C++
- Ostatné objekty virtuálneho stroja, ktoré nemapujú Smalltalk objekty

4.3.1 Mapovanie Smalltalk objektov na C++ objekty

V terminológii virtuálneho stroja systému Strongtalk sa používa *Objected Oriented Pointer* (Objektovo orientovaný ukazateľ, ďalej len Oop) ako referencia na Smalltalk objekt. Sú to skutočné Smalltalk referencie na objekty, ktoré tvoria väčšinu dát na Smalltalk halde. Oop nie je možné použiť priamo ako ukazateľ na prístup k štruktúre objektu. Je potrebné najprv získať adresu C++ objektu a ujistiť sa, že sa nejedná o referenciu, ktorá nie je ukazateľom (napríklad `SmallInteger`). Na získanie použiteľného C++ ukazateľa je používaná frekventovaná metóda `addr()`. Kód, ktorý sa stará o Objektovo orientované ukazatele (Oop-s) je možné nájsť v adresári `Strongtalk/vm/oops`, viz. príloha A.

4.3.2 `OopDesc` Hierarchia

Keď je použitá metóda `addr()`, vracia táto metóda ukazateľ na C++ objekt `OopDesc *`. Pre každý typ Oop existuje vhodná trieda, ktorá dedí od triedy `OopDesc`.

Hierarchia `OopDesc` obsahuje C++ triedy, ktoré sú mapované priamo do pamäťového rozloženia Smalltalk objektov. Toto priame mapovanie umožnuje virtuálnemu stroju, aby bol navrhnutý na vysokej úrovni, objektovo orientovaným spôsobom, na rozdiel od mnohých iných virtuálnych strojov, pretože s rôznymi druhmi Smalltalk objektov je možné priamo manipulovať prostredníctvom abstraktného C++ rozhrania.

Avšak toto riešenie má problém. Pretože `OopDesc` je prekrytie skutočnej štruktúry Smalltalk objektov, má `OopDesc` hlavičku Smalltalk objektu, namiesto hlavičky C++ objektu. Teda nie je možné použiť tabuľku virtuálnych metód (vtable) C++ objektov. To znamená, že trieda `OopDesc` a triedy dediace od triedy `OopDesc` nemôžu obsahovať virtuálne metódy.

Vzhľadom k tomu, že absencia virtuálnych metód by striktne obmedzila užitočnosť tejto C++ triedy, je potrebné vytvoriť vhodný *work-around*. Jedným zo zrejmých riešení tohto problému by bolo prideľiť pole vtable každému Smalltalk objektu. To by bolo však neprijateľným plytvaním priestoru na halde, pretože väčšina Smalltalk objektov má lén pár inštančných premenných a hlavička objektu má veľkosť dvoch slov (*word*, t.j. 2 byty). To znamená, že pridanie, čo i len jedného slova na každý objekt, by výrazne zväčšilo veľkosť Smalltalk haldy, ako aj zvýšilo tlak na pamäťový subsystém.

Pre rozumnejšiu alternatívu je potreba si uvedomiť, že Smalltalk objekty už majú triedu, ktorá je zdieľaná naprieč všetkými jej instanciami, takže trieda je zrejmé miesto, kam uložiť C++ vtable, ktoré by mohlo byť použité na delegovanie odosielania inštancií. Avšak Smalltalk triedy sú tiež regulárne Smalltalk objekty, takže majú tiež len Smalltalk hlavičky.

4.3.3 `Klass`: Špeciálny C++ objekt integrovaný v Smalltalk objekte triedy

Strongtalk Virtuálny stroj rieši absenciu vtable integrovaním C++ objektu Klass do štruktúry Smalltalk triedy, ktorá je mapovaná do `KlassOopDesc` v C++. Narozdiel od `OopDesc`, triedy `Klass` majú vtable, a teda hierarchia `Klass` objektov môže obsahovať virtuálne metódy. Keď je potrebné definovať virtuálnu funkciu v triede `OopDesc`, táto je definovaná namiesto toho v asociovanej triede `Klass` a potom je v triede `OopDesc` definovaná inline nevirtuálna funkcia, ktorá vedie k virtuálnej funkcii triedy `Klass`.

Takže tak ako všetky typy Smalltalk objektov, každá Smalltalk trieda je priamo mapovaná na inštanciu triedy, ktorá dedí od triedy `OopDesc`, alebo špecifickejšie, triedy, ktorá dedí od triedy `KlassOopDesc`, ktorá ako všetky triedy `OopDesc` nemá vtable. Ale trieda `KlassOopDesc` obsahuje v sebe integrovanú inštanciu triedy `Klass`, ktorá nie je `OopDesc`, takže môže mať vtable. V podstate `KlassOopDesc` zabaľuje Smalltalk objekty pre integrovaný C++ `Klass` objekt.

V praxi z hľadiska skutočnej štruktúry objektov to znamená, že každá trieda ako Smalltalk objekt má vložené pole C++ vtable bezprostredne za hlavičkou Smalltalk objektu. `KlassOopDesc` a `Klass` pracujú spoločne, spoločne tvoria objekt triedy, ktorý má hlavičku Smalltalk objektu, za ktorú je pridaná C++ vtable hlavička, takže pridaním konštantného off-setu k adrese Smalltalk triedy je možné získať C++ objekt s vtable hlavičkou. To nám umožňuje zaobchádzať s objektmi Smalltalk tried ako so Smalltalk objektmi alebo ako s C++ objektmi, jednoducho a podľa potreby.

Toto avšak prináša ďalší problém. Keďže vtable triedy `Klass` je neoznačená hodnota a nie je to Oop, ale je integrovaná ako pole v každom objekte Smalltalk triedy na Smalltalk halde, musí byť poznaná garbage collectorom, že nie je Oop, takže s ňou nie je omylom zachádzané ako s referenciou na Smalltalk objekt. Toto ilustruje, že väčšina polí v Smalltalk objektoch sú Oop, niektoré vnútorné polia nimi nie sú.

Vo väčšine virtuálnych strojov je potrebné zaviesť ďalšiu komplikovanosť do garbage collectoru, pretože potrebuje poznať všetky detaily o štruktúre všetkých typov Smalltalk objektov, aby vedel ako lokalizovať Oop pri prechode ukazateľmi. Ale keďže existuje, ako bolo ukázané, metóda na nepriamu definíciu virtuálnych funkcií v triede `OopDesc` pre každý typ objektu, môžeme namiesto toho navrhnúť garbage collector objektovo orientovaným spôsobom tak, že každá trieda `OopDesc` môže nepriamo definovať virtuálnu metódu na výpočet vložených Oop pre garbage collector. To robí garbage collector nezávislím na detailnej štruktúre Smalltalk objektov a uľahčuje vnútornú zmenu formátu Smalltalk objektov vo virtuálnom stroji.

