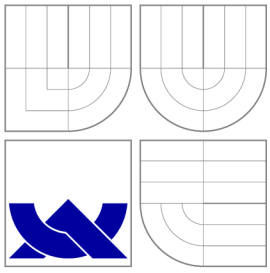
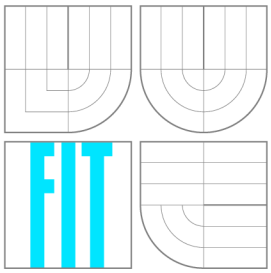


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

VYLEPŠENÍ ANALÝZY ŽIVÝCH PROMĚNNÝCH POMOCÍ POINTS-TO ANALÝZY

IMPROVEMENT OF LIVE VARIABLE ANALYSIS USING POINTS-TO ANALYSIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL RAISKUP

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAMIL DUDKA

BRNO 2012

Abstrakt

Jazyky, jako je C, hojně využívají práce s ukazateli. Implemetace dynamických datových struktur vázaných ukazateli a operací nad nimi však není jednoduchá – významně zvyšuje rizika zanášení chyb do zdrojových kódů. Jedna z cest, jakými lze eliminovat množství těchto chyb, je použití statické analýzy. Tato práce se tedy zabývá vylepšením architektury Code Listner, která nabízí rozhraní pro tvorbu statických analyzátorů. Vlastností tohoto rozhraní je, že poskytuje takovému analyzátoru k rozboru potřebné informace o programu – ku příkladu databázi proměnných, graf toku řízení čí graf volání funkcí. Součástí implementace Code Listeneru je také algoritmus pro analýzu živých proměnných, umožňující odstranit, neboli zabít proměnné, které nejsou v daném místě grafu toku řízení potřeba. Původní algoritmus ale nedovedl z důvodu bezpečnosti zabít žádné proměnné, na něž byla kdekoliv ve zdrojovém kódu vzata adresa. Předpokládalo se, že taková proměnná může být zpřístupněna pomocí reference kdekoliv v programu. Cílem práce tedy bylo navrhnout a implementovat algoritmus pro points-to analýzu, která dovede vyloučit existenci některých referencí v daném kontextu programu a umožní tedy zefektivnit analýzu živých proměnných.

Abstract

Languages such as C use pointers very heavily. Implementation of operations on dynamically linked structures is, however, quite difficult. This can cause the programmer to make more mistakes than usual. One method for dealing with this situation is to use the static analysis tools. This thesis elaborates on the extension to the Code Listener architecture which is an interface for building static analysis tools. Code Listener is able to construct a call-graph or a control flow graph for a given source code and send it to the analyzing tool. One ability of the architecture is that it can conduct the live variable analysis internally. It detects places in the control flow graph where some subset of variables may be killed. The problem was that every variable for which a pointer address was assigned could not been killed, before. This decision had been made because there was no assurance that the variable could never been used through the pointer. So the goal of this work was to design and incorporate a points-to analysis which is able to exclude some references from the set of considered pointers to improve the live variable analysis.

Klíčová slova

Statická analýza, analýza živých proměnných, analýza ukazatelů, points-to analýza, analýza aliasů, ukazatelová aritmetika, Code Listener, gcc zásuvný modul, formální verifikace

Keywords

Static analysis, live variable analysis, pointer analysis, points-to analysis, alias analysis, pointer arithmetic, Code Listener, gcc plugin, formal verification

Citace

Pavel Raiskup: Vylepšení analýzy živých proměnných
pomocí points-to analýzy, diplomová práce, Brno, FIT VUT v Brně, 2012

Vylepšení analýzy živých proměnných pomocí points-to analýzy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana inženýra Kamila Dudky.

.....
Pavel Raiskup
24. května 2012

Poděkování

Děkuji Kamilu Dudkovi za trpělivost, cenné rady, připomínky a za metodické vedení diplomové práce.

Obsah

Kapitola 1

Úvod

Moje diplomová práce tématem zapadá do oboru formální analýzy a verifikace, částečně také do oblasti formálních jazyků. V jejím rámci jsem pracoval na úpravě knihovny Code Listener, což je nástroj, který slouží jako rozhraní pro výstavbu statických analyzátorů. Tato knihovna je vyvíjena v jazyce C++ a je dílem lidí z týmu VeriFIT[1].

Architektura Code Listeneru je navržena tak, že dovede zpracovat zdrojový program v Jazyce C a nabídnout analyzátoru jistou formu jeho mezikódu. Tento mezikód ale není jediným produktem, který může analyzátor využít. Již v rámci daného rozhraní se provádí některé obecné analýzy, o kterých je známo, že mají široké využití. Jednou z těchto analýz je také analýza živých proměnných, jejíž úprava byla cílem mojí práce. Tento algoritmus má za úkol najít pro každou proměnnou místa v programu, ve kterých není potřeba se touto proměnnou zabývat, např. protože hodnota proměnné není v dané oblastí nikdy použita.

Předchozí implementace byla schopna takto analyzovat pouze lokální proměnné, na něž zároveň v rámci celého programu nebyla nikde vzata adresa. Moje rozšíření algoritmu proto spočívalo v tom, že jsem umožnil, aby také nad cíli některých ukazatelů mohla být prováděna tato analýza. K tomu jsem využil tzv. points-to analýzu, kterou se tato práce bude z velké části zabývat. Na základě jejích výsledků dovede rozšířený algoritmus pro analýzu živých proměnných pracovat s cíli některých ukazatelů, jako by to byly lokální proměnné.

Práce je dělena do několika kapitol. Kapitola 2 se věnuje teorii, kterou bylo pro úspěšné zpracování tématu nastudovat. Další kapitola 3 schématicky vysvětluje algoritmy, které jsem v práci implementoval, případně je zvažoval jako alternativy. Architekturu rozhraní Code Listener popisuje kapitola 4. Analýze živých proměnných — jako nejdůležitějšímu prvku Code Listeneru pro diplomovou práci — je věnována zvláštní kapitola 5. Tato kapitola také uvádí motivační zdrojový kód, který vedl ke vzniku této diplomové práce. Postup návrhu rozšíření a jeho implementaci rozebírám v kapitole 6. Dosažené výsledky shrnuje kapitola 7.

[1] Výzkumná skupina automatizované analýzy a verifikace – www.fit.vutbr.cz/research/groups/verifit

Kapitola 2

Teoretické základy

Jazyky, v nichž se při práci využívá ukazatelů (např. jazyk C), jsou stále velmi populární[1]. Přestože implementace datových struktur pomocí ukazatelů či referencí není nijak jednoduchá činnost, jedná se o velmi oblíbenou a výkonově efektivní techniku, která má ale jednu velkou nevýhodu – náročnost práce s datovými strukturami vede ke zvýšenému zanášení chyb do výsledných kódů.

Průmyslová praxe si tedy žádá existenci podpůrných nástrojů, které dovedou co možná nejspolehlivěji odhalit vznikající implementační problémy (a to samozřejmě nejen pří práci s ukazateli) vzniklé vinou programátora. Tyto nástroje využívají různých typů programových analýz. V mojí práci bude diskutována tzv. „points-to“ analýza a analýza živých proměnných, které spadají do kategorie analýzy statické. Cílem práce je zefektivnění analýzy živých proměnných s využitím points-to analýzy.

2.1 Statická a dynamická analýza programu

Rozdílem v definicích těchto analýz je v tom, že pro provedení analýzy statické není nutné spouštět program – jeho rozbor je proveden čistě na základě zdrojového kódu. Dynamická analýza program naopak spouští a kontroluje chyby za běhu. Neplatí to ale úplně doslova, technika *symbolické exekuce* je známá ze statické analýzy a mírně stírá jejich rozdíl.

Další podstatný rozdíl dynamické proti statické analýze je ten, že kontroluje obvykle jen jednu cestu grafem toku řízení (viz. 2.3). Je pravda, že se to může zdát někdy i jako výhoda, protože dynamická analýza zkoumá pouze běh programu, který je reálně proveditelný. Tomuto se občas statické analýze nemusí podařit vyhnout – příkladem pro představu může být analýza chování funkce `scanf("%s", s)`, která v tomto tvaru načítá ze standardního vstupu řetězec do proměnné s. Počet variací obsahu řetězce s je po provedení funkce obrovský, i když se jedná o tak jednoduchou operaci. Poté i kontrola vzniklého stavového prostoru bude způsobovat problém. Při tom všem může být ale jasné, že vzhledem k prostředí (tj. proměnné prostředí, argumenty programu, apod.), ve kterém program běží, je známých několik málo možností, kterých může s nabývat.

Statická analýza či verifikace někdy musí tedy prozkoumat všechny teoreticky možné stavy proměnné s (a určit jejich korektnost) a mít jistotu, že nemůže nastat jiná – nepokrytá situace. Statická analýza tedy i ve funkčním programu může najít potenciální chyby, které teoreticky mohou nastat. Oproti tomu dynamická analýza bude nejspíše sloužit k hledání existujících reprodukovatelných problémů v programu, kdy dovedeme běžícímu programu

[1] Zajímavá sada statistik oblíbenosti programovacích jazyků je k dispozici na http://langpop.com/.

vynutit chybující cestu grafem toku řízení a zachytit vznikající problémy během právě této jediné cesty. Pro statickou analýzu hraje svým způsobem také fakt, že dává programu, ve kterém nenalezne problémy, jakýsi status bezchybnosti. Aby dynamická analýza tento status mohla zařídit, musela by nasimulovat všechny možné situace, do kterých se program může reálně dostat. Prozkoumání tak velkého stavového prostoru by tedy bylo pravděpodobně nerealizovatelné. Výhoda statické proti dynamické analýze bude, že zmiňovaná exploze stavů půjde často řešit zaváděním abstrakcí, které spojí množinu svými vlastnostmi podobných stavů v jeden.

Z toho všeho ale také plyne jistá jednoduchost dynamické analýzy v tom smyslu, že to hlavní a důležité je „jen“ interpretace programu[2]. Statická analýza musí jít při realizaci dále a tedy i implementace bude většinou složitější.

2.2 Formální verifikace a analýza

K definici těchto pojmů si vypůjčím termíny z materiálů [18]. Pokud je řeč o formální verifikaci, má se na mysli proces odvozování informace obvykle binárního charakteru. Otázka je například typu, zda daný systém (program nebo jeho model), splňuje nějakou předem specifikovanou vlastnost. Odpověď na ni je poté „ano“ nebo „ne“ (případně proč ne, má-li verifikátor navést jeho uživatele k vyřešení problému). Například dotaz, zda-li program může někdy uváznout, je tohoto charakteru. Verifikační otázka navíc bude také často směřována na korektnost systému.

Naopak, analýza si klade dotazy obecnějšího charakteru. Na tyto otázky obvykle nejde odpovědět binární odpovědí „ano“ nebo „ne“, popřípadě pokud jde, ještě stále to nevypovídá nic o správnosti systému. Navíc, výsledek analýzy může být využitý pro odlišné záměry než zrovna zjistit, zda-li program pracuje korektně – může být použit například k optimalizaci, syntézu či generování kódu.

Příklady nástrojů pro analýzu. Jako některé známé statické analyzátory uvedu například Sparse[3] – sloužící pro rozbor zdrojových kódu jádra Linuxu (tedy pro jazyk C), Splint[4] – nástroj pro hledání chybových vzorů v tomtéž jazyce, Coverity[5] – komerční nástroj pro statickou analýzu jazyků C/C++ a Java, či CodeSonar[6]. Jako příklady některých známých nástrojů pro dynamickou analýzu programu můžu zmínit nástroj Valgrind[7], který je otevřeným nástrojem pro UNIX systémy. Je určen pro hledání paměťových úniků, monitorování vyrovnávacích pamětí či hledání souběhů v programu. Z komerčních nástrojů pro systémy Windows bych jako zástupce dynamické analýzy zmínil projekt Insure++[8]. Za zmínku stojí poměrně ucelené seznamy nástrojů pro formální analýzu programů [19, 17].

[2] mimochodem, ladící nástroje jako je *gdb* také plní jakousi formu polo-automatického až manuálního dynamického analyzátoru

[3] https://sparse.wiki.kernel.org/ – nástroj pro statickou analýzu jádra linuxu

[4] http://www.splint.org/ – statická analýza jazyka C

[5] http://www.coverity.com/ – komerční statická analýza pro jazyk C/C++/Java

[6] http://www.grammatech.com/products/codesonar/ – komreční, statická analýza C/C++

[7] http://valgrind.org/ – dynamická analýza běhu programu

[8] http://www.parasoft.com/jsp/products/insure.jsp – komerční, dynamická analýza pro Windows/Linux

2.3 Graf toku řízení

Aby bylo možné provádět nad zdrojovým kódem statickou analýzu, je potřeba si jej nějak reprezentovat uvnitř analyzátoru. K tomuto lze použít tzv. graf toku řízení (anglicky Control flow graph [20]), pro který budu dále používat pouze zkratku CFG. Formát grafu toku řízení může být implementován různě, proto se v následujících definicích budu držet konvencí zavedených v rozhraní Code Listener. Jak tedy CFG uvnitř tohoto rozhraní vypadá?

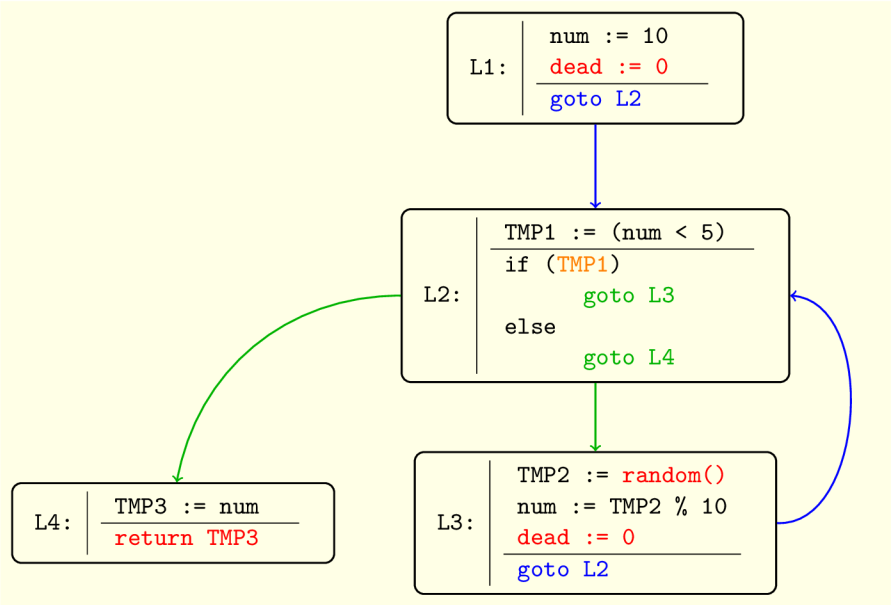
Program v jazyce C lze z logického pohledu rozložit na množinu funkcí. Každá z těchto funkcí poté bude obsahovat sekvence programových instrukcí. Chování vnořených sekvencí bohužel nemá přímo ideální charakter k analýze. Jednotlivé instrukce mohou být voláním jiných funkcí, instrukcemi skoku, podmíněného skoku a podobně, a analýza by byla zbytečně náročná, měla-li by se brát v úvahu každá instrukce jako zcela nezávislý prvek v rámci analyzované funkce. Proto se pro účely zjednodušení zavádí pojem základního bloku (anglicky basic block), který znamená jistou logickou pod-sekvenci instrukcí dané funkce. Tyto základní bloky poté tvoří množinu uzlů CFG odpovídající funkce.

Základní blok. Základní bloky jsou dány jako maximálně dlouhé sekvence neterminálních instrukcí – ukončené instrukcí terminální. Terminálními instrukcemi rozumíme instrukce skoku (ty nám definují hrany grafu toku řízení), instrukce ukončení podprogramu (instrukce `return`) či instrukce ukončení celého programu (například `abort()`). Každou sekvenci instrukcí dané funkce lze rozdělit do základních bloků podle uvedených konvencí až na izomorfizmus jednoznačně.

Základní blok je vždy uvozen jedinečným identifikátorem, takzvaným návěštím bloku. Jedná se o vstupní místo pro vykonávání základního bloku programem. Prakticky to poté znamená, že základní blok bude programem vykonán jako atomická posloupnost operací, do které se řízení programu nedostane jinak, než přes dané návěští, a ven ze základního bloku se nedostane jinak, než pomocí terminální instrukce. Toto je také, kromě zjemnění granularity abstrakce, pozitivní pro další analýzy (třeba právě analýza živosti proměnných, viz. 3.1). Příklad grafu toku řízení funkce `main()` následujícího programu je na obrázku 2.1.

```
1  int main()
2  {
3      int num = 10;
4      int dead = 0;
5
6      while (num > 5) {
7          num = random() % 10;
8          dead = 0;
9      }
10
11     return num;
12 }
```

Uzly grafu na tomto obrázku jsou základními bloky funkce a hrany jsou, jak už bylo řečeno, definovány terminálními instrukcemi. V tomto případě – modré hrany značí instrukci nepodmíněného skoku, zelené naopak skok podmíněný. Terminální instrukce bloku `L4` je příkaz `return`, který znamená nejen konec bloku, ale i celého podprogramu. Všimněme si, že volání funkce `random()` v rámci bloku `L2` netvoří hranu CFG – tvoří ale hranu v grafu volání, který bude předmětem kapitoly 2.4. Dalším příkladem grafu toku řízení je později uvedený obrázek 4.2.



Obrázek 2.1: Příklad grafu toku řízení pro jednoduchou funkci v jazyce C

2.4 Graf volání

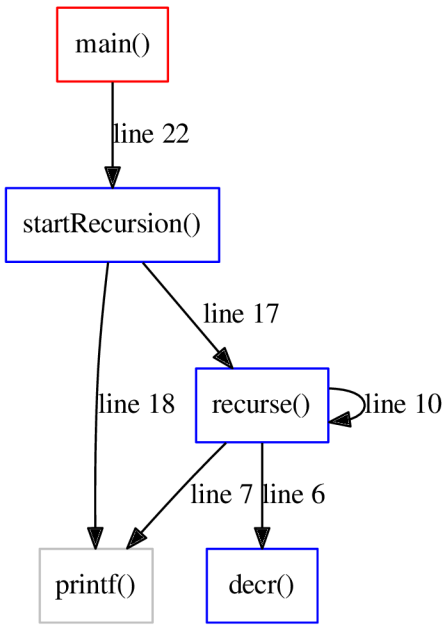
Další důležitá struktura při analýze je graf volání programu (anglicky Call Graph). Opět zde budu popisovat tvar grafu volání tak, jak je tomu v Code Listeneru. Uzly tohoto grafu tvoří množina funkcí (dalo by se říct, že množina CFG grafů daných funkcí). Graf volání tvoří doplňující informaci k množině CFG v tom, že nám tvoří hrany vedoucí od volající k volané funkci. Implementace grafu volání v Code Listeneru také přesněji identifikuje instrukce, která dané hrany tvoří – ke každé hraně grafu odpovídá jedna instrukce grafu volání. Jedná se tedy o multigraf, jelikož jedna funkce může volat druhou z více míst (více instrukcemi volání).

Na obrázku 2.2 uvádím příklad grafické podoby grafu volání. Počáteční uzel (tedy funkce `main`) je značena červeně, lokálně definované funkce jsou vykresleny modře a externí funkce šedě.

2.5 Analýza ukazatelů v programu

Tato sekce uvede základní dělení analýz ukazatelů, aby bylo objasněno, co to vlastně points-to analýza je, že to není jediný způsob rozboru programu z hlediska hodnot ukazatelů, a k čemu vlastně různé typy analýz ukazatelů slouží. Konkrétním algoritmům points-to analýzy bude věnována zvláštní kapitola 3.2. Dále se zde zmíním o tom, jakým způsobem se obvykle hodnotí kvalita points-to algoritmu a co znamená bezpečnost v kontextu points-to.

Analýza ukazatelů (pointer analysis). Jde o analýzu, jež je prováděna v době překladu, a která se pokouší určit, jakých hodnot nabývají ukazatele. Jde o obecnou nadmnožinu ostatních popisovaných analýz a její kompletní řešení představuje nerozhodnutelný problém [13]. Proto existuje mnoho algoritmů a článků, které se věnují řešením jejích aproximací.




```c
int decr(int i) {
    return i - 1;
}

int recurse(int i) {
    i = decr(i);
    printf("%d\n", i);

    if (i > 0)
        return recurse(i);
    else
        return i;
}

int startRecursion()
{
    int i = recurse(10);
    printf("%d\n", i);
}

int main() {
    startRecursion();
}
```

Obrázek 2.2: Příklad grafu volání a odpovídajícího programu.

Aliasová analýza (pointer alias analysis). Ta se snaží nacházet místa, kde dva ukazatele ukazují na stejná data. Pokud jsou dva ukazatele, například x a y, nastaveny na stejnou hodnotu, poté ukazují na stejné místo. Můžeme tedy prohlásit, že `*x` je alias pro `*y` a opačně.

Points-to analýza. Její snahou je pro každý ukazatel programu určit, na jaká data, neboli points-to množiny, *může* (anglicky may pointer analysis) ukazovat. Na základě výsledků points-to analýzy lze do jisté míry vyřešit problém předchozí – aliasovou analýzu. V dalším textu budu používat zkratku PT pro výraz points-to.

Dále bych ještě rád uvedl jiná logická dělení typů analýz ukazatelů a zmínil se o pojmech, které budu dále v textu běžně používat.

Interprocedurální a intraprocedurální analýza ukazatelů. Dělení je do jisté míry samovysvětlující – intraprocedurální analýza nebere v úvahu koexistenci analyzované funkce s ostatními funkcemi programu, interprocedurální ano. V points-to kontextu bude tedy interprocedurální analýza stavět pro každou funkci jí vlastní PT graf (který klidně nemusí být závislý na kontextu volání). Naopak, interprocedurální bude stavět jeden PT graf pro celý program (ten může být klidně stavěn v závislosti na grafu volání).

Kontextově senzitivní analýza (context-sensitive analysis) určuje, zda-li je, či není brán během výpočtu v úvahu kontext volání daných funkcí, tedy jestli propagovat PT informace z volané funkce do volající a naopak. Tato analýza připadá do úvahy jak pro interprocedurální, tak pro intraprocedurální analýzu.

Tokově senzitivní analýza (flow-sensitive analysis) bere do úvahy pořadí příkazů v rámci funkcí analyzovaného programu. Tokově insenzitivní je méně přesná – nezáleží na pořadí příkazů, které ovlivňují konstrukci grafu. Graf tokově senzitivní analýzy bude zřejmě rozsáhlejší.

Analýza typu může či musí. Na několika místech práce uvidím zvýrazněno slovo „moci“, protože analýza, kterou řeším v rámci diplomové práce je právě typu *může*. Charakter této analýzy je, že jakmile je šance — byť jen teoretická — že během celého programu *může* být některý ukazatel aliasem k jinému, musí být tato informace zahrnuta do výsledného grafu (a tedy budou sloučeny cíle uzlů, které těmto ukazatelům odpovídají). Naopak, analýza typu *musí* by promítla tuto informaci do výsledného grafu jedině tehdy, pokud by bylo pro každé místo programu (či podprogramu) zřejmé, že tento alias je platný.

Bezpečnost points-to analýzy je jedna z důležitých vlastností algoritmu. Pokud se budu odkazovat na bezpečnost, budu tím mít na mysli obvykle bezpečnost z hlediska analýzy živosti proměnných – pokud má být proměnná zabita, musím si být stoprocentně jistý, že *nemůže* existovat žádný ukazatel, který by obsahoval její adresu, a že tedy nemůže být za žádných okolností přes ukazatel použita. Jinak by šlo o *nebezpečné* odstranění. Pokud celý algoritmus nedělá žádnou nebezpečnou operaci, lze označit za bezpečný.

Pesimistická analýza. V kontextu programových analýz, konkrétně i u analýzy živosti, může být řeč o pesimistické analýze. V pesimistické analýze se vždy předpokládá to nejhorší tak, aby se zachovala bezpečnost. Pokud by jen teoreticky existovala šance, že by zabití proměnné způsobilo nekorektní operaci, zabití se neprovede.

Kvalita points-to algoritmu [11]. Běžným hodnocením bývá „přesnost“ a „efektivita“. Při vyjadřování přesnosti algoritmu se bere v úvahu, jak dobré informace nabídne implementacím jiných vyšších algoritmů (např. analýza živosti, analýza dosažitelných definic, propagace konstant, apod.). Zatímco efektivita algoritmu obvykle označuje nepřímou úměrnost se složitostí, neboli cenou výpočtu. Tyto dvě vlastnosti obvykle jdou proti sobě a volí se mezi nimi kompromis. Je potřeba ještě říct, že vlivem nepřesnosti v points-to analýze může dojít ke zpomalení algoritmu, jenž výsledek points-to analýzy využívá. Článek [10] se tomuto hledisku věnuje poměrně podrobně. Právě kvůli tomu, že nekvalitní PT algoritmus může být efektivní z hlediska vlastního výpočtu, ale může způsobit markantní zpomalení ostatních algoritmů, je volba kompromisu ještě složitější.

Analýza celého programu. Někdy nemusí být samozřejmé, že lze (či lze pohodlně) provést analýzu celého programu a dělat úsudky nad jeho vlastnostmi jako nad celkem. Poté se provádí analýza jednotlivých funkcí (nebo až základních bloků) separátně. Pokud to ale možné je, bavíme se o analýze celého programu.

Shrnutí pojmů. Pokud bychom volili algoritmus pro získání kvalitních points-to relací pro analýzu živosti proměnných, v ideálním případě bychom použili kontextově i tokově senzitivní algoritmus, který by byl bezpečný a zároveň efektivní. Dále by byl takový, aby pracoval nad celým programem. Takový algoritmus ale neexistuje, jelikož se jedná o protichůdné požadavky. Je tedy nutné zvážit, jaké hledisko je pro nás v dané situaci nejdůležitější

a najít vhodný kompromis. Problematice volby algoritmu se věnuje článek [9], ve kterém je mj. rozsáhlé empirické srovnání rychlosti běžných typů PT algoritmů.

Kapitola 3

Algoritmy pro analýzu živých proměnných a points-to analýzu

V této kapitole se budu věnovat rozboru algoritmu pro analýzu živých proměnných a algoritmů pro points-to analýzu, protože právě implementace kombinace těchto algoritmů je cílem mojí práce.

Motivace. Pro verifikaci programu je důležité redukovat stavový prostor, jenž je analyzován. Více proměnných samozřejmě znamená větší stavový prostor a to implikuje jednak více práce a jednak větší paměťovou náročnost. Proto je potřeba tuto problematiku nějak řešit. Jak ale poznat, kdy a kde některá z proměnných není potřeba?

3.1 Algoritmus pro analýzu živých proměnných

Je zřejmé, že proměnná, která byla deklarována a nebyla použita (a nebo byla využita pro zápis a pak následně nebyla nikdy čtena), nemusela v programu ani existovat. Byl to zbytečně alokovaný prostor, který mohl být využitý k něčemu užitečnému. Proto, aby k této situaci nedocházelo a přeložený kód nehýřil pamětí, byly objeveny algoritmy pro eliminaci takzvaných mrtvých proměnných během překladu programu. Původně šlo tedy o algoritmus určený spíše k optimalizaci kódu při překladu.

Z hlediska diplomové práce je ale důležité, že z tohoto algoritmu dobře vytěží i analyzátor či verifikátor, jelikož s menším počtem proměnných jí zbude méně práce při zkoumání vlastností těchto proměnných a bude potřebovat taktéž méně paměti. Pokud budu v následujícím textu používat pojem „živost“, budu tím myslet vždy živost proměnné, nikoliv živost ve smyslu „živost systému“.

Definice živé proměnné. Na úvod je potřeba říct, že „živost“ je stavová informace – proměnná může být během programu několikrát usmrcena a oživena. Na množinu živých či mrtvých proměnných se proto budeme ptát pro každé místo v kódu u každé instrukce zvlášť (jedná se tedy o tokově senzitivní analýzu).

Uvažujme pouze jeden základní blok. Předpokládejme dále, že každá instrukce je označena svým jedinečným návěštím. Mluví-li se o pozici v kódu „po provedení instrukce“, myslí se tím doba (či místo v kódu), kdy už je daná instrukce s daným návěštím zpracována ale ještě nebyla započata instrukce následující. Definice živosti proměnné poté zní:

Definice 1. *Proměnná je živá po provedení instrukce, existuje-li cesta z jejího návěští k použití této proměnné v CFG, během které nedojde k redefinici této proměnné.*

Tato definice byla převzata z [12], kapitola 2.1.4.

Analýza živých proměnných poté zjišťuje, které množiny proměnných na konci konkrétních instrukcí splňují tuto definici. Základní algoritmus pro hledání množin živých proměnných $V_L(i)$ v základním bloku $B = (i_1, i_2, i_3, ..., i_n)$[1] je dán následovně:

1. Předpokládejme, že všechny proměnné na konci bloku jsou mrtvé.
   $V_{LIVE} = V_{KILL} = V_{GEN} = \emptyset$ – implicitní stav „po“ poslední instrukci (žádná živá proměnná).

2. Pro každou instrukci $I \in B$ v **reverzním** pořadí:

   (a) $V_L(i) := V_{LIVE}$ – uchování informace o živosti pro danou instrukci.

   (b) $V_{LIVE} := (V_{LIVE} \setminus V_{KILL}) \cup V_{GEN}$, kde
   $V_{KILL} = \{v \mid \text{do } v \text{ je v } I \text{ prováděn zápis}\}$
   $V_{GEN} = \{v \mid v \text{ se je čtena v dané instrukci}\}$

Algoritmus je uveden pouze schématicky (volná reprodukce z [12]). V reálném programu nemáme samozřejmě pouze jeden separátní základní blok. Je proto nutné provést distribuci informace o živosti napříč všemi základními bloky funkce. Také nelze jen předpokládat, že všechny proměnné na konci bloku jsou mrtvé (následující základní bloky v CFG mohou číst některé proměnné a tyto jsou tedy pro aktuální analyzovaný blok — už z definice živosti — samozřejmě živé).

K řešení tohoto problému se využívá algoritmu výpočtu pevného bodu (anglicky fixed point), jak také uvádí citovaná literatura. Tuto metodu popíšu přesněji na konkrétní implementaci CL v kapitole 5.

Ideální by bylo samozřejmě vzít do úvahy nejen základní bloky analyzované funkce, ale počítat živost proměnných interprocedurálně, a to jak pro lokální, tak pro globální funkce, či dokonce pro proměnné, na které může být vzata adresa a uložena v nějakém ukazateli. Tím se ale analýza patřičně komplikuje. Analyzátor nemá pod kontrolou např. externí funkce a neví, jaké vedlejší efekty mohou mít na svědomí (mohou například měnit či číst viditelné proměnné pomocí ukazatele a analýza takové proměnné je proto problematická). Samozřejmě, pokud si nejsme jisti, vždy můžeme prohlásit proměnnou za živou. Na algoritmus to nebude mít vliv z hlediska bezpečnosti, pouze bude výsledek méně efektivní. K efektivnějšímu zabíjení se hodí nějaký jiný podpůrný mechanismus, například právě points-to analýza, o které bude řeč nyní.

3.2 Algoritmy pro points-to analýzu

V této sekci se pokusím shrnout vlastnosti některých points-to algoritmů. Nepůjde o vyčerpávající výčet všech existujících. V praktických implementacích jsou totiž myšlenky PT algoritmů podobné, a nebo jsou kombinací myšlenek zde uvedených algoritmů [9]. Proto není nezbytně nutné se všemi podrobně zabývat. Jeden algoritmus zde ale rozeberu podrobněji — alg. *FICS* (viz. sekce 3.2). Ten jsme se s vedoucím diplomové práce rozhodli implementovat, protože právě kombinace jeho vlastností je vhodná pro účely zadání. Na začátku sekce se ale ještě podívám na některé důležité pojmy.

[1] základní blok je definován jako sekvence po sobě jdoucích instrukcí

Points-to množina je množina paměťových míst (tedy obvykle proměnných), na které může daný ukazatel ukazovat.

Points-to graf. Některá literatura, např. [15], operuje při vysvětlování různých PT algoritmů s více typy PT grafu či PT množin – ke každému algoritmu nadefinuje jiný tvar grafu tak, jak to vyhovuje danému algoritmu. Dovolím si tedy raději mírně zobecnit definici PT grafu tak, aby výsledná struktura grafu byla vhodná pro reprezentaci výstupu všech diskutovaných algoritmů. Bude se to mimo jiné hodit i posléze při praktické implementaci proto, že struktura (či třída) reprezentující PT graf v Code Listeneru bude moci být výstupem všech typů algoritmů (minimálně těch diskutovaných) a tedy i dané programové rozhraní bude více generické.

Nyní tedy předpokládejme program P, definovaný jako dvojici:

$$P = (V, F)$$

Prvky této dvojice budou V – množina proměnných existujících v programu a F – množina funkcí, ze kterých se program skládá. Označme si tedy PT graf dané funkce $f \in F$ jako dvojici $G^f_{PT} = (N, E)$, kde N je množina uzlů grafu a E je množina hran. Každý uzel $n \in N$ je definován jako:

$$n \subseteq V \cup H$$

Množina H je množina reprezentující místa v paměti (obvykle na haldě), na něž se vrací ukazatel – například pomocí systémové funkce `malloc()`. Každé volání funkce `malloc()` bude znamenat unikátní prvek $h \in H$. Dále každá hrana $e \in E$ bude definována jako:

$$e \in N \times N \quad \text{a tedy} \quad E \subseteq N \times N$$

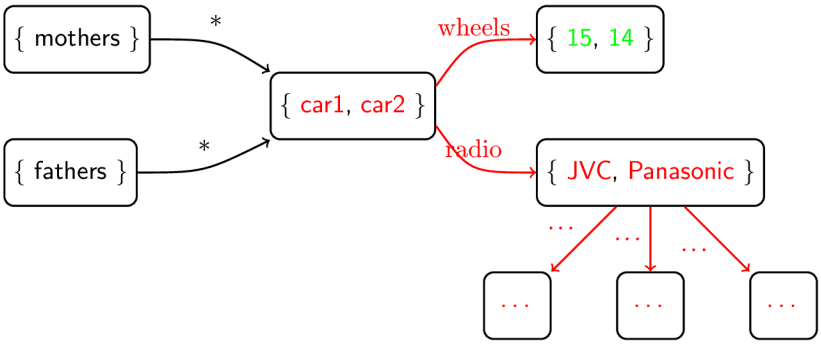
Důležité je říct, že nejsou na strukturu grafu z obecného hlediska kladeny žádné další nároky. Omezení si může klást konkrétní algoritmus (může omezit počty hran, vícenásobnou existenci některých proměnných, atd.), ale graf takto definovaný by měl vyhovět všem následujícím algoritmům. Důsledkem je, že v tomto obecném grafu se může daná proměnná $v \in V$ vyskytovat ve více uzlech, stejně jako může být množina proměnných příslušejících jednomu uzlu PT grafu i prázdná. Nutno podotknout, že originální článek [10] operuje s velice podobným grafem, který ještě do n-tice G^f_{PT} přidává další prvek, řekněme L, který je zobrazením

$$L : E \to StructFields(V) \cup *$$

kde symbol ***** značí jednoduchou dereferenci a funkce $StructFields$ vrací všechny možné názvy polí strukturovaných datových typů (`struct`). Tedy, že z každého uzlu může vést více hran různě pojmenovaných podle možných položek proměnných typu struktura, jež jsou obsaženy v daném uzlu. V praxi to má poskytnout zpřesněné informace o tom, že proměnná nějakého typu může ukazovat na nějakou další proměnnou (nebo dokonce na sebe sama) přes svoji konkrétní položku (typu ukazatel), nestačí říct, že struktura celá na tuto proměnnou ukazuje. Tohle vede na jak na složitější implementaci algoritmů, ale také na to, že formát grafu ne zcela přesně charakterizuje možnosti operací nad strukturami v jazyce C. Nicméně příklad takového grafu sestrojeného algoritmem FICS je na obrázku 3.1. O tom, proč jsme od této myšlenky až později v kapitole 6.4.1. Nyní ale už k popisu jednotlivých PT algoritmů.

„Proměnná je odkazována“ (*address-taken*)

Jde o tokově insenzitivní algoritmus, často použitý v produkčních překladačích [9],



Obrázek 3.1: Příklad reprezentace dat grafu G_{PT} ve FICS algoritmu

který si pouze poznamenává u každé proměnné, je-li na ni někde během programu uložena adresa. Pokud ano, znamená to, že je vyloučena z některých analýz – například ji nejde nikdy „zabít“ v algoritmu analýzy živých proměnných. Algoritmus je zde zmíněn proto, že je to obvyklá technika a dosavadní implementace analýzy živých proměnných v Code Listeneru této heuristiky využívá.

Steensgaardův algoritmus [16]

Je tokově insenzitivní, kontextově insenzitivní a interprocedurální algoritmus. Jeho výsledkem je konstrukce PT-Grafu, v němž každý uzel představuje ekvivalenční třídu proměnných. Tato ekvivalence je stavěna vzhledem k tomu, jaký ukazatel ukazuje na dané proměnné – nebo jinak, pokud jsou dvě proměnné odkazovány během programu pomocí toho samého ukazatele, náleží do stejné ekvivalenční třídy a musí být vloženy do stejného uzlu grafu.

Konstrukce grafu probíhá tak, že pro každou instrukci přiřazení v programu, která může způsobit vznik nového aliasu či jinak ovlivnit PT graf (přiřazuje do proměnné typu ukazatel či nějakého kompozitního typu), se vynutí, aby cíle ukazatele z levé strany přiřazení patřily do stejné ekvivalenční třídy, jako cíl proměnné pravé strany. Náležitost do stejné ekvivalenční třídy tedy znamená, že tyto cíle budou patřit do stejného uzlu.

Steensgaardův algoritmus pracuje s relativně nízkou asymptotickou složitostí – stačí mu jeden průchod grafem toku řízení k poskytnutí bezpečného points-to grafu. PT-množina daného ukazatele bude dána množinou paměťových míst v uzlu, který je (jediným) následovníkem uzlu, ve kterém je tento ukazatel.

Vlastností grafu tohoto typu je mimo jiné také to, že jedna proměnná se v rámci jednoho grafu může objevit v maximálně jednom z jeho uzlů. Existence ve více uzlech není povolena. Rozložení již sestavené množiny proměnných v uzlu již není nikdy možné a proto je zaručena bezpečnost.

Složitost algoritmu je téměř lineární, jak už prezentuje i název článku[16]. Algoritmu sice skutečně stačí jeden průchod grafem toku řízení, nicméně operace pro jednotlivé instrukce CFG nejsou vždy konstantní složitosti (i pokud budeme považovat sjednocení množin proměnných dvou uzlů jako konstantní operaci). Může dojít k situaci, kdy je potřeba provést sjednocení množin dvou uzlů, kde oba mají další následující disjunktní uzly. Vzhledem k tomu, že uzel grafu musí mít vždy pouze jednu výstupní hranu, je nutné provést běh rekurzivně sjednocení nad sekvencemi následníků

těchto uzlů. Tento postup budu podrobněji popisovat také u implementace algoritmu FICS 6.4.1.

Andersenův algoritmus [1]
Andersenův algoritmus využívá uzel grafu pro reprezentaci jednoho paměťového místa (nikoliv jako ekvivalenční třídu). Proto každý uzel obsahuje pouze jednu proměnnou (přesněji, dle předchozí definice PT grafu, množinu, která obsahuje právě jednu proměnnou či paměťové místo). Points-to množina (tedy množina paměťových míst, na která daná proměnná ukazatele může ukazovat) bude dána množinou uzlů, jež jsou cílem hran vedoucích z ukazateli odpovídajícího uzlu.

Výstavba grafu v Andersenově algoritmu vždy musí zajistit, aby množina cílů levé strany přiřazení pracující s ukazateli byla nadmnožinou cílů pravé strany tohoto přiřazení. Při průchodu grafem CFG, tedy při detekci takového přiřazení, se chybějící hrany do levé strany přiřazení doplní. Z toho je zřejmé, že pokud by se prošel CFG programu pouze jednou, nebylo by zaručeno, že je graf bezpečný. Proto je součástí algoritmu výpočet pevného bodu, jež iterativně přidává chybějící hrany do doby, než je bezpečnost grafu zaručena.

Andersenův algoritmus je také tokově insenzitivní. Graf vypočtený tímto algoritmem je obvykle rozsáhlejší, než Steensgaardův, ale informace jím získané budou přesnější. Výpočet pevného bodu však vede oproti Steensgaardovu algoritmu k časové složitosti až $O(N^3)$ vzhledem k velikosti programu [10].

Choi a kol. algoritmus [8, 2]
zde uvádím jako zástupce tokově-senzitivních algoritmů. Tento algoritmus vytváří informace o PT-relacích pro každý bod grafu toku řízení. Jeho vysoká přesnost je ale vyvážena jak složitostí výpočtu, tak náročností implementace. Vzhledem k tomu, že zadání diplomové práce si nežádá natolik přesné informace o points-to relacích a vyžaduje se spíše soustředit na efektivitu a rychlost výpočtu, algoritmem jsem se zabýval pouze přehledově. Dále, publikace [9] zvažovala ve svých srovnáních i tento algoritmus s výsledkem, že přináší pro jednotlivé typy analýz programu (včetně analýzy *Def/Use*, což je základ pro analýzu živosti) minimální, či žádný přínos.

FICS algoritmus

Nyní přichází na řadu popis algoritmu, který byl zvolen k implementaci do Code Listeneru. Název je dán dle zkratky Flow-Insensitive/Context-Sensitive, česky tokově insenzitivní a kontextově senzitivní, a zní tedy FICS.

Před tím, než se pustím do podrobnějšího popisu algoritmu, uvedu, že asymptotická složitost tohoto algoritmu je téměř lineární [10], čímž se efektivitou přibližuje k algoritmu Steensgaardovu. Přesto se přesností, s níž lze z výsledného grafu vyčíst PT množiny, téměř vyrovná Andersenovu algoritmu [10]. Autor tohoto článku také tvrdí, že může jít o nejlepší PT volbu, pokud má analýza sloužit pro analýzu celého programu. Tato tvrzení jsou autorem podložena empirickými studiemi na poměrně rozsáhlých zdrojových kódech.

Podobnost FICS se Steensgaardovým algoritmem je velká – dalo by se říct, že FICS je jeho vylepšením. Rozdílem v těchto algoritmech je, že oproti Steensgaardovu, FICS přináší kontextovou senzitivitu a tedy, že každá z funkcí má svůj vlastní (přesnější) PT graf. Jak jsem popsal dříve, Steensgaardův algoritmus produkuje pouze jeden společný graf pro celý program.

V rámci diplomové práce jsem sice — po diskuzi s vedoucím práce — od implementace následující myšlenky upustil, ale publikace algoritmu FICS počítá mj. trochu přesněji s položkami datových struktur (vzhledem k mojí funkční implementaci FICS algoritmu, kterou podorobně popíšu v 6.4.1). Pro představu výstupu takového řešení byl také přiložen obrázek 3.1, který byl sestaven pro následující kód.

```
1  struct TRadio {
2      const char      *manufacturer;
3  };
4  struct TCar {
5      struct TWheels  *wheels;
6      struct TRadio   *radio;
7  };
8  int main(int argc, char **argv) {
9      struct TRadio JVC;
10     struct TRadio Panasonic;
11     int w15 = 15;
12     int w14 = 14;
13     const char *season = argv[1];
14     struct TCar car1 =
15         { &w15, &JVC };
16     struct TCar car2 =
17         { &w14, &Panasonic };
18
19     struct TCar *mothers;
20     struct TCar *fathers;
21     if (STREQ(season, "winter")) {
22         mothers = &car1;
23         fathers = &car2;
24     }
25     else {
26         mothers = &car2;
27         fathers = &car1;
28     }
29     return 0;
30 }
```

Jména proměnných z uvedeného obrázku psány černou barvou značí ukazatele, červené názvy značí struktury a zelené běžné – nestrukturované proměnné. Černé hrany označené symbolem * vyjadřují, na kterou(é) část(i) paměti *může* ukazatel ukazovat. Z každého uzlu příslušejícího ukazateli vede maximálně jedna *-hrana. Červené hrany vyjadřují, na kterou část paměti můžou odkazovat zanořené ukazatele uvnitř dané struktury. Každá taková hrana musí mít pro daný uzel jedinečné označení, které odpovídá názvu položky dané struktury.

Pseudokód FICS algoritmu je dostupný v [10]. Nyní se jej pokusím popsat v následující části textu. Jak lze z referenčního pseudokódu vidět, algoritmus FICS pracuje ve třech fázích:

Fáze 1 – výpočet PT informací pro jednotlivé funkce.
FICS algoritmus prvně sestavuje „lokální“ PT graf pro každou z funkcí programu. To se děje zpracováním každé instrukce přiřazení do ukazatele, které existují v rámci dané funkce — tedy stejný postup jako u Steensgaardova alg. — jen se zůstává v kontextu dané funkce, nikoliv celého programu. Instrukce volání funkcí se prozatím přeskakují. Zpracování jednotlivých instrukcí přiřazení způsobí to, že se graf dané funkce přetvaruje tak, aby platilo, že proměnná levé strany přiřazení ukazuje na tu samou sadu uzlů, jako proměnná pravé strany přiřazení.

Nyní by za určitých okolností šlo prohlásit PT grafy funkcí za hotové a bezpečně sestavené. Tyto okolnosti by byly, že by (1) všechny instrukce daných funkcí neoperovaly s globálními proměnnými a zároveň, (2) veškeré instrukce volání programu by probíhaly bez parametrů. Tím by se zaručilo to, že se nebudou PT relace či aliasy distribuovat napříč grafem volání. Protože se jedná ale pouze o teoretickou myšlenku, následují v popisu algoritmu další dvě fáze, které řeší všechny možnosti, které mohou v reálu nastat.

Fáze 2 – propagace informace směrem k volajícím.
Existují celkem tři možnosti, ve kterých může přiřazení, týkající se ukazatelů, ovlivnit PT-graf funkce F a přitom se nacházet mimo F:

- Ve funkci F je použit globální ukazatel, jehož hodnota je nastavována ještě uvnitř jiných funkcí. Kvůli tomuto případu FICS staví ještě, kromě grafů pro jednotlivé funkce, také jeden graf globální, který tímto vzniklé aliasy zaznamenává a propaguje je posléze (ve fázi 3) do všech ostatních funkcí.
- Pokud funkce F volá funkci G, ve které tvoří alias některé její formální paretry navzájem[2], některé globální proměnné navzájem, a nebo jejich libovolně vzdálení následovníci. Takový alias z volané funkce se musí určitě promítnout do volající. Uvedu příklad.

```
void callee(void *a1, void *a2) {
    /* tady by byl samozrejme mrtvy kod, predpokladejme, ze je
     * priklad vytrzen z kontextu a s promennymi a1 pripadne a2
     * provadime jeste neco uzitecneho.  Podstatne je, ze alias
     * nasledovniku 'a1' a 'a2' se pote musi promitnout do
     * volaneho.
     */
    a1 = a2;
}
void caller() {
    void *p1, *p2;
    callee(p1, p2);
}
```

  Volání funkce `callee()` zde musí způsobit, že následníci proměnných `p1` a `p2` z volající funkce tvoří alias – tedy `*p1` bude aliasem pro `*p2`. Dalo by se říct, že následníci byli voláním funkce G „svázáni“.

- A poslední možnost, která se musí zohlednit v PT grafu je, pokud volající funkce předá odkaz na paměťová místa parametrem (nebo přes globální proměnnou) při volání jiné funkce:

```
void callee(void *arg) {
    /** zde muze byt cokoliv **/
}
void caller() {
    int data;
    void *ptr = &data;
    callee(ptr);
}
```

  Pokud volající funkce (třeba právě `caller()`) má ukazatel (`ptr`), který ukazuje na nějakou (byť i lokální) proměnnou (`data`), a předá tento ukazatel (nebo i ukazatel na tento ukazatel, atd.) ve formálním parametru volanému, je zřejmé, že volaný má přístup k téže cílové proměnné. Toto musí způsobit, že daná proměnná bude sídlit i v odpovídajícím uzlu grafu volané funkce. Tedy výsledkem musí být, že `*arg` bude aliasem na proměnnou `data` na straně volaného.

Fáze 2 tedy propaguje informace z volaných funkcí do funkcí volajících a zároveň buduje PT graf pro globální proměnné (podobně jako pro jednotlivé funkce bude vznikat informace o PT relacích mezi globálními proměnnými), jenž můžeme nazvat například G_{glob}. Tato fáze tedy „tlačí“ informace v grafu volání zdola nahoru (anglicky bottom-up). Protože je brán v úvahu graf volání, což je graf, jenž může obsahovat klidně smyčky, je potřeba iterativně vypočítat pevný bod daných PT relací napříč

[2]Návratová hodnota se chápe v tomto případě jako formální parametr.

funkcemi. Po této fázi již je graf G_{glob} hotový (nemůže nastat situace, že by se ve 3. fázi ještě nějak rozšířil).

Fáze 3 – propagace informace směrem k volaným.
Ve fázi 3 nakonec dojde (1) k propagaci proměnných, které sídlí v uzlech grafu G_{glob} do všech grafů procedur, posléze podobně (2) k propagaci proměnných z volajících funkcí do volaných. Tato propagace začíná (1) počínaje uzly grafu, které odpovídají globálním proměnným a (2) počínaje uzly, které odpovídají proměnným navázaným na parametry na straně volajícího. Třetí fáze tedy prochází graf volání shora dolů (angl. top-down order) a také probíhá iterativně (kvůli možné rekurzi) do té doby, dokud nejsou informace stabilizovány.

Vzhledem k tomu, že PT informace jsou nejprve propagovány zdola-nahoru a až posléze shora-dolů, nemůže dojít k propagaci, která by vyústila ve vytvoření nerelevantních aliasů [10] v důsledku neplatných sekvencí volání-návrat (anglicky call-return). Díky této vlastnosti lze prohlásit FICS algoritmus za kontextově senzitivní. Dle uvedeného článku je FICS možno také rozšířit o podporu volání přes funkční ukazatel. Proto je potřeba brát identifikátor funkce jako obecnou proměnnou ukazatele, která ale ukazuje na kód místo na data. Během běhu programu poté tento ukazatel *může* ukazovat na množinu funkcí. Přestože algoritmus implementovaný v CL nyní tuto funkčnost nepodporuje, určitě je možno ji v budoucnu doprogramovat. V tuto chvíli existuje prozatím detekce použití ukazatele na funkci. Jakmile se na jeho použití narazí během analýzy programu, FICS algoritmus zkrátka skončí a označí graf jako neúplný. Další algoritmy jej tedy nebudou moci použít. Podobný mechanizmus je použit ve více nepodporovaných situacích i mimo doménu funkčních ukazatelů.

Kapitola 4

Architektura Code Listener

V tomto projektu jsem se zabýval úpravou nástroje Code Listener (dále jen zkratkou CL). Jedná se o nástroj, který poskytuje infrastrukturu, jež slouží jako základ pro jednoduchou implementaci statických analyzátorů. Kód Code Listeneru je distribuován v rámci projektu Predator, o kterém řeknu pár slov později. Zaměřím se zpočátku na vlastnosti CL, jenž pracuje (alespoň prozatím) jako rozhraní nad překladačem gcc. Rozhraní v tom smyslu, že transformuje interní mezikód tohoto překladače do svého zjednodušeného formátu, nad kterým mj. provádí různé analýzy, jejichž výsledek má poté programátor statického analyzátoru či verifikátoru k dispozici.

Lexikální, syntaktická a sémantická analýza je vyřešena. Protože obvykle „mozek“ statického analyzátoru pracuje nad nějakým programovým mezikódem (například tříadresný kód), je potřeba nějakým způsobem tento kód pro analyzátor opatřit (či si jej musí vytvořit sám). Zmiňované analýzy zdrojového kódu jsou, vzhledem k formě implementace rozhraní, vyřešeny automaticky pomocí gcc. CL si dále vnitřní mezikód sice transformuje ještě trochu podle svých potřeb na svůj formát, ale i tak to jde mimo programátora analyzátoru a nemusí něco podobného řešit sám. To je první výhoda CL, o které se zmiňuji.

Rozhraní pro rozbor zdrojového kódu. Sázkou na budoucnost je, že CL má ve svém návrhu komponentu Code Parsing Interface, viz. schéma na obrázku 4.1. Tato část CL slouží jako rozhraní k navázání na již existující analyzátory zdrojových kódů, které poskytují nějaký mezikód (viz. předchozí odstavec). Reprezentace mezikódem bývá často implementována právě v rámci překladačů jazyka. Adaptér pro využití gcc k rozboru zdrojového kódu je již implementován a je součástí CL, další mohou být implementovány v případě potřeby.

Nasazení do praxe. Výhoda vystavění mezikódu někým jiným (rozumějme překladačem) není pouze v jednoduchosti. Předností taky může být snazší proces nasazení analýzy do praxe. Při delegaci rozboru kódu na překladač je zaručeno, že *právě když* překlad zdrojového kódu proběhne v pořádku, má analyzátor k dispozici data, nad kterými bude moci pracovat. Další výhody související s navázáním na překladač jsou:

- Dochází k úspoře času i energie, protože je zbytečné provádět rozbor kódu v daném jazyce dvakrát,
- programátor si nemusí pamatovat dva dialekty analyzovaného jazyka, jeden pro statický analyzátor, druhý pro překladač a

- schopnost analyzovat více jazyků. Moderní překladače bývají modulární tak, že jsou schopny přes svoje rozhraní navázat moduly věnující se konkrétnímu jazyku a provést pomocí nich základní analýzu zdrojového kódu. Výsledek je poté předán jádru překladače, který nad ním (mj.) provádí další optimalizace a vytváří cílový kód. Pokud je CL schopen (tak jako je tomu v případě `gcc`) připojit se k překladači takovým způsobem, že získá ten typ mezikódu, jež je pro všechny podporované jazyky společný, autor analyzátoru je schopen tento analyzátor psát obecně pro celou tuto množinu jazyků bez přidaného úsilí.

Hlášení chyb zpět programátorovi. CL nabízí také analyzátoru jednotné rozhraní pro hlášení chyb. Výsledkem poté jsou stejně vypadající chybové hlášky, na něž je uživatel překladače zvyklý. Samozřejmostí je, že je umožněno lokalizovat pozici chyby v rámci zdrojového kódu. Ku příkladu proměnné v rámci databáze proměnných `CodeStorage::VarDb` (viz. 4.2) mají v sobě uloženou informace o pozici definice, či instrukce CFG mají odpovídající pozici kódu také jako jednu položku deklarované třídy. V případě chyb týkajících se těchto prvků je lze přímo ukázat v kódu.

Nevýhody přístupu využívání překladačů. Hlavní nevýhoda je zřejmá – rozhraní není samostatně stojící nástroj. Je na překladači naprosto závislý a je nutné jej tedy pro distribuci obstarat. V rámci CL je tohle vyřešeno tím způsobem, že jeho instalátor automaticky dovede získat zdrojové kódy překladače `gcc`, vhodně je přeložit a správně využít binární výstup překladu. Je s tím spojena také další nevýhoda – zavedení podpory tohoto překladače stojí jednak nějaké úsilí (také proto zatím, přes obecnost návrhu CL, je implementován pouze překladač `gcc`), ale především následná údržba rozhraní pro aktuální verze překladače bude vždy stát nějakou práci.

Shrnutí vlastností. Navržené rozhraní CL je přívětivé – programátor analyzátoru má k ruce předem zpracovaný zdrojový kód ve formě CFG, dále má k dispozici graf volání a po mých úpravách také výsledek points-to analýzy, či vylepšené analýzy živost. To vše pod otevřenou licencí a bez další nutně vynaložené práce. Jako nevýhoda může být brán způsob získání potřebného překladače a závislost na něm. V tuto chvíli se jedná pouze o gcc, které je ale také šířeno pod otevřenou licencí.

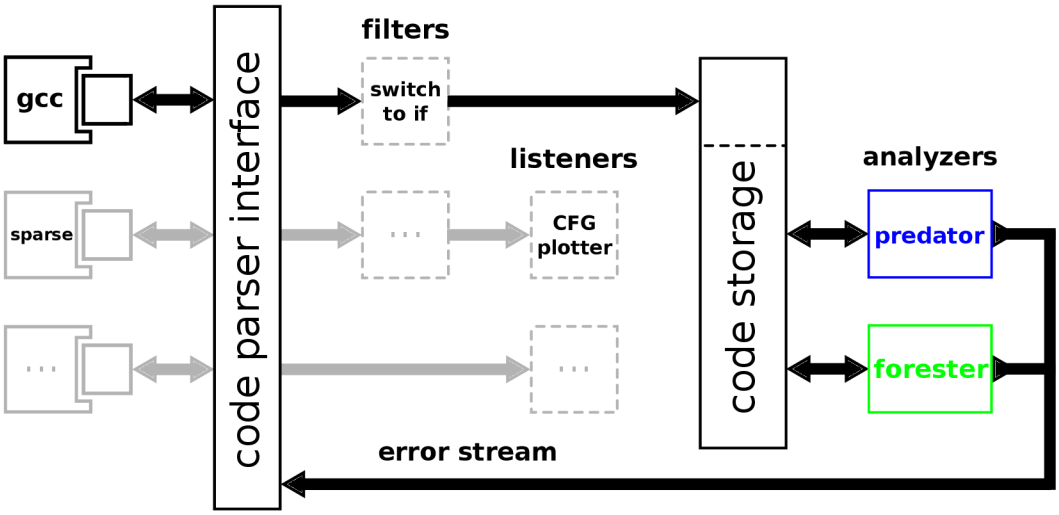
4.1 Spolupráce s překladačem gcc

Pro komunikaci s `gcc` využívá CL dynamických knihoven (anglicky dynamic linking library, zkratkou dll), které jsou zaváděny jako zásuvné moduly (anglicky plug-ins) do překladače. Pro tuto činnnost má příkaz `gcc` parametry:

`-fplugin=PLUGIN.so`
: Parametr slouží pro specifikaci cesty k dynamické knihovně (modulu) daného rozšíření,

`-fplugin-arg-PLUGIN-ARGUMENT=VALUE`
: Slouží pro specifikací argumentů, které budou uvnitř `gcc` předány danému zásuvnému modulu.

Přesně tímto způsobem jsou implementovány nástroje Predator a Forester (viz. sekce 4.3). Po stažení zdrojových kódů těchto nástrojů a jejich překladu vzniknou dynamicky linkované



Obrázek 4.1: Architektura CodeListener, přezvzato z [6]

knihovny (`libsl.so` pro Predator a `libfa.so` pro Forester). Pokud chce programátor využít tyto nástroje pro verifikaci svých zdrojových kódů, stačí překladači `gcc` zadat pomocí výše popsaných parametrů cestu k těmto modulům v souborovém systému.

Překlad testované aplikace poté probíhá stále stejným způsobem, jako by analyzující modul nebyl zaveden — s jediným rozdílem — že je-li ve zdrojovém kódu nějaká chyba, kterou detekoval zásuvný modul, rozšíří se chybový výstup překladače o popis těchto chyb.

CL tedy dodržuje aplikační rozhraní[1] pro zásuvné moduly `gcc`. Uvnitř si zpracovává struktury interního mezikódu překladače, a převádí je na svůj dokumentovaný formát CFG, který je dostupný využívajícím nástrojům. [2]Reprezentace CFG je poté uložena uvnitř komponenty CodeStorage (viz. obrázek 4.1). Tato komponenta je v CL implementována jako jmenný prostor, jehož nejdůležitější částí je třída `CodeStorage::Storage`, která je právě obálkou reprezentací grafu toku řízení (viz. 2.3).

4.2 Graf toku řízení v CL

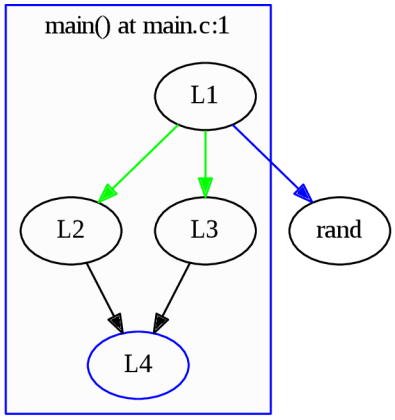
Na úvod řeknu pár slov k tomu, jak probíhá výstavba CFG z vnitřní reprezentace kódu `gcc`. O tuto transformaci se stará funkce `handle_fnc_cfg()`, ve které dojde k postupnému naplnění databáze funkcí, databáze proměnných, apod. Celý mechanizmus je implementován přes zpětné volání z jádra gcc, kde je pomocí základní funkce aplikačního rozhraní `gcc` — funkce `plugin_init()` — zaregistrována adresa funkce `handle_fnc_cfg()` a následně zavolána v době, kdy je interní kód ke transformaci připraven.

Typ kódu, který funkce `handle_fnc_cfg()` využívá pro plnění informací programového skladu `CodeStorage::Storage` se nazývá *GIMPLE*[3]. Výsledkem poměrně komplikované hierarchie zpětných volání funkcí CL je naplnění vnitřních struktur jmenného prostoru `CodeStorage`, a to (1) databáze funkcí `FcnDb`, (2) databáze proměnných `VarDb` a databáze

[1]Specifikace pro aplikační rozhraní `gcc`: http://gcc.gnu.org/wiki/GCC_PluginAPI

[2]CL API dokumentace: http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/api/

[3]Pokud čtenáře zajímá, jaké mezikódy `gcc` využívá, může spustit překlad s parametrem `-fdump-tree-all`, v adresáři odkud byl překlad proveden vznikne několik člověkem čitelných souborů, které jednak slouží k ladění `gcc` samotného, ale jednak můžou pomoci při studiu.




```
int main()
{
    int i, odd, even;

    i = rand(); odd = 0; even = 0;

    if (i % 2 == 0)
        even += i;
    else
        odd += i;

    return even;
}
```

Obrázek 4.2: Příklad CFG grafu pro jednoduchý zdrojový kód

typů (3) `TypeDb`. Tyto prvky tvoří asi nejdůležitější koncepční část CL z hlediska mojí diplomové práce, ne-li vůbec. Proto následuje krátký popis těchto struktur:

`CodeStorage::VarDb`
: obsahuje seznam proměnných programu s jejich příznaky (například typ proměnné, příznak zda se jedná o globální proměnnou, zda-li na ni *může* ukazovat nějaký ukazatel, atd.). Z hlediska CFG je podstatné, že instrukce v jednotlivých funkcích pracují s operandy, které mohou odkazovat na proměnné této databáze. Každá proměnná má v rámci `CodeStorage` přiřazen unikátní celočíselný identifikátor, jež bude také použit v analýze živosti proměnných (výstupem algoritmu je jeden seznam identifikátorů proměnných pro každou instrukci programu – tyto proměnné můžou být považovány za mrtvé po zpracování dané instrukce).

`CodeStorage::FncDb`
: Je databáze objektů, které reprezentují jednotlivé funkce programu. Co je důležité, že každý z objektů odpovídajících **definované** funkci obsahuje vnořenou strukturu `FncDb::cfg`. Tato struktura je funkci odpovídající graf toku řízení. Analýza živých proměnných i PT analýza má tuto strukturu jako hlavní zdroj informací. Dále, každá z funkcí má také unikátní číselný identifikátor. Tato čísla jsou mj. volena tak, aby neexistovala kolize identifikátoru funkce a proměnné.

`CodeStorage::TypeDb`
: Databáze typů. Její položky obsahují informace o typech objektů v programu, například druh (ukazatel, funkce, struktura, celé číslo, ...), lokaci deklarace, jméno typu, velikost odvozeného objektu, atd. Každá proměnná poté obsahuje odkaz na jednu položku této databáze, která udává její typ.

Jako ilustrační příklad uvádím obrázek 4.2 a odpovídající zdrojový kód. Ty zjednodušeně reprezentují to, jak vypadá CFG přímo v Code Listeneru. Obrázek byl vygenerovaný automaticky při překladu uvedeného zdrojového kódu pomocí `gcc` se zavedeným modulem Predator.[4] Celá funkce `main()` uvedeného programu (ohraničena modrou linkou) obsahuje

[4]Pro výpis podobný výstup stačí spustit gcc se zavedeným pluginem Predator a s parametrem `-fplugin-arg-libsl-gen-dot=graph.dot`, případně si lze vždy nechat napovědět pomocí parametru `-fplugin-arg-libsl-help`.

čtyři základní bloky ($L1, L2, L3, L4$). Černé hrany grafu značí nepodmíněné skoky, zelené podmíněné. Modrá hrana znamená obecné volání funkce, v tomto případě volání funkce externí. Pro zajímavost či ujasnění čtenáři přidávám ladící výstup ve formě instrukčního kódu odpovídajícího tomuto zdrojovému kódu v příloze A, aby bylo možné si představit, jaké instrukce uvedeného programu se vlastně ve kterých blocích nachází.

4.3 Nástroje Predator a Forester

V této sekci se jen krátce zmíním o dvou nástrojích, které jsou na architektuře Code Listener vystavěny. Oba dva nástroje jsou dílem lidí z *Fakulty informačních technologií, Vysokého učení technického v Brně*. Prvním z nich je nástroj Predator [6]. Jedná se o nástroj pro verifikaci zdrojových kódů jazyka C, pracujících nad dynamickými datovými strukturami a ke své práci používá formalismu Separační logiky. Nástroj je poskytován zdarma pod licencí GPL[5]. Také samotná architektura Code Listener je distribuována v rámci projektu Predator. Z hledika zadání práce je nástroj Predator důležitý, protože požadavek na efektivnější analýzu živých proměnných v CL vzešel z potřeby zrychlit průběh jeho analýzy.

Nástroj Forester také slouží pro verifikaci programů, které manipulují s komplexními datovými strukturami [7]. Využívá jiného formalismu – reprezentuje stavový prostor programu pomocí regulárních stromových automatů [3]. Stejně jako Predator, funguje na bázi zásuvného modulu do `gcc`. Je distribuován pod toutéž licencí – společně s Predatorem.

Shrnutí. Hlavním produktem projektů jsou dynamické knihovny `libsl.so` a `libfa.so`, které lze pomocí parametru `-fplugin=lib*.so` přímo zavést do `gcc`, což postačuje pro jejich uvedení do provozu. Moje práce se nepřímo týká obou zmiňovaných nástrojů, protože oba dva využívají informace poskytované analýzou živosti proměnných, nicméně, práce nepožaduje hlubší pochopení formalismů či implementace těchto nástrojů, proto se tady také nebudu těmito nástroji více zabývat. Moje praktické úpravy se týkají čistě Code Listeneru, což vede pouze na vedlejší efekt zefektivní práce těchto analyzátorů.

[5] Licence viz. stránky projektu: https://github.com/kdudka/predator

Kapitola 5

Implementace analýzy živých proměnných v Code Listeneru

Současná implementace analýzy živých proměnných je inspirována algoritmem z [12], o které jsem také mluvil v sekci 3.1. Stejně jako v této knize i algoritmus v CL počítá živost proměnných nejprve pro základní bloky a posléze provádí výpočet pevného bodu pro distribuci informace o živosti napříč základními bloky. Související zdrojový kód lze najít v `cl/killer.cc`, vstupním bodem analýzy je funkce `killLocalVariables()`. Tato funkce přistupuje především k interní reprezentaci grafu toku řízení a k databázi proměnných (`VarDb`).

Úkolem implementovaného algoritmu však — na rozdíl od referenčního — není hledat přímo mrtvé proměnné, ale nalézt pro každou instrukci CFG množinu proměnných, které (1) jsou před provedením instrukce živé, a zároveň (2) po provedení této instrukce se stávají mrtvými.

Výsledek algoritmu. Algoritmus ve skutečnosti poskytuje ještě trochu přesnější informace, viz. dále. Rozhraním či výstupem algoritmu jsou dva prvky struktury `Insn`, která reprezentuje každou instrukci CFG. Jsou jimi `varsToKill` – množina identifikátorů proměnných, jež mají být po provedení instrukce vždy zabity, ale navíc zmiňovaná přesnější informace – vektor `killPerTarget`, který je (a) definován pouze pro terminální instrukce základního bloku a (b) obsahuje vektor množin identifikátorů proměnných (jedna množina pro každou odchozí hranu terminální instrunkce), které mají být zabity právě, když se řízení programu rozhodne jít z aktuálního bloku přes terminální instrukci danou cílovou hranou (terminální instrukce `switch` může mít teoreticky neomezený počet cílů).

Jakmile jsou zmiňované sady identifikátorů proměnných vypočteny, analýza proměnných končí a nic víc nedělá. Využití těchto výsledků je již na analyzátoru (např. Predatoru) samotném.

Postup výpočtu. Algoritmus pracoval před mými úpravami celkem ve třech fázích, které jsem, dá-li se to tak říci, o jednu fázi rozšířil. Nicméně původní fáze zůstaly nezměněny, proto o nich budu mluvit v přítomném čase. První fáze prozkoumává celý program s tím, že pro každý existující základní blok prochází v dopředném směru všechny instrukce a sestavuje dvě množiny proměnných – množinu `gen` a množinu `kill`. Definice sémantiky množiny `gen` je:

Definice 2. *Každá proměnná v rámci množiny gen je během provádění tohoto bloku prvně čtena a až následně může či nemusí být zabita přiřazením.*

Důsledky množiny jsou takové, že pokud je daná proměnná v množině `gen` daného bloku, všichni jeho předchůdci v CFG musí předávat řízení programu s touto proměnnou v živém stavu. Proměnná je živá vždy **před** provedením první instrukce.

Definice sémantiky množiny `kill` zní:

Definice 3. *Každá proměnná v rámci množiny kill je během provádění tohoto bloku alespoň jednou usmrcena.*

Význam proměnné této množiny hraje roli především při výpočtech první fáze – uchovává jakousi informaci o tom, které proměnné již byly zabity. Dále, pokud je proměnná v této množině, znamená to pro distribuci v rámci výpočtu pevného bodu jakousi bariéru – viz. dále. Následující fáze je výpočet onoho pevného bodu. Cílem je zjistit, které množiny proměnných jsou na **konci** daných základních bloků živé. K tomu vždy budou sloužit množiny `gen` z **následovníků**. Z tohoto také poté plyne ono upřesnění pro jednotlivé cíle `killPerTarget` – ne vždy je potřeba distribuovat živost proměnné všemi směry, ale tak dalece ve vysvětlování nepůjdu. Výpočet pevného bodu posléze probíhá tím způsobem, že:

- Naplníme pomocnou množinu bloků všemi základními bloky funkce a,
- dokud není tato množina prázdná, vezmeme první blok z množiny, jenž budeme zpracovávat a odstraníme jej z pomocné množiny bloků.
- Následuje jeho zpracování
  - Doplníme množinu `gen` bloku o identifikátory všech proměnných, které jsou v množinách `gen` následovníků, *a zároveň* nejsou v lokální množině `kill`.
  - pokud došlo ke změně množiny `gen` tohoto bloku, naplánujeme provedení všech jeho předchůdců tím, že je znova přidáme do pomocné množiny bloků.

Pozn. při výpočtu počítám se standardní definicí množiny – tedy, že dva prvky nemůžou existovat v jedné množině vícekrát. Přidání jíž existujícího prvku je tedy prázdná operace.

Poslední fáze algoritmu už má k dispozici vypočtené množiny živých proměnných pro každý konec základního bloku. Na základě tohoto lze provést jeden finální průchod instrukcemi každého základního bloku ve **zpětném** pořadí a (1) při čtení proměnné v rámci některého operandu proměnnou „oživit“, případně při (2) zápisu do proměnné ji prohlásit za mrtvou (pro posloupnost předcházejících instrukcí).

Problém s odkazovanými proměnnými. Jak už jsem se v kapitole 3.2 zmínil, před mým rozšířením bylo využito základní heuristiky, že pokud *mohla* být proměnná během programu (byť jen po dobu trvání jedné instrukce) odkazována, automaticky byla vyloučena ze seznamu kandidátů pro zabití v rámci celého programu. Tady tento přístup zajišťoval bