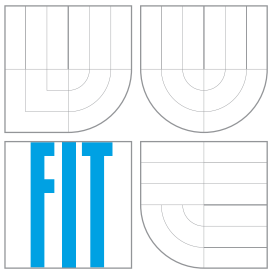
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VÝVOJ LLVM ADAPTÉRU PRO INFRASTRUKTURU CODE LISTENER

DEVELOPMENT OF AN LLVM ADAPTER FOR THE CODE LISTENER INFRASTRUCTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VERONIKA ŠOKOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAMIL DUDKA

BRNO 2014

Zadání bakalářské práce/15896/2013/xsokov00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2013/2014

Zadání bakalářské práce

Řešitel: **Šoková Veronika**
Obor: Informační technologie
Téma: **Vývoj LLVM adaptéru pro infrastrukturu Code Listener**
Development of an LLVM Adapter for the Code Listener Infrastructure
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se s nástrojem Clang/LLVM pro překlad a analýzu C/C++ programů, infrastrukturou Code Listener pro tvorbu nástrojů na statickou analýzu a nástroji Predator a Forester pro verifikaci operací s dynamickými datovými strukturami.
2. Prostudujte interní reprezentaci kódu v infrastruktuře LLVM. Zaměřte se zejména na reprezentaci instrukcí pro práci s pamětí.
3. Implementujte Clang/LLVM adaptér pro infrastrukturu Code Listener tak, aby bylo možné nástroje Predator a Forester používat nezávisle na GCC.
4. Vytvořený adaptér otestujte na sadách testů dodávaných spolu s nástroji Predator a Forester.
5. Srovnejte vámi vytvořený adaptér s adaptérem pro GCC a zhodnoťte přínos vámi vytvořeného adaptéru pro nástroje Predator a Forester.

Literatura:

- K. Dudka, P. Peringer, and T. Vojnar. An Easy to Use Infrastructure for Building Static Analysis Tools. In Proc. of 13th International Conference on Computer Aided Systems Theory - EUROCAST'11, Las Palmas, Spain, volume 6927 of LNCS, pages 527-534, 2012. Springer-Verlag.
- K. Dudka, P. Müller, P. Peringer, and T. Vojnar. A Tool for Verification of Low-level List Manipulation. In Proc. of 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - TACAS'13, Rome, Italy, volume 7795 of LNCS, pages 627-629, 2013. Springer-Verlag.
- Domovská stránka nástroje Clang/LLVM: http://clang.llvm.org/

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Dudka Kamil, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2013
Datum odevzdání: 21. května 2014

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá vývojem LLVM adaptéru pro infrastrukturu Code Listener, která usnadňuje tvorbu statických analyzátorů jako jsou Predator a Forester. Ty jsou vyvíjeny a využívany v rámci skupiny VeriFIT. Popisuje překladový systém LLVM, jeho interní reprezentaci kódu a frontend Clang. Součástí práce je implementace daného adaptéru. K dnešnímu dni je schopen analyzovat omezenou množinu programů jazyka C. Je schopen generovat CFG k funkcím. Některé testy pro Predator a Forester projdou. Dále je naznačen budoucí vývoj adaptéru.

Abstract

This Bachelor's thesis deals with the development of an LLVM adapter for the Code Listener Infrastructure, which simplifies the creation of static analyzers such as the Predator and the Forester. They are developed and used within the group VeriFIT. It describes LLVM compiler system, the internal representation of the code and frontend Clang. Part of this work is the implementation of the adapter. Up to this date, it is able to analyze a limited set of programs in C. It is able to generate CFGs. Some tests for Predator and Forester pass. It is also hinted at future developments.

Klíčová slova

LLVM, infrastruktura Code Listener, plugin, statická analýza

Keywords

LLVM, Code Listener Infrastructure, plug-in, static analysis

Citace

Veronika Šoková: Vývoj LLVM adaptéru pro infrastrukturu Code Listener, bakalářská práce, Brno, FIT VUT v Brně, 2014

Vývoj LLVM adaptéru pro infrastrukturu Code Listener

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Kamila Dudku. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Veronika Šoková
21. mája 2014

Obsah

Kapitola 1

Úvod

Táto práca sa zaoberá vývojom LLVM adaptéru k rozhraniu Code Listener, ktoré uľahčuje tvorbu statických analyzátorov. K dnešnému dňu je podporovaný prekladač GCC a sémantický analyzátor Sparse. Všeobecne adaptér umožňuje spoluprácu častiam programu s rôznymi vnútornými štruktúrami ich vzájomným namapovaním.

Cieľom tejto práce je vytvoriť alternatívny prístup k používaniu analyzátorov ako je Predator či Forester a tak odhaľovať prípadné chyby v programoch jazyka C v spojení s iným prekladovým systémom ako bol používaný doposiaľ.

Na miestach, kde neexistuje slovenský ekvivalent anglického výrazu, sú použité anglické termíny (viď Slovník pojmov na str. 29).

Práca je členená nasledujúcim spôsobom. Úvod do problematiky statických analyzátorov sa nachádza v kapitole 2. Rozhranie Code Listener je popísane v 3. kapitole. Štruktúra prekladového systému LLVM a front-endu Clang, tak ako aj návrh napojenia rozširujúcich modulov, približuje kapitola 4. V 5. je popis vlastnej implementácie adaptéru. Výsledky testovania sú prezentované v kapitole 6. Ďalšia kapitola 7 načrtne možné budúce rozšírenie. Výsledky testovania a samotnej práce s prípadnými nedostatkami sú zhrnuté v kapitole 8.

1.1 Motivácia

Statické analyzátory sú prostriedkom na kontrolu správnosti zdrojového kódu a dopomáhajú odhaľovať chyby programátora. Takými nástrojmi sú aj práve vyvíjané analyzátory Predator a Forester. Aktuálne pracujú nad prekladačom GCC. Aby testovanie programov nemuselo byť obmedzené len na jeden prekladový systém, táto práca ponúka rozšírenie už stávajúcej infraštruktúry Code Listener o LLVM adaptér.

Samozrejme existuje veľké spektrum iných prekladačov, avšak LLVM je podobne ako GCC šírený pod slobodnou licenciou. Jedná sa o moderný softwarový systém, pôvodne vytváraný ako univerzitný projekt. Dnes už so širokou a aktívnou komunitou vývojárov. LLVM je často používaná infraštruktúra pre nástroje na verifikáciu kódu (napr. Clang Static Analyzer).

Kapitola 2

Statická analýza

Statická analýza informuje o správaní systému na základe zdrojového popisu bez jeho vykonávania (alebo vykonávanie s odľahčenou sémantikou). Tento druh analýzy nevyžaduje model systému, používa sa i pre optimalizáciu a generovanie kódu. Jednou z techník statickej analýzy je symbolická exekúcia, ktorá vykonáva program, nie s dátami, ale množinami hodnôt, ktoré sú popísané formou logiky alebo automatmi.

V rámci formálnej verifikácie rozlišujeme aj iné postupy ako sú napr. Model Checking, Theorem Proving a SAT Solving.[1]

2.1 Analýza tvaru

Analýza tvaru (angl. *shape analysis*, podrobnejšie [1]) je druh statickej analýzy, ktorý si dáva za cieľ verifikovať kompozitné štruktúry ako sú rôzne cyklické a acyklické zoznamy. Existujú prístupy k analýze tvaru, ktoré potrebujú vopred vedieť tvar dátových štruktúr. Alebo sú poloautomatizované tak, že sa dodefinuje invariant cyklu či indukčný predikát.

Bohužiaľ dnešné analyzátory neponúkajú prostriedky na analýzu dynamických kompozitných dátových štruktúr používaných v oblasti priemyselného softwaru.

2.2 Vybrané nástroje tvarovej analýzy

Ďalej priblížim dva vybrané analyzátory vyvíjané pod záštitou skupiny VeriFIT. Tvorbu nástroja Predator vedie Kamil Dudka a nástroja Forester Ondřej Lengál (predtým Jiří Šimáček). Analyzátory sú postavené na rozhraní Code Listener (viď kapitola 3), preto sa aktuálne spúšťajú ako plug-in prekladača GCC. Zaoberajú sa verifikáciou programov s dynamickým prideľovaním pamäte. Oba sú publikované pod GNU GPLv3 licenciou.

Predator

Predator [2] je úplne automatizovaný nástroj ukazujúci nový prístup k tvarovej analýze programov s prepojenými zoznamami používajúci nízkoúrovňové operácie s pamäťou. Je založený na separačnej logike a symbolických grafoch pamäti (SMG).

Je schopný detekovať cyklické, vnorené (do ľubovoľnej hĺbky) a zdieľané jedno- či obojsmerne viazané zoznamy v programoch napísaných v jazyku C. Špeciálne sa zame-

[1] viď kurz FAV http://www.fit.vutbr.cz/study/courses/FAV/

riava na zoznamy používané v Linuxovom jadre. Kontroluje prácu so smerníkovou aritmetikou (angl. *pointer arithmetic*), neplatné ciele, blokové operácie s pamäťou, reinterpretáciu obsahu pamäti, zarovnanie adries a iné.

Nástroj je stále vo vývoji, no už teraz dosahuje výborné výsledky[2] (najmä čo do rýchlosti a zároveň presnosti).

Forester

Forester [4] je experimentálny nástroj založený na stromových automatoch (TA) a abstraktnom stromovom regulárnom model-checkingu (ARTMC). Názov analyzátora vychádza z pojmu lesný automat (FA), ktorý slúži na reprezentáciu množiny dosiahnuteľných konfigurácií programu. Vzhľadom na možnosť reprezentácie zložitých dátových štruktúr na halde, môžu byť FA hierarchicky zanorené tým spôsobom, že FA sú použité ako symboly abecedy iných FA (FA vyššej úrovne).

Zaoberá sa verifikáciou programov so všeobecnými dátovými štruktúrami. Automaticky detekuje základné chyby pri práci s dynamicky pridelenou pamäťou (napr. neplatná dereferencia, nedefinovaný smerník, viacnásobné uvoľnenie pamäte či pretečenie). Podporuje analýzu sekvenčných nerekurzívnych C programov pracujúcich s haldou. V súčasnosti pracuje dobre nad jedno- či obojsmerne viazanými zoznamami (i cyklickými), *skip*-zoznamami a stromami (s pridanými smerníkmi).