4.3.4 Objekty virtuálneho stroja, ktoré nemapujú Smalltalk objekty

Všetky triedy virtuálneho stroja (iné než triedy `OopDesc` hierarchie a súvisiacich tried, ktoré slúžia k mapovaniu na Smalltalk haldu, viz. kapitola 4.3.2 musia byť podtriedami nasledujúcich alokačných tried:

- Pre objekty alokované v Resource Area, 4.2.1

  - `ResourceObj`
    - `PrintableResourceObj`
- Pre objekty alokované na C halde (spravované pomocou free a malloc):
  - `CHeapObj`
    - `PrintableCHeapObj`
- Pre objekty alokované na zásobníku:
  - `StackObj`
    - `PrintableStackObj`
- Pre vložené objekty:
  - `ValueObj`
- Pre triedy použité ako menné priestory (všetky členy triedu sú statické):
  - `AllStatic`

Odsadenie naznačuje dedenie. Podtriedy s prefixom `Printable` sú používané na ladenie a definíciu virtuálnych funkcií pre tlač.

Kapitola 5

Implementácia virtuálneho stroja

Úlohou práce bolo upraviť zdrojové kódy virtuálneho stroja systému Strongtalk tak, aby bolo možné používať virtuálny stroj systému Strongtalk v operačných systémoch GNU-Linux a Windows. V prvom rade bolo ale potrebné preložiť systém Strongtalk v prostredí Windows, čo ukázalo prvé problémy.

Samotný systém Strongtalk bol navrhnutý pre systém Windows 95. Pôvodný systém, ktorý vydala firma Sun ako open source nebolo možné preložiť na vtedajších systémoch Windows XP. Po úpravách je možné zo stránky `www.strongtalk.org` stiahnuť verziu 2.0 systému Strongtalk, ktorá je určená pre otvorenie a preklad vo VisualStudio 2005. Pri otvorení vo VisualStudio 2010, ktoré mám k dispozícii k štúdijným účelom preklad prebehol neúspešne. Teda samotný systém určený pre systém Windows je nefunkčný. To je spôsobené rozdielmi v prekladači verzie z roku 2005 a 2010.

Pre samotný preklad bolo ešte potrebné použiť utilitu `Makedeps` pre vytvorenie hlavičkových súborov.

5.1 Makedeps

Utilita Makedeps slúži k vytvoreniu hlavičkových súborov pre jednotlivé súbory so zdrojovým kódom. Tieto vytvorené hlavičkové súbory obsahujú len direktívy `#include` s hlavičkovými súbormi pre daný súbor so zdrojovým kódom.

Utilita Makedeps je software firmy Sun Microsystems, Inc. Utilita prečíta databázu hlavičkových súborov. Táto databáza obsahuje dvojice neprázdnych slov (slov bez medzier), kde prvé slovo je názov súboru, ktorý potrebuje, aby do neho bol vložený súbor pomocou direktívy `#include` ako druhé slovo z dvojice. Tieto dvojice sú oddelené znakmi nového riadku, pri čom medzi slovami, ale aj dvojicami, môže byť ľubovoľný počet prázdnych znakov. Pre každý súbor `*.cpp` je tak vytvorený hlavičkový súbor `_*.cpp.incl`. Aplikácia pri tom vytvára tieto súbory tak, aby hlavičkové súbory boli vložené v správnom poradí a detekuje cykly.

Databáza hlavičkový súborov je uložená v súboroch `Strongtalk/vm/deps/includeDB` a `Strongtalk/vm/deps/includeDB2`, pri čom výsledné hlavičkové súbory sú uložené do adresára `Strongtalk/build/incls`. Adresárovú štruktúru je možné nájsť v prílohe A.

5.2 Postup práce

Prvým krokom bolo upraviť zdrojové kódy tak, aby boli preložiteľné v prostredí VisualStudio 2010.

Toto obnášalo úpravu triedy pre výpis na štandardný výstup a štandardný chybový výstup, ktorá používala slovo `std`, ktoré je definované v C++ ako priestor mien. Celú triedu bolo potrebné prepísať a kvôli potrebe používať slovo `std` vo virtuálnom stroji bez ďalších zmien, bol priestor mien `std` obídený pomocou makra, ktoré nahrádza priestor mien `std` tokom pre tlač na štandardný výstup.

Ďalším problémom bolo chýbajúce kľúčové slovo `class` pri definícií spriatelených tried. Namiesto definície `friend class <trieda>;` obsahovali jednotlivé definície spriatelených tried len definíciu `friend <trieda>;`. Toto bolo potrebné dopísať v celom systéme. Úloha to bola však jednoduchá, pretože sa dala zvládnuť vytvorením jednoduchého skriptu, ktorý za pomoci regulárneho výrazu doplnil chýbajúce slovo `class` všade tam, kde to bolo potreba.

Opäť jednoduchý problém bola definícia čisto virtuálnych metód (*pure virtual methods*), ktoré boli deklarované pomocou `NULL`. Pre preklad však bolo potrebné zmeniť `<metóda> = NULL` na `<metóda> = 0`. Táto úloha sa tiež dala zvládnuť pomocou jednoduchého skriptu.

Ďalším problémom bolo veľmi často sa opakujúce použitie nedefinovanej triedy, teda problémy s hlavičkovými súbormi a utilitou `Makedeps` 5.1, kde bolo potrebné upraviť databázu hlavičkových súborov a doplniť chýbajúce dvojice.

Pri spustení pod systémom Windows sa ukázali prvé problémy, viz kapitola 5.5. Systém neprešiel všetkými testami, ale len niektorými. Tento problém ostal neriešený do doby, než bude samotný virtuálny stroj prenositeľný medzi systémami Windows a GNU-Linux.

Ďalším krokom bol prenos virtuálneho stroja na operačný systém GNU-Linux. Prvou úlohou bolo vytvoriť funkčný `makefile` a preložiť utilitu Makedeps. Utilita makedeps však bola preložiteľná aj v operačnom systéme GNU-Linux, takže nebolo potrebné ju upravovať.

Potom bolo potrebné upraviť vložený asembler z Intel kódu na AT&T kód[10] v kóde všade tam, kde bol vložený assembler použitý.

V niektorých častiach kódu sa namiesto vloženého asembleru používa knižnica `libnasm`. Túto knižnicu sa podarilo nájsť pre systém Windows aj GNU-Linux, avšak bez verejných zdrojových kódov a bez akejkoľvek dokumentácie.

Ďalším krokom bolo upraviť všetky použité WINAPI. Keďže v samotnom zdrojom kóde existoval hlavičkový súbor `Strongtalk/vm/runtime/os.hpp`, s triedou `os`, ktorý mal zdrojový súbor `os.cpp`, ktorý využíval služby operačného systému Windows, bolo potrebné premenovať súbor `os.cpp` na `os_nt.cpp` a vytvoriť súbor `os_linux.cpp`, ktorý bude implementovať metódy triedy `os` pre systém GNU-Linux. Pri čom bolo potrebné vyriešiť tri základné problémy:

- konvencie volania procedúry
- manažment vlákien
- rozdieli v operačných systémoch mimo triedu `os`

Posledným krokom by bolo vytvorenie funkčného GUI. To však už nebolo mojou tejto bakalárskej práce.

Konvencie volania procedúry Toto sa týka najmä vloženého asembleru v kóde. Primitívne volania procedúr sa vo Windowse a Linuxe nelíšia, pretože volaný a volajúci sa môžu dohodnúť na vzájomnej konvencii. Rozdiel nastáva pri dynamických knižniciach. Windows

používa `stdcall`, pri čom Linux knižnice používajú všeobecne `cdecl`. Podpora volania procedúr knižníc funguje za predpokladu, že všetky konvencie sú `stdcall`. Tento problém je obídený tak, že je vytvorený pomocný extra zásobník, ktorý sa stará o nastavenie registrov `ESP` a `EBP` na pôvodné hodnoty.

Manažment vlákien Vlákna sú spracované pomocou knižnice `pthread`. ID vlákien sú zabalené a operácie nad nimi v triede `Thread`, ktorá je privátna trieda[1] v súbore `os_linux.cpp`. Prístup k tejto metóde má len spriatelená trieda `os`. Pri tom trieda `Thread` využíva triedu `Event` na správu jednotlivých vlákien, paralélny prístup a podobne.

Rozdieli v operačných systémoch mimo triedu `os` Všetok zdrojový kód, ktorý nebol napísaný vo vloženom asemblery a bol preložiteľný v systéme Windows, avšak nie v systéme GNU-Linux a nenachádzal sa v triede `os`, sa presnul do triedy `os`, všade tam, kde to bolo možné, pre lepšiu čitateľnosť zrojového kódu. Miesto bývalého použita tohto kódu bolo nahradené volaním statickej metódy triedy `os`. Jednalo sa o:

- Spacovanie parametrov.
- Spracovanie výnimiek operačného systému.
- Použitie funkcií `malloc`, `calloc` a `free` pre malé alokácie.

5.3 Preklad Virtuálneho stroja

Samotný virtuálny stroj je možné preložiť pod systémom Windows alebo GNU-Linux. Výsledný súbor je vždy 32 bitová aplikácia. Aj v prípade, že sa jedná o preklad na 64 bitovej platforme.

Nazvime koreňový adresár systému Strongtalk ako `$STRTLK` v systéme Windows aj GNU-Linux, kde `$STRTLK` je absolútna cesta k tomu koreňovému adresáru.

5.3.1 Preklad pod systémom Windows s použitím VS 2010

1. Najprv je potrebné zostaviť utilitu `makedeps`. Otvoríme adresár `$STRTLK\tools\makedeps`. Vo VisualStudiu otvoríme súbor pre projekt `makedeps`, `makedeps.vcproj`. "Build Solution"z menu. Toto by malo vytvoriť spustiteľný súbor `$STRTLK\tools\makedeps.exe`

2. Potom je potrebné použiť aplikáciu makedeps na vytvorenie súboru závislostí vložených súborov. Z Príkazového riadku: `cd` do adresára `$STRTLK\bin`. A spustiť `namke -f Makefile.win32 lists`.

3. Ďalej otvoríme pomocou VisualStudia projekt systému Strongtalk `$STRTLK\build.win32\strongtalk.sln`.

4. V prostredí VisualStudia zvolíme "Build → Batch Build". Označíme konfiguráciu projektu, ktorú chceme. To znamená jednu z Debug, Fast alebo Product (je potrebné ignorovať možnosť Release) a stlačíme "Rebuild All".

5. Môžme odstrániť prebytočné súbory pomocou príkazovej riadky a to v adresári `$STRTLK\bin` pomocou príkazu `‘nmake -f Makefile.win32 clean‘`.

[1] Privátna trieda znamená, že žiadna iná trieda nemá prístup k metódam tejto triedy

5.3.2 Preklad pod systémom GNU-Linux

Preklad po systémom GNU-Linux je veľmi jednoduchý, je potrebné len spustiť `makefile` z adresára `$STRTLK/build`, ktorý vytvorí spustiteľný súbor v koreňovom adresári. Je však potrebné ho spustiť ako `make ROOT_DIR=$STRTLK` alebo upraviť v súbore `$STRTLK/build.linux/makefile-macros.incl` premennú `ROOT_DIR` tak, aby to bola úplná cesta ku koreňovému súboru. Preklad prebehne s množstvom varovaní, no bez chýb.

Keďže `makefile` neobsahuje cieľ `install`, ktorý by prekopíroval potrebné knižnice do adresárov systému GNU-Linux, je potrebné nastaviť systémovú premennú `LD_LIBRARY_PATH`. `export LD_LIBRARY_PATH=$STRTLK/bin`

5.4 Parametre príkazového riadku

Virtuálny stroj možne spustiť s niekoľkými parametrami príkazového riadku a to:

- `-t` vypne používanie časovačov a potom sa ignoruje parameter -f
- `-b <súbor>`, kde `súbor` je image systému Strongtalk, teda súbor `strongtalk.bst`, ktorý sa nachádza v koreňovom adresári. Pokiaľ nie je parameter zadaný, virtuálny stroj hľadá tento súbor vo svojom umiestnení.
- `-f <súbor>` kde `súbor` obsahuje parametre virtuálneho stroja. Parametre sú v tomto súbore zapísané ako:
  - `+<parameter>` nastavenie `bool` parametru na `true`.
  - `-<parameter>` nastavenie `bool` parametru na `false`.
  - `<parameter>=#` kde `#` je celé číslo pre nastavenie celočíselného parametru.
  - `#` znak mriežka predstavuje komentár do konca riadku.

  Jednotlivé parametre virtuálneho stroja sú popísane v prílohe C. Zmena týchto parametrov by mala slúžiť hlavne k debugovacím účelom.
- `-script <súbor>`, kde `súbor` je Smalltalk zdrojový kód, ktorý bude vykonaný Strongtalk systémom.
- `-?` Vypíše s akými parametrami, viz príloha C, bude spustený virtuálny stroj a ukončí virtuálny stroj.

5.5 Problémy a nedostatky výslednej práce

Systém Strongtalk neobsahuje GUI v systéme GNU-Linux, preto je potrebné ho spúšťať s parametrom `-script`, viz. kapitola 5.4.

Samotný komplexný test virtuálneho stroja je problémový, pretože pri vykonávaní testov, niektoré testy skončia s chybou, ktorá však nie je chybou virtuálneho stroja, ale chyba je pravdepodobne v samotnom systéme Strongtalk. Ten je k dispozícií ako image v binárnej forme, súbor `strongtalk.bst`. Čo je bytekód vytvorený zo Smalltalk kódu. Tento Smalltalk kód je v rozsahu takmer 1150 súborov, kde vždy jeden súbor tvorí jednu triedu. Preštudovanie týchto kódov, nájdenie chyby by vyžadovalo ďalšie prostriedky (najmä časové).

Debug virtuálneho stroja ukázal, že chyba nastane niekde vo vloženom asembleri. Kód vloženého asembleru je však prevedený niekoľkokrát bez chyby a až neskôr v ňom nastane

problém. Čo ukazuje na problém v systéme Strongtalk. Lepšie preskúmanie správania vloženého asembleru v tomto mieste kódu je veľmi problematické, pretože tento asembler kód je generovaný pomocou knižnice `libnasm`. Túto knižnicu sa mi podarilo nájsť na ftp serveri a stiahnuť. Bohužiaľ existuje len ako knižnica pre systém Windows, `libnasm.dll` a pre systém GNU-Linux `libnasm.so`. Nie sú však pre túto knižnicu zverejnené zdrojové kódy a neexistuje, respektíve nepodarilo sa mi nájsť žiadnu dokumentáciu k tejto knižnici.

Knižnica `libnasm` vytvára asembler kód pomocou metód triedy `nasm`. Tento kód je uložený vnútorne a je možné ho vyvolať pomocou makra. Preto nie je možné vložiť do funkcie vytvorenej pomocou knižnice `libnasm` debugovací výpis. Ak je vložený do metódy, ktorá vytvára funkciu pomocou knižnice `libnasm`, teda medzi metódy triedy `nasm` z knižnice `libnasm` je tento výpis ignorovaný, pretože funckie sú vytvorené pri inicializácii virtuálneho stroja.

Ďalším znakom toho, že problém je v samotnom systéme Strongtalk, je použitie staršej verzie imageu systému Strongtalk. Pri použití tejto staršej verzie virtuálny stroj skončí s chybovým hlásením, že sa nepodarilo načítať zdieľanú knižnicu `kernel32.dll`, čo je systémová knižnica operačného systému Windows. Teda samotný systém Strongtalk pravdepodobne obsahuje nejaké systémové volania.

Virtuálny stroj pri tom vykazuje rovnaké správanie v systéme GNU-Linux ako v systéme Windows. Pre lepšie spracovanie a pochopenie chyby by bolo potrebné naštudovať a upraviť virtuálny stroj a samotný systém Strongtalk zároveň a tak lepšie pochopiť jednotlivé vzájomné vzťahy medzi systémom a virtuálnym strojom. To by však vyžadovalo značné prostriedky, ktoré sú nad rámec jednej bakalárskej práce.

Kapitola 6

Testovanie

Pre overenie správnej funkčnosti vlastného virtuálneho stroja a pre porovnanie jeho výkonnosti s pôvodným virtuálnym strojom bolo použitých niekoľko základných testov. A to Richards Benchmark, Smopstone Benchmarks, Slopstone Benchmarks, kolekcia testov pod názvom OO Standford Benchmarks a DeltaBlue.

Testovanie prebiehalo v dvoch operačných systémoch, a to v operačnom systéme *Windows XP Professional 32-bit (SP3)*, kde sa testoval pôvodný virtuálny stroj, ktorý je možné stiahnuť zo stránok projektu. Tento virtuálny stroj je stiahnuteľný ako spustiteľný súbor, preložený v prostredí Visual Studio 2005, určený pre systém Windows XP. Vlastný virtuálny stroj bol potom testovaný v operačnom systéme GNU–Linux, v distribúcii *Ubuntu 11.10 (Oneiric Ocelot) i386*.

Testovanie prebiehalo na počítači PC s nasledujúcou konfiguráciou:

- CPU: Intel Core i7 CPU 930 @ 2,80 GHz
- RAM: 6 GB RAM (1600 MHz)
- OS: Ubuntu 12.04 precise x86_64

Kvôli potrebe testovať pôvodný virtuálny stroj na 32 bitovej platforme a hlavne kvôli zjednodušeniu inštalácie systému Windows XP, prebiehalo samotné testovanie na virtualizovaných operačných systémoch so vždy rovnakou konfiguráciou:

- CPU: povolené 1 jadro CPU bez obmedzenia výkonu
- RAM: Vyhradené 2048 MB operačnej pamäte

Pre virtualizáciu boli zapnuté všetky hardware-ové akcelerácie. Operačné systémy boli virtualizované pomocou programu `VirtualBox`.

6.1 Richards Benchmark

Richards simuluje správanie prepínania kontextu (dispatcher) v jadrách operačných systémov. Pôvodný test napísal Martin Richards na Cambridge University v Anglicku. Tento test bol preložený do mnohých jazykov, ako napríklad C, C++, Smalltalk, Java apod.

Jadrom testu je prepínač kontextu. Do jadra prichádzajú 4 rôzne úlohy reprezentované rôznymi triedami, pri čom dedia od jednej spoločnej triedy. Každá úloha obsahuje jednu pracovnú funkciu. Na začiatku testu sa vytvorí náhodný mix úloh, potom sú jednotlivé

úlohy naplánované, každá úloha zavolá svoju pracovnú funckiu. Pri čom sledujeme čas, za ktorý sa test dokončí, teda všetky úlohy budú splnené.

Výsledky testu ukazuje tabuľka 6.1.

	Čas [ms]	
Iterácia	Windows	Linux
1	39	63
2	10	24
3	7	6
4	5	5

Tabuľka 6.1: Výsledky Richards Benchmark

6.2 Smopstone Benchmarks

Smopstone Benchmarks (Smalltalk Medium level OPeration Stones Benchmark) testuje stredné operácie virtuálneho stroja ako sú napríklad rekurzívne bloky, volanie metód, vytváranie kolekcií a zoznamov a prácu s reťazcami. Test konkrétne obsahuje 7 algoritmov. Generovanie Fibonacciho postupnosti (tzv. fractonaccis), generovanie prvočísel, vytváranie malých tokov, vytváranie reťazcov, tvorenie sústav, radenie reťazcov a algoritmus *sorcerer's apprentice*.

Algoritmus *sorcerer's apprentice* testuje efektívnosť rekurzívneho volania blokov a zahrňuje množstvo celočíselných operácií, tvorenia kolekcií a operácií nad týmito kolekciami. Najprv sa vytvorí kolekcia pseudonáhodných obdĺžnikov. Potom sa vytvorí nová kolekcia všetkých priesečníkov týchto obdĺžnikov. Algoritmus rekurzívne pokračuje pokiaľ existujú nejaké priesečníky. Nakoniec vráti počet obdĺžnikov, ktoré majú spoločné priesečníky pre každý obdĺžnik.

Testovanie prebieha v desiatich opakovaniach. Výsledkom testu je čas, za ktorý bol jednotlivý test dokončený a hodnota smopstones, vypočítaná pomocou heuristiky, slúžiaca na ohodnotenie testu. Čím vyššie číslo tým lepšie. Hodnota smopstones slúži pre porovnávanie testov medzi operačnými systémami

Samotné výsledky testu sú však zkresľujúce, pretože test v systéme GNU–Linux nemožno dokončiť, viz. kapitola 5.5. Test vždy skončí s chybou a to pri *Vytváraní malých tokov*. Teda jednotlivé iterácie neprebiehali ako pri testovaní pôvodného virtuálneho stroja, ale medzi jednotlivými iteráciami bol celý virtuálny stroj znovu spustený, inicializovaný a načítaný image systému Strongtalk.

Výsledky ukazujú tabuľky 6.2, 6.3, 6.4 a 6.5. Prázdne políčka tabuliek označujú nenamerané hodnoty.

6.3 Slopstones Benchmarks

Slopstones Benchmarks (Smalltalk Low level OPeration Stones Benchmark) testuje malé operácie virtuálneho stroja. Konkrétne testuje 7 operácií a to celočíselné sčítanie, sčítanie v pohyblivej rádovej čiarke, prístup do reťazca, vytvorenie objektu, kopírovanie objektu, použite selektorov a výpočet blokov.

Čas [s]		smopstones		
Windows	Linux	Windows	Linux	Popis
0,094	0,18	335,851064	175,388889	Generovanie Fibonacciho postupnosti
0,027	0,062	415,925926	181,129032	Generovanie prvočísel
0,05	0,414	218,2	77,375887	Vytváranie malých tokov
0,048		643,958333		Vytváranie reťazcov
0,025		499,8		Tvorenie sústav
0,091		564,725275		Radenie reťazcov
0,071		788,843239		Algoritmus sorcerer's apprentice

Tabuľka 6.2: Výsledky Smopstone Benchmarks, 1. iterácia, 10 opakovaní

Čas [s]		smopstones		
Windows	Linux	Windows	Linux	Popis
0,067	0,177	471,19403	178,361582	Generovanie Fibonacciho postupnosti
0,027	0,064	415,925926	175,46875	Generovanie prvočísel
0,05	0,14	218,2	77,928571	Vytváranie malých tokov
0,035		883,142857		Vytváranie reťazcov
0,017		686,470588		Tvorenie sústav
0,074		694,459459		Radenie reťazcov
0,049		1143,061224		Algoritmus sorcerer's apprentice

Tabuľka 6.3: Výsledky Smopstone Benchmarks, 2. iterácia, 10 opakovaní

Testovanie sa opakuje 16 000-krát. Výsledkom tetu je počet opakovaní krát 1000, čas dokončenia, počet opakovaní za jednu sekundu krát 1000 a hodnota slopstones, vypočítaná pomocou heuristiky, slúžiaca na ohodnotenie testu. Čím vyššie číslo, tým lepšie. Hodnota slopstones slúži pre porovnávanie testov medzi operačnými systémami a vlastného virtuálneho stroja s virtuálnym strojom systému Strongtalk.

Výsledky ukazujú tabuľky 6.6, 6.7, 6.8 a 6.9.

6.4 OO Standford Benchmarks

Standforské testy je kolekcia testov implementovaných pomocou objektovej orientácie. Obsahuje niekoľko základných algoritmov, ktoré sú jednotlivo spúšťané a je meraný čas ich dokončenia. Keďže je čas dokončenia algoritmu meraný v celočíselných hodnotách, sú pri niektorých algoritmoch časy dokončenia uvedené s hodnotou 0. Túto skutočnosť treba interpretovať ako veľmi krátky čas dokončenia. Čas 0 vznikol ako nepresnosť merania času.

Tieto testy skončia chybou pri načítaní posledného algoritmu `Parser`. Nemožno test dokončiť, viz kapitola 5.5. Je potrebné považovať výsledky tohto testu za skresľujúce, pretože medzi jednotlivými iteráciami testu, musel byť celý virtuálny stroj znovu spustený, inicializovaný a musel byť načítaný image systému Strongtalk.

Výsledky ukazuje tabuľka 6.10.

Čas [s]		smopstones		
Windows	Linux	Windows	Linux	Popis
0,055	0,178	574,0	177,359551	Generovanie Fibonacciho postupnosti
0,018	0,062	623,888889	181,129032	Generovanie prvočísel
0,036	0,139	303,055556	78,489209	Vytváranie malých tokov
0,034		909,117647		Vytváranie reťazcov
0,02		583,5		Tvorenie sústav
0,08		642,375		Radenie reťazcov
0,047		1191,702128		Algoritmus sorcerer's apprentice

Tabuľka 6.4: Výsledky Smopstone Benchmarks, 3. iterácia, 10 opakovaní

Čas [s]		smopstones		
Windows	Linux	Windows	Linux	Popis
0,055	0,177	574,0	178,361582	Generovanie Fibonacciho postupnosti
0,017	0,061	660,588235	184,098361	Generovanie prvočísel
0,036	0,139	303,055556	78,489209	Vytváranie malých tokov
0,036		858,411111		Vytváranie reťazcov
0,019		614,210526		Tvorenie sústav
0,08		642,375		Radenie reťazcov
0,049		1143,061224		Algoritmus sorcerer's apprentice

Tabuľka 6.5: Výsledky Smopstone Benchmarks, 4. iterácia, 10 opakovaní

6.5 DeltaBlue

DeltaBlue je algoritmus riešiča obmedzujúcich podmienok. Modeluje prístup, kedy neriešime zadaný problém pomocou jednoznačne popísaného algoritmu, ale pomocou množiny obmedzujúcich podmienok, ktoré sú predané riešiču a ten na ich základe vráti výsledok.

Výsledok ukazuje tabuľka 6.11.

6.6 Vyhodnotenie testovania

Pri vyhodnotení testov nebudeme brať do úvahy Smopstone Benchmarks a OO Standford Benchmarks, pretože tieto testy neprešli v poriadku a ich jednotlivé iterácie prebiehali inak, než mali, pretože medzi jednotlivými iteráciami musel byť celý virtuálny stroj znovu spustený, inicializovaný a načítaný image systému Strongtalk.

Richards Benchmark prebehol v systéme Windows v prvých dvoch iteráciách rýchlejšie, ďalšie dve iterácie sa časy v oboch operačných systémoch vyrovnali.

Slopstone Benchmarks ukazuje rovnaké správanie, prvé iterácie ukazujú horšie výsledky v operačnom systéme GNU-Linux, ale posledná iterácia ukazuje rovnaké hodnotenie testov v obidvoch systémoch.

Nakoniec aj posledný DeltaBlue test ukazuje rovnaké správanie. Prvé iterácie prebehnú s horším časom v systéme GNU-Linux než tie v systéme Windows.

Toto správanie môže byť spôsobené rozdielnym správaním systémov pri inicializácii a vytváraní nového priestoru v pamäti. Možno by výsledky vylepšilo použitie prepínača `-O3`,

	Čas [s]		Slopstones		
Iterácií [x1000]	Windows	Linux	Windows	Linux	Popis
3808	0,022	0,055	26,225895	10,490358	Celočíselné sčítanie
544	0,015	0,028	151,111111	80,952381	Sčítanie vo float
960	0,023	0,236	47,323277	4,612014	Prístup do reťazca
320	0,012	0,057	75,757576	15,948963	Vytvorenie objektu
160	0,01	0,024	149,53271	62,305296	Kopírovanie objektu
480	0,038	0,163	29,721362	6,928907	Použitie selektoru
896	0,039	0,063	31,73254	19,643953	Výpočet bloku

Tabuľka 6.6: Výsledky Slopstone Benchmarks, 1. iterácia, 16 000 opakovaní

	Čas [s]		Slopstones		
Iterácií [x1000]	Windows	Linux	Windows	Linux	Popis
3808	0,017	0,02	33,939394	28,848485	Celočíselné sčítanie
544	0,012	0,017	188,888889	133,333333	Sčítanie vo float
960	0,023	0,232	47,323277	4,691532	Prístup do reťazca
320	0,013	0,055	69,93007	16,528926	Vytvorenie objektu
160	0,009	0,031	166,147456	48,235358	Kopírovanie objektu
480	0,037	0,168	30,524642	6,722689	Použitie selektoru
896	0,037	0,068	33,447812	18,199545	Výpočet bloku

Tabuľka 6.7: Výsledky Slopstone Benchmarks, 2. iterácia, 16 000 opakovaní

ktorý nebol použitý pri preklade virtuálneho stroja z dôvodu, že optimalizácia prekladačom nebola použitá ani v projekte pre VisualStudio pôvodného virtuálneho stroja. Rozdieli taktiež mohlo spôsobiť použitie preloženého spustiteľného súboru v operačnom systéme Windows, o ktorom nie je jasné, akým spôsobom bol preložený. Môžme však povedať, že rozdieli nie sú príliš veľké a vlastný virtuálny stroj ponúka uspokojivé výsledky.

	Čas [s]		Slopstones		
Iterácií [x1000]	Windows	Linux	Windows	Linux	Popis
3808	0,004	0.031	144,242424	18,611926	Celočíselné sčítanie
544	0,001	0.002	2266,666667	1133,333333	Sčítanie vo float
960	0,001	0.004	1088,435374	272,108844	Prístup do reťazca
320	0,001	0,033	909,090909	27,548209	Vytvorenie objektu
160	0,009	0,028	166,147456	53,404539	Kopírovanie objektu
480	0,037	0,174	30,524642	6,490872	Použitie selektoru
896	0,002	0,009	618,78453	137,507673	Výpočet bloku

Tabuľka 6.8: Výsledky Slopstone Benchmarks, 3 iterácia, 16 000 opakovaní

	Čas [s]		Slopstones		
Iterácií [x1000]	Windows	Linux	Windows	Linux	Popis
3808	0,001	0,001	576,969697	576,969697	Celočíselné sčítanie
544	0,001	0,001	2266,666667	2266,666667	Sčítanie vo float
960	0,001	0,001	1088,435374	1088,435374	Prístup do reťazca
320	0,001	0,001	909,090909	909,090909	Vytvorenie objektu
160	0,008	0,009	168,915888	166,147456	Kopírovanie objektu
480	0,036	0,039	31,372549	28,959276	Použitie selektoru
896	0,001	0,001	1237,569061	1237,569061	Výpočet bloku

Tabuľka 6.9: Výsledky Slopstone Benchmarks, 4. iterácia, 16 000 opakovaní

	1. Iterácia [ms]		2. Iterácia [ms]		3. Iterácia [ms]		4. Iterácia [ms]	
Algoritmus	Win	Linux	Win	Linux	Win	Linux	Win	Linux
BubbleSort	6	6	5	7	3	6	3	6
BubbleSort2	4	5	2	5	2	5	2	5
IntMM	1	6	1	6	1	6	1	6
IntMM2	2	7	1	6	1	7	1	6
MM	3	11	4	11	3	11	4	11
MM2	5	15	4	15	3	16	3	15
Perm	1	1	2	1	1	1	1	1
Perm2	1	1	1	1	1	1	0	1
Queens	1	1	1	1	1	1	1	1
Queens2	1	1	1	1	1	1	1	1
QuickSort	1	2	1	2	1	2	1	2
QuickSort2	1	3	1	3	1	3	1	3
Towers	2	3	1	3	1	3	1	3
Towers2	0	1	0	1	0	1	0	1
TreeSort	2	10	2	9	2	10	1	9
TreeSort2	2	7	1	7	1	7	1	7
Puzzle	18	35	36	35	14	36	14	35
Sieve	3	3	1	3	1	3	1	3
SumTo	6	6	5	6	5	6	5	6
Recurse	1	1	1	1	1	1	1	1
AtAllPut	0	0	0	0	0	0	0	0
IncrementAll	1	1	0	1	0	1	0	1
NestedLoop	4	4	5	5	4	4	4	4
Tak	0	0	0	0	0	0	0	0
Takl	10	8	7	7	7	8	6	7
Dictionary	38	67	16	67	16	70	17	67
OrderedCollection	31	35	32	34	15	35	17	34
Livermore	15	46	7	42	7	43	8	42
Parser	26		0		0		0	

Tabuľka 6.10: Výsledky OO Standfords Benchmarks

	Čas [ms]	
Iterácia	Windows	Linux
1	56	75
2	18	20
3	11	12
4	11	11

Tabuľka 6.11: Výsledky DeltaBlue

Kapitola 7

Záver

Táto práca popisuje implementáciu virtuálneho stroja systému Strongtalk, ktorý je implementáciou programovacieho jazyka a systému Smalltalk-80. Úlohou práce bolo na základe už existujúceho virtuálneho stroja napísaného v MS C++ vytvoriť vlastný virtuálny stroj, respektíve prepísať pôvodný virtuálny stroj tak, aby bol prenositeľný medzi operačnými systémami Windows a GNU-Linux.

Túto úlohu sa podarilo splniť a samotný virtuálny stroj vykazuje rovnaké správanie v operačnom systéme Windows a v operačnom systéme GNU-Linux. Toto správanie avšak nie je úplne korektné. V niektorých situáciach končí prevádzanie Smalltalk kódu chybou. Táto chyba avšak nie je pravdepodobne chyba virtuálneho stroja, ale je to chyba samotného systému Strongtalk, ktorý beží nad týmto virtuálnym strojom, viz. 5.5.

Samotný systém je k dispozícii ako image, prípadne sú k dispozícii jeho zdrojové kódy, ktoré tvoria takmer 1150 súborov, kde vždy jeden súbor tvorí popis jednej triedy programu. Preskúmanie a úprava týchto tried by však vyžadovalo ďalšie prostriedky (najmä časové), ktoré nie sú v rámci tejto bakalárskej práce k dispozícií. Preštudovanie zdrojových kódov systému Strongtalk by však nie len že odhalilo chybu v systéme Strongtalk, ale možno by odhalilo skrytú chybu v samotnom virtuálnom stroji, pretože niektoré konštrukcie použité vo virtuálnom stroji sú nejasné bez bližšej znalosti systému Strongtalk.

Testovanie obmedzené na testy, ktoré dokáže systém Strongtalk previesť ukázalo, že systém je o niečo pomalší v systéme GNU-Linux, než v systéme Windows. Avšak testovanie prebiehalo v systéme Windows na spustiteľnom súbore, ktorý je možné priamo stiahnuť zo stránok projektu, pri čom nebol známy presný spôsob ako bol tento súbor vytvorený, teda preložený zo zdrojových kódov a pri testovaní v systéme GNU-Linux nebol použitý prepínač pre prekladač `-O3`, ktorý slúži k optimalizácii zdrojové kódu, pretože optimalizácie neboli použité ani v projekte pre VisualStudio pre pôvodný virtuálny stroj. Testovanie prebiehalo vždy v štyroch iteráciách, kde posledná iterácia vždy ukázala veľmi podobné výsledky testovania v oboch systémoch. Rozdiel v prvých iteráciách môže teda spôsobovať rozdiel v správaní operačných systémov pri vytváraní nového priestoru v pamäti a jeho inicializácii.

Výsledná práca spĺňa zadanie, pre jej úplné dokončenie by bolo ešte potrebné spraviť podporu GUI pre systém Strongtalk za pomoci virtuálneho stroja. Ďalej by bolo potrebné preštudovať Smalltalk zdrojové kódy systému Strongtalk a nájsť v nich chybu, respektíve za ich pomoci odhaliť skrytú chybu vo virtuálnom stroji.

Literatúra

[1] *ANSI Smalltalk Standard*. [online], 2010, [cit. 2013-04-19].
Dostupné z: http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html

[2] Borman, S.: *Sensible sanitation – Understanding the IBM Java Garbage Collector, Part 2: Garbage collection*. [online], 2002, [cit. 2013-04-19].
Dostupné z: http://www.ibm.com/developerworks/java/library/j-jtp10283/

[3] Craig, I. D.: *Virtual Machines*. 2010, ISBN 1-85233-969-1.

[4] Gilad Bracha, David Griswold: *Strongtalk: Typechecking Smalltalk in a Production Enviroment*. [online], [cit. 2013-04-19].
Dostupné z: http://www.bracha.org/oopsla93.ps

[5] Goetz, B.: *Java theory and practice: A brief history of garbage collection*. [online], 2003, [cit. 2013-04-19].
Dostupné z: http://www.ibm.com/developerworks/java/library/j-jtp10283/

[6] Kay, A.: *The early history of Smalltalk*. [online], 2007, [cit. 2013-04-19].
Dostupné z: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

[7] Křivánek, P.: *Squeak: návrat do budoucnosti*. [online], 2004, [cit. 2013-04-19].
Dostupné z: http://www.root.cz/clanky/squeak-navrat-do-budoucnosti/

[8] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, Urs Hölzle: *Mixins in Strongtalk*. [online], 2002, [cit. 2013-04-19].
Dostupné z: http://strongtalk.org/downloads/mixins-paper.ps

[9] Liu, C.: *Smalltalk, Objects, and Design*. iUniverse reprint, 2000, ISBN 1-58348-490-6.

[10] Phillip: *Using Assembly Language in Linux*. [online], 2001, [cit. 2013-04-19].
Dostupné z: http://asm.sourceforge.net/articles/linasm.html

[11] Zbyněk Křivka, Dušan Kolář: *Principy programovacích jazyků a objektově orientovaného programování IPP – II*. Fakulta informačních technologií VUT v Brně, 2008.

Príloha A

Obsah CD

- ↪ `Strongtalk/` – Adresár so systémom Strongtalk
  - ↪ `bin/` – Adresár s knižnicami potrebnými pre beh systému Strongtalk
  - ↪ `build/` – Adresár so spustiteľným súborom `Makedeps` a makefileom pre preklad v operačnom systéme GNU-Linux.
  - ↪ `build.linux/` – Adresár so súbormi potrebnými pre preklad v operačnom systéme GNU-Linux.
  - ↪ `build.win32/` – Adresár so súbormi potrebnými pre preklad v operačnom systéme Windows vo VisualStudio 2010.
  - ↪ `tools/` – Adresár s pomocnými skriptami a utilitou Makedeps 5.1.
  - ↪ `vm/` – Adresár so zdrojovými kódmi virtuálneho stroja.
  - ↪ `StrongtalkSource/` – Adresár so zdrojovými kódmi systému Strongtalk.
  - ↪ `strongtalk` – Spustiteľný súbor virtuálneho stroja získaný prekladom na školskom servery merlin.
  - ↪ `strongtalk.bst` – Image systému Strongtalk.
- ↪ `doc/` – Adresár s touto dokumentáciou a s LaTeX zdrojovými kódmi
- ↪ `runme.sh` – Spustiteľný shell skript, viz. B.
- ↪ `README.txt` – Popis virtuálneho stroja, spôsobobu prekladu a jeho parametrov príkazového riadku a popis skriptu `runme.sh`.

Príloha B

Skript `runme.sh`

Skript `runme.sh` je jednoduchý shell skript slúžiaci k jednoduchému overeniu funkčnosti virtuálneho stroja, vďaka ktorému nie je potrebné poznať postup prekladu virtuálneho stroja alebo jeho parametre príkazového riadku.

Skript počíta s umiestnením skriptu a relatívnych ciest k virtuálnemu stroju, imageu systému Strongtalk, skriptu pre systém Strongtalk `bench.dlt`, a súboru `makefile` tak, ako na priloženom CD. Pri zmene umiestnenia niektorého z týchto súborov sa počíta s tým, že ten, kto tieto súbory premiestnil pozná samotný systém Strongtalk a jeho virtuálny stroj a tak sa stáva tento skript pre neho neužitočný.

B.1 Parametre skriptu

Pri spustení bez parametrov je prevedený už vytvorený spustiteľný súbor, ak tento existuje a na ňom sú vykonané tri testy: Richards Benchmark, Slopstone Benchmark a DeltaBlue.

Pri použití parametru `-c` alebo `--compile` je virtuálny stroj najprv preložený a až na tomto preloženom virtuálnom stroji sú spustené testy. Pozor, tento parameter prepíše priložený spustiteľný súbor a preto nebude fungovať pri spustení na priloženom CD. Kombináciou s parametrom `-d` alebo `--donttest` je možné zabrániť spusteniu testov na novovytvorenom spustiteľnom súbore, teda skript len preloží virtuálny stroj.

Príloha C

Parametre virtuálneho stroja

Tabuľka C.1: Parametre virtuálneho stroja, ktoré je možné meniť pomocou parametru `-f`

parameter	typ (`bool` alebo `int`)
ZapResourceArea	bool
PrintResourceAllocation	bool
PrintResourceChunkAllocation	bool
PrintHeapAllocation	bool
PrintOopAddress	bool
PrintObjectID	bool
PrintLongFrames	bool
LogVMMessages	bool
AlwaysFlushVMMessages	bool
VerifyBeforeScavenge	bool
VerifyAfterScavenge	bool
PrintScavenge	bool
PrintGC	bool
WizardMode	bool
VerifyBeforeGC	bool
VerifyAfterGC	bool
VerifyZoneOften	bool
PrintVMMessages	bool
CompiledCodeOnly	bool
UseRecompilation	bool
UseNMethodAging	bool
UseInlineCaching	bool
EnableTasks	bool
CompressPcDescs	bool
UseAccessMethods	bool
UsePredictedMethods	bool
UsePrimitiveMethods	bool
PrintStackAtScavenge	bool
PrintInterpreter	bool
PrintStubRoutines	bool
UseInliningDatabase	bool

UseInliningDatabaseEagerly	bool
UseSlidingSystemAverage	bool
UseGlobalFlatProfiling	bool
EnableOptimizedCodeRecompilation	bool
CountParentLinksAsOne	bool
GenerateSmalltalk	bool
GenerateHTML	bool
UseTimers	bool
SweeperUseTimer	bool
EnableProcessPreemption	bool
HasActivationClass	bool
TraceOopPrims	bool
TraceDoublePrims	bool
TraceByteArrayPrims	bool
TraceDoubleByteArrayPrims	bool
TraceDoubleValueArrayPrims	bool
TraceObjArrayPrims	bool
TraceSmiPrims	bool
TraceProxyPrims	bool
TraceBehaviorPrims	bool
TraceBlockPrims	bool
TraceDebugPrims	bool
TraceSystemPrims	bool
TraceProcessPrims	bool
TraceVframePrims	bool
TraceCallBackPrims	bool
TraceLookup	bool
TraceLookupint	bool
TraceLookupAtMiss	bool
TraceBytecodes	bool
TraceAllocation	bool
TraceExpansion	bool
TraceBootstrap	bool
TraceMethodPrims	bool
TraceMixinPrims	bool
TraceVMOperation	bool
TraceDLLLookup	bool
TraceDLLCalls	bool
TraceGC	bool
TraceMessageSend	bool
TraceInlineCacheMiss	bool
TraceProcessEvents	bool
TraceDeoptimization	bool
TraceZombieCreation	bool
TraceResults	bool
TraceApplyChange	bool
TraceInliningDatabase	bool
TraceCanonicalContext	bool

ActivationShowExpressionStack	bool
ActivationShowBCI	bool
ActivationShowFrame	bool
ActivationShowContext	bool
ActivationShowCode	bool
ActivationShowNameDescs	bool
ShowMessageBoxOnError	bool
BreakAtWarning	bool
PrintCompilerWarnings	bool
PrintInliningDatabaseCompilation	bool
CountBytecodes	bool
ProfilerShowMethodHolder	bool
UseMICs	bool
UseLRUInterrupts	bool
UseNewBackend	bool
TryNewBackend	bool
UseFPUStack	bool
ReorderBBs	bool
CodeForPint	bool
PrintInlineCacheInvalidation	bool
PrintCodeReclamation	bool
PrintCodeSweep	bool
PrintCodeCompaction	bool
PrintMethodFlushing	bool
MakeBlockMethodZombies	bool
CompilerDebug	bool
EnableIntint	bool
VerifyCode	bool
VerifyDebugInfo	bool
GenTraceCalls	bool
TraceCalls	bool
MaterializeEliminatedBlocks	bool
Inline	bool
InlinePrims	bool
ConstantFoldPrims	bool
TypePredict	bool
TypePredictArrays	bool
TypeFeedback	bool
CodeSizeImpactsInlining	bool
OptimizeIntegerLoops	bool
OptimizeLoops	bool
EliminateJumpsToJumps	bool
EliminateContexts	bool
LocalCopyPropagate	bool
GlobalCopyPropagate	bool
BruteForcePropagate	bool
Splitting	bool
EliminateUnneededNodes	bool

DeferUncommonBranches	bool
MemoizeBlocks	bool
DebugPerformance	bool
PrintInlining	bool
PrintSplitting	bool
PrintLocalAllocation	bool
PrintGlobalAllocation	bool
PrintEliminateContexts	bool
PrintCompilation	bool
PrintRecompilation	bool
PrintRecompilationint	bool
PrintCode	bool
PrintAssemblyCode	bool
PrintJumpElimination	bool
PrintRegAlloc	bool
PrintCopyPropagation	bool
PrintUncommonBranches	bool
PrintRegTargeting	bool
PrintExposed	bool
PrintEliminateUnnededNodes	bool
PrintHexAddresses	bool
GenerateLiteScopeDescs	bool
PrintRScopes	bool
PrintLoopOpts	bool
PrintStackAfterUnpacking	bool
PrintDebugInfo	bool
PrintDebugInfoGeneration	bool
PrintCodeGeneration	bool
PrintPRegMapping	bool
PrintMakeConformantCode	bool
CreateScopeDescInfo	bool
GenerateFullDebugInfo	bool
UseNewMakeConformant	bool
EventLogLength	int
StackPrintLimit	int
MaxElementPrintSize	int
ReservedHeapSize	int
ObjectHeapExpandSize	int
EdenSize	int
SurvivorSize	int
OldSize	int
ReservedCodeSize	int
CodeSize	int
ReservedPICSize	int
PICSize	int
JumpTableSize	int
ThreadStackSize	int
CompilerInstrsSize	int

CompilerScopesSize	int
CompilerPCsSize	int
MaxFnInlineCost	int
MaxBlockInlineCost	int
MinBlockCostFraction	int
BlockArgAdditionalAllowedInlineCost	int
InvocationCounterLimit	int
LoopCounterLimit	int
MaxNmInstrSize	int
MinSendsBeforeRecompile	int
MaxFnInstrSize	int
BlockArgAdditionalInstrSize	int
MaxBlockInstrSize	int
MaxRecursionUnroll	int
MaxTypeCaseSize	int
UncommonRecompileLimit	int
UncommonInvocationLimit	int
UncommonAgeBackoffFactor	int
MinInvocationsBeforeTrust	int
NMethodAgeLimit	int
MaxRecompilationSearchLength	int
MaxInterpretedSearchLength	int
CounterHalfLifeTime	int
MaxCustomization	int
StopInterpreterAt	int
TraceInterpreterFramesAt	int
NumberOfContextAllocations	int
NumberOfBlockAllocations	int
NumberOfBytecodesExecuted	int
ProfilerNumberOfInterpreterMethods	int
ProfilerNumberOfCompiledMethods	int
HeapSweeperInterval	int
PrintProgress	int
InliningDatabasePruningLimit	int