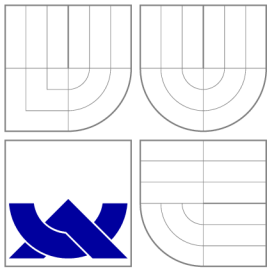
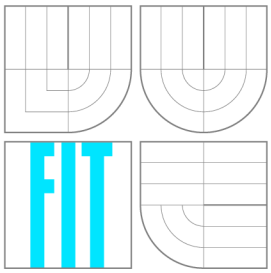


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ANALÝZA PRÁCE S DYNAMICKÝMI DATOVÝMI STRUKTURAMI V C PROGRAMECH

ANALYSIS OF C PROGRAMS WITH DYNAMIC LINKED DATA STRUCTURES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VERONIKA ŠOKOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Šoková Veronika, Bc.**
Obor: Inteligentní systémy
Téma: **Analýza práce s dynamickými datovými strukturami v C programech**
Analysis of C Programs with Dynamic Linked Data Structures
Kategorie: Formální verifikace

Pokyny:

1. Nastudujte problematiku formální analýzy chování programů s dynamickými datovými strukturami (shape analysis) a její implementaci v nástroji Predator.
2. Analyzujte nedostatky současné implementace formální analýzy chování programů s dynamickými datovými strukturami v nástroji Predator a navrhněte jejich řešení. Zaměřte se zejména na možnost vstupu z LLVM bitkódu a na lepší otevřenost a rozšiřitelnost implementace.
3. Navržená řešení implementujte a otestujte na vhodných programech ze sady testovacích programů nástroje Predator, soutěže SV-COMP, případně dalších Vámi zvolených programech.
4. Diskutujte dosažené výsledky a možnosti jejich dalšího rozvoje v budoucnu.

Literatura:

- Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation, In: Proc. of SAS'13, LNCS 7935, Springer-Verlag, 2013.
- Loc Le, Q., Gherghina, C., Qin, S., Chin, W.-N.: Shape Analysis via Second-Order Bi-Abduction, In: Proc. of CAV'14, LNCS 8559, Springer-Verlag, 2014.
- Muller, P., Vojnar, T.: CPAlien: Shape Analyzer for CPAChecker (Competition Contribution), In: Proc. of TACAS'14, LNCS 8413, Springer-Verlag, 2014.
- The LLVM Compiler Infrastructure, online dokumentace, http://llvm.org/docs/.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% 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: **Vojnar Tomáš, prof. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Táto diplomová práca sa zaoberá analýzou dynamických dátových štruktúr pomocou analýzy tvaru použitej v nástroji Predator. Popisuje zvolenú abstraktnú doménu pre reprezentáciu pamäte vo forme symbolických grafov pamäte. Ďalej sa zaoberá návrhom prostredia pre vývoj statických analyzátorov nad clang/LLVM. Prínosom tejto práce je vytvorenie a otestovanie transformačných priechodov zjednodušujúcich LLVM IR medzikód. Ďalším prínosom je optimalizácia parametrov paralelnej nadstavby Predatora opakovaným spúšťaním testov z medzinárodnej súťaže SV-COMP'16, kde táto verzia nástroja Predator získala zlatú medailu v kategórii *Heap Data Structures*. Posledným prínosom je návrh architektúry samotného verifikačného jadra s ohľadom na SMG doménu.

Abstract

This master's thesis deals with the analysis of dynamic linked data structures using shape analysis used in the Predator tool. It describes the chosen abstract domain for heap representation — symbolic memory graphs. It deals with the design of framework for the development of static analyzers based on Clang/LLVM. The main contribution is implementing and testing LLVM's transformation passes that simplify the LLVM IR. Second contribution is the optimization of parameters for parallel run of several variants of the Predator tool. Parameters are tuned for benchmark from SV-COMP'16, where our tool won gold medal in *Heap Data Structures* category. Last contribution is the design of verification core with the focus on the SMG domain.

Kľúčové slová

LLVM, symbolický graf pamäte, statická analýza, analýza tvaru, Predator, framework, abstraktná interpretácia, dynamické dátové štruktúry

Keywords

LLVM, symbolic memory graph, static analysis, shape analysis, Predator, framework, abstract interpretation, dynamic linked data structures

Citácia

ŠOKOVÁ, Veronika. *Analýza práce s dynamickými datovými strukturami v C programech.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Vojnar Tomáš.

Analýza práce s dynamickými datovými strukturami v C programech

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracovala samostatne pod vedením pána profesora Tomáša Vojnara. Ďalšie informácie mi poskytol pán doktor Petr Peringer. Uviedla som všetky literálne pramene a publikácie, z ktorých som čerpala.

. . . . . .
Veronika Šoková
25. mája 2016

Poďakovanie

Chcela by som poďakovať môjmu vedúcemu prof. Ing. Tomášovi Vojnarovi Ph.D. a kolegovi Ing. Tomášovi Fiedorovi za cenné rady a pripomienky k textu práce. Ďalej by som chcela poďakovať Dr. Petrovi Peringrovi za pomoc pri návrhu.

© Veronika Šoková, 2016.
Táto práca vznikla ako školské dielo na FIT VUT v Brně. Práca je chránená autorským zákonom a jej využitie bez poskytnutia oprávnenia autorom je nezákonné, s výnimkou zákonne definovaných prípadov.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

V priebehu posledných rokov sa zmenil spôsob života natoľko, že si ho len ťažko vieme predstaviť bez osobných počítačov, vstavaných systémov a tzv. inteligentných zariadení.

Kód programov vytvárajú programátori a z rastúcou zložitosťou programov, rastie aj veľkosť nimi vytvoreného kódu. Pretože mýliť sa je ľudské, aj nimi vytvorené programy obsahujú chyby. Je nežiaduce, keď sa systém dostane do neočakávaného resp. chybového stavu. U osobného počítača to môže spôsobiť ukončenie videohovoru, či vypnutie internetového prehliadača, čo môže byť pre užívateľa nepríjemné a tým ho odradíme od používania daného programu a spoločnosti prichádzajú o zisky. Nepríjemnejšie sú však život ohrozujúce situácie. Keď sa program riadiaci inteligentné vozidlo dostane do neošetreného chybového stavu, môže to viesť k havárii.

Aby nedochádzalo ku škodám v dôsledku chýb v programoch, finančných, či v prípade kritických programov aj ujmám na zdraví či životoch, potrebujeme zaručiť, že programy budú pracovať korektne vzhľadom k danej špecifikácií. K riešeniu daného problému môžeme použiť dva prístupy: *testovanie a dynamickú analýzu* alebo *formálnu analýzu a verifikáciu* [1]. V prvom prípade hľadáme chyby. Spúšťame program nad nami zvolenou testovacou množinou, pričom sa snažíme pokryť čo najviac možných behov programu.

V druhom prípade sa snažíme dokázať spoľahlivosť, čiže program neobsahuje žiadne chyby. Typickými technikami sú *model checking*, u ktorého sa overujú požadované vlastnosti systematickým generovaním a preskúmaním stavového priestoru. Ďalším prístupom je *theorem proving* — deduktívna metóda založená na matematickom dokazovaní. Ďalšou, v praxi asi najrozšírenejšou technikou, je *statická analýza*, ktorá informuje o správaní systému na základe zdrojového popisu bez nutnosti jeho vykonávania (alebo vykonávania s odľahčenou sémantikou). Z toho dôvodu je možné aspoň niektoré z uvedených analýz spúšťať aj nad nedokončeným alebo čiastočným kódom, ktorý nie je možné skompilovať a následne spustiť. Narozdiel od dynamického testovania, často nachádza chybu, kde vzniká a nie kde sa prejavuje.

Táto práca spadá práve do oblasti statickej analýzy. Konkrétne sa týka statickej analýzy programov s dynamickými dátovými štruktúrami, tzv. analýzy tvaru dátových štruktúr (angl. *shape analysis*), avšak niektoré jej časti sú všeobecnejšie. Cieľom práce je diskutovať analýzu tvaru dátových štruktúr pomocou symbolických grafov pamäte a jej implementáciu v nástroji Predator. Práca identifikuje slabé miesta v návrhu nástroja Predator a navrhuje ich zlepšenie a to ako z pohľadu fungovania nástroja Predator, tak z pohľadu možného jeho ďalšieho rozvoja. Konkrétne sú takto pokryté tri oblasti. Prvou oblasťou je zložitosť vstupného formátu programu, nad ktorým Predator funguje. Tento formát komplikuje implementáciu dodatočných analýz v Predatorovi tým, že vývojár príslušnej analýzy by sa

musel vyrovnať s veľkým množstvom vstupných konštrukcií. Druhým diskutovaným problémom sú nedostatky v paralelnej nadstavbe nástroja Predatora, tzv. Predator Hunting Party. Treťou oblasťou je problematická rozšíriteľnosť funkčnosti Predatora pre príliš optimalizovaným vnútorným dátovým štruktúram.

Prvá vyššie spomenutá oblasti je v práci riešená návrhom sady zjednodušujúcich transformácií vstupných konštrukcií jazyka, ktorý môže byť použitý nástrojom, ktorý v budúcnu nahradí Predatora. Tieto zjednodušujúce konštrukcie boli navrhnuté nad LLVM IR, lebo skupina VeriFIT, ktorá vyvinula nástroj Predator a aktuálne vyvíja jeho nástupcu, sa rozhodla prejsť od vstupu z GCC, ktorý je implicitne použitý v Predatorovi, k novšiemu LLVM. Uvedené transformácie samozrejme môžu byť použité nie len v nástupcovi Predatora, ale i v ľubovolných ďalších analyzátoroch, ktoré skupina bude v budúcnu vyvíjať.

Druhá oblasť sa zaoberá paralelnou nadstavbou nástroja Predator, ktorá bola pôvodne navrhnutá pre potreby medzinárodnej súťaže vo verifikácii SV-COMP. Táto nadstavba spúšťa paralelne rôzne verzie nástroja Predator líšiace sa tým, či používajú alebo nepoužívajú abstrakciu a tiež spôsobom priechodu stavovým priestorom. V rámci práce je diskutovaná optimálna kombinácia týchto verzií s ohľadom na rýchlosť a presnosť analýzy.

V tretej oblasti je navrhnutá nová štruktúra jadra analyzátoru založeného na symbolických grafov pamäte. Narozdiel od stávajúceho návrhu nástroja Predator, je tu kladený dôraz ľahkú udržovateľnosť a budúcu rozšíriteľnosť, skôr ako na maximálnu optimalizáciu. Tento návrh tiež zúročuje skúsenosti získané dlhodobým používaním nástroja Predator, ktoré neboli k dispozícií v dobe jeho vývoja a nemohli teda byť premietnuté do jeho návrhu.

Členenie práce

Práca je členená nasledujúcim spôsobom. Úvod do problematiky statickej analýzy sa nachádza v sekcii 2.1. V 2.3. sekcii je popísaná statická analýza pomocou symbolických grafov pamäte použitá v nástroji Predator. Štruktúra prekladového systému LLVM a front-endu Clang, ktorý je v práci použitý, približuje kapitola 3. Nedostatky súčasnej implementácie nástroja Predator sú diskutované a riešené v nasledujúcich troch kapitolách. Návrh a implementácia sady transformačných priechodov zjednodušujúcich stávajúci LLVM medzikód je popísaná v kapitole 4. Popis a analýza paralelnej nadstavby Predatora sa nachádza v kapitole 5. Kapitola 6 popisuje návrh generického jadra analýzy s napojením na SMG doménu. Zhrnutie práce a budúci vývoj je popísaný v kapitole 7.

Kapitola 2

Statická analýza pomocou symbolických grafov pamäte

Táto kapitola prezentuje jeden z prístupov formálnej analýzy a verifikácie — statickú analýzu. Vymedzujeme sa voči analýze tvaru a to konkrétne voči symbolickým grafom pamäte použitým v nástroji Predator.

2.1 Statická analýza

Overovanie programu bez jeho spúšťania sa nazýva statická analýza (SA) [1, 2]. SA informuje o správaní systému na základe zdrojového popisu bez jeho vykonávania (alebo vykonávania s odľahčenou sémantikou). Z toho dôvodu je možné spúšťať analýzu aj nad nedokončeným kódom, ktorý nie je možné skompilovať a následne spustiť. Tento druh analýzy nevyžaduje model systému. Narozdiel od dynamického testovania, často nachádza chybu, kde vzniká a nie kde sa prejavuje.

Okrem skúmania korektnosti programu sa SA používa aj pre optimalizáciu a generovanie vnútorného kódu.

Jednou z techník statickej analýzy je **symbolická exekúcia** [3], ktorá vykonáva program, nie s dátami, ale množinami hodnôt, ktoré sú popísané formou logiky alebo automatmi. Berú sa v úvahu všetky dosiahnuteľné behy programu, čo vedie na potenciálne nekonečný stavový priestor (stavovú explóziu).

Ďalším kľúčovým prístupom je **abstraktná interpretácia** [4], ktorá však môže siahať aj do model checkingu. Analyzovaný program je vykonávaný na určitej abstraktnej úrovni pre všetky možné vstupy. Pôvodným inštrukciám v programe sémanticky zodpovedajú abstraktné transformátory definované nad abstraktnou doménou. Ku každej inštrukcií je priradený zoznam abstraktných kontextov reprezentujúci jej možné hodnoty v určitej abstraktnej podobe. Pri spájaní ciest po vetveniach sa používa operátor *join*, na zredukovanie počtu kontextov. Pri neobmedzených cykloch nám pomáha k nájdeniu pevného bodu operátor *widening*, tým urýchľuje termináciu. Ak s pomocou *wideningu* získame príliš hrubý odhad, môžeme ho skúsiť obmedziť pomocou operátoru *narrowing*, ktorý spresňuje výslednú hodnotu. Abstrakciou však strácame časť informácií. Analýza je teda zvyčajne spoľahlivá, ale neúplná. Čo často vedie na falošné hlásenia, napr. falošné pozitíva — čiže v programe bez chýb nájde chybu. Vyhnúť sa tomuto dôsledku je samozrejme, s ohľadom na nekonečnosť stavového priestoru, problematické.

Formálna verifikácia programov pracujúcich s dynamickými dátovými štruktúrami si kladie za cieľ overiť program vzhľadom na chyby spojené s pamäťou: neplatné dereferencie, nedefinované smerníky, viacnásobné uvoľnenie pamäte, pretečenie a iné. Môže sa kontrolovať práca so smerníkovou aritmetikou (angl. *pointer arithmetic*), blokové operácie s pamäťou, reinterpretáciu obsahu pamäte a zarovnanie adries.

2.2 Analýza tvaru

Analýza tvaru (angl. *shape analysis*, podrobnejšie viď [5]) je druh statickej analýzy, ktorý je použitý napr. v nástroji Predator. Jej cieľom je charakterizovať *tvar* (podobu) všetkých dynamických dátových štruktúr viazaných smerníkmi, s ktorými sa v programe pracuje. Verifikujú sa kompozitné štruktúry ako sú rôzne cyklické a acyklické zoznamy, skip–listy, stromy (s pridanými smerníkmi) a iné zložité štruktúry. Existujú prístupy k analýze tvaru, ktoré potrebujú vopred vedieť tvar dátových štruktúr. Alebo sú poloautomatizované tak, že sa dodefinuje invariant cyklu či indukčný predikát. U tých automatizovaných nástrojov je zas obmedzená množina tvarov (dátových štruktúr), ktoré sú schopné analyzovať.

Na reprezentáciu pamäte sa používajú separačné logiky [6], 3–hodnotové logiky (TVLA) [7], grafy pamäte [8] či lesné automaty [9].

2.3 SMG ako model reprezentácie pamäte

Sekcia popisuje reprezentáciu pamäte použitú v nástroji Predator. Na analýzu implementovanú v tomto nástroji možno pozerať ako na *abstraktnú interpretáciu* (viď sekciu 2.3.3), kde ako abstraktná doména figurujú **symbolické grafy pamäte** (SMG) a ako operátor *widening* je použitá abstrakcia. Narozdiel od klasickej definície, neberie v úvahu dve postupne vygenerované abstraktné konfigurácie, ale len konfiguráciu aktuálnu. Konvergencia je pritom zaistená iba pre podmnožinu programov so zoznamami. Pre ostatné dátové štruktúry nie je v Predatorovi implementovaná abstrakcia, ktorá by zvládla konečne reprezentovať nekonečné množiny daných štruktúr.

Koncept SMG je inšpirovaný a do istej miery podobný *separačnej logike* [6] s induktívnymi predikátmi typu zoznam. Jedná sa však o čisto grafovo-orientovaný prístup, doplnený o podporu niektorých nízkoúrovňových operácií so smerníkmi.

SMG modeluje pamäť do podoby orientovaného bipartitného grafu s dvomi typmi uzlov a dvomi typmi hrán s popisnou funkciou v závislosti na druhu objektu. Uzly môžu byť buď *objekty* alebo *hodnoty*. Objekty ďalej rozlišujeme na *regióny*, ktoré predstavujú konkrétny blok pamäte ležiaci na halde, zásobníku alebo v globálnom priestore, a *abstraktné objekty*, ktoré môžu byť jedno- alebo obojsmerne viazané segmenty zoznamov (SLS alebo DLS).

2.3.1 Symbolický graf pamäte

Táto kapitola je založená na definíciách z [10].

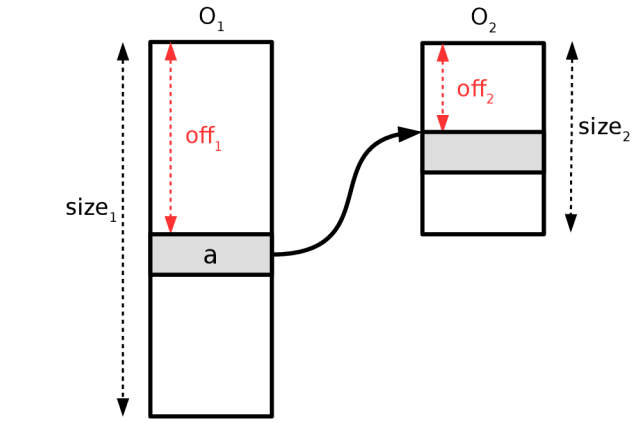
Definícia 1. *Nech* $\mathbb{B} = \{false, true\}$, $\mathbb{T}$ *je množina typov,* $\mathbb{K} = \{reg, sls, dls, ...\}$ *je množina druhov abstraktných objektov a regiónu a* $\mathbb{S} = \{reg, fst, lst, all\}$ *je množina bližšie určujúca druh cieľového objektu: obyčajný región, prvý/posledný prvok zoznamu alebo všetky prvky nadzoznamu.* Symbolický graf pamäte *je potom pätica* $G = (O, V, \Lambda, H, P)$*, kde*

- O *– konečná množina objektov vrátane nulového objektu* #*,*

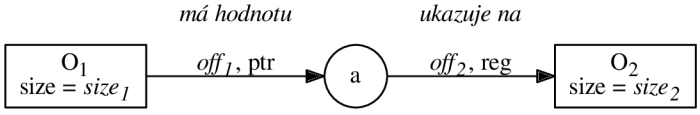
- V – *konečná množina hodnôt taká, že* $O \cap V = \emptyset \wedge 0 \in V$ *(adresa nulového objektu),*
- Λ – *n-tica nasledujúcich popisných funkcií určujúcich vlastnosti uzlov:*
  - *druh objektu* **kind***:* $O \to \mathbb{K}$*, kde* $kind(\#) = reg$*,*
  - *úroveň zanorenia objektu a hodnôt* **level***:* $O \cup V \to \mathbb{N}$ *(najvyššia úroveň je 0),*
  - *platnosť objektu* **valid***:* $O \to \mathbb{B}$*,*
  - *veľkosť objektu* **size***:* $O \to \mathbb{N}$*,*
  - *špecifické vlastnosti pre* $D = \{d \in O \mid kind(d) \in \{sls, dls\}\}$*:*
    - *minimálna dĺžka segmentov* **len***:* $D \to \mathbb{N}$*,*
    - *offsety položiek segmentu* **hoff, noff, poff***:* $D \to \mathbb{N}$*.*
- H – *čiastočná funkcia* $O \times \mathbb{N} \times \mathbb{T} \rightharpoonup V$ *popisujúca hranu typu „má hodnotu“,*
- P – *čiastočná injektívna funkcia* $V \leftharpoonup \mathbb{Z} \times \mathbb{S} \times O$ *popisujúca hranu typu „ukazuje na“.*

Pre znázornenie SMG zavádzame konvenciu, že oválne uzly reprezentujú hodnoty a hranaté uzly objekty. Ak hrana bude viesť z uzlu reprezentujúceho objekt do uzlu typu hodnota, bude to hrana typu „má hodnotu“ (niekedy odlíšená modrou farbou), v opačnom prípade sa jedná o hranu „ukazuje na“.

Význam hrán môžeme ukázať na regiónoch. Nech $a \in V, o_1, o_2 \in O$, $t = \text{ptr}$, $off_1 \in \mathbb{N}$, $off_2 \in \mathbb{Z}$ a $tg \in \mathbb{S}$, potom matematický zápis $o_1 \xrightarrow{off_1,t} a \xrightarrow{off_2,tg} o_2$ bude značiť, že v bloku pamäte o_1 na offsete off_1 je uložený smerník a ukazujúci do pamäte v bloku o_2 na offsete off_2 (obr. 2.1).



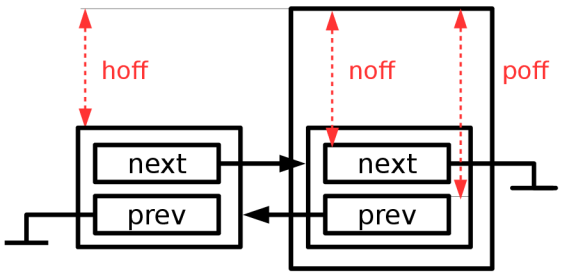
(a)



(b)

Obr. 2.1: Sémantika hrán: (a) zobrazenie objektov o_1 a o_2 v pamäti a (b) reprezentácia v SMG.

Pri práci so zoznamami môžeme použiť nasledujúce offsety. Offset *hoff* (*header offset*) určuje počiatočné umiestnenie štruktúry, ktorá je prepojená zoznamom. Takéto zoznamy sú súčasťou záznamov používaných v Linuxovom jadre (obr. 6.2). Offsety *noff* (*next offset*) a *poff* (prev offset) určujú spätné a dopredné smeníky pri lineárne viazaných zoznamoch. Pre jednoduchosť sa budeme ďalej zaoberať jednosmerne viazanými zoznamami.



Obr. 2.2: Offsety — Linuxové jadro

.

Definícia 2. Nulový objekt # *je neplatný, má veľkosť a úroveň zanorenia 0 a leží na adrese 0, čiže* $valid(\#) = false, size(\#) = level(\#) = 0$ *a* $0 \xrightarrow{0,reg} \#$.

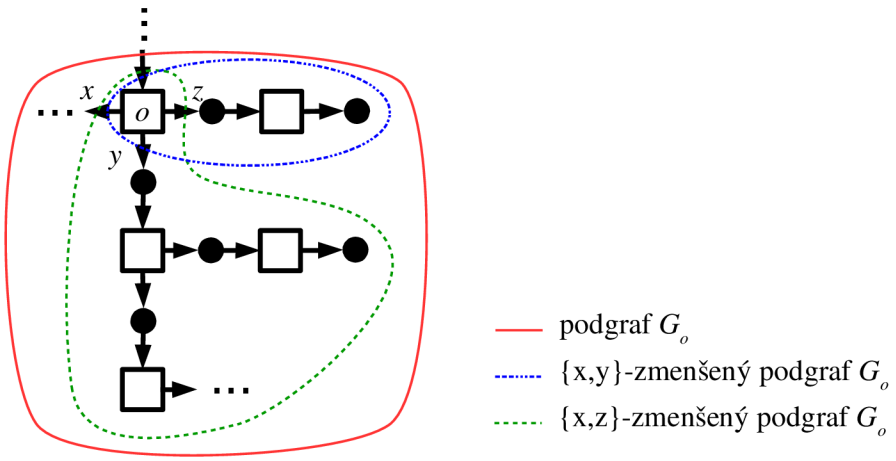
V ukážkových príkladoch nebudeme # za hodnotou 0 (`NULL`) uvádzať (jedná sa o neplatný objekt).

Definícia 3. Prázdny SMG *je definovaný ako SMG* $G = (\{\#\}, \{0\}, \Lambda, \emptyset, P)$. *Čiže pozostáva výhradne z objektov #, adries 0 a hrán typu „ukazuje na“ medzi nimi.*

Definícia 4. Podgraf SMG $G' = (O', V', \Lambda', H', P')$ *grafu* $G = (O, V, \Lambda, H, P)$ *je vtedy, keď* $O' \subseteq O, V' \subseteq V$ *a* H', P' *a* Λ' *sú také podmnožiny* H, P, Λ *vzťahujúce sa k* O' *a* V'.

Definícia 5. *Nech* $x \in O \cup V$. *Potom* G_x *je taký najmenší podgraf grafu* G *s koreňom* x, *ktorý zahŕňa* x *a všetky objekty a hodnoty dosiahnuteľné z* x.

Definícia 6. *Nech* $F \subseteq \mathbb{N}$. *Potom* F–zmenšený podgraf SMG *grafu* G_x *je taký najmenší podgraf obsahujúci objekty a hodnoty dosiahnuteľné z* x *mimo tých na adresách* $A_F = \{H(x, off, \text{ptr}) \mid off \in F\}$ a uzlov dosiahnuteľných z x cez A_F.



Obr. 2.3: Znázornenie podgrafov z objektu o obmedzených offsetmi x, y, z.

Definícia 7. Graf pamäte *(angl. memory graph, MG) je SMG* $G = (R, V, \Lambda, H, P)$*, kde* $R = \{r \in O \mid kind(r) = reg\}$ *je množina regiónov.*

Význam objektov. SMG reprezentujú triedu grafov MG, ktoré sú získané viacnásobným aplikovaním dvoch transformácií:

1. *materializácia* — vyňatie konkrétneho regiónu zo začiatku alebo konca abstraktného objektu,
2. *odstránenie* — nulovej sekvencie abstraktného objektu (napr. 0+SLS[1], ktorý sme nakoniec získali materializáciou).

Definícia 8. Konkrétny obraz pamäte *(angl. concrete memory image, MI). Sémanticky MG* $G = (R, V, \Lambda, H, P)$ *reprezentuje množinu pamäťových obrazov* $\mu : \mathbb{N} \to \{0, ..., 255\}$ *mapujúcich konkrétne adresy na bajty tak, že existuje funkcia* $\pi : R \to \mathbb{N}$*, pre ktorú platí:*

1. *iba nulový objekt je na adrese 0:* $\forall r \in R : \pi(r) = 0 \Leftrightarrow r = \#$
2. *žiadne dva platné regióny sa neprekrývajú:*

$$\forall r_1, r_2 \in R : valid(r_1) \wedge valid(r_2) \Rightarrow \langle \pi(r_1), \pi(r_1) + size(r_1)) \cap \langle \pi(r_2), \pi(r_2) + size(r_2)) = \emptyset$$

3. *smerníkové položky obsahujú konkrétne adresy regiónov, na ktoré odkazujú:* $\forall r_1 \xrightarrow{off_1, \text{ptr}} a \xrightarrow{off_2, reg} r_2 \in H, P$ *platí*

$$addr(bseq(\mu, \pi(r_1) + off_1, size(\text{ptr}))) = \pi(r_2) + off_2, \text{ kde}$$

   - $bseq(\mu, p, size)$ *je sekvencia bajtov* $\mu(p)\mu(p+1)...\mu(p+size-1)$ *pre* $\forall p, size > 0$
   - $addr(\sigma)$ *je konkrétna adresa zakódovaná sekvenciou bajtov* σ

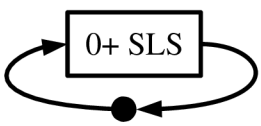
4. *položky majúce rovnakú hodnotu obsahujú rovnakú konkrétnu hodnotu*[2]*:* $\forall r_1 \xrightarrow{off_1, t_1} v,\ r_2 \xrightarrow{off_2, t_2} v \in H$*, kde* $v \neq 0$ *platí*

$$bseq(\mu, \pi(r_1) + off_1, size(t_1)) = bseq(\mu, \pi(r_2) + off_2, size(t_2))$$

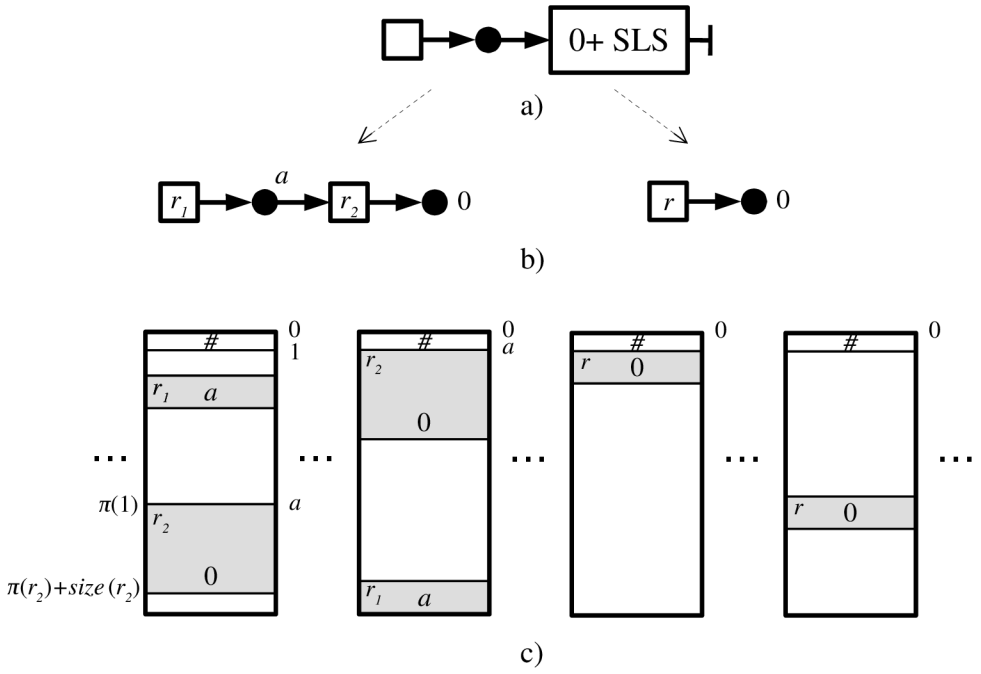
5. *nulové položky obsahujú nuly:* $\forall r \xrightarrow{off, t} 0$ *z* H *platí*

$$\mu(\pi(r) + off + i) = 0, \quad 0 \leq i < size(t)$$

Konečne môžeme povedať, že pre SMG G máme $\text{MI}(G) = \bigcup_{G' \in MG(G)} \text{MI}(G')$. (S)MG nespĺňajúce vyššie uvedené podmienky sú *sémanticky prázdne*. Názorný príklad takého grafu je na obrázku č. 2.4. Jeden z možných MG je prázdny zoznam a ten nemôže na seba ukazovať. Tento prípad je nutné ošetriť pri abstrakcii.



Obr. 2.4: Cyklický 0+SLS bez definovanej sémantiky.



Obr. 2.5: Úrovne reprezentácie pamäte: (a) SMG(G), (b) množina MG(G) a (c) nekonečná množina MI(G).