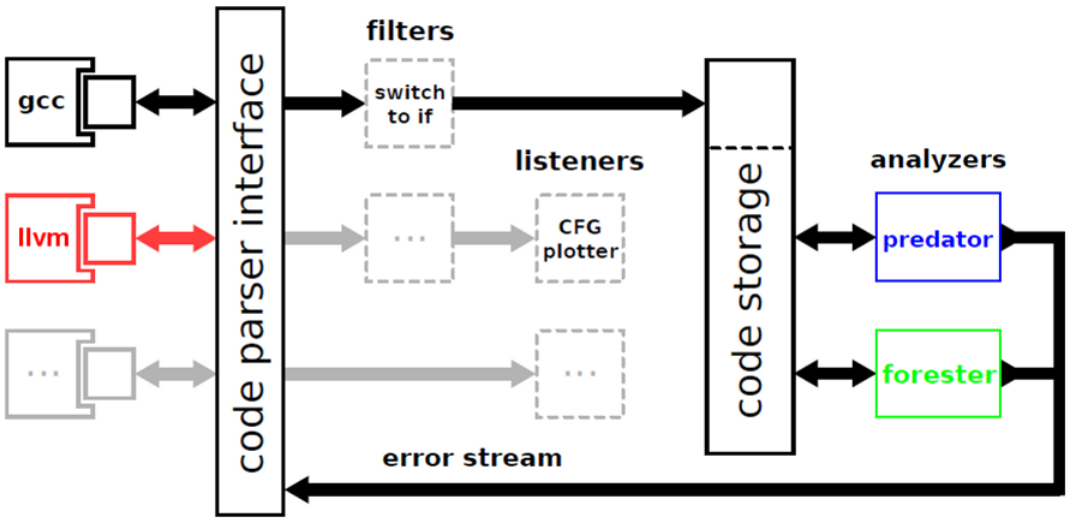
[2] SV-COMP'14 [3], kategória *Heap Manipulation*

Kapitola 3

Infraštruktúra Code Listener

Infraštruktúra Code Listener (CL, [5]) je framework uľahčujúci tvorbu statických analyzátorov vyvíjaný skupinou VeriFIT. Tvorí abstraktnú vrstvu medzi výstupom syntaktickej analýzy prekladača a vyvíjaným nástrojom. Je distribuovaná pod GPLv3 licenciou ako C++ knižnica[1] pre tvorbu GCC plug-inu. Vnútorná reprezentácia prekladaného programu je úplne nezávislá od GCC. Vďaka čomu sa môže napojiť na výstup iného syntaktického analyzátora (napr. Clang, viď sekcia 4.1) bez nutnosti meniť svoju vnútornú štruktúru.

Infraštruktúra CodeListener je schematicky znázornená na obr. 3.1. Blok *code parser interface* definuje Code Listener API pre syntaktické analyzátory využívajúci vlastný medzikód a *code storage* je C++ API pre statické analyzátory. Ďalej sú tu filtre ako napr. „switch to if" blok, ktorý transformuje `SWITCH` inštrukciu na `COND` inštrukcie. A bloky pracujúce iba so vstupom generujúce graf riadenia toku (CFG) `gen-dot`, graf typov `type-dot` alebo linearizovaný medzikód `dump-pp`.



Obr. 3.1: Architektúra Code Listener (inšpirované podľa [5, str. 531])

[1] spoločne s nástrojmi Predator a Forester v repozitári https://github.com/kdudka/predator

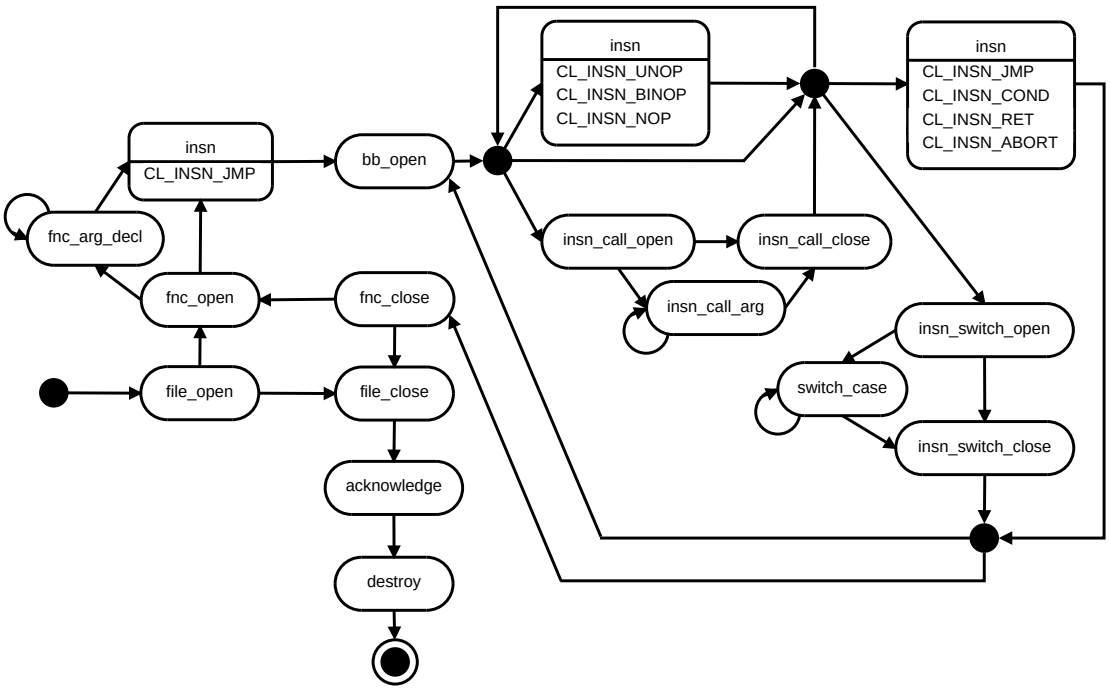
3.1 Medzikód

Medzikód používaný v Code Listener API je inšpirovaný GIMPLE používaným v prekladači GCC. Je podstatne jednoduchší. Funkcie programu sú popísané pomocou CFG. Inštrukcie sú rozdelené do dvoch skupín: ukončujúce (`COND`, `SWITCH`, `RET` a `ABORT`) a neukončujúce (`UNOP`, `BINOP` a `CALL`). Ukončujúca inštrukcia sa môže nachádzať iba na konci základného bloku (BB). Hrany v grafe určujú cieľ tejto inštrukcie. Pričom neukončujúca inštrukcia nesmie ukončiť BB. Popis inštrukcií v sekcií 5.3.

Inštrukcie pristupujú pomocou operandov (`cl_operand`) ku konštante (`cl_cst`) alebo premennej programu (`cl_var`). Konštanta reprezentuje číselný alebo reťazcový literál, či funkciu. Ku každej premennej a funkcií je priradená úroveň (`GLOBAL`, `STATIC` či `LOCAL`) a unikátne `uid`, ktoré ju identifikuje. Sémantiku operandu môžeme meniť pomocou tzv. *accessorov* (`cl_accessor`). Z premennej vytvoríme smerník na premennú a pod. Viacnásobná dereferencia v zdrojovom kóde sa v medzikóde reprezentuje pomocou postupnosti inštrukcií, kde v každej z nich je najviac jedna dereferencia. Každá z vyššie menovaných štruktúr si uchováva informáciu o type (popis v sekcií 5.2). Pričom typy sú definované rekurzívne. Kompozitné typy sú dané typom ich zložiek.

3.2 Rozhranie infraštruktúry Code Listener

Infraštruktúra sa skladá z dvoch rozhraní. Najskôr priblížim prvé, ktoré bude uplatnené pri implementácií.



Obr. 3.2: Diagram volania funkcií pri vytváraní objektu CL

Všetko potrebné na použitie Code Listener API pre tvorbu adaptéru sa nachádza v hlavičkovom súbore <`cl/code_listener.h`>. Definuje potrebné štruktúry, vymenované zoznamy symbolov a funkcie: inicializáciu (`cl_global_init`, `cl_global_init_defaults`),

vytvorenie objektu zapuzdrujúceho CL (`cl_code_listener_create`), zreťazenie viacerých objektov (`cl_chain_create`, `cl_chain_append`) a uvoľnenie zdrojov (`cl_global_cleanup`).

V objekte CL sú definované funkcie spätného volania (viď obr. 3.2) pre vytvorenie funkcií, BB a inštrukcií. Po vložení všetkých súčastí programu sa zavolá `acknowledge`, ktorý značí, že všetky funkcie spätného volania boli odoslané a objekt CL je validný. Na záver sa zavolá deštruktor (`destroy`).

Na rozdiel od predchádzajúceho je Code Storage API napísané v C++. Jeho použitie je ukázané na jednoduchom programe (`fwnull`) hľadajúcom `NULL`-smerník dereferencie.

Kapitola 4

LLVM

Low-Level Virtual Machine [6] je komplexný prekladový systém distribuovaný pod NCSA licenciou. Jedná sa o silne typovaný nízkoúrovňový modulárne riešený systém napísaný v jazyku C++, ktorý je veľmi dobre dokumentovaný. Pozostáva z troch častí: front-endu, optimalizéru a back-endu. Tým sa stáva samotný preklad jazykovo nezávislý. Systém ponúka prostriedky pre napísanie vlastných kompilátorov. Na nasledujúcich riadkom priblížim niektoré aktuálne vyvíjané projekty.

Front-end zabezpečuje lexikálnu, syntaktickú a sémantickú analýzu programu. Výstupom je vygenerovaný LLVM IR kód, ktorý sa ďalej spracuje. Najznámejší je **Clang** (viď 4.1), ktorý podporuje okrem C a C++ aj Objectiv-C a Objectiv-C++. Iné projekty podporujú napr. Javu a Scheme.

Gro systému predstavuje súbor knižníc nazvaných **LLVM Core**, ktoré tvoria optimalizér a back-end (generátor kódu) prekladača. Podporuje architektúry typu X86, X86-64, PowerPC, ARM, SPARC a iné. Okrem klasickej kompilácie je pre niektoré platformy podpora JIT-kompilácie.

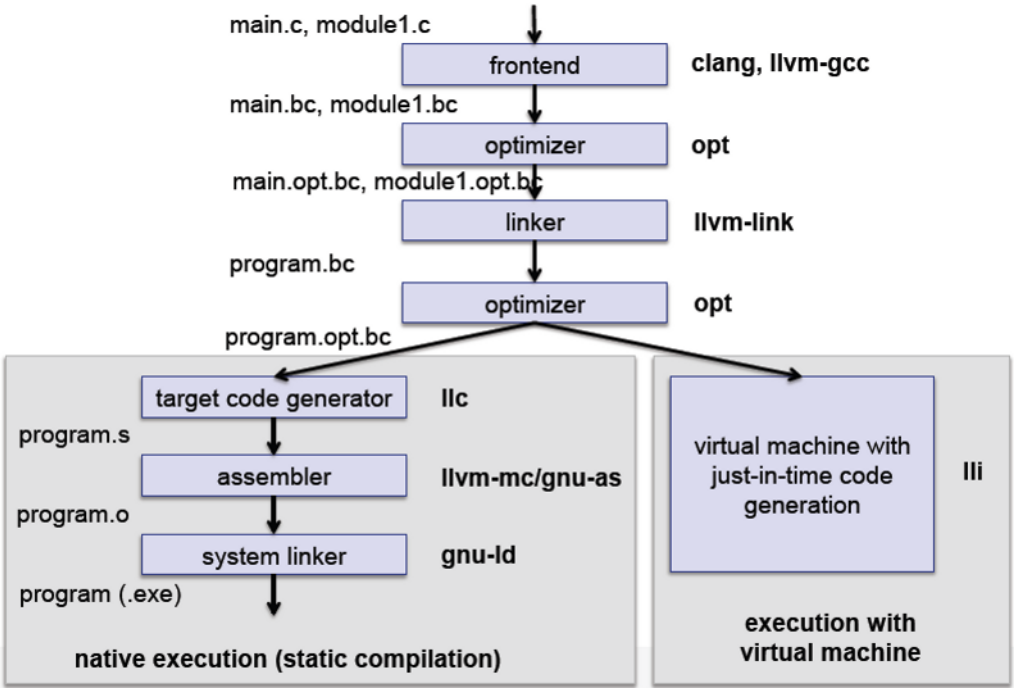
Ďalším projektom je **DragonEgg**, ktorý použije LLVM namiesto GCC back-endu. Napojí sa na GCC a výstup syntaktickej analýzy prevedie do podoby LLVM IR kódu. Aktuálne plne podporuje jazyky Ada, C, C++ a Fortran a čiastočne jazyky ako Go či Java.

Proces prekladu zdrojového kódu je znázornený na obrázku 4.1. Pri analýze kódu nás bude zaujímať časť medzi front-endom a optimalizérom.

4.1 Front-end Clang

Projekt Clang [8] tvorí front-end pre jazyky C, C++, Objective-C a Objective-C++. Je založený na prekladači GCC, má však ľahko pochopiteľnú štruktúru a je publikovaný pod NCSA licenciou. Ponúka API pre vývoj nástrojov používajúcich statickú a sémantickú analýzu.

Výstupom statickej analýzy je abstraktný syntaktický strom (AST) pre každú funkciu zvlášť, ktorý si uchováva podrobné informácie o typoch a umiestnení v zdrojovom súbore. Vďaka tomu je schopný generovať užitočné chybové hlásenia. Následne sa transformuje na CFG. Pri prevode do LLVM IR je schopný kód optimalizovať. Ponúka štyri úrovne optimalizácie, pričom väčšina prebieha na úrovni 2.



Obr. 4.1: Proces prekladu zdrojových súborov – ľavá vetva predstavuje statickú kompiláciu a pravá JIT (prebrané z [7, snímka č. 5])

4.2 LLVM IR

Medzikód (angl. *assembly language* ev. *intermediate representation*, [9]) tvorí gro kompilátoru. Jedná sa o jazykovo-nezávislý a silne typovaný systém založený na load-store architektúre a RISC inštrukciách. Má jednoznačne definovanú sémantiku. Kde každý operand má typ a aj návratová hodnota je typovaná.

Pracuje s potenciálne nekonečným počtom virtuálnych registrov v **SSA forme** (angl. *Static Single Assignment*, [10]). To znamená, že do každého registra sa môže priradiť hodnota iba raz. Z dominantnej premennej sa vytvoria nezávislé premenné pri každom priradení, avšak, ak je to možné, pristúpi k vykonaniu cesty v CFG ako prvá. Pre platnosť invariantov pri vetveniach a cykloch sa využíva Φ-inštrukcia volaná iba na začiatku BB. Tá zabezpečuje použitie správnej verzie premennej.

SSA uľahčuje analýzu závislostí medzi premennými, detekciu mŕtveho kódu a ďalšie optimalizácie. Napr. optimalizácia `-mem2reg` konvertuje ne-SSA formu LLVM IR do SSA podoby. Vkladá alokované premenné, ktoré sú použité iba v inštrukciách load a store do registrov (redukcia počtu inštrukcií).

Medzikód je dostupný v troch reprezentáciách: uložený v pamäti prekladača, na disku v binárnej podobe (pri JIT-preklade) a v textovej podobe.

Textová podoba

Je čitateľne formátovaná. Rozpoznáva lokálne a globálne symboly. Lokálne začínajú znakom `'%'` a sú to registre, lokálne premenné a definície typov. Globálne symboly sú označované znakom `'@'` a identifikujú funkcie a globálne premenné. Anonymným symbolom sa priradí meno pomocou prepínača `-instnamer`, inak sú postupne číslované (unikátne len

v rámci svojej úrovne). Komentár začína znakom ';'. Výpis pozostáva z popisu cieľovej architektúry, definícií typov, globálnych premenných, funkcií, aliasov a pomenovaných metadát.

Obsah zdrojového súboru `source.c`:

```
struct point {int x; int y;};

int A = 0;

int suma(int a, int b) {
        return a+b;
}

int main(void){
        return A;
}
```

Reprezentácia v medzikóde:

```
; ModuleID = 'source.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16
-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:1
28:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128" ; # cielova architektura
target triple = "x86_64-redhat-linux-gnu"

%struct.point = type { i32, i32 }                    ; # definicie typov

@A = global i32 0, align 4                           ; # globalne premenne

; Function Attrs: nounwind                           ; # funkcie
define i32 @suma(i32 %a, i32 %b) {                   ; # s parametrami
bb:
  %tmp4 = add nsw i32 %a, %b
  ret i32 %tmp4
}

; Function Attrs: nounwind
define i32 @main() {
bb:
  %tmp1 = load i32* @A, align 4
  ret i32 %tmp1
}
```

Podoba IR uložená v pamäti

Hlavným kontajnerom pre IR je `Module` [11]. Všetky objekty sú reprezentované pomocou obojsmerne viazaných zoznamov. `Module` obsahuje zoznam objektov `Function` (definície a deklarácie funkcií) a `GlobalVariable` (globálne premenné). `Function` obsahuje zoznam objektov `BasicBlock` (základné bloky) a `Argument` (argumenty funkcie). Približne odpovedá funkciám v C. `BasicBlock` obsahuje zoznam objektov `Instruction` (inštrukcie). `Instruction` sa pretypováva na správnu podtriedu podľa svojho `opcode`. Samotná predstavuje návratovú hodnotu. Môže obsahovať vektor operandov, pričom všetky a aj výsledok samotnej inštrukcie sú typované. Typ popisuje trieda `Type`, na ktorú vždy ukazuje trieda `Value` a od nej sú odvodené všetky triedy reprezentujúce premenné, konštanty a inštrukcie. Znázornené v prílohe A.

4.3 Možnosti napojenia

Pre vytvorenie adaptéru potrebujeme dostať program do tej správnej podoby, aby sa dal ľahko namapovať na vnútornú reprezentáciu Code Listenera (viď 3.1). Potrebujeme získať prístup k výsledku syntaktickej analýzy. Či už v podobe CFG alebo LLVM IR (viď 4.2). Do úvahy prichádzajú nasledujúce prostriedky.

LLVM priechod

LLVM priechod (angl. *LLVM Pass*, [12]) je LLVM framework pre nástroj, ktorý môže vykonávať transformácie a optimalizácie nad kódom. Vstupom je program v bitecode podobe. Programom je dynamická knižnica, ktorá sa nahrá do optimalizéra prepínačom: `opt -load`. V závislosti na zvolenej triede máme plnú alebo čiastočnú kontrolu nad LLVM IR.

Každá trieda popisuje tri virtuálne metódy: `doInitialization` (spúšťaná pred samotným spracovaním), `runOn<meno_triedy>` (hlavný program - spúšťaná nad každým objektom) a `doFinalization` (spúšťaná po spracovaní všetkých objektov). Nasleduje popis tried:

- trieda **`ImmutablePass`**: najmenej zaujímavá, podáva informácie o konfigurácií kompilátora, pričom nie je volaná, nič nerobí a nemôže nič meniť
- trieda **`ModulePass`**: najvšeobecnejšia, spracováva program ako celok v podobe modulu
- trieda **`CallGraphSCCPass`**: spracováva graf volaní, má dostupnosť len k funkciám, ktoré sú priamo volané
- trieda **`FunctionPass`**: spracováva každú funkciu zvlášť
- trieda **`LoopPass`**: spracováva každý cyklus funkcie zvlášť
- trieda **`RegionPass`**: podobná `LoopPass`, ale vykonáva pre každý jeden vstupný jeden výstupný región funkcie
- trieda **`BasicBlockPass`**: spracováva každý BB funkcie zvlášť
- trieda **`MachineFunctionPass`**: podobná `FunctionPass`, ale spracováva funkciu v podobe závislej na cieľovej architektúre

Kapitola 5

Implementácia adaptéru

V tejto kapitole je ukázaný postup tvorby samotného adaptéru. Ukážky LLVM inštrukcií a postupy vychádzajú z programového manuálu [11] a príručky [9]. Identifikátory použité v tejto kapitole sa nachádzajú v mennom priestore `llvm`, ak nezačínajú prefixom `CL_`. Tie označujú symboly infraštruktúry Code Listener.

Zvoleným jazykom je C++ podľa normy C++11. Z viacerých dostupných možností (viď 4.3) bolo najvhodnejšie zvoliť si `ModulePass`, pretože naraz spracúva jeden súbor, čo korešponduje s CL objektom.

5.1 Popis tried

Trieda `CLPass` je vlastný adaptér implementovaný ako zásuvný modul. Obsahuje dve tabuľky (použité asociačné pole) uchovávajúce si typy a premenné. Jedná sa o dvojice smerníkov na objekty LLVM a CL vyskytujúce sa v analyzovanom súbore. Nebolo možné použiť číselný identifikátor, nakoľko v LLVM je unikátny len v rámci vlastného kontextu a iba ak je objekt pomenovaný.

```
typedef std::unordered_map<Type *, struct cl_type *> TypeMap;

typedef std::unordered_map<Value *, struct cl_var *> VarMap;
```

Je použitý smerník na hlavnú triedu a ďalej sa používa pretypovanie. LLVM rozširuje RTTI mechanizmu. Je to podobné norme C++98. Operátor `dynamic_cast<>` pracuje iba nad triedami s tabuľku virtuálnych metód, čo u `dyn_cast<>` neplatí. Operátor `isa<>` určuje, či sa jedná o inštanciu danej triedy a na základe jeho výsledku robí `cast<>` statické pretypovanie. Hierarchiu tried viď príloha A.

Prepisuje už spomínané tri virtuálne metódy triedy `ModulePass`: `doInitialization`, `runOnModule` a `doFinalization`. V prvej menovanej sa spracujú parametre príkazového riadku (viď tab. 5.5) a podľa nich sa vytvorí príslušný CL objekt. V druhej metóde sa tento objekt napĺňa, tak ako je to naznačené na obr. 3.2. V cykle sa prechádzajú funkcie. Pre každú funkciu sa spracujú základné bloky a pre každý blok všetky jeho inštrukcie. V poslednej menovanej metóde sa označí CL objekt za validný (prebehne analýza), uvoľnia sa zdroje a upraví sa návratový kód. Ďalej sú v triede definované pomocné metódy začínajúce `handle`, ktorých funkcionalitu popíšem ďalej.

Trieda `CLPrint` slúži len na presmerovanie výpisov na `errs()`.

5.2 Mapovanie typov

V LLVM je použité podobné rozdelenie typov ako v CL (viď tab. 5.1). Sú tu však menšie odchýlky. Napr. nie sú definované uniony, ale iba štruktúry. Takže union je vlastne štruktúra o jednom prvku (najväčšom) a pri načítaní sa používa inštrukcia pretypovania `bitcast`.

Nerozlišuje sa medzi znamienkovým a neznamienkovým celočíselným typom. U znamienkového predpokladá dvojkový doplnok, takže vo výsledku sú všetky čísla v CL znamienkové. Typ `ENUM` je tu chápaný ako celočíselný typ. LLVM pozná viacero typov reálnych čísel, ktoré sú reprezentované jedinou triedou a to `APFloat`, nezávislou na cieľovej architektúre.

Nedovoľuje použiť `void*`. V kóde je reprezentovaný ako `i8*`, čo plug-in prekladá na typ `char*`. Táto nekonzistencia robila problém napr. pri volaní funkcie `free`. Keďže smerníky špceifikujú konkrétne miesto v pamäti, všetky globálne premenné sú vždy typu `pointer`.

Typ sa určuje v metóde `handleType`, kde je každý nový vložený do tabuľky typov a je mu pridelený jedinečný identifikátor. Pre konkrétnejšie určenie sa volajú metódy `handleStructType`, `handleIntegerType` a `handleFunctionType`.

LLVM		**Code Listener**
`Type::TypeID`	IR	`cl_type_e`
`VoidTyID`	`void`	`CL_TYPE_VOID`
`HalfTyID`	`half`	`CL_TYPE_REAL`
`FloatTyID`	`float`	
`DoubleTyID`	`double`	
`X86_FP80TyID`	`x86_fp80`	
`FP128TyID`	`fp128`	
`PPC_FP128TyID`	`ppc_fp128`	
`LabelTyID`	`label`	(inštrukcia)
`MetadataTyID`	–	`CL_TYPE_UNKNOWN`
`X86_MMXTyID`	–	`CL_TYPE_UNKNOWN`
`IntegerTyID`	`iN`	`CL_TYPE_INT`
	`i8`	`CL_TYPE_CHAR`
	`i1`	`CL_TYPE_BOOL`
`FunctionTyID`	`<ret_type> (<params>)`	`CL_TYPE_FNC`
`StructTyID`	`%struct.<name> {<list>}`	`CL_TYPE_STRUCT`
	`%union.<name> {<type>}`	`CL_TYPE_UNION`
`ArrayTyID`	`[<num> x <type>]`	`CL_TYPE_ARRAY`
`PointerTyID`	`<type> *`	`CL_TYPE_PTR`
`VectorTyID`	–	`CL_TYPE_UNKNOWN` (C++)

Tabuľka 5.1: Reprezentácia typov

5.3 Mapovanie inštrukčnej sady

Popis inštrukcií vychádza zo súboru `llvm/IR/Instruction.def`. Niektoré inštrukcie sa mapujú priamo (viď tab. 5.4), iné potrebujú vytvárať ďalšie typy alebo základné bloky. V niektorých prípadoch je podpora pomocných funkcií zo strany CL (napr. `call`). Iné inštrukcie majú funkciu *accessorov*. Pri vysvetľovaní je ukázaný program v troch podobách: v jazyku C, v LLVM IR a ako výstup CL v podobe linearizovaného kódu.

Časť inštrukcií nie je podporovaná. Buď preto, lebo sa nachádzajú v testovaných programoch zriedka, ak vôbec, alebo sa týkajú jazyka C++.

Samotné spracovanie prebieha v metóde `handleInstruction`.

Ukončujúce inštrukcie

Definované triedou `TerminatorInst` (viď tab. 5.2). Uzatvárajú každý základný blok. Inštrukcie `Ret` a `Unreachable` presne zodpovedajú inštrukciám `CL_INSN_RET` a `CL_INSN_ABORT`. Skokovú inštrukciu `Br` je potrebné spracovať zvlášť v metóde `handleBranchInstruction`. Zistí sa, či sa jedná o podmienený skok alebo nie a podľa toho sa použije príslušná inštrukcia v CL.

S prepínačom `-lowerswitch` sa automaticky rozkladá inštrukcia `Switch` na postupnosť BB a inštrukcií `Br`, preto nie je nutná podpora tejto inštrukcie v rámci plug-inu.

LLVM		**Code Listener**
opcode	trieda	`cl_inst_e`
`Ret`	`ReturnInst`	`CL_INSN_RET`
`Br`	`BranchInst`	`CL_INSN_COND` `CL_INSN_JMP`
`Switch`	`SwitchInst`	`CL_INSN_SWITCH`
`IndirectBr`	`IndirectBrInst`	(nepodporuje)
`Invoke`	`InvokeInst`	(ret pre výnimky C++)
`Resume`	`ResumeInst`	(propagácia výnimky C++)
`Unreachable`	`UnreachableInst`	`CL_INSN_ABORT`

Tabuľka 5.2: Reprezentácia ukončujúcich inštrukcií

Φ-inštrukcia

Inštrukcia `PHI` sa môže nachádzať len na začiatku BB a to len v prípade, že sa nejedná o vstupný blok funkcie. Podľa toho, z akého bloku sa skočilo, určuje hodnotu, ktorá sa priradí do registra alebo premennej. Tým v LLVM zabezpečuje SSA formu. Je reprezentovaná triedou `PHINode`. Pozostáva z postupnosti dvojíc $< val_i, bb_i >$, kde val_i predstavuje hodnotu, ktorá sa má použiť, ak bolo skočené z bloku bb_i.

V CL je nutné ju eliminovať. O to sa stará metóda `testPhi`, ktorá je volaná len pred nepodmieneným skokom (u ostatným podporovaných ukončujúcich inštrukcií to nemá význam, lebo sa `PHI` negeneruje). Zistí, či prvá inštrukcia v BB, na ktorý sa skáče, je `PHI`. Ak áno, vloží inštrukciu priradenia s hodnotou viazanou na aktuálny blok.

V príklade nekonečného cyklu sa to týka premennej `%a.0`. Ak sa nachádzame v bloku `%1` a skáčeme na blok `%2`, ktorý začína inštrukciou `PHI`, ešte pred vykonaním daného skoku sa vloží do premennej `%a.0` hodnota `0`. Obdobne pre blok `%2`.

```
int a=0;
while(1) a++;


; <label>:1
  br label %2

; <label>:2                                          ; preds = %2, %1
  %a.0 = phi i32 [ 0, %1 ], [ %r3, %2 ]
  %r3 = add nsw i32 %a.0, 1
  br label %2

        L1:     [int:4]%mF2:a.0 := 0
                goto L2

        L2:     [int:4]%r3 := ([int:4]%mF2:a.0 + 1)
                [int:4]%mF2:a.0 := [int:4]%r3
                goto L2
```

Inštrukcia `Select`

Táto inštrukcia predstavuje ternárny výraz v jazyku C. V CL preň nie je priama podpora a i keď je `Select` chápaný ako neukončujúca inštrukcia, je transformovaný metódou `handleSelectInstruction` na podmienený skok (`CL_INSN_COND`). Vytvárajú sa tri pomocné bloky (v príklade sú to `L2`, `L3` a `L4`) a samotný `Select` sa delí na dve inštrukcie priradenia (`CL_UNOP_ASIGN`) pre register `%r3`.

```
int a = (x>3)? 4 : 6;


%r2 = icmp sgt i32 %r1, 3
%r3 = select i1 %r2, i32 4, i32 6

        L1:     [bool:1]%r2 := ([int:4]%r1 > 3)
                if ([bool:1]%r2)
                        goto L2
                else
                        goto L3

        L2:     [int:4]%r3 := 4
                goto L4

        L3:     [int:4]%r3 := 6
                goto L4
```

Neukončujúce inštrukcie

Inštrukcie s binárnym operátorom (triedy `BinaryOperator`) sú namapované priamo. Stará sa o to metóda `handleBinInstruction`, ktorá si len potrebuje zistiť o aký typ sa jedná (`getCLCode`). Ostatné rozoberiem na nasledujúcich riadkoch.

Inštrukcie pre prácu s pamäťou

Inštrukcia `Alloca` prideľuje pamäť na zásobník. Po ukončení funkcie sa automaticky uvoľní. Pri lokálnych premenných sa tomu dá z časti zabrániť pomocou prepínača `-mem2reg`. Všade, kde to bude možné, bude premenná reprezentovaná ako register. Jej spracovanie prebieha v metóde `handleAllocaInstruction`. V rámci CL je namiesto inštrukcie generovaná funkcia `__alloca`. Pre Predator ju bolo potrebné doplniť do tabuľky modelových externých funkcií v súbore `sl/symbin.cc`.

Inštrukcia `Load` číta z pamäte na príslušnej adrese a `Store` ukladá hodnotu na miesto v pamäti. Obe sú reprezentované inštrukciou `CL_UNOP_ASIGN` s príslušnými *accessormy* dereferencie.

V príklade je načítanie do pamäť ako `%r4 := [%b]` a uloženie hodnoty `[%b] := 3`.

```
int back(void) {
  int b=3;
  return b;
}
```

```
define i32 @back() {
  %b = alloca i32, align 4
  store i32 3, i32* %b, align 4
  %r4 = load i32* %b, align 4
  ret i32 %r4
}
```

```
[int ()():0]back():
                goto L1

        L1:     [int *:8]%mF3:b := [int * ()(int, int):0]__alloca(4, 4)
                [int:4]*%mF3:b := 3
                [int:4]%r4 := [int:4]*%mF3:b
                ret [int:4]%r4
```

Inštrukcie pretypovania

V LLVM sú združené pod triedu `CastInst` a v tabuľke 5.4 sú to inštrukcie od `Trunc` po `IntToPtr`. Spracovávajú sa v metóde `handleCastInstruction`. V CL sú reprezentované inštrukciou `CL_UNOP_ASSIGN`, až na prevod z celočíselného typu na reálny (špeciálna inštrukcia `CL_UNOP_FLOAT`).

Ak je operand `%a` smerník na union, použije sa inštrukcia pretypovania `BitCast`. Tento prípad bližšie rozoberiem v časti o *accessoroch*.

```
%b = <opcode> <type1> %a to <type2>       ; [type2]%b := [type1]%a
```

Porovnávacie inštrukcie

Jedná sa o binárne inštrukcie triedy `CmpInst` s tým, že okrem dvoch operandov majú aj predikát <cond>. Ten určuje o aké pretypovanie sa jedná (viď tab. 5.3). Oba operandy musia byť rovnakého typu a to buď celočíselného alebo reálneho. Spracovanie prebieha obdobne ako pri binárnych inštrukciách, len kód sa zistí pomocou `getCLCodePredic` volanej z metódy `handleCmpInstruction`.

```
%result = icmp <cond> <ty> <op1>, <op2>   ; %result je typu bool (i1)
```

CmpInst::Predicate		CL_INSN_BINOP
ICmp	FCmp	cl_binop_e
eq	oeq, ueq	CL_BINOP_EQ
ne	one, une	CL_BINOP_NE
ugt, sgt	ogt, ugt	CL_BINOP_GT
uge, sge	oge, uge	CL_BINOP_GE
ult, slt	olt, ult	CL_BINOP_LT
ule, sle	ole, ule	CL_BINOP_LE
	false, true ord, uno	CL_BINOP_BAD

Tabuľka 5.3: Argumenty inštrukcií `ICmp` a `FCmp`

Inštrukcia `Call`

Pri vkladaní do CL objektu sa nepoužíva priamo inštrukcia `CL_INSN_CALL`, ale postupnosť volaní `insn_call_open`, `insn_call_arg` a `insn_call_close` (znázornené na obr. 3.2). Pričom argumenty sú voliteľné. V LLVM je volanie funkcie reprezentované triedou `CallInst`, ktorá obsahuje vektor operandov. Spracováva sa v metóde `handleCallInstruction`.

Ak by sa jednalo o jednoduché volanie funkcie, najpr sa zavolá `insn_call_open` pre volanú funkciu (vráti metóda `getCalledValue` v `CallInst`) a následne sa pomocou cyklu budú vkladať operandy ako argumenty funkcie. Na záver sa ukončí inštrukcia `Call` s `insn_call_close`. Výsledok je uvedený na príklade volania funkcie `suma`.

```
suma(1, 3);
```

```
%r1 = call i32 @suma(i32 1, i32 3)
```

```
[int:4]%r1 := [int ()(int, int):0]suma(1, 3)
```

Problém nastáva, ak je argumentom konštantný výraz (viď ďalšia sekcia). S použitím načrtnutého postupu by sa vytvorila inštrukcia obsluhujúca konštantný výraz vo vnútri volania funkcie, čo by viedlo na chybu. Rieši sa to s použitím pomocného pola, ktoré si uchová všetky argumenty, a až po jeho naplnení sa otvorí volanie funkcie, argumenty sa načítajú z pola a volanie sa ukončí.

V druhom príklade sa to týka reťazca "%d", ktorý je v LLVM predstavovaný ako pole znakov na globálnej úrovni (avšak viditeľný len z našej funkcie). Inštrukcia `GetElementPtr` sa zavolá ako prvá a jej výsledok `%r1` sa použije ako argument pri volaní funkcie `printf`.

```
printf("%d", 4);
```

```
; globalne symboly
  @.str2 = private unnamed_addr constant [3 x i8] c"%d\00", align 1
; vo funkcii main
  %r2 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
                                ([3 x i8]* @.str2, i32 0, i32 0), i32 4)
```

```
[char *:8]%r1 := [char *:8]&*%mG0:.str2[0]
[int:4]%r2 := [int ()(char *):0]printf([char *:8]%r1, 4)
```

Operandy a accessory

Operand môže byť reprezentovaný ako globálna premenná, funkcia, globálny alias, lokálna premenná, návestie, konštanta či konštantný výraz. O správnu identifikáciu sa stará metóda `handleOperand`.

Globálne premenné sú reprezentované triedou `GlobalVariable` a musia byť inicializované. V prípade, že sa jedná o externú premennú, ktorá je iba deklarovaná, nemusí byť inicializovaná. Globálne premenné môžu obsahovať zoznam príslušných inicializačných inštrukcií. Aktuálne je podporované iba priradenie konštanty a nie určenie pomocou konštantného výrazu. Globálna premenná je vždy smerník na svoju skutočnú hodnotu.

Lokálne premenné spolu s globálnymi premennými sú ďalej spracovávané metódou `handleVariable`. Každá nová premenná je vkladaná do tabuľky premenných a je jej pridelený jedinečný identifikátor v rámci celého programu, nie len v rámci funkcie. Nezáleží na tom, či sa jedná o skutočnú premennú alebo nepomenovaný register, lebo vyhľadávanie prebieha podľa adresy.

Funkcia sa spracováva metódou `handleFncOperand` a má návratový typ, prípadne zoznam argumentov. Ak sa nejedná o deklaráciu (v prípade externej funkcie), obsahuje minimálne jeden BB.

V prípade **aliasu**, resp. druhého mena premennej, funkcie alebo i iného aliasu, je rekurzívne volaná metóda `handleOperand` nad hodnotou, ktorú vráti metóda `getAliasee` triedy `GlobalAlias`.

Všetky **konštanty** (cháp literály) rieši metóda `handleConstant`, kde sa pretypujú na príslušnú podtriedu.

Ak sa jedná o **konštantný výraz**, je to mierne zložitejšie. Predtým, než sa vráti správny operand, je volaná metóda `handleInstruction` pre inštrukciu, ktorú reprezentuje daný konštantný výraz (získaná metódou `getAsInstruction` triedy `ConstantExpr`). Potom ako operand figuruje výsledok tejto inštrukcie.

CL nereprezentuje **návestie** (trieda `BasicBlock`) ako operand, ale iba ako reťazec. Keďže LLVM nie vždy pomenováva svoje návestia, v rámci programu sa generujú unikátne mená, ktorými sa príslušné BB pomenujú. To zaručuje, že pre návestie, na ktoré sa odkazovala napr. skoková inštrukcia, bude v budúcnosti analyzované pod tým istým menom (zavolaná funkcia `bb_open`).

V CL môžeme meniť sémantiku operandu pomocou *accessorov*. V LLVM sú na to vyhradené vlastné inštrukcie. Jedná sa o `BitCast`, `GetElementPtr` (GEP), `ExtractValue`, `InsertValue` a pár inštrukcií pre prácu s vektormi (v rámci C++).

Inštrukcie `ExtractValue` a `InsertValue` pracujúce nad štruktúrou alebo polom, aktuálne, nie sú podporované. Ani raz sa nevyskytli v testoch. Rozdiel oproti GEP je ten, že nepracujú so smerníkom, ale priamo s typom. Inštrukcia `ExtractValue` sa zavolá, napr. ak je návratová hodnota z funkcie štruktúra, pričom väčšinou (a tak je to aj v testoch) sa vracia smerník na štruktúru.

Inštrukcia `BiteCast` nemení obsah pamäte, len pohľad naň, t.j. typ. V prípade unionu sa v rámci CL vytvárajú *accessory*. Prvý je dereferencia, lebo v skutočnosti je na vstupe smerník na union, potom prístup k prvku a následne referencia, lebo sa vracia smerník na prvok (adresa sa nemení).

Na príklade je vidieť, že LLVM naozaj chápe union ako štruktúru o najväčšom prvku, teda `double`.

```
union u {int i; double d; char c;};
// vo funkcii main
union u uno;
uno.i = 3;
```

```
; globalne symboly
  %union.u = type { double }
; vo funkcii main
  %r2 = bitcast %union.u* %uno to i32*
  store i32 3, i32* %r2, align 4

[int *:8]%r2 := [int *:8]&%mF7:uno->[+0]<anon_item>
[int:4]*%r2 := 3
```

Veľmi dôležitou inštrukciou je `GetElementPtr`. Nikdy nepristupuje k pamäti. Vykonáva iba výpočet smerníka. Na rozdiel od `ExtractValue` je prvým operandom smerník, ktorý musí byť indexovateľný. Pri štruktúrach je index vždy konštanta. Pri poliach to môže byť aj premenná. Všetky náležitosti sú dobre popísané v rámci LLVM dokumentácie[1].

Na prvom príklade GEP inštrukcie je ukázaný prístup k prvkom pola. Ako prvý parameter je smerník na pole, druhý značí dereferenciu a tretí, že sa jedná o druhý prvok pola (klasicky sa čísluje od nuly). Návratovou hodnotou je adresa, preto sa použije ešte referencia.

```
char arr[3];
char c = arr[1];
```

```
%arr = alloca [3 x i8], align 1
%r1 = getelementptr inbounds [3 x i8]* %arr, i32 0, i64 1

[char []*:8]%mF7:arr := [char []* ()(int, int):0]__alloca(3, 1)
[char *:8]%r1 := [char *:8]&*%mF7:arr[1]
```

Naviažeme na prvý príklad s polom, no tentoraz bude indexované podľa offsetu. Prvý GEP nám vráti smerník na prvý prvok pola a druhý GEP sa posunie o konštantu 2.

```
// nad rovnakym polom arr
char d = arr + 2;
```

```
%r2 = getelementptr inbounds [3 x i8]* %arr, i32 0, i32 0
%r3 = getelementptr inbounds i8* %2, i64 2

[char *:8]%r2 := [char *:8]&*%mF7:arr[0]
[char *:8]%r3 := [char *:8]%r2<+2>
```

[1] *The Often Misunderstood GEP Instruction*, http://llvm.org/docs/GetElementPtr.html

U štruktúr je to obdobné. Prvý operand predstavuje smerník na štruktúru, druhý dereferenciu a tretí index položky v štruktúre. LLVM si neuchováva názvy položiek. V našom prípade je výsledkom operátor ->, ktorý predstavuje `(*%r1).[+0]<anon_item>`.

```
struct S { int i; };
// vo funkcii main
struct S *ps;
int i = ps->i;
```

```
; globalne symboly
  %struct.S = type { i32 }
; vo funkcii main
  %ps = alloca %struct.S*, align 8
  %r1 = load %struct.S** %ps, align 8
  %r2 = getelementptr inbounds %struct.S* %r1, i32 0, i32 0
  %r3 = load i32* %r2, align 4
  store i32 %r3, i32* %i, align 4

[struct struct.S **:8]%mF5:ps := [struct struct.S ** ()(int, int):0]__alloca(8, 8)
[struct struct.S *:8]%r1 := [struct struct.S *:8]*%mF5:ps
[int *:8]%r2 := [int *:8]&%r1->[+0]<anon_item>
[int:4]%r3 := [int:4]*%r2
[int:4]*%mF8:i := [int:4]%r3
```

LLVM		Code Listener	
opcode	trieda	cl_inst_e	cl_unop_e
Add, FAdd	BinaryOperator	CL_INSN_BINOP	CL_BINOP_PLUS
Sub, FSub			CL_BINOP_MINUS
Mul, FMul			CL_BINOP_MULT
UDiv, SDiv			CL_BINOP_TRUNC_DIV CL_BINOP_EXACT_DIV
FDiv			CL_BINOP_RDIV
URem, SRem FRem			CL_BINOP_TRUNC_MOD (zvyšok po delení REAL)
Shl			CL_BINOP_LSHIFT
LShr (logic.) AShr (aritm.)			CL_BINOP_RSHIFT
And Or Xor			CL_BINOP_BIT_XOR CL_BINOP_BIT_IOR CL_BINOP_BIT_AND
Alloca Load Store	AllocaInst LoadInst StoreInst	CL_INSN_UNOP	CL_UNOP_ASSIGN
GetElementPtr	GetElementPtrInst		(*accessor*)
Fence	FenceInst	CL_INSN_NOP	
Trunc ZExt SExt FPToUI FPToSI	TruncInst ZExtInst SExtInst FPToUIInst FPToSIInst	CL_INSN_UNOP	CL_UNOP_ASSIGN
UIToFP SIToFP	UIToFPInst SIToFPInst		CL_UNOP_FLOAT
FPTrunc FPExt PtrToInt IntToPtr	FPTruncInst FPExtInst PtrToIntInst IntToPtrInst		CL_UNOP_ASSIGN
BitCast	BitCastInst		(*accessor*)
ICmp, FCmp	ICmpInst, FCmpInst	CL_INSN_BINOP	(viď tab. 5.3)
PHI	PHINode		(Φ-inštrukcia)
Call	CallInst	CL_INSN_CALL	
Select	SelectInst	CL_INSN_COND	
VAArg	VAArgInst		(nepodporuje)
ExtractElement InsertElement ShuffleVector	ExtractElementInst InsertElementInst ShuffleVectorInst		(vektor C++)
ExtractValue InsertValue	ExtractValueInst InsertValueInst		(*accessor*)
LandingPad	LandingPadInst		(interná záležitosť LLVM)

Tabuľka 5.4: Reprezentácia neukončujúcich inštrukcií

5.4 Registrácia zásuvného modulu

Spúšťanie pomocou opt

Použije sa šablóna `RegisterPass<t>`. Tá zaregistruje plug-in do internej databázy, ktorú spravuje `PassManager`. Primárnym manažérom pre moduly je `MPPassManager`.

```
RegisterPass<CLPass> X("cl", "Code␣Listener␣Pass");
```

Následne sa analyzovaný zdrojový súbor preloží pomocou Clang do bitecode podoby a predá sa ako vstup optimalizéru. Výhodou je, že sa ľahko povolia ďalšie optimalizácie, napr. pre rozklad `switch` inštrukcií alebo zamedzenie ukladaniu na zásobník.

```
clang -S -emit-llvm source.c -o - | opt -lowerswitch -mem2reg -load libcl.so -cl
```

Spúšťanie pomocou clang

Plug-in sa opäť zaregistruje u `PassManager`-a. Registrácia je podmienená úrovňou 0 – bez optimalizácií. Metóda `RegisterStandardPasses` vytvorí statickú inštanciu našej triedy. To dovolí použiť privátnu metódu, ktorú `PassManagerBuilder` použije na pridanie nášho priechodu. Čo zaručí automatické spúšťanie plug-inu už v rámci Clangu.

```
static void registerMyPass(const PassManagerBuilder &pb, PassManagerBase &pm)
{
        if (pb.OptLevel == 0 && pb.SizeLevel == 0)
                pm.add(new CLPass);
}

static RegisterStandardPasses
    RegisterMyPass(PassManagerBuilder::EP_EnabledOnOptLevel0 , registerMyPass);
```

V tomto prípade sa zdrojový súbor prekladá iba Clangom.

```
clang -Xclang -load -Xclang libcl.so source.c
```

5.5 Ovládanie programu

Aktuálne sa program spúšťa ako zásuvný modul optimalizéru `opt`. Nasledujúca tabuľka popisuje použitie jednotlivých parametrov príkazového riadku. Program očakáva na štandardnom vstupe alebo ako posledný parameter zdrojový súbor v bitecode podobe.

Parameter	**Popis**
`-help`	vypíše ovládanie programu.
`-args=peer_args`	predá argumenty analyzátoru
`-dry-run`	nespúšťa analyzátor
`-dump-pp[=filename]`	vypíše linearizovaný kód
`-dump-types`	pridá informácie o typoch
`-gen-dot[=filename]`	generuje CFG
`-pid-file=filename`	zapíše PID do súboru
`-preserve-ec`	analýza neovplyvní návratový kód
`-type-dot=filename`	generuje graf typov
`-verbose=uint`	nastaví úroveň výpisu ladiacich správ

Tabuľka 5.5: Parametre plug-inu

Kapitola 6

Výsledky testovania

Samotné testovanie prebiehalo v dvoch fázach. Prvá predstavovala vytváranie CL objektu, čiže preklad LLVM IR do medzikódu infraštruktúry Code Listener. Druhou fázou bolo spúšťanie analyzátora. Postup bude rozobraný na nasledujúcich riadkoch.

Cieľom testov v prvej fázi bolo odhaliť nekorektné volania funkcií pri vytváraní CL objektu. Napríklad, že pri deklarácií funkcie sa vôbec nevolá dvojica `fnc_open` a `fnc_close`. Alebo, ak sa nachádza konštantný výraz vo volaní funkcie, nie je prípustné vkladať inštrukciu pomocou `insn`, ale iba argumenty pomocou `insn_call_arg`. V podstate, že implementácia odpovedá diagramu predstavujúceho postupnosti volaní funkcií napĺňajúcich CL objekt na obrázku 3.2.

Testované súbory boli korektné programy napísané v jazyku C a C++ (bez použitia výnimiek, streamov, vektorov a podobne). Ich úlohou bolo overiť, či sa programové konštrukcie prekladajú správne. Porovnával sa výstup adaptéru v podobe linearizovaného kódu s výstupom adaptéru pre GCC. Generovali sa grafy riadenia toku. Na záver tejto fázy sa spúšťali regresné testy dodávané s Code Listenerom. Množina testov sa spúšťala skriptom `./tests.sh cl` v adresári `src` na priloženom CD. Úspešne prešlo 41 testov z 42, pričom súbor `pt-0906.c` obsahoval volanie intrinstic inštrukcie `memcpy`, ktorá nie je podporovaná.

Potom sa prešlo k integrácií s už existujúcimi analyzátormi a to nástrojmi Predator a Forester. Aby sa správne zostavila zdieľaná knižnica, bolo nutné pridať vstupný bod pre linker a to `void plugin_init(void){}` na koniec súboru z dôvodu načítaniu externých symbolov. Ďalším problém bolo, keď Predator nebol schopný rozpoznať modelové externé funkcie (ako `abort`, `__VERIFIER_plot`, či `___sl_error`), pretože nebola korektne nastavená návratová hodnota (týkalo sa to void operandu).

Následne sa pristúpilo k testovaniu. Opäť sa spúšťal skript `./tests.sh sl`, ktorý spustil regresné testy dodávané s nástrojom Predator. V testovanej množine bolo 279 testovacích prípadov, ktoré sa spúšťali trikrát, vždy s iným parametrom. Testy zahrňovali smerníkovú aritmetiku, prácu s jedno-/dvojsmerne viazanými zoznamami (SLL/DLL), zoznamami používanými v Linuxovom jadre, triediace algoritmy nad zoznamami a podobne. Chyby v programoch sa týkali nesprávnej manipulácie s dynamicky alokovanou pamäťou. Obsahovali od triviálnych chýb ako je neinicializovaná premenná, dvojnásobné uvoľnenie pamäti a podobne, až po špecifické chyby.

Vo výsledku úspešne prešlo 6% zo všetkých spustených testov. Jednalo sa triviálne prípady, ktoré zahrňovali neplatnú dereferenciu, `NULL` hodnotu, neplatné volanie `free()` a podobne (súbory `test-00{02, 03, 20, 25}.c`). Potom to boli testy, ktoré kontrolo-

vali prístup k položkám zložených typov, ako napr. anonymné uniony v štruktúrach (`test-0091.c`) či rušenie SLL (`test-000{5,6}.c`) Ďalej to bol test na nekonečnú rekurziu (`test-0041.c`) a podobne.

Niektoré testy boli označené za neúspešné, i keď v skutočnosti prešli. Jednou z príčin bolo zlé filtrovanie výstupu pri testoch `test-0{004, 013, 177, 178, 179, 316}.c`. Zostávalo varovanie o nedosiahnuteľnom návestí, ktoré generuje CL, pričom na výstupe sa očakávali hlásenia produkované Predatorom. Ďalším dôvodom, prečo neprešli niektoré testy, bola nepodporovaná lokalizácia inštrukcií. To spôsobilo, že súbor, do ktorého sa ukladal graf využitia hromady sa generoval s nesprávnym menom. Týkalo sa to testov `test-0{062, 070, 071, 166, 170, 194}.c` a testov prebraných z nástroja Forester (`test-05{00, 01, 04, 05, 09, 10, 12, 15, 18}.c`), ktoré krásne graficky ukazujú alokáciu zložitých štruktúr. Na rozdiel od výstupu GCC plug-inu, je pridaná aj alokácia na zásobník spôsobená volaním funkcie `__alloca`. Posledným dôvodom je v prípade testu `0015` vypršaný časový limit, ktorý bol nastavený pre iné testovacie prípady.

Najčastejším dôvodom, prečo neprešli testy, bola nesprávna inicializácia globálnych premenný. Pri vykonávaní drvivej väčšiny súborov (napr. `test-00{01, 12, 64}.c`) program spadol na signál SIGSEGV pri volaní inicializéru. Najčastejšie je to spôsobené volaním funkcie `__VERIFIER_plot`, ktorá má ako parameter reťazcový literál. Ukázalo sa, že toto nie je dobre doriešené v rámci plug-inu. Ďalej program spadne už pri samotnom spracovávaní LLVM IR, ak sa jedná o nepriame volanie funkcie (súbory `test-000{7, 8}.c`).

Opakom SIGSEGV je, že sa program zacyklí. U 16 súborov (prípady `0161` až `0164`, `0225` až `0233` a `0193`) bola symbolická exekúcia ukončená vypršaním časového limitu. Cyklenie začalo po tom, čo sa opravila chyba „error: dereferencing object of size 8B out of bounds“ vyvolaná zlým nastavením veľkosti smerníka.

Tri testy (`test-046{4, 6, 7}.c`) vôbec Clang nepripustil k prekladu, lebo nepoznal funkciu `__builtin_va_arg_pack`. V rámci všetkých súborov, obsahoval vytvorený LLVM IR kód 107-krát intrinsic inštrukcie, ktoré nie sú podporované (najčastejšie to viedlo na neplatnú dereferenciu, keďže sa v rámci plug-inu ignoruje). Jednalo sa o `memset`, `memcpy` a `expect`.

Vo výsledku prešlo 14% testov dodaných k Predatoru (čo je 114 z 837).

Analýzu kódu pomocou nástroja Forester sa bohužiaľ nepodarilo spustiť.

Kapitola 7

Budúci vývoj

Modul je stále vo vývoji, preto v tejto kapitole priblížim aktuálny stav implementácie.

Nový adaptér pre Clang/LLVM prekladač podporuje analýzu zdrojových súborov písaných v štýle C. Čiže prekladá C aj C++ programy, lebo LLVM používa jednotnú reprezentáciu inštrukcií pre všetky jazyky. Do budúcna sa dá urobiť podpora aj pre moderné C++. Ako je podpora výnimiek, typu vektor a inštrukcií nad ním (naznačené v kapitole 5). Samozrejme za predpokladu rozšírenia infraštruktúry Code Listener.

Čo sa týka výpisov, nie je podporovaná lokalizácia chýb v rámci zdrojového kódu. Je to z dôvodu, že kód sa na vstup optimalizéra dostáva v bitecode podobe, ktorá neobsahuje tieto informácie. Ak by sa Clang volal s parametrom `-g`, do výstupného LLVM IR by sa pridali potrebné metadáta. Avšak to generuje intrinstic inštrukciu, ktorá sa pretaví ako varovanie pri mapovaní inštrukcií. Jedná sa o `dbg`, takže by bolo možné ju v rámci analýzy, takpovediac, prehliadnuť. Toto sa môže vyriešiť v blízkej budúcnosti.

V niektorých prípadoch je nepraktické spúšťanie analýzy cez optimalizér (napr. ak sa nejedná o zautomatizované testy, ale priamo analýza jedného súboru z príkazového riadku). Riešením by bolo použiť registráciu priechodu ako je to načrtnuté v sekcií 5.4 a spúšťať zásuvný modul priamo Clangom. Aktuálne to nie je možné, lebo nie je podpora pre `Switch`, ktorý sa eliminuje vďaka prepínaču `-lowerswitch`. Nie je ťažké ho prepísať do podoby akceptovateľnej pre CL. Na druhú stranu, spúšťanie pomocou optimalizéru dovolí pridávanie ďalších optimalizácií, ktoré sa vykonajú ešte pred vlastnou analýzou.

Modul nerieši inline assembler (ktorý pri analýze tvaru nie je využívaný) a volanie intrinstic inštrukcií, ktoré v LLVM IR reprezentujú zväčša niektoré funkcie štandardnej C knižnice. V zdrojovom súbore je to napr. definícia a zároveň inicializácia polí a štruktúr, priraďovanie štruktúr rovnakého typu alebo priamo volania funkcií zo štandardnej knižnice C. Dalo by sa to obísť generovaním inštrukcie `call`, ako sa to riešilo v prípade inštrukcie `alloca`.

Zatiaľ nie sú podporované úplne všetky konštrukcie, najmä tie atypické, nakoľko je jazyk C rozsiahli.

Ako ukázali testy, v najbližšej dobe by bolo dobré sa zamerať na korektnú inicializáciu globálnych premenných. Vrátane reťazcových literálov.

```
char *p ="Analyza sa spusta";

@.str = private unnamed_addr constant [18 x i8] c"Analyza sa spusta\00", align 1
@p = global i8* getelementptr inbounds ([18 x i8]* @.str, i32 0, i32 0), align 8
```

V podstate sa jedná o globálnu premennú typu pole znakov (nejedná sa o smerník), avšak viditeľnú len z príslušnej funkcie. A ak sa jedná o takú globálnu premennú, potom je inicializovaná konštantným výrazom (viď príklad), čo by viedlo na nasledujúcu postupnosť inicializačných inštrukcií: priradenie konštanty a priradenie pomocou GEP. A pri inštrukcií GEP sa musí doriešiť, ako sa bude rozpoznávať prístup k reťazcu od prístupu k poľu.

Kapitola 8

Záver

Cieľom práce bolo zoznámiť sa s prekladovým systémom Clang/LLVM a preštudovať vnútornú reprezentáciu kódu, ktorú používa. Ďalej sa zoznámiť so statickými analyzátormi Predator a Forester a pochopiť framework Code Listener. Nadobudnuté vedomosti následne pretaviť do podoby LLVM adaptéru pre infraštruktúru Code Listener. Implementovaný adaptér porovnať s už existujúcim pre prekladač GCC.

Podarilo sa vytvoriť zásuvný modul optimalizéru, ktorý prevádza zdrojový program z vnútornej reprezentácie prekladača LLVM do medzikódu Code Listener. Nepodarila sa síce úplná integrácia s oboma analyzátormi, no ponúka preklad z LLVM IR kódu do podoby CL. Je schopný generovať grafy riadenia toku a graf použitých typov. Nie je pravdou, že by nedokázal analyzovať zdrojové súbory, len musia byť zo špecifickej množiny (ako bolo rozobrané v kapitole 7). V tom prípade pracuje korektne. Tvorí pevný základ, na ktorom sa dá stavať.

Vytvorený adaptér určite nepokrýva všetky konštrukcie a prvky jazyka C potrebné k analýze dynamicky alokovaných štruktúr a práce s nimi. Ak by sme zamedzili použitie globálnych premenných, výsledky by boli lepšie. Taktiež mu robia problém aj nepriame volania funkcií. Napriek tomu z dodávaných testov pre Predator prejde slušných 14%. Z tohto ohľadu je už existujúci GCC plug-in dobre odladeným nástrojom.

Na druhú stranu, LLVM priechod bol písaný takým štýlom, aby bolo jednoduché zakomponovať rozšírenia potrebné pre spracovanie C++ programov. Úspechom práce je, že je schopný akceptovať na vstupe aj triedy s nevirtuálnymi metódami obsahujúce premenné. Dokáže z nich generovať príslušné grafy a poslať na ďalšiu analýzu.

Dá sa povedať, že sa podarilo splniť zadanie. Jedná sa o projekt vo vývoji publikovaný pod GNU GPLv3 licenciou, tak ako aj vlastný Code Listener a spomínané analyzátory. Aktuálna podoba zdrojových kódov je verejne dostupná v git repozitári Kamila Dudku na adrese `https://github.com/kdudka/predator/tree/llvm`.

Slovník pojmov

API **rozhranie pre programovanie aplikácií** (angl. *application programming interface*)

ARTMC *abstract regular tree model checking* je technika verifikácie nekonečných stavových systémov s použitím konečných TA k reprezentácií potenciálne nekonečnej množiny dosiahnuteľných konfigurácií systému

AST **abstraktný syntaktický strom** (angl. *abstract syntax tree*) reprezentuje kód počas syntaktickej analýzy programu

BB **základný blok** (angl. *basic block*) je sekvencia maximálneho počtu inštrukcií, ktoré sa musia vykonať ako celok, bez toho, aby sa skočilo do iného BB rovnakej funkcie, pričom skokové inštrukcie môže obsahovať iba na konci [5, str. 529]

CFG **graf riadenia toku** (angl. *control flow graph*) je graf reprezentujúci program (jednu funkciu), v ktorom uzly sú BB

dereferencia operátor sprístupňujúci obsah uložený na adrese, na ktorú ukazuje

FA **lesný automat** (angl. *forest automata*) je tvorený n-ticami TA, ktoré kódujú množiny grafov haldy, pričom ich listy môžu spätne odkazovať na korene týchto komponent

framework softwarová štruktúra poskytujúca základnú funkcionalitu vývojárom pri programovaní

funkcia spätného volania (angl. *callback*) oddeľuje vykonávanie funkcie volaného knižnice od volajúceho knižnice

GIMPLE 3-adresná reprezentácia medzikódu používaná v prekladači GCC `http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`

GCC prekladač jazyka C *GNU C Compiler* alebo súbor kompilátorov *GNU Compiler Collection* `http://gcc.gnu.org/`

GPLv3 *GNU General Public License* verzie 3 je licencia slobodného softwaru `http://gplv3.fsf.org/`

invariant cyklu podmienka, ktorá musí byť splnená pred a po vykonaní každého cyklu

JIT *Just In Time* metóda prekladu programu

load-store architektúra pre bežné inštrukcie je nutné načítať dáta (`LOAD`) z pamäte do registru a výsledok uložiť (`STORE`) do pamäte

LTO *link-time optimization* je optimalizácia až počas zostavovania programu

Model Checking overovanie vlastností systematickým generovaním a skúmaním stavového priestoru daného systému

NCSA *University of Illinois/NCSA Open Source License* je tolerantná licencia slobodného softwaru, ktorá umožňuje jeho distribúciu pod inou licenciou `http://opensource.org/licenses/NCSA`

NP problém vypočítateľný nedeterministickým Turingovým strojom v polynomiálnom čase

plug-in zásuvný modul programu

predikát výraz, ktorého výsledkom je pravdivostná hodnota

RISC **redukovaná inštrukčná sada** (angl. *reduced instruction set computer*) obsahuje jednoduché inštrukcie (vykonávané zväčša v 1 takte)

RTTI **typová identifikácia za behu** (angl. *Run-Time Type Identification*)

SAT Solving riešenie NP-úplneho problému splniteľnosti boolovských formulý (angl. *Boolean Satisfiability Problem*)

SIGSEGV signál generovaný pri porušení ochrany pamäti

SMG **symbolický graf pamäte** (angl. *symbolic memory graph*) je tvorený dvomi typmi uzlov: objektami O (alokovaná pamäť) a hodnotami V (adresy), ktoré sú spojené hranami $O \to V$ „má hodnotu“ a $V \to O$ „ukazuje na“

SSA Static single assignment forma

SV-COMP *Competition on Software Verification* `http://sv-comp.sosy-lab.org/`

symbolická exekúcia vykonáva program, nie s dátami, ale množinami hodnôt, ktoré sú popísané formou logiky alebo automatmi

TA **stromový automat** (angl. *tree automata*) je druh konečného automatu zložený zo stromových štruktúr

Theorem Proving deduktívna metóda podobná matematickému dokazovaniu

Turingov stroj teoretický model pre popis formálnych jazykov

VeriFIT výskumná skupina automatizovanej analýzy a verifikácie na FIT, VUT

Literatúra

[1] Berdine, J., Calcagno, C., Cook, B. et al. Shape Analysis for Composite Data Structures. In Damm, W. a Hermanns, H. (ed.). *Computer Aided Verification*. [b.m.]: Springer Berlin Heidelberg, 2007. S. 178–192. Lecture Notes in Computer Science, sv. 4590. Dostupné na: http://dx.doi.org/10.1007/978-3-540-73368-3_22. ISBN 978-3-540-73367-6.

[2] Dudka, K., Peringer, P. a Vojnar, T. *Byte-Precise Verification of Low-Level List Manipulation* [online]. 2013 [cit. 2014-04-27]. 48 s. Tech. rep. Dostupné na: http://www.fit.vutbr.cz/research/view_pub.php?id=10330.

[3] Dudka, K., Peringer, P. a Vojnar, T. Predator: A Shape Analyzer Based on Symbolic Memory Graphs. In Ábrahám, E. a Havelund, K. (ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. [b.m.]: Springer Berlin Heidelberg, 2014. S. 412–414. Lecture Notes in Computer Science, sv. 8413. Dostupné na: http://dx.doi.org/10.1007/978-3-642-54862-8_33. ISBN 978-3-642-54861-1.

[4] Holík, L., Lengál, O., Rogalewicz, A. et al. Fully Automated Shape Analysis Based on Forest Automata. In Sharygina, N. a Veith, H. (ed.). *Computer Aided Verification*. [b.m.]: Springer Berlin Heidelberg, 2013. S. 740–755. Lecture Notes in Computer Science, sv. 8044. Dostupné na: http://dx.doi.org/10.1007/978-3-642-39799-8_52. ISBN 978-3-642-39798-1.

[5] Dudka, K., Peringer, P. a Vojnar, T. An Easy to Use Infrastructure for Building Static Analysis Tools. *Lecture Notes in Computer Science*. 2012, roč. 2012, č. 6927. S. 527–534. Dostupné na: http://www.springerlink.com/content/750240l1tk386572/. ISSN 0302-9743.

[6] Lattner, C. a al et. *The LLVM Compiler Infrastructure* [online]. 2007 [cit. 2014-04-27]. Dostupné na: http://llvm.org/.

[7] Plessl, J.-P. D. C. *Introduction to the LLVM Compiler Framework* [online]. 1.1.0. 20012-04-24 [cit. 2014-05-02]. Dostupné na: http://homepages.uni-paderborn.de/plessl/lectures/2012-Codesign/slides/02-Compiler-LLVM.pdf.

[8] *Clang: a C language family frontend for LLVM* [online]. [cit. 2014-05-04]. Dostupné na: http://clang.llvm.org/.

[9] *LLVM Language Reference Manual* [online]. 2003, 2014-05-01 [cit. 2014-05-02]. Dostupné na: http://llvm.org/docs/LangRef.html.

[10] Zhao, J., Nagarakatte, S., Martin, M. M. K. et al. Formal Verification of SSA-based Optimizations for LLVM. *SIGPLAN Not.* June 2013, roč. 48, č. 6. S. 175–186. Dostupné na: http://doi.acm.org/10.1145/2499370.2462164. ISSN 0362-1340.

[11] *LLVM Programmer's Manual* [online]. 2003, 2014-05-01 [cit. 2014-05-02]. Dostupné na: http://llvm.org/docs/ProgrammersManual.html.

[12] *Writing an LLVM Pass* [online]. 2003, 2014-05-01 [cit. 2014-05-02]. Dostupné na: http://llvm.org/docs/WritingAnLLVMPass.html.

Dodatok A

Hierarchia tried v LLVM

Hierarchia tried[1] predstavujúca objekty LLVM IR, kde každý prvok (funkcia, globálna premenná, základný blok, inštrukcia a pod.) má pridelený svoj typ. Symboly sa nachádzajú v mennom priestore `llvm`.

- ▼ `Value`
  - ► `Argument`
  - ► `BasicBlock`
  - ► `InlineAsm`
  - ► `MDNode`
  - ► `MDString`
  - ► `User`
    - ▽ `Constant`
    - ▽ `Instruction`
    - ▽ `Operator`

- ▼ `Type`
  - ► `CompositeType`
    - ▽ `SequentialType`
      - ▷ `ArrayType`
      - ▷ `PointerType`
      - ▷ `VectorType`
    - ▽ `StructType`
  - ► `FunctionType`
  - ► `IntegerType`

[1] prevzaté z LLVM API dokumentácie http://www.llvm.org/docs/doxygen/html/hierarchy.html

- Constant
  - BlockAddress
  - ConstantAggregateZero
  - ConstantArray
  - ConstantDataSequential
    - ConstantDataArray
    - ConstantDataVector
  - ConstantExpr
    - BinaryConstantExpr
    - CompareConstantExpr
    - ExtractElementConstantExpr
    - ExtractValueConstantExpr
    - GetElementPtrConstantExpr
    - InsertElementConstantExpr
    - InsertValueConstantExpr
    - SelectConstantExpr
    - ShuffleVectorConstantExpr
    - UnaryConstantExpr
  - ConstantFP
  - ConstantInt
  - ConstantPointerNull
  - ConstantStruct
  - ConstantVector
  - GlobalValue
    - Function
    - GlobalAlias
    - GlobalVariable
  - UndefValue

- Instruction
  - AtomicCmpXchgInst
  - AtomicRMWInst
  - BinaryOperator
  - CallInst
    - IntrinsicInst
      - DbgInfoIntrinsic
        - DbgDeclareInst
        - DbgValueInst
      - MemIntrinsic
        - MemSetInst
        - MemTransferInst
          - MemCpyInst
          - MemMoveInst
      - VACopyInst
      - VAEndInst
      - VAStartInst
  - CmpInst
    - FCmpInst
    - ICmpInst
  - ExtractElementInst
  - FenceInst
  - GetElementPtrInst
  - InsertElementInst
  - InsertValueInst
  - LandingPadInst
  - PHINode
  - SelectInst
  - ShuffleVectorInst
  - StoreInst
  - TerminatorInst
    - BranchInst
    - IndirectBrInst
    - InvokeInst
    - ResumeInst
    - ReturnInst
    - SwitchInst
    - UnreachableInst
  - UnaryInstruction
    - AllocaInst
    - CastInst
      - AddrSpaceCastInst
      - BitCastInst
      - FPExtInst
      - FPToSIInst
      - FPToUIInst
      - FPTruncInst
      - IntToPtrInst
      - PtrToIntInst
      - SExtInst
      - SIToFPInst
      - TruncInst
      - UIToFPInst
      - ZExtInst
    - ExtractValueInst
    - LoadInst
    - VAArgInst

Dodatok B

Obsah CD

Adresárová štruktúra priloženého kompaktného disku.

```
.
|-- doc                    vlastné zdrojové súbory technickej správy
|-- src                    zdrojové súbory pre Predator/Forester
|    |-- ...
|    \-- cl
|         |-- ...
|         \-- llvm          vlastné zdrojové súbory adaptéru
|              |-- ...
|              |-- api      dokumentácia vygenerovaná doxygenom
|              \-- README   návod na inštaláciu
|-- projekt.pdf             táto technická správa
\-- README                  tento popis
```