Na obrázku č. 2.5 môžeme vidieť jednotlivé úrovne abstrakcie pre jednosmerne viazaný zoznam. Abstraktný objekt (a) sa skonkretizuje na dva zoznamy: 1–prvkový a prázdny (b). Pre ne existuje nekonečné množstvo kombinácií v pamäti (c). Musia však spĺňať obmedzenia dané príslušným MG, napr. blok pamäte reprezentujúci región r_1 bude obsahovať adresu a a na adrese a bude ležať blok reprezentujúci región r_2, ktorého položka obsahuje adresu 0 (`NULL`). Berieme v úvahu lineárne rozloženie pamäte (angl. *flat address space*), ktoré pokrýva väčšinu praktických prípadov.

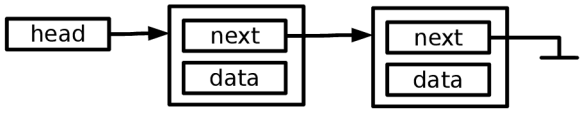
[1] n+SLS značí objekt d s minimálnou dĺžkou segmentov $len(d) = n$

[2] až na vynulované bloky pamäte, ktoré sa môžu líšiť v dĺžke

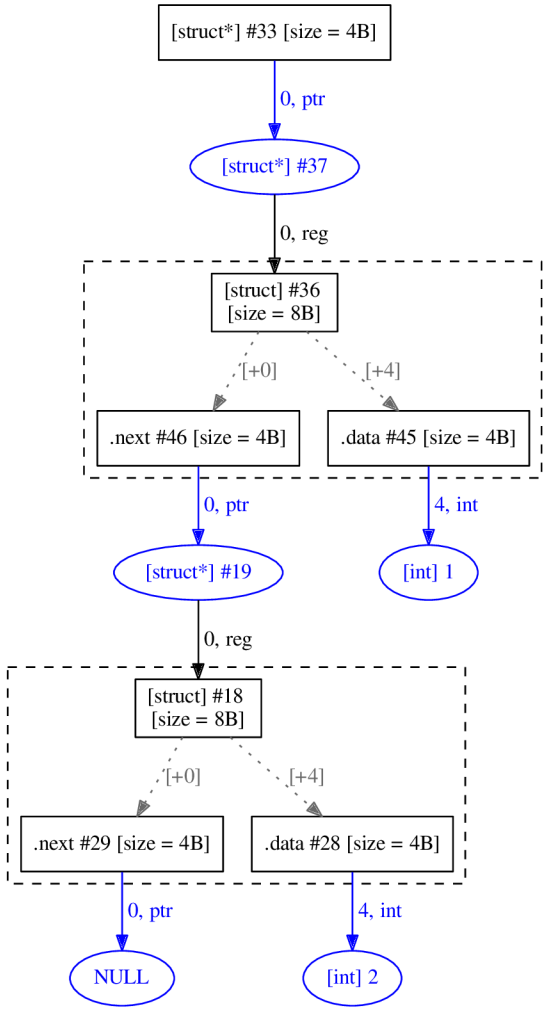
Ukážkové príklady

V tejto časti si ukážeme ako reprezentovať vybrané typy jednosmerne viazaných zoznamov pomocou SMG.

Graf na obr. 2.7 je zjednodušená verzia získaná pomocou funkcie `__VERIFIER_plot()` nástroja Predator v príslušnom programe popisujúcom prácu so zoznamom znázorneným na obr. 2.6. Prekladané pre 32-bitovú architektúru, t.j. adresy sú 4–bajtové.



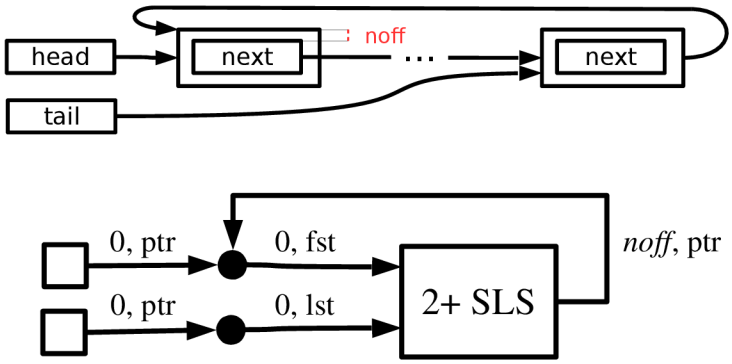
Obr. 2.6: Jednoduchý jednosmerne viazaný zoznam o 2 prvkoch.



Obr. 2.7: Konkrétna MG reprezentácia zoznamu generovaného nástrojom Predator.

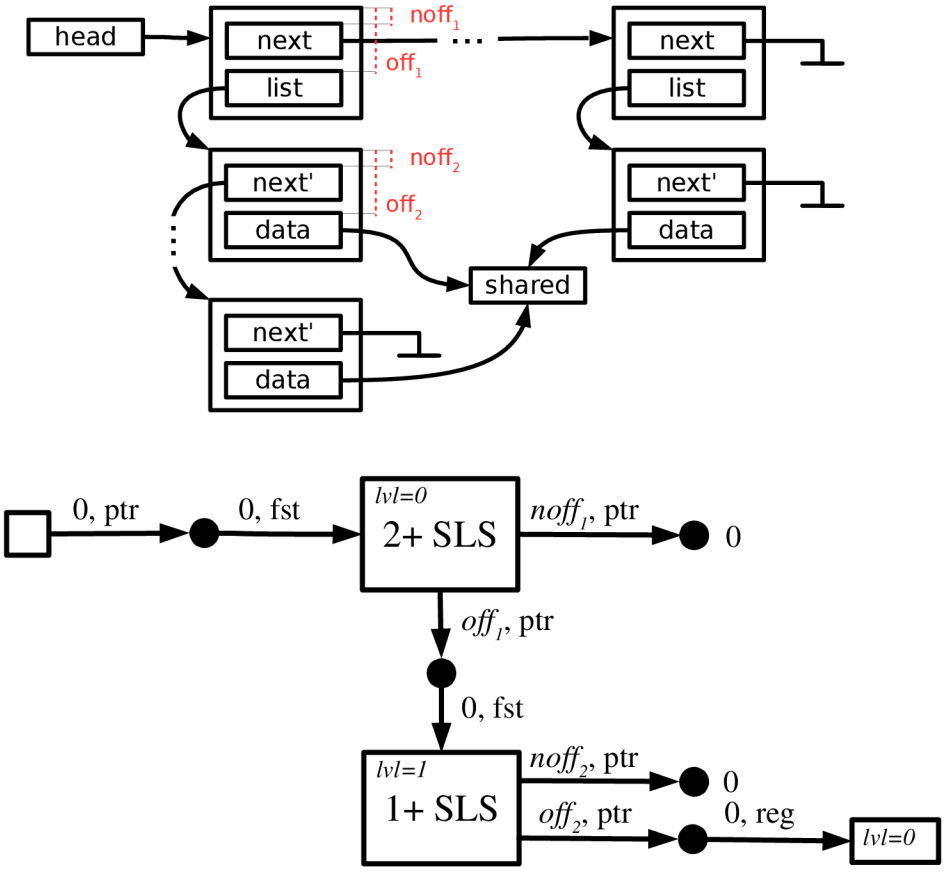
Nasleduje znázornenie cyklického zoznamu so smerníkmi na začiatok a koniec zoznamu

(obr. 2.8), modelované v popise príslušnej hrany ako druh cieľového objektu *fst* a *lst* (prvý a posledný prvok).



Obr. 2.8: Cyklický jednosmerne viazaný zoznam.

Pre reprezentáciu hierarchických dátových štruktúr sa zavádza atribút úrovne (*level*). Pri každom zanorení sa inkrementuje úroveň hodnôt aj objektov o 1. Pričom najvyššia úroveň je 0. Ak sa jedná o zdielaný objekt naprieč rôznymi stupňami zanorenia, jeho úroveň je vždy 0 (v obr. 2.9 objekt `shared`). Ale ak už má zdieľaný objekt nastavenú úroveň a zdieľajú ho objekty s rovnakou, ale vyššou úrovňou, tak sa nenastavuje na 0.



Obr. 2.9: Hierarchicky vnorené jednosmerne viazané zoznamy so zdieľaným objektom.

2.3.2 Operácie nad SMG

Táto časť popisuje základné algoritmy pracujúce nad SMG (pre SLS), ktoré sa využívajú pri verifikácií programov uvádzané v [10]. Jedná sa o reinterpretáciu dát, spájanie a abstrakciu.

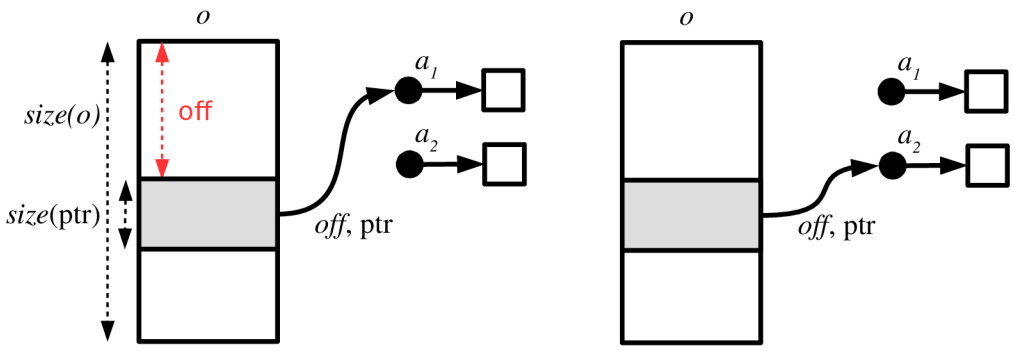
Reinterpretácia dát

Definovať reinterpretáciu pre všetky možné dátové typy (a ich hodnoty) je náročné. Preto sa uvažuje minimum operácií, ktoré vedú na spoľahlivosť[3] formálnej verifikácie. S ohľadom na program, ktorý verifikujeme, sa jedná o pretypovanie a čítanie a zápis do pamäte. Špeciálne je popísaná reinterpretácia nad vynulovanou pamäťou, čo sa využíva pri spracovaní štandardných funkcií `calloc()` a `memset()`.

Čítanie dát. Algoritmus berie na vstupe SMG G, objekt $o \in O$, v ktorom je položka typu t na offsete *off*, ktorej hodnotu chceme prečítať. Výstupom je dvojica (G', v), kde v je hodnota danej položky. Ak je hodnota iného typu, tak ju syntetizuje. Ak je súčasťou vynulovaného bloku, vráti 0. Ak neexistuje, vráti nedefinovanú hodnotu a príslušný uzol vloží do grafu. A G' je taký SMG s množinou hrán H' takých, že $H'(o, off, t) = v \neq \bot$[4] $\wedge$ MI(G)=MI(G').

Zápis dát. Algoritmus očakáva na vstupe SMG G, objekt $o \in O$, v ktorom je položka typu t na offsete *off*, do ktorej chceme zapísať hodnotu v. Výstupom je SMG G' s množinou hrán H' takých, že $H'(o, off, t) = v \wedge \text{MI}(G) \subseteq \text{MI}(G'')$. Kde G'' je vlastne G' bez hrany $e : o \xrightarrow{off,t} v$. Čiže sa uistíme, že výsledné SMG obsahuje hranu e. Nemôžeme požadovať rovnosť, lebo by mohla nová hrana kolidovať s už existujúcimi hranami.

Na obrázku č. 2.10 je znázornený zápis novej adresy a_2 do položky typu smerník objektu o. Jedná so o realizáciu pseudopríkazu `(ptr+off)=a2;`, kde premenná *ptr* ukazuje na objekt o.

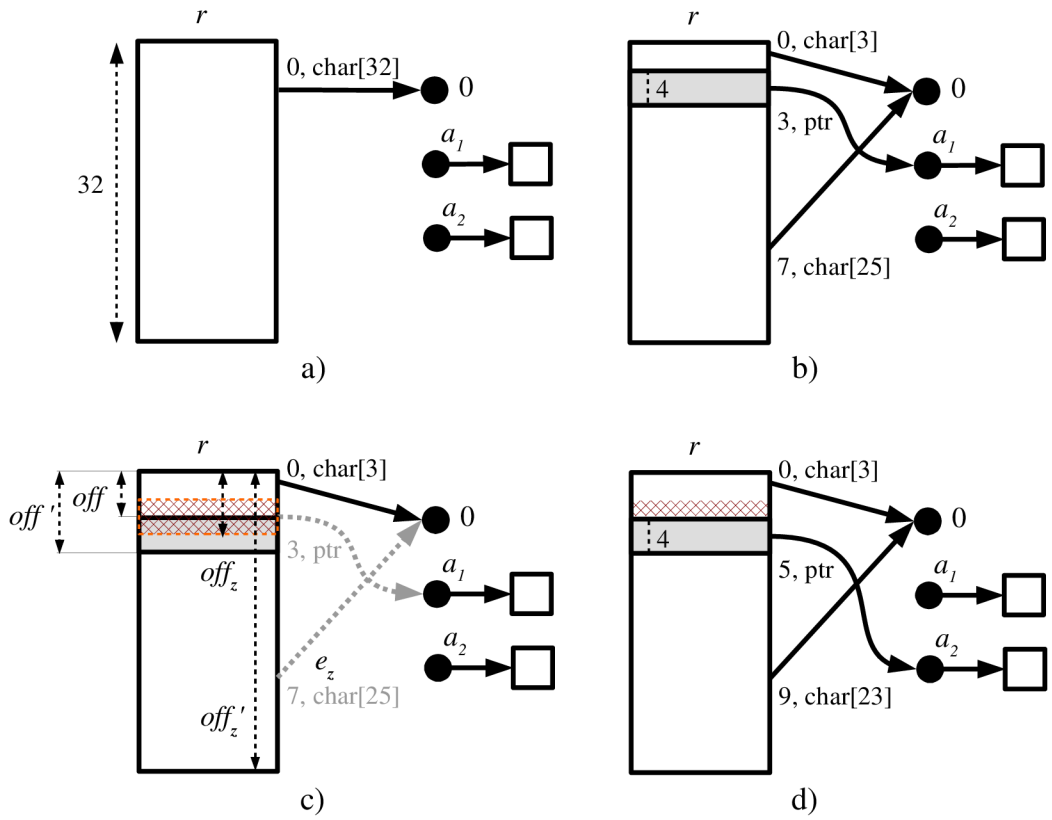


Obr. 2.10: Ilustrácia zápisu dát v položke bez prekryvu.

Na obrázku č. 2.11 je ilustrovaný priebeh zápisu dát pre položky vo vynulovanom regióne r veľkosti $size(r) = 32$. Nastane tak po prevedení nasledujúcej sekvencie pseudopríkazov: `memset(ptr, 0, 32); (ptr+3) = a1; (ptr+5) = a2;`, kde premenná *ptr* ukazuje na región r. Pričom adresy sú 4–bajtové. Nad vynulovaným regiónom sa 2–krát po sebe zavolá

[3] ak verifikátor odpovie, že je program korektný, tak korektný je
[4] symbol $\bot$ značí situáciu, keď H alebo P nie je definované

funkcia `WriteValue(G,o,off,t,v)` ilustrujúca zápis hodnoty v v SMG G do objektu o na jeho položku (off, t). Na obrázku (c) je znázornený medzistav vo vnútri cyklu pri druhom volaní funkcie. Odstraňuje sa hrana e_z. Pomocné premenné sú nastavené nasledovne: $off = 5, off' = 9$ (hranice položky, do ktorej zapisujeme) a $off_z = 7$, $off'_z = 32$ (hranice vynulovaného bloku, s ktorým sa kryje prepisovaná položka). Vo výsledku budú mať 2 bajty na offsete 3 nedefinovanú hodnotu (na obrázku (d) šrafovaná oblasť).



Obr. 2.11: Ilustrácia zápisu dát, keď sa položky prekrývajú: (a) počiatočné SMG G, (b) SMG G' získané zavolaním $WriteValue(G, r, 3, \text{ptr}, a_1)$, (c, d) SMG G'' počas a po zavolaní $WriteValue(G', r, 5, \text{ptr}, a_2)$.

Spájanie (Join)

Spájanie je výpočtovo náročná operácia, pri ktorej dochádza k strate informácií. Po spojení nemusia byť zachované pôvodné minimálne dĺžky zoznamových segmentov, môžu byť abstrahované offsety cieľov smerníkov a i. Bežne sa spúšťa pri spájaní viacerých kontextov vznikajúcich pri vetveniach a cykloch (sekcia 2.3.4).

Jedná sa o súbežné prehľadávanie do hĺbky (DFS) oboch grafov, ktoré chceme spájať, z určených počiatočný uzol (o_1 a o_2). Spájané objekty musia byť kompatibilné: rovnaká úroveň zanorenia, veľkosť, offsety... Z každého objektu potom prebieha spájanie rekurzívne. Výsledok spájania určuje status $s \in \mathbb{J}$, $\mathbb{J} = \{\simeq, \sqsupset, \sqsubset, \bowtie\}$. Symbol $\simeq$ značí, že sú podgrafy izomorfné, $\sqsupset$, že ľavý je všeobecnejší než pravý, $\sqsubset$, že pravý je všeobecnejší než ľavý a $\bowtie$, ak sú neporovnateľné. Výsledný stav spájania nám určuje funkcia $UpdateJoinSatus : \mathbb{J} \times \mathbb{J} \to \mathbb{J}$

$s_1 \backslash s_2$	$\simeq$	$\sqsupset$	$\sqsubset$	$\bowtie$
$\simeq$	$\simeq$	$\sqsupset$	$\sqsubset$	$\bowtie$
$\sqsupset$	$\sqsupset$	$\sqsupset$	$\bowtie$	$\bowtie$
$\sqsubset$	$\sqsubset$	$\bowtie$	$\sqsubset$	$\bowtie$
$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$

Tabuľka 2.1: Funkcia `UpdateJoinStatus()` definovaná tabuľkou.

(tab č. 2.1). Napr. pre abstraktné objekty platí:

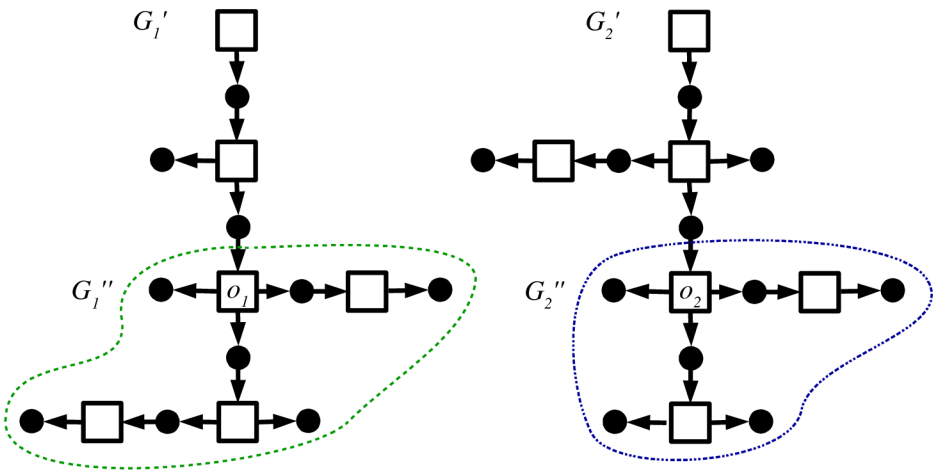
$$n{+}\text{SLS} \sqsubset m{+}\text{SLS}, \text{ kde } n > m$$

V prípade, že v druhom SMG chýbajú položky alebo objekty, na ich doplnenie sa skúša použiť reinterpretácia alebo pridanie najvšeobecnejšieho abstraktného objektu (napr. 0+SLS). Join algoritmus očakáva na vstupe grafy G_1, G_2 (zdrojové) a G (cieľový) a vracia G'_1, G'_2, G'. Ich sémantika je nasledujúca:

- $\text{MI}(G_1) \subseteq \text{MI}(G'_1)$ a $\text{MI}(G_2) \subseteq \text{MI}(G'_2)$, kde G_1, G_2 môžu byť rôzne od G'_1, G'_2 (tab.2.2)
- podgrafy G''_1, G''_2 grafov G'_1, G'_2 od koreňov o_1, o_2 sú spájané v podgraf G'' grafu G' od koreňa o (obr. 2.12), je požadované, aby $\text{MI}(G''_1) \subseteq \text{MI}(G'') \supseteq \text{MI}(G'_1)$
- podgraf $G' - G''$ grafu G je zložený z objektov a hodnôt, ktoré sa v priebehu algoritmu nemenia

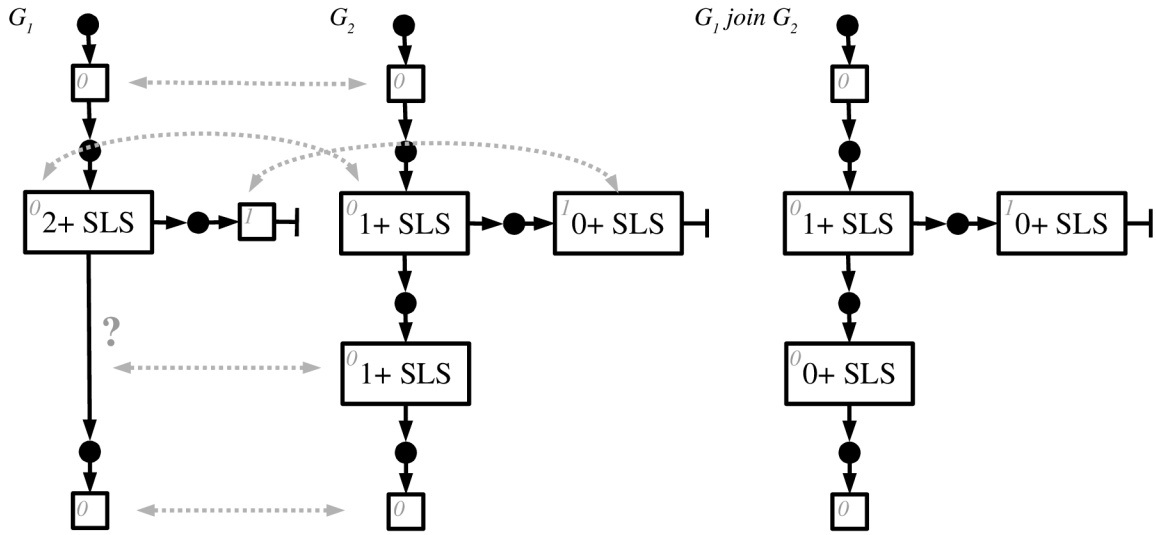
s	sémantika G'_1	sémantika G'_2
$\simeq$	$\text{MI}(G_1) = \text{MI}(G'_1)$	$\text{MI}(G_2) = \text{MI}(G'_2)$
$\sqsupset$	$\text{MI}(G_1) = \text{MI}(G'_1)$	$\text{MI}(G_2) \subset \text{MI}(G'_2)$
$\sqsubset$	$\text{MI}(G_1) \subset \text{MI}(G'_1)$	$\text{MI}(G_2) = \text{MI}(G'_2)$
$\bowtie$	$\text{MI}(G_1) \subset \text{MI}(G'_1)$	$\text{MI}(G_2) \subset \text{MI}(G'_2)$

Tabuľka 2.2: Porovnanie sémantiky vstupných grafov SMG G_1, G_2 pri reinterpretácii (1. krok algoritmu spájania).



Obr. 2.12: Príklad vstupných grafov SMG G_1, G_2 spájania po reinterpretácií.

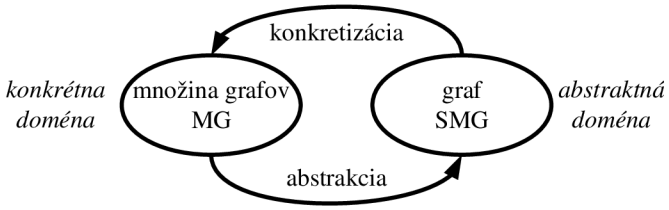
Na obrázku 2.13 je znázornený priebeh spájania dvoch grafov SMG G_1, G_2. Pre jednoduchosť nie sú uvádzané popisy hrán a objektov (sivým je označená úroveň zanorenia). Predpokladá sa, že objekty a hodnoty v jednej rovine sú kompatibilné a všetky platné. Majú rovnakú veľkosť, offsety, sú rovnakého typu... Prvé objekty sú zhodné, pre druhé platí 2+SLS $\sqsubseteq$ 1+SLS. Vnorený región možno chápať ako 1–prvkový zoznam. V ďalšom kroku ideme prehľadávať druhú hranu, kde nám chýba objekt. Tak si ho skúsime domyslieť ako 0+SLS a následne spojiť. A posledné objekty sú opäť zhodné. Výsledný graf je vľavo.



Obr. 2.13: Ilustrácia sémantiky operátoru *join*, inšpirované podľa [10, str. 29].

2.3.3 Abstraktná interpretácia

Neoddeliteľnou súčasťou statickej analýzy je abstrakcia (a prípadná konkretizácia), ktorá sa snaží zredukovať veľkosť skúmaného stavového priestoru. Viď obrázok 2.14.



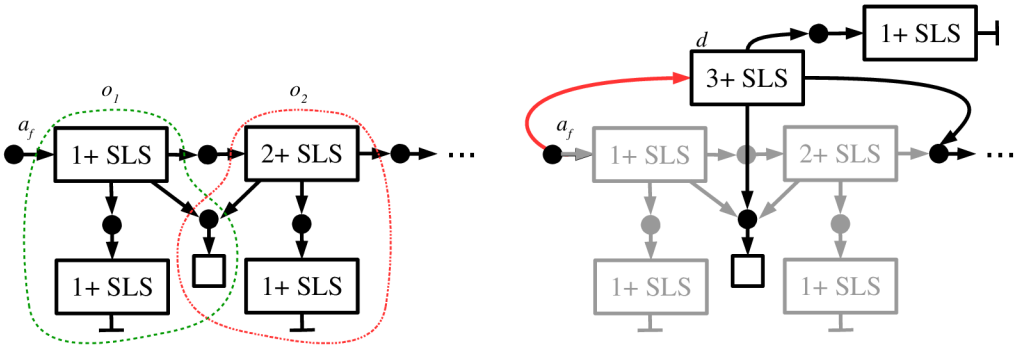
Obr. 2.14: Abstrakcia a konkretizácia.

U SMG si najprv vyhľadáme kandidátnu neprerušovanú sekvenciu. Jedná sa o množinu hrán a objektov, ktoré budeme spájať v jeden abstraktný objekt. Pre SLS musí platiť, že všetky susedné objekty majú rovnaký *noff* offset. Pre jednoduchosť, hľadáme n-ticu $(o_c, hoff, noff)$, kde $o_c \in O$ a $hoff, noff \in \mathbb{N}$, pre ktorú platí:

1. o_c je platný objekt na halde
2. o_c má susedný objekt $o \in O$, ktorý je s ním jednosmerne zviazaný nasledovne (a je adresa): $H(o_c, noff_c\text{,ptr}) = a$, $P(a) = (hoff_c, tg_1, o)$ pre $tg_1 \in \{fst, reg\}$

Ak by sme *hoff* neuvažovali (zoznamy v Linuxovom jadre), nastavíme ho na nulu. Následne zavoláme algoritmus pre *join* nad naším grafom ($G_1 = G_2$) z počiatočnými objektami, ktoré chceme zlučovať. Nazačiatku dva objekty sekvencie, potom k výsledku pridávame ďalší objekt sekvencie.

Po zavolaní funkcie treba previazať a_f (ak existuje) na nový objekt d. Čiže hrana $P(a_f) = (o_1, hoff_c, tg)$, kde $tg \in \{fst, reg\}$ sa zmení na $P(a_f) = (d, hoff_c, fst)$. Tým sa stanú oba podgrafy objektov o_1 a o_2 nedosiahnuteľné a budú zmazané. Ukázané je to na obr. 2.15, kde vľavo je znázornený vstupný graf a vpravo výsledok spájania.



Obr. 2.15: Zlúčenie dvoch objektov, kde čiary predstavujú $\{noff\}$-zmenšené podgrafy z koreňov o_1 a o_2, inšpirované podľa [10, str. 16].

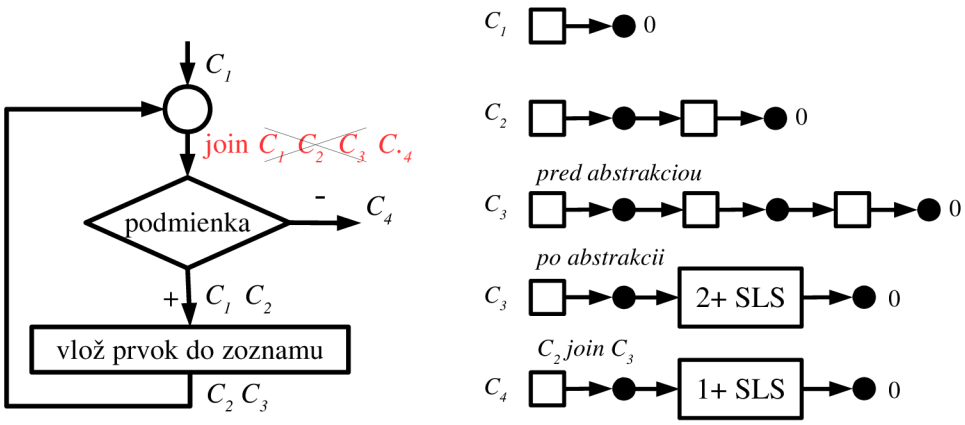
2.3.4 Konfigurácie programu

Symbolická konfigurácia programu (SPC, angl. *symbolic program configuration*) značí dvojicu $C = (G, \nu)$ tvorenú obsahom pamäte v podobe SMG G a mapovaním ν premenných zdrojového programu na regióny v SMG.

$$\nu : Var \to R,\ \forall x \in Var : level(\nu(x)) = 0 \wedge valid(\nu(x))$$

SPC sa vytvára po symbolickom prevedení každej inštrukcie. Po vetvení a cykloch sa spájajú konfigurácie pomocou operátoru *join* (obr. 2.16), aby sa zredukoval počet SMG, ktoré sú priradené jednotlivým uzlom *grafu riadenia toku* (CFG) a je nutné ich preskúmať. Naviac sa môže po vykonaní cyklu urobiť abstrakcia, aby sa sekvencia regiónov tvoriacich zoznam konkrétnej dĺžky n nahradila segmentom s dĺžkou $n+$. Tým dochádza k akcelerácii výpočtu, ktorého cieľom je zaistiť konečnosť behu analýzy i pri práci so zoznamami s nekonečne veľa rôznymi dĺžkami.

Spájanie prebieha nad každou premennou programu. Lebo bežne nie je SMG súvislý graf. Očakáva sa, že všetky objekty na halde budú dosiahnuteľné pomocou statických objektov alebo objektov na zásobníku (globálne a lokálne premenné programu). Na obrázku 2.16 je znázornené spájanie konfigurácií pri spracovaní cyklu. C_i nám určuje konfiguráciu programu po vykonaní predchádzajúceho uzlu grafu.



(a) Vývojový diagram programu.

(b) SMG jednotlivých konfigurácií.

Obr. 2.16: Program vytvárajúci jednosmerne viazaný zoznam do premennej x. V C_1 ukazuje x na `NULL`. Do x sa vloží 1 prvok a vznikne C_2. Pri ďalšom cyklení sa do C_2 vloží ďalší prvok, *abstrakcia* to zovšeobecní na 2+SLS a vznikne C_3. *Join* spojí C_2 a C_3 do jednej konfigurácie C_4, ktorá bude 1+SLS. Ak by bola podmienka vyhodnocovaná nedeterministicky, bolo by nutné v `else` vetvi skúmať všetky konfigurácie spracované `then` vetvou (C_1, C_2, C_3, C_4). S *join* operátorom stačí skúmať iba C_4.

Kapitola 3

LLVM

Táto kapitola vychádza z [2]. LLVM (*Low-Level Virtual Machine*) [11] 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 naviac ponúka prostriedky pre napísanie vlastných kompilátorov.

Front-end zabezpečuje lexikálnu, syntaktickú a sémantickú analýzu programu. Výstupom je vygenerovaný LLVM IR kód. Najznámejším front–endom je **Clang** (viď 3.1), ktorý podporuje okrem C a C++ aj Objective-C a Objective-C++. Existujú aj iné projekty podporujúce napr. Javu a Scheme.

Jadro 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.

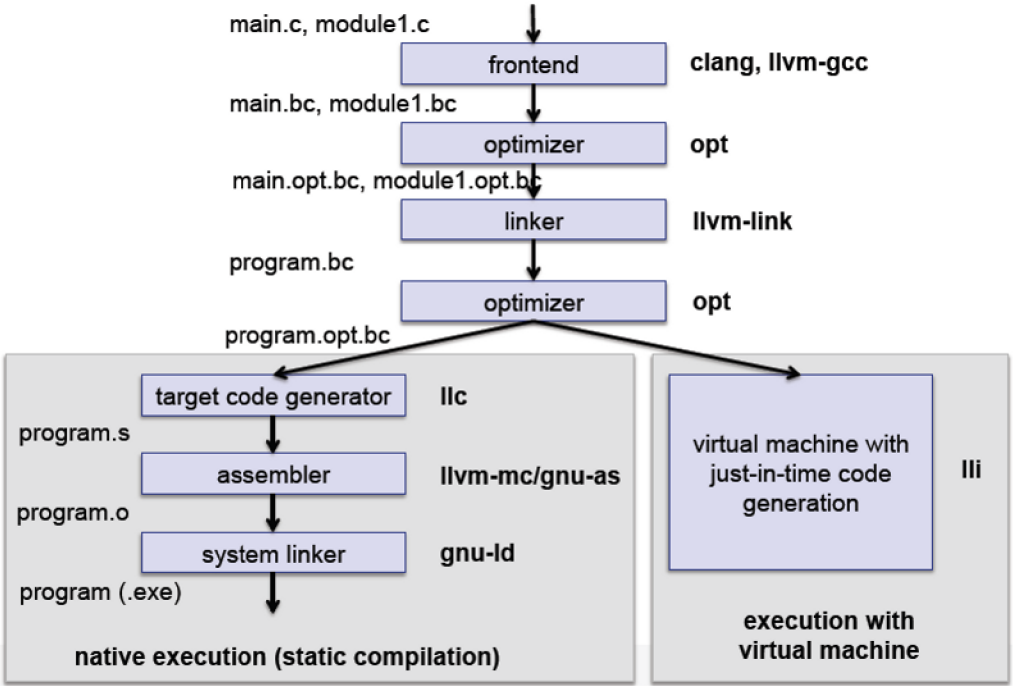
Ďalším projektom je **DragonEgg**, ktorý použije LLVM namiesto GCC back-endu. Napojí sa na GCC a výstup syntaktickej analýzy prevedie do podoby LLVM IR kódu. Aktuálne plne podporuje jazyky Ada, C, C++ a Fortran a čiastočne jazyky ako Go či Java.

Proces prekladu zdrojového kódu je znázornený na obrázku 3.1. Pri analýze kódu nás bude zaujímať časť medzi front-endom a optimalizérom.

3.1 Front-end Clang

Projekt Clang [13] 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. Naviac ponúka API pre vývoj nástrojov využívajúcich syntaktickú a sémantickú analýzu.

Výstupom syntaktickej 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. AST sa následne transformuje na graf riadenia toku (CFG). Pri prevode do LLVM IR je schopný kód optimalizovať. Ponúka štyri úrovne optimalizácie, pričom väčšina prebieha na úrovni 2. Pre vstup statického analyzátora nebudeme uvažovať žiadnu optimalizáciu Clang-om získaného kódu (úroveň 0).



Obr. 3.1: Proces prekladu zdrojových súborov — ľavá vetva predstavuje statickú kompiláciu a pravá JIT, prebrané z [12, snímka č. 5].

3.2 LLVM IR

Medzikód (angl. *assembly language* ev. *intermediate representation*, [14]) tvorí jadro kompilátoru. Jedná sa o jazykovo-nezávislý a silne typovaný systém založený na load-store a RISC inštrukciách. Má jednoznačne definovanú sémantiku, kde každý operand má typ a návratová hodnota je typovaná.

Pracuje s potenciálne nekonečným počtom virtuálnych registrov v **SSA forme** (angl. *Static Single Assignment*, [15]). 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 základných blokov. 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.

3.2.1 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-m:e-i64:64-f80:128-n8:     ; # cielova architektura
  16:32:64-S128"
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
}
```

3.2.2 Podoba IR uložená v pamäti

Hlavným kontajnerom pre IR je `Module` [16]. Všetky objekty sú reprezentované pomocou obojsmerne viazaných zoznamov. `Module` obsahuje zoznam objektov `Function` (definície a deklarácie funkcií) a `GlobalVariable` (globálne premenné). `Function` obsahuje zoznamy objektov `BasicBlock` (základné bloky) reprezentujúce telo funkcie 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.

3.3 LLVM priechod

LLVM priechod (angl. *LLVM Pass*, [17]) 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`: 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.

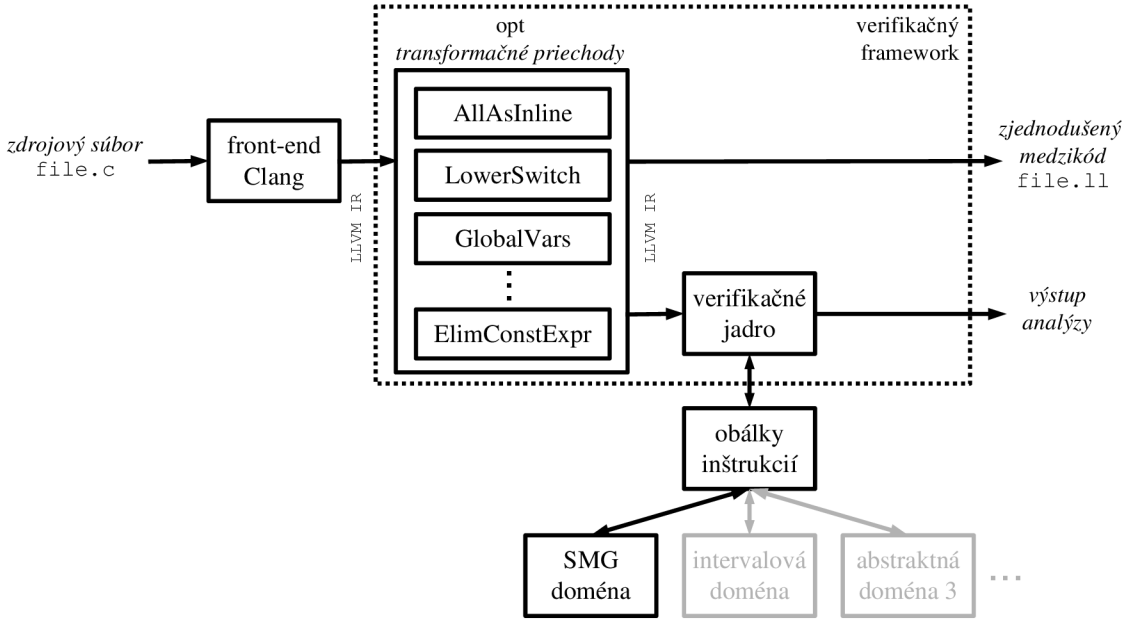
Ako základ statického analyzátora je vhodné voliť najkomplexnejší priechod, a to konkrétne `ModulePass`. Pre potreby vykonania transformácií zjednodušujúcich vstupný LLVM IR môžu postačovať špecifickejšie priechody.

Kapitola 4

Zjednodušenie vstupu pre analýzu

Táto kapitola vychádza z [18]. Rozoberá sa prvý problém spomenutý v Úvode (1), čiže zložitosť vstupného programu, kedy sa pri akomkoľvek rozšírení analyzátoru musí autor tohoto rozšírenia vysporiadať s rôznymi konštrukciami na vstupe. Preto navrhujeme upraviť stávajúcu architektúru tak, ako je prezentované na obr. 4.1. Konkrétne pred vlastné verifikačné jadro bude predradená sada transformačných priechodov, ktoré zjednodušia vstupný kód.

Na obr. 4.1 predpokladáme použitie prostredia Clang/LLVM. Toto prostredie bolo zvolené pre jeho komplexnosť, prehľadnosť medzikódu a ľahkú napojiteľnosť ďalších analýz v dobe prekladu.



Obr. 4.1: Konceptuálny návrh frameworku pre verifikačné nástroje.

Dodajme ešte, že v súvislosti s previazaním výstupu prekladača a verifikačného jadra (či už založeného na SMG alebo ľubovolnej inej symbolickej reprezentácii) bol zvažovaný tiež preklad do špeciálnej vnútornej reprezentácie, či vytvorenie architektúry založenej na spätnom volaní funkcií, pomocou ktorých by bol výsledok prekladu dostupný analyzátorom. Nakoniec bolo ale rozhodnuté vykonať iba zjednodušenie LLVM IR a použiť štandardné

pamäťové reprezentácie LLVM IR. Nad tou budú postupne implementované rôzne vzorové verifikačné cykly (viď sekcia 6.2) a autori rôznych analyzátorov budú môcť niektorý z nich prevziať a upraviť pre svoje vlastné potreby, prípadne si vytvoriť vlastný cyklus.

Cieľom je vytvoriť prostredie, ktoré by bolo čo najjednoduchšie a čo najmenej obmedzujúce pre autorov budúcich analýz z toho hľadiska, ako majú svoju analýzu písať. Preto, narozdiel napr. od prostredia CPAchecker [19] nebude vynucované to, že analýza musí byť písaná formou reakcie na určité udalosti generované pri priechode CFG, že analýza musí podporovať tie či oné operácie a pod. Ďalším dôsledkom tohoto rozhodnutia je, že medzikód, nad ktorým prebieha analýza, bude aj naďalej kompilovateľný (narozdiel od prostredia IKOS [20]).

4.1 Úprava medzikódu

Účelom transformačných priechodov je zjednodušiť podobu LLVM IR, aby sa dosiahlo redukcie množiny konštrukcií jazyka, ktoré musia vytvárané analýzy podporovať.

Jedná sa o redukciu inštrukčnej sady a redukciu druhov operandov jednotlivých inštrukcií.

Štandardná práca s LLVM vyzerá tak, že najprv *predná časť* (Clang) prevedie preklad do LLVM IR. Následne nad ním spustí *optimalizátor* (opt), ktorý vyprodukuje opäť LLVM IR. Nakoniec *zadná časť* vyprodukuje cieľový spustiteľný program. Pre potreby zjednodušenia kódu sme sa v našom prípade rozhodli napojiť hneď za výstup Clang-u (popis v sekcii 3.1) a tým mať väčšiu kontrolu, ktoré optimalizácie sa vykonajú. Pridané transformačné priechody sú vo forme špecifickej optimalizácie, i keď z pohľadu prekladu sa nejedná o reálne optimalizácie. Z oficiálnych [21] potom budú použité len tie, ktoré pre nás majú zmysel, menovite:

- `-mem2reg`,
- `-lowerswitch`.

Pridané transformačné priechody zahŕňajú:

- `-all-inline`,
- `-global-vars`,
- `-lowermem`,
- `-lowerselect`,
- `-elim-const-expr`,
- `-elim-phi`.

4.1.1 Redukcia operandov

Operandom inštrukcie môže byť iná **inštrukcia** (predstavuje register obsahujúci výsledok danej inštrukcie), **konštanta** alebo **konštantný výraz**. V rámci analýz je vhodnejšie pracovať len s jednoduchými konštantami ako sú číslicové prípadne reťazcové literály. LLVM IR však môžu obsahovať aj **zložené literály** vznikajúce pri inicializácií poľa alebo štruktúry. Takéto zložené inicializátory globálnych premenných predstavujú špeciálnu konštrukciu,

ktorú je potom potrebné v analyzátoroch zvlášť implementovať. Tieto konštrukcie možno odstrániť priechodom `-global-vars` (popísaný nižšie). V prípade lokálnej inicializácie, sa v LLVM umelo vytvára statická premenná inicializovaná príslušným zloženým literálom, ktorý je pomocou `llvm.memcpy` nahraný do lokálnej premennej. To odstraňuje `-lowermem` priechod. Daná inicializácia je rozložené na sekvenciu dvoch inštrukcií `getelementptr` rátajúcich adresu a `store`, ktoré už vkladajú jednoduchý literál na danú adresu.

Ďalšou komplikáciou je, ak bude operandom **konštantný výraz**, lebo musíme podporovať viac konštrukcií pri analýze. Naša transformácia vytvorí novú inštrukciu, ktorej výsledok predstavuje vyhodnotenie konštantného výrazu. Táto inštrukcia nahradí konštantný výraz na mieste operandu. Rieši to `-elim-const-expr`.

1. príklad Globálna premenná inicializovaná zloženým literalom v jazyku C a následne v LLVM IR:

```
struct bus {
    int x; struct bus* y;
};
struct bus glob = {.x = -1, .y = 0};

%struct.bus = type { i32, %struct.bus* }
@glob = global %struct.bus { i32 -1, %struct.bus* null }
```

2. príklad Globálna premenná x inicializovaná zloženým inicializátorom (sekvenciou inštrukcií), čiže vnorenými konštantnými výrazmi:

```
int n[] = {1,2,3};
int *x = &n[1];
```

```
@n = global [3 x i32] [i32 1, i32 2, i32 3], align 4
@x = global i32* bitcast (i8* getelementptr (i8, i8* bitcast ([3 x i32]* @n to i8*), i64 4)
to i32*)
```

Rozbitie inicializátoru premennej x pomocou priechodu `-elim-const-expr` po vykonaní priechodu `-global-vars`, ktorý premiestni inicializátor z globálneho priestoru vďaka `store`.

```
; # po -global-vars => instrukcia v umelo vytvorenej funkcii
  store i32* bitcast (i8* getelementptr (i8, i8* bitcast ([3 x i32]* @n
        to i8*), i64 4) to i32*), i32** @x

; # po -elim-const-expr
  %1 = bitcast [3 x i32]* @n to i8*
  %2 = getelementptr i8, i8* %1, i64 4
  %3 = bitcast i8* %2 to i32*
  store i32* %3, i32** @x
```

3. príklad Inicializácia lokálnej premennej *loc* vo funkcii `main()`. LLVM vytvorí privátny globálny symbol `main.loc` a pomocou `llvm.memcpy` sa jeho obsah dostane do nami naprogramovanej premennej *loc*.

```
struct bus loc = {.x = 1, .y = 0};

; # globalny priestor
@main.loc = private unnamed_addr constant
            %struct.bus { i32 1, %struct.bus* null }, align 8
```

```
; # vo funkcii
  %loc = alloca %struct.bus, align 8
  %1 = bitcast %struct.bus* %loc to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %2, i8* bitcast (%struct.bus*
       @main.loc to i8*), i64 16, i32 8, i1 false)
```

Stav po priechode `-lowermem` už nebude obsahovať globálny symbol a inicializácia bude rozdelená. Obsah funkcie `main()` bude nasledovný:

```
  %loc = alloca %struct.bus, align 8
  %1 = getelementptr inbounds %struct.bus, %struct.bus* %loc, i32 0, i32 0
  store i32 1, i32* %1
  %2 = getelementptr inbounds %struct.bus, %struct.bus* %loc, i32 0, i32 1
  store %struct.bus* null, %struct.bus** %2
```

4.1.2 Redukcia inštrukčnej sady

Ďalším aspektom programov, ktorý nie je treba u jednoduchších analyzátorov uvažovať, je členenie programov na funkcie. Tohoto členenia sa dá zbaviť, pre nerekurzívne programy, vložením kódu funkcií na miesto ich volania. Toto samozrejme znižuje efektivitu výslednej analýzy, ale zjednodušuje jej tvorbu. Sploštený kód vytvoríme priechodom `-all-inline`, ktorý je do značnej miery inšpirovaný priechodom `-always-inline`, avšak inlineované funkcie nemusia mať nastavený atribút *„always inline“*.

Ďalšou úpravou, ktorú je možno aplikovať, je odstránenie inštrukcie `alloca` pomocou `-mem2reg`. Tým sa zamedzí alokácii na zásobníku. Dá sa aplikovať iba na programy, kde je každá premenná použitá len raz v `load` a `store` inštrukciách.

Analýzu môže tiež zjednodušiť, ak sa nainicializuje pamäť pre globálne a statické premenné na jednom mieste a nie až pri prvom použití (zamedzuje optimalizácií, kedy sa vytvárajú globálne objekty až pri ich prvom použití). Pomocou `-global-vars` sa vytvorí nová funkcia `__initGlobalVar()`, v ktorej budú priradenia pre všetky globálne premenné (inštrukcie `store`) a bude volaná ako prvá z `main()`. Tento priechod však nemožno použiť pri analýze otvoreného kódu. Bez tohoto priechodu nemožno použiť ani priechod `-elim-const-expr`, pretože by bola ignorovaná inicializácia globálnych symbolov. Tak, ako je naznačené v sekcii 4.1.1 v 2. príklade pre premennú x. Inštrukcia `store` ukladajúca do premennej `@x`, ktorá vznikla po priechode `-global-vars` bude umiestnená vo funkcii `__initGlobalVar()`. Sem sa tiež uložia vygenerované inštrukcie vzťahujúce sa k `store` pomocou `-elim-const-expr`.

Priechod `-lowermem` je užitočný, ak analyzátor ešte nepodporuje spracovanie C štandardných funkcií pre prácu s pamäťou. Tieto funkcie sa nachádzajú v LLVM IR, aj keď ich nepoužije priamo programátor, a to pri inicializácii štruktúr a polí. Takto LLVM implicitne vytvorené `llvm.memcpy` a `llvm.memset` sa dajú nahradiť bez straty informácií. Pri programátorom použitých funkciách sa nekopíruje výplň (angl. *padding*) medzi položkami štruktúr. Nahradenie prebehne, len ak sú zdrojové a cieľové miesta typovo a veľkostne rovnaké. V nasledujúcom prípade sa nepodarí `memcpy()` odstrániť.

```
char *from = "abc";
char to[10];
memcpy(&to,from,4);
```

Nasledujúci program ilustruje odstránenia funkcie `memset()`. Pre polia a `memcpy()` by to bolo obdobné.

```c
#include <stdio.h>
struct bus {
    int x; struct bus* y;
};
int main() {
    struct bus trup;
    memset(&trup, 0, sizeof(struct bus));
    return 0;
}
```

Po zavolaní priechodu `-lowermem` sa vo vygenerovanom LLVM IR medzikóde nahradí volanie funkcie `llvm.memset` dvomi priradeniami, keďže štruktúra má dve položky.

```
; # pred priechodom
%struct.bus = type { i32, %struct.bus* }

define i32 @main() {
  %trup = alloca %struct.bus, align 8
  %1 = bitcast %struct.bus* %trup to i8*
  call void @llvm.memset.p0i8.i64(i8* %1, i8 0, i64 16, i32 8, i1 false)
  ret i32 0
}

declare void @llvm.memset.p0i8.i64(i8* nocapture, i8, i64, i32, i1)

; # po priechode
%struct.bus = type { i32, %struct.bus* }

define i32 @main() {
  %trup = alloca %struct.bus, align 8
  %1 = getelementptr inbounds %struct.bus, %struct.bus* %trup, i32 0, i32 0
  store i32 0, i32* %1
  %2 = getelementptr inbounds %struct.bus, %struct.bus* %trup, i32 0, i32 1
  store %struct.bus* null, %struct.bus** %2
  ret i32 0
}
```

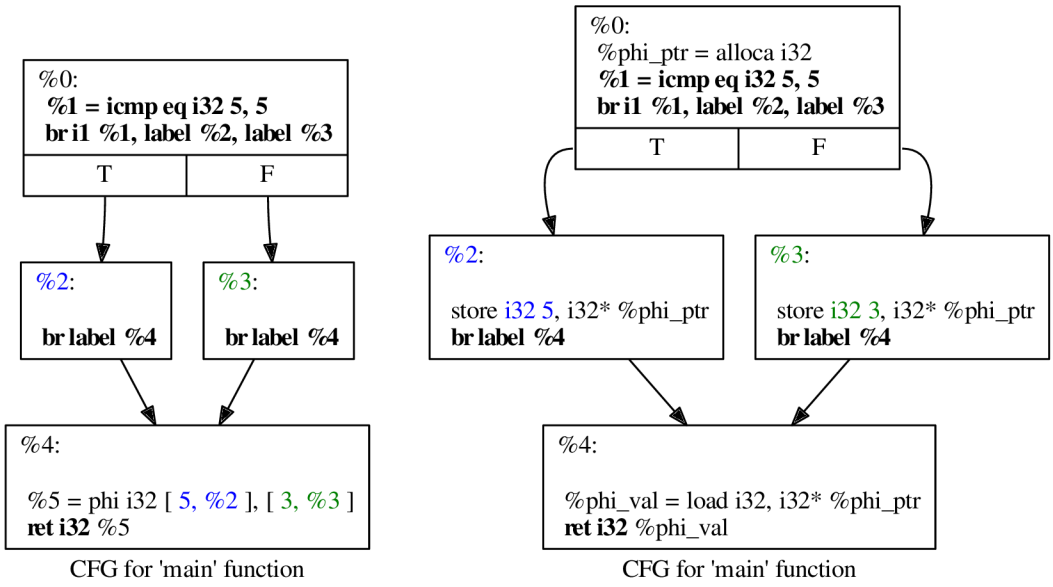
V prípade, že programátorom vytváraná analýza nevyžaduje **SSA formu** [15], môžeme pomocou priechodu `-elim-phi` odstrániť **Φ-inštrukcie** (`phi`). Pozostávajú z postupnosti dvojíc (val_i, bb_i), kde val_i predstavuje hodnotu, ktorá sa priradí do premennej, ak bolo skočené z bloku bb_i. Odstránenie prebieha podľa algoritmu `EliminatePhi` (alg. 1). Na vstupe očakáva CFG G a výstupom bude CFG bez Φ-inštrukcií. Obdobný problém rieši aj oficiálny priechod `-reg2mem`, ten však generuje na začiatku funkcie viac pomocných lokálnych premenných vytvorených inštrukciou `alloca`.

Nasleduje jednoduchý kód v C, pričom pri jeho preklade sa namiesto inštrukcie `select` vytvára Φ-inštrukcia. Je to spôsobené tým, že jeden z priraďovaných prvkov nie je konštanta. Φ-inštrukcia je však dosiahnuteľná len z blokov končiacich nepodmieneným skokom. Pre úsporu miesta bude na tomto aj ďalších dvoch obrázkoch znázornená varianta CFG po priechode `-mem2reg`. Z tohoto dôvodu sa na obrázku 4.2 zobrazuje konštanta 5 namiesto premennej b, lebo tá bola touto úpravou odstránená. Táto úprava však nebráni ilustrácii algoritmu.

```c
int main() {
    int i = 5;
    int *b = &i;
    int j = (i == 5)? *b : 3;
    return j;
}
```

Algoritmus 1: ELIMINATEPHI(G)

1: $S \leftarrow \emptyset$
2: **for** $\forall bb_x \in G$ **do**
3: **for** $\forall i \in bb_x$ **do**
4: **if** i je nepodmienený skok na bb_y **then**
5: **while** bb_y začína $j : var \leftarrow \Phi(var_x, bb_x)$ **do**
6: vlož pred i v bb_x: $var \leftarrow var_x$
7: $S \leftarrow S \cup \{j++\}$
8: **if** i je podmienený skok na bb_y **then**
9: **while** bb_y začína $j : var \leftarrow \Phi(var_x, bb_x)$ **do**
10: vlož bb_{xy} do G
11: nahraď v i: bb_y za bb_{xy}
12: vlož do bb_{xy}: $var \leftarrow var_x$ a nepodmienený skok na bb_y
13: $S \leftarrow S \cup \{j++\}$
14: zmaž $\forall i \in S$ // všetky Φ-inštrukcie
15: **return** G



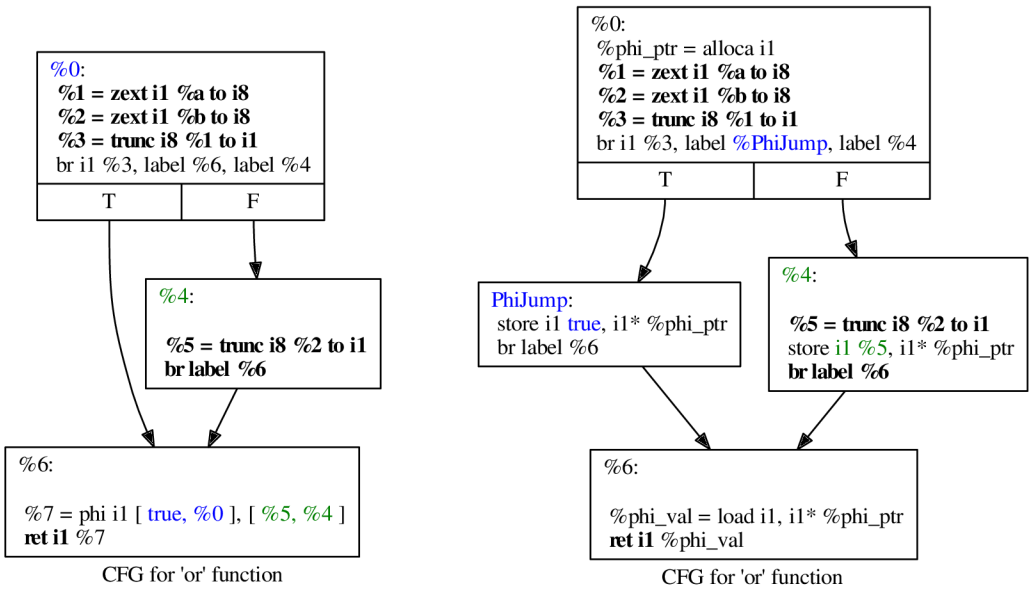
(a) CFG pred priechodom.

(b) CFG po priechode.

Obr. 4.2: Zmena LLVM IR pre funkciu `main()` vďaka priechodu `-elim-phi`. Grafy sú generované pomocou priechodu `-dot-cfg`.

Na obrázku 4.3 je znázornená komplikovanejšia varianta, a to, ak sa podmienene skáče do bloku začínajúceho Φ-inštrukciu. CFG predstavuje nasledujúcu funkciu `or()` v jazyku C, kde prekladač použije skrátené vyhodnocovanie logického operátoru:

```
#include <stdbool.h>
bool or(bool a, bool b) {
    return a || b;
}
```



(a) CFG pred priechodom.

(b) CFG po priechode.

Obr. 4.3: Zmena LLVM IR pre funkciu `or()` vďaka priechodu `-elim-phi`. Grafy sú generované pomocou priechodu `-dot-cfg`.

Ďalšiou špecifickou konštrukciou je **ternárny výraz** v C, ktorý je v LLVM IR reprezentovaný ako inštrukcia `select`, ktorú možno odstrániť použitím priechodu `-lowrselect`. Ak základný blok bb obsahuje danú inštrukciu $var \leftarrow (cond)\ ?\ var_{true} : var_{false}$, rozdelíme ho v mieste inštrukcie a jeho druhá časť bude tvoriť bb'. Inštrukciu `select` nahradíme podmieneným skokom na bb_{true} a bb_{false}. Potom bb_{true} bude obsahovať priradenie $var \leftarrow var_{true}$ a nepodmienený skok na bb'. bb_{false} bude obsahovať priradenie $var \leftarrow var_{false}$ a nepodmienený skok na bb'. Ilustrácia priechodu je znázornená na obr. 4.4, ktorý reprezentuje CFG nasledujúceho programu v C:

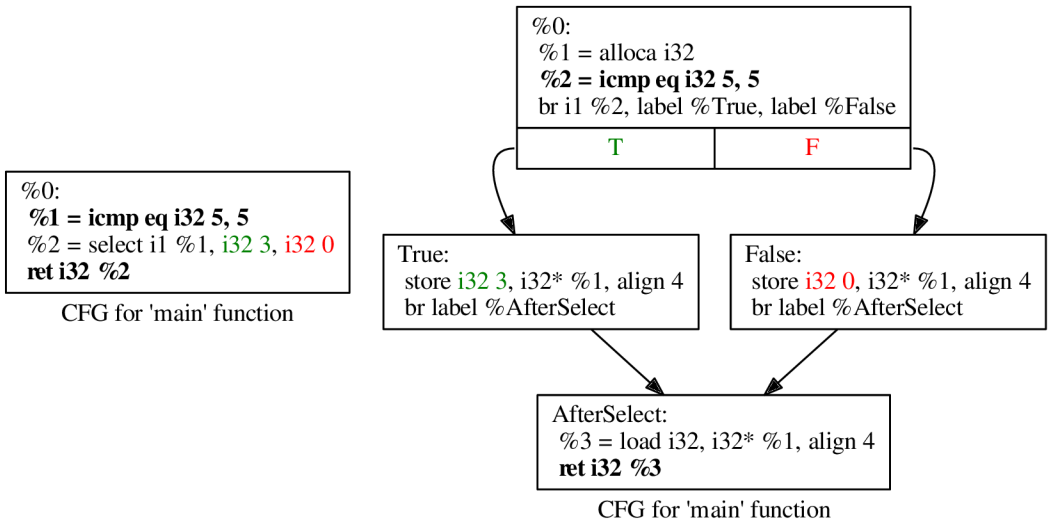
```
int main() {
    int a = 5;
    int b = (a == 5)? 3 : 0;
    return b;
}
```

Na záver možno bez straty informácií odstrániť aj LLVM inštrukciu `switch` reprezentujúcu konštrukciu switch v jazyku C. Jedná sa už o oficiálny priechod `-lowerswitch`, preto nebude ďalej rozoberaný.

4.2 Implementácia a testovanie

Popísané priechody boli implementované[1] v jazyku C++ pomocou rozhrania, ktoré je uvedené v sekcii 3.3. Pomocou triedy `BasicBlockPass` je implementovaný transformačný priechod `-elim-const-expr` ako `EliminateConstExprPass`, lebo mení inštrukcie len v rozsahu základného bloku. Všeobecnejšiu triedu `FunctionPass` používajú priechody `-elim-phi` ako

[1] Framework „Pro Statické Analyzatory“, dostupný na: https://github.com/VeriFIT/ProStatA



(a) CFG pred priechodom. (b) CFG po priechode.

Obr. 4.4: Zmena LLVM IR pre funkciu `main()` vďaka priechodu `-lowerselect`. Grafy sú generované pomocou priechodu `-dot-cfg`.

`EliminatePHIPass` a `-lowerselect` ako `LowerSelectPass`, pretože menia podobu funkcie a vytvárajú nové základné bloky. Trieda `Inliner` je špecifickou triedou `CallGraphSCCPass`, kde sa využíva graf volaní, z ktorého sa zistí či a kam sme schopný vložiť funkciu a bola použité pre priechod `-all-inline` ako `AllAsInlinePass`. V najvšeoubecejšej triede `ModulePass` sa pracuje s globálnymi symbolmi a preto bola použitá v priechodoch `-global-vars` ako `GlobalVarsPass` a `-lowermem` ako `LowerMemIntrinsicPass` s parametrom `justGV`. Ak je nastavený na `true`, eliminujú sa len inicializácie lokálnych premenných, pri ktorých sa vytvára privátny globálny symbol.

Priechody sa spúšťajú pomocou rovnomenných prepínačov optimalizéru `opt`. Vstupom je program v podobe LLVM získaný napr. týmto spôsobom:

```
clang -S -emit-llvm source.c -o - | opt -load libpasses.so -<meno priechodu>
```

Pre otestovanie funkčnosti jednotlivých transformačných priechodov boli vytvorené regresné testy. Pozostávajú z jednoduchých programov v jazyku C a pokrývajú rôzne konštrukcie vznikajúce v LLVM IR. Overuje sa zhodnosť výstupov pred a po použití priechodu. Okrem testovania jednotlivých priechodov sa testovala aj ich kombinácia.

Kapitola 5

Optimalizácia paralelnej nadstavby Predatora

Táto kapitola je založená na výsledkoch publikovaných v [22] v rámci konferencie TACAS'16. Kapitola pojednáva o paralelnej nadstavbe nástroja Predator vytvorenej pre SV-COMP'15. Rozoberá jednotlivé parametre analýzy a súbežné spúšťanie takto upravených Predatorov s cieľom získať takú kombináciu nastavení, aby bola analýza čo najrýchlejšia a najpresnejšia. Výsledná kombinácia Predatorov bola prezentovaná na SV-COMP'16 [23] a obdržala zlatú medailu v kategórii *Heap Data Structures*.

5.1 Predator Hunting Party

V tejto sekcii sa nachádza samotný popis paralelnych Predatorov (Predator Hunting Party, PredatorHP[1]).

Pôvodný PredatorHP (pre SV-COMP'15) pozostával z klasického Predatora (*Predator verifier*), ktorý je spoľahlivý. Používa abstrakciu, čím nadaproximováva priestor, čo však môže viesť na falošné hlásenia. Z tohoto dôvodu sa berie jeho výsledok, len ak analýza označí program za korektný. V prípade, ak nájde chybu, je spúšťaný **neobmedzený Predator** (*BFS Hunter*). Operátor *join* je povýšený na izomorfizmus (neuvažujeme spájanie), stavový priestor sa prehľadáva do šírky pričom nie je nijako obmedzený (len časovými a pamäťovými zdrojmi). Ďalej je vypnutá abstrakcia nad zoznamami, ale zostáva zapnuté obmedzenie pre aritmetické operácie. V prípade, že skončí, je jeho výsledok spoľahlivý, čiže ak program označí za korektný, vravíme, že je korektný a v opačnom prípade je chybný.

Okrem týchto spoľahlivých Predatorov sú spúšťané aj tzv. **obmedzené Predatory** (*DFS huntery*). Podobne ako *BFS hunter* je v nich vypnutá abstrakcia nad zoznamami a operátor *join* je povýšený na izomorfizmus. Je však použité obmedzené prehľadávanie stavového priestoru do hĺbky, pričom obmedzením je počet GIMPLE inštrukcií generovaných prekladačom GCC. V pôvodnom PredatorHP boli ad hoc vytvorené tri *DFS huntery* a to s obmedzeniami 400, 700 a 1000 inštrukcií. Ich výsledok sa berie v úvahu, len ak nájdu chybu.

Po akceptovaní výsledku hocakého Predatora podľa vyššie uvedených podmienok, sú ostatný Predatory ukončený. V prípade neprijatia žiadneho výsledku, je výstup analýzy neznámy (*unknown*).

[1] Predator Hunting Party, dostupný na: http://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp/

Výsledné nastavenie a spúšťanie verzií Predatora je riešené pomocou skriptov v Bashi a Python3.

5.2 Optimalizácia nastavení Predatorov

Pre SV-COMP'16 boli znovu prehodnotené nastavenia atribútov Predatorov vzhľadom na výsledky získané opakovaným spúšťaním nad testovaciou množinou[2] kategórie *Heap Data Structures*. Tá pozostáva z dvoch subkategórií: *HeapReach* (analýza dátových štruktúr na halde) a *HeapMemSafety* (overovanie *memory safety* chýb — neplatné dereferencie, neplatné uvoľnenia...).

Graf na obrázku 5.1 znázorňuje maximálnu hĺbku prehľadávaného stavového priestoru v počte GIMPLE inštrukcií vzhľadom k počtu programov s touto hĺbkou z testovacej množiny SV-COMPu. Bolo zistené, že (a) vo viac ako 80% (uvažované programy s chybami) prípadoch je chyba nájdená s limitom 200 inštrukcií. (b) V asi 96% prípadoch sa chyba nájde do 900 inštrukcií. (c) Vo zvyšných prípadoch by protipríklad pri chybe obsahoval viac ako 50 000 inštrukcií. Čo je príliš veľa, aby bolo také nastavenie použité nad všetkými testovacími príkladmi. V niektorých testovacích prípadoch môže byť generovaný protipríklad vedúci k chybe podstatne dlhší, ale stavový priestor nie je tak rozvetvený, takže chybu stihne nájsť *BFS hunter*. Z ohľadom na zistené skutočnosti boli v ďalšej verzii PredatoraHP použité: *Predator verifier*, *BFS Hunter* a dva *DFS huntery* s obmedzeniami 200 a 900 inštrukcií. Pričom všetky budú spúšťané naraz.

Zmenou spracovania výstupu Predatorov v používanom Bash-skripte a úpravou Python-skriptu sa ušetril celkový čas behu PredatoraHP až o cca 38%. Tým sa v niektorých prípadoch zamedzilo prekročeniu povoleného času na analýzu a výsledok analýzy nebol *unknown*.

Nasledujúci graf na obr. 5.2 znázorňuje, koľko času spotrebovali verifikačné nástroje (vrátane nástroja PredatorHP) spustené nad testovacou množinou *Heap Data Structures* vzhľadom k získaným bodom. Body boli prideľované podľa výsledku analýzy znázornené v tab. 5.1. PredatorHP získal 298 bodov za 1 100 sekúnd.

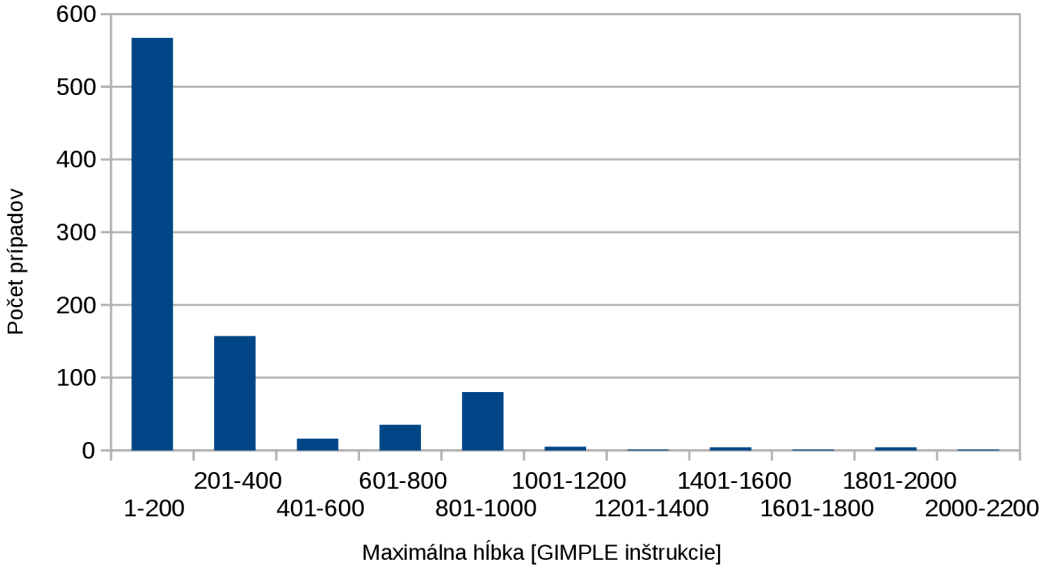
Experimenty sa spúšťali pod Linuxom v systéme s CPU Intel Xeon E5-2650 v2 @ 2,60 GHz s počtom jadier 32, frekvenciou 3,4 GHz a RAM 135 149 MB, pričom boli obmedzené zdroje nasledovne. Časový limit bol nastavený na 900 s a maximálna pridelená pamäť bola 15 000 MB na 8 jadrách.

Body	Výsledok	Popis
0	*unknown*	ukončená analýza z dôvodu chyby či vyčerpania zdrojov
+1	správne *false*	správne určený chybný program a dodaný protipríklad
−16	nesprávne *false*	nájdenie chyby v správnom programe — neúplná analýza
+2	správne *true*	správne určený korektný program
−32	nesprávne *true*	nenájdenie chyby v chybnom programe — nespoľahlivá analýza

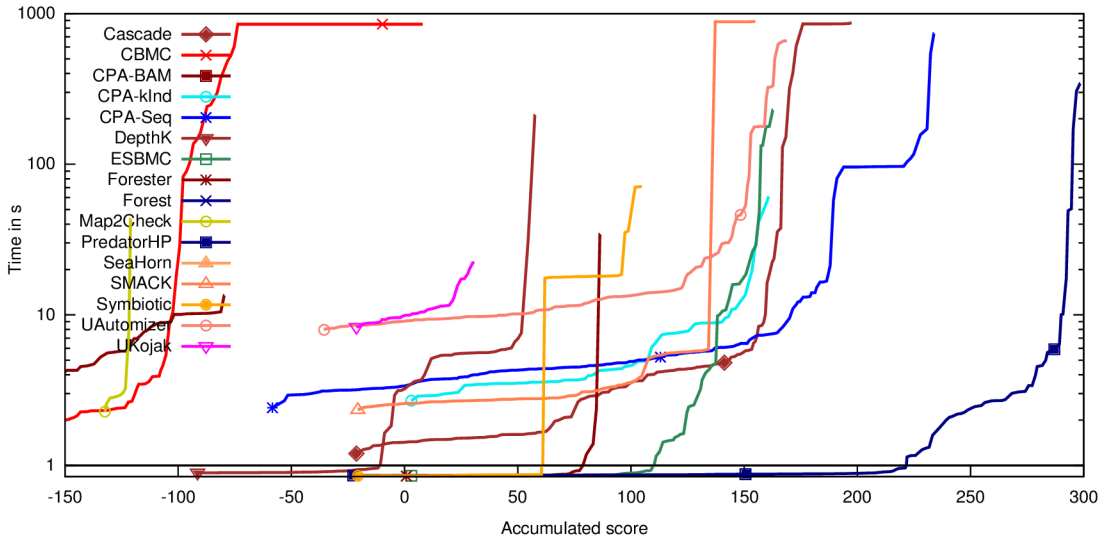
Tabuľka 5.1: Bodové ohodnotenie výsledku analýzy v súťaži SV-COMP'16 (prebrané z [23]).

Podobná analýza môže byť vykonaná aj nad inými testovacími sadami pre ešte lepšiu optimalizáciu.

[2]SV-COMP Benchmark Verification Tasks, dostuné na: http://sv-comp.sosy-lab.org/2016/benchmarks.php



Obr. 5.1: Stĺpcový graf znázorňujúci koľko testovacích prípadov z SV-COMP'16 testovacej sady malo akú maximálnu hĺbku v počte GIMPLE inštrukcií.



Obr. 5.2: Kvantilový graf výsledkov súťaže SV-COMP'16 pre kategóriu *Heap Data Structures*, prebrané z [24].

Kapitola 6

Návrh novej architektúry pre analýzu pomocou SMG

Táto kapitola rozširuje popis návrhu prezentovaný v [18]. Obsahom úzko nadväzuje na kapitolu 4. Najprv sú popísané nedostatky nástroja Predator vedúce k jeho novému návrhu. Následne je popísané samotné jadro analyzátora a zameranie sa na SMG doménu.

6.1 Obmedzenia stávajúcej implementácie

Táto sekcia je zameraná na hraničné limity stávajúcej implementácie nástroja Predator. Zásadným problémom Predatora je jeho prílišná optimalizácia, ktorá zamedzuje rozšíriteľnosť domény o iné abstraktné objekty, napr. neschopnosť spracovať skip-listy alebo stromy.

Pri analýze [10] pamäťového alokátoru založeného na **NSS arénach**[1], sa ukázala potreba práce so smerníkmi, ktorých hodnota nie je určená presne, ale nachádza sa v určitom intervale adries. Naviac sa ukázala potreba sledovať si usporiadanie adries v pamäti. Zatiaľ čo prvý problém sa podarilo v súšasnej implementácii zohľadniť, druhý problém už nebol riešený, pretože doplnenie usporiadania nad adresami do reprezentácie SMG sa ukázalo príliš komplikované.

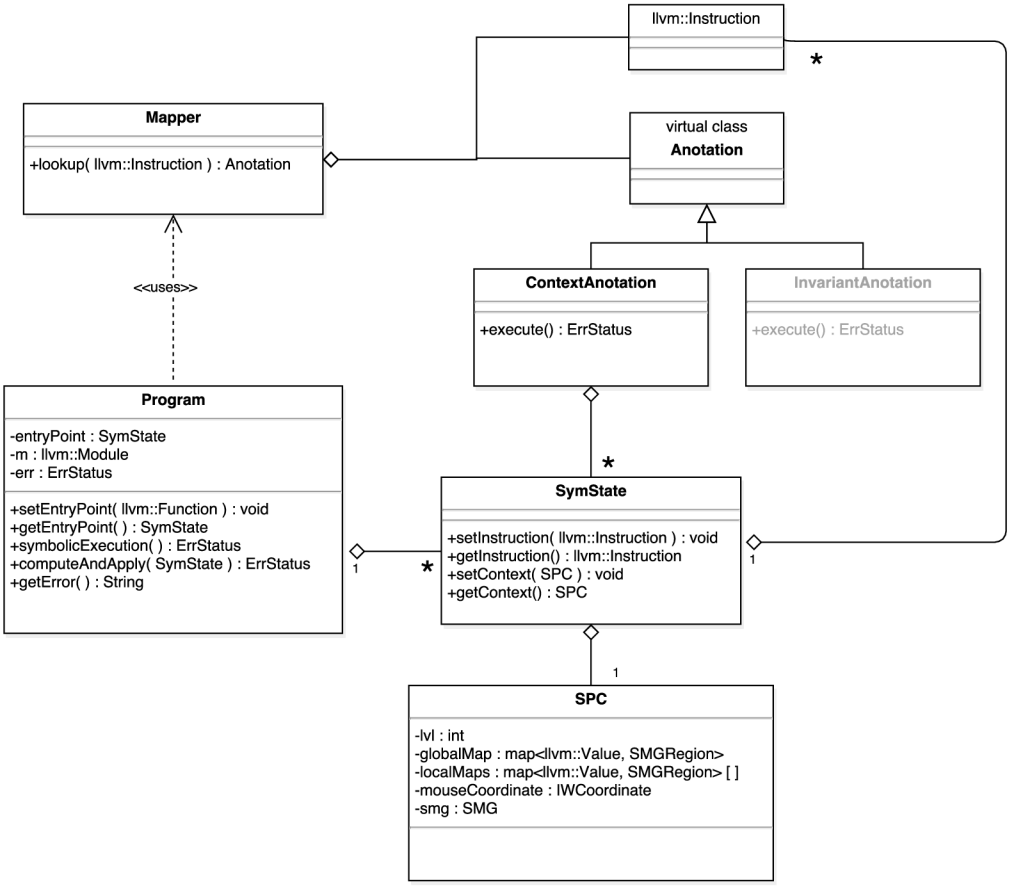
V Predatorovi sú **aritmetické operácie** vykonávané presne po určitú pevne zadanú hranicu. Nad touto hranicou je premenným priradená hodnota *unknown*, čiže o nej nevieme povedať nič. Naviac pri adresách sú použité intervaly s pevnými hranicami. Taktiež sa neuvažuje nad obmedzeniami pre dve a viac premenných. V novom návrhu by oboje malo byť zovšeobecnené pomocou kombinácie SMG s vhodnými doménami pre popis množín číselných hodnôt: intervaly s premennými hranicami, oktagony [25], DBM [26] a pod.

6.2 Verifikačné jadro

Ako je naznačené na obrázku 4.1, vstupom pre verifikačný cyklus bude LLVM IR medzikód v podobe `llvm::Module` ako je popísané v sekcii 3.2.2. Ten bude zapuzdrený do triedy `Program`, ktorá riadi celú analýzu, ako je zrejmé na objektovom návrhu statického analyzátora na obrázku 6.1. `Program` bude obsahovať aj vstupný bod analýzy. Jedná sa o stav programu `State` tvorený inštrukciou a prázdnym kontextom, ktorý je v Predatorovi reprezentovaný pomocou SPC (viď sekcia 2.3.4). V prípade klasickej analýzy sa prehľadávanie

[1] NSS Memory allocation, dostupné na: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Memory_allocation

začína od prvej inštrukcie vo funkcii `main()`. Pri analýze otvoreného kódu sa bude prehľadávať z určitej funkcie, no nebude to `main()` a teda veľa vecí nebude inicializovaných.

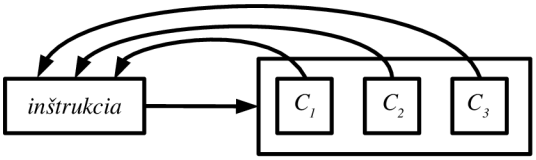


Obr. 6.1: Objektový návrh verifikačného jadra.

Tvorca analyzátora dostane `Program` obsahujúci už zjednodušený `llvm::Module` vďaka ním nastaveným transformačným priechodom. V ňom si nastavý vstupný bod a môže sa rozhodnúť, či bude prehľadávať stavový priestor pomocou DFS alebo BFS. V prvom prípade sa budú rozpracované stavy ukladať do fronty a v druhom prípade na zásobník. Potom sa už môže program symbolicky spustiť podľa algoritmu 2 a pre vyhodnocovanie jednotlivých inštrukcií alg. 3.

Keďže si chceme pri inštrukciách (`Instruction`) uchovávať dodatočné informácie, ako sú invarianty, počiatočné a koncové podmienky (angl. *pre/postcondition*) a najmä zoznam kontextov, v ktorých sa nachádzal program na tomto mieste, programátor si musí dodefinovať triedu `Anotation`. Tá bude pomocou asociačného poľa priradená každej inštrukcií:

```
std::unordered_map<llvm::Instruction*, std::unique_ptr<Anotation>>
```



Obr. 6.2: Ilustrácia zoznamu kontextov (SPC) priradených k jednej inštrukcii.

Programátor tiež musí doimplementovať jednotlivé obálky inštrukcií LLVM IR, ktoré budú spúšťané vo verifikačnom cykle (virtuálna metóda `execute()`). Taktiež musí doimplementovať operátory *join*, abstrakciu a *entailment checking*, ktorý otestuje, či pridávaný stav programu už nie je pokrytý existujúcim stavom (v rámci metódy `add()`).

Algoritmus 2: SymbolicExecution(program, worklist)

Vstup: *program* obsahuje llvm::Module a vstupný SymState
Výstup: *status* vracia OK, ak analýza prebehla bez chyby, inak je nastavený príznak chyby

```
1: start ← program.getEntryPoint()
2: worklist.add(start)
3: status ← OK
4: do
5:     state ← worklist.getAndRemove()
6:     status ← program.ComputeAndApply(state)
7: while status = OK and !worklist.empty()
8: return status
```

6.3 Abstraktná doména

Pre potreby reimplementácie Predatora je čiastočne implementovaný formálny model[2] predstavený v kapitole 2.3. V náveznosti na predchádzajúcu sekciu, je symbolický stav reprezentovaný LLVM inštrukciou a symbolickou programovou konfiguráciou (viď 2.3.4), ktorý obsahuje aktuálnu kópiu SMG po symbolickom vykonaní danej inštrukcie a mapovanie všetkých používaných premenných programu. Každej inštrukcii zodpovedá len priradený zoznam stavov (obsah triedy `Anotation`). Zatiaľ nie je nič viac potrebné.

Mapovanie premenných programu je riešené hierarchicky. O správne mapovanie premenných programu na regióny SMG sa starajú prevodníky typu `Value*` (LLVM IR) → `Region` (SMG doména). Jeden sa vytvára pre globálny priestor, v prípade, že program obsahuje globálne symboly ako sú globálne a statické premenné, globálne aliasy, a niekoľko prevodníkov pre lokálne priestory. V prípade rekurzie bude rovnaká lokálna LLVM premenná `%var` v rôznych úrovniach zanorenia namapovaná na iný región. Prevodníky sa prehľadávajú hierarchicky. Počnúc od aktívneho lokálneho priestoru, práve spracovávaného rámca zásobníka (angl. *stack frame*), po globálny priestor. Pri každom zavolaní LLVM inštrukcie `call` sa inkrementuje aktuálny priestor a pri každom návrate z funkcie sa dekrementuje.

[2]reimplementácia SMG z Javy do C++, dostupná na: https://github.com/VeriFIT/smg3

Algoritmus 3: ComputeAndApply(state)

Vstup: SymState *state* predstavuje analyzovaný stav programu
Výstup: *status* vracia OK, ak spustenie inštrukcie prebehlo bez chyby, inak chybu

```
1: instruction ← state.getInstruction()
2: if instruction je podmienený skok then
3:     for i ← 0 to 1 do
4:         bb ← instruction.getOperand(i)// základný blok kam skočiť
5:         instruction ← bb.begin()
6:         anotation ← map.lookup(instruction)
7:         anotation.contexts.add(state)
8:         worklist.add(state)
9:     return OK
10: if instruction nie je nepodmienený skok then
11:     anotation ← map.lookup(instruction)
12:     (status, setOfStates) ← anotation.execute() // symbolické vyhodnotenie
        inštrukcie
13:     if status ! = OK then
14:         return status
15:     instruction + +
16: else
17:     bb ← instruction.getOperand(0)      // základný blok kam skočiť
18:     instruction ← bb.begin()
19:     setOfStates ← {state}
20: anotation ← map.lookup(instruction)
21: for ∀states ∈ setOfStates do
22:     anotation.contexts.add(state)       // pridáva iba ak nie je spracovaný,
23:     worklist.add(state)                 // či pokrytý iným stavom
24: return OK
```

Kapitola 7

Záver

Hlavným cieľom práce bolo analyzovať nedostatky súčasnej implementácie nástroja Predator a navrhnúť nové prostredie pre vývoj statických analyzátorov, konkrétne Predatora. Súčasťou toho bolo naštudovanie princípov, na ktorých je postavený. Nástroj Predator bol analyzovaný z troch rôznych uhlov. Prvým bola zložitosť vstupného formátu programov, ktoré sa analyzujú. Rozhodlo sa, že sa ustúpi od prekladača GCC, ktorý je primárne používaný v súčasnom Predatorovi, a nové prostredie tak nadviaže na Clang/LLVM. Bolo tak rozhodnuté pre jeho komplexnosť a prehľadnosť medzikódu. Na zjednodušenie konštrukcií tohoto medzikódu boli navrhnuté, implementované a otestované transformačné priechody. Tento koncept bol prezentovaný na študentskej konferencii Excel@FIT'16, kde bol aj ocenený jedným z partnerov konferencie.

Ďalšou oblasťou bola analýza paralelnej nadstavby nástroja Predator navrhnutá v roku 2015. Súčasťou tohoto bodu bolo vykonávané opakované spúšťanie Predatorov v rôznych konfiguráciách za účelom nájdenia optimálnej kombinácie Predatorov. Spúšťanie sa vykonávalo nad testovacou množinou dodávanou k medzinárodnej súťaži verifikácie softvéru SV-COMP'16, na ktorej náš nástroj — PredatorHP (riešné spolu s T. Vojnarom, P. Peringrom a M. Kotounom) získal zlatú medailu v kategórii *Heap Data Structures*. Výsledky testovania boli prezentované na konferencii TACAS'16 v rámci bloku SV-COMP.

Poslednou oblasťou bol návrh štruktúry samotného jadra analyzátoru založeného na SMG. Narozdiel od súčasného nástroja Predator, bol kladený dôraz na budúcu rozšíriteľnosť a prehľadnosť. Boli riešené možnosti napojenia ďalších domén, napr. pre aritmetické operácie.

V budúcnosti sa plánuje doimplementovať návrh prostredia pre vývoj analyzátorov a nad ním implementovať nového Predatora. Rozšíriť túto implementáciu do funkcionality stávajúceho Predatora tak, aby podporoval aj nízkoúrovňové operácie s pamäťou a prípadne ho rozšíriť o nové domény pre číselnú aritmetiku, či analýzu nepodporovaných dátových štruktúr, ako sú skip-listy alebo stromy.

Slovník pojmov

API	**rozhranie pre programovanie aplikácií** (angl. *application programming interface*)
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 [27, str. 529]
BFS	**prehľadávanie do šírky** (angl. *breadth first search*)
CFG	**graf riadenia toku** (angl. *control flow graph*) je graf reprezentujúci program (jednu funkciu), v ktorom uzly sú BB
DFS	**prehľadávanie do hĺbky** (angl. *depth first search*)
DLS	*doubly-linked segment*
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/
invariant cyklu	podmienka, ktorá musí byť splnená pred a po vykonaní každého cyklu
JIT	*Just In Time* metóda prekladu programu
LLVM	prekladový systém *Low-Level Virtual Machine*
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
RISC	**redukovaná inštrukčná sada** (angl. *reduced instruction set computer*) obsahuje jednoduché inštrukcie (vykonávané zväčša v 1 takte)
SLS	*single-linked segment*
SSA	*static single assignment* forma
SV-COMP	*Competition on Software Verification* http://sv-comp.sosy-lab.org/
VeriFIT	výskumná skupina automatizovanej analýzy a verifikácie na FIT, VUT http://www.fit.vutbr.cz/research/groups/verifit/

Literatúra

[1] Vojnar, T. a Lengál, O. *Prednášky FAV* [online]. [cit. 2015-11-26]. Dostupné z: http://www.fit.vutbr.cz/study/courses/FAV/public.

[2] Šoková, V. *Vývoj LLVM adaptéru pro infrastrukturu Code Listener*. Brno: FIT VUT v Brně, 2014. 39 s. Bakalářská práce.

[3] King, J. C. Symbolic Execution and Program Testing. *Commun. ACM*. July 1976, roč. 19, č. 7. S. 385–394. ISSN 0001-0782.

[4] Cousot, P. a Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*. [b.m.]: ACM, 1977. S. 238–252.

[5] Berdine, J., Calcagno, C., Cook, B. et al. Shape Analysis for Composite Data Structures. In *Proc. of CAV'07*. [b.m.]: Springer, 2007. S. 178–192. LNCS, sv. 4590.

[6] Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. [b.m.]: IEEE Computer Society, 2002. S. 55–74.

[7] Sagiv, M., Reps, T. a Wilhelm, R. Parametric Shape Analysis via 3-valued Logic. In *Proc. of POPL'99*. [b.m.]: ACM, 1999. S. 105–118. ISBN 1-58113-095-3.

[8] Dudka, K., Peringer, P. a Vojnar, T. *Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic*. 2011. 23 s. Dostupné z: http://www.fit.vutbr.cz/research/view_pub.php?id=9723.

[9] Holík, L., Lengál, O., Rogalewicz, A. et al. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*. [b.m.]: Springer, 2013. S. 740–755. LNCS, sv. 8044. ISBN 978-3-642-39798-1.

[10] Dudka, K., Peringer, P. a Vojnar, T. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*. [b.m.]: Springer, 2013. S. 215–237. LNCS, sv. 7935.

[11] Lattner, C. *The LLVM Compiler Infrastructure* [online]. 2007 [cit. 2014-04-27]. Dostupné z: http://llvm.org/.

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

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

[14] *LLVM Language Reference Manual* [online]. 2003, 2016-01-08 [cit. 2016-01-09]. Dostupné z: http://llvm.org/docs/LangRef.html.

[15] 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. ISSN 0362-1340.

[16] *LLVM Programmer's Manual* [online]. 2003, 2016-01-08 [cit. 2016-01-09]. Dostupné z: http://llvm.org/docs/ProgrammersManual.html.

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

[18] Šoková, V. Prostredie pre vývoj statických analyzátorov nad LLVM. In *Excel@FIT'16* [online]. 2016 [cit. 2016-05-24]. Dostupné z: http://excel.fit.vutbr.cz/submissions/2016/012/12.pdf.

[19] Beyer, D. *CPAchecker: The Configurable Software-Verification Platform* [online]. [cit. 2016-01-09]. Dostupné z: http://cpachecker.sosy-lab.org/.

[20] Koga, D. *IKOS: Inference Kernel for Open Static Analyzers* [online]. [cit. 2016-01-09]. Dostupné z: http://ti.arc.nasa.gov/opensource/ikos/.

[21] *LLVM's Analysis and Transform Passes* [online]. 2016-04-08 [cit. 2016-04-09]. Dostupné z: http://llvm.org/docs/Passes.html.

[22] Vojnar, T., Peringer, P., Šoková, V. et al. Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark (Competition Contribution). In *Proc. of TACAS'16*. [b.m.]: Springer, 2016. S. 942–945. LNCS, sv. 9636. ISBN 978-3-662-49674-9.

[23] Beyer, D. *SV-COMP 2016* [online]. 2016 [cit. 2016-05-23]. Dostupné z: http://sv-comp.sosy-lab.org/2016/.

[24] Beyer, D. *Results of the Competition in Heap Category* [online]. 2016-01-20 [cit. 2016-05-25]. Dostupné z: http://sv-comp.sosy-lab.org/2016/results/results-verified/quantilePlot-Heap.svg.

[25] Miné, A. The octagon abstract domain. *Higher-Order and Symbolic Computation.* 2006, roč. 19, č. 1. S. 31–100. ISSN 1573-0557.

[26] Jeannet, B. a Miné, A. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proc. of CAV'09*. [b.m.]: Springer, 2009. S. 661–667. LNCS, sv. 5643.

[27] Dudka, K., Peringer, P. a Vojnar, T. An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proc. of EUROCAST'11*. 2012. S. 527–534. LNCS, sv. 6927.

Prílohy

Zoznam príloh

Príloha A

Hierarchia tried v LLVM

Hierarchia tried prevzatá z LLVM API dokumentácie prekladača Clang/LLVM 3.8. 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`


▼ 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

Príloha B

Obsah DVD

Adresárová štruktúra priloženého kompaktného disku.

```
.
|-- llvm-3.8                zdrojové súbory prekladača clang/LLVM
|-- PredatorHP              zdrojové súbory nástroja Predator Hunting Party
|    |-- ...
|    \-- sv-benchmark       testovacia sada z SV-COMP'16
|-- ProStatA                zdrojové súbory LLVM transformačných priechodov
|    |-- ...
|    \-- sv-benchmark       testy
|-- doc                     zdrojové súbory technickej správy
|-- projekt.pdf             táto technická správa
\-- README                  tento popis
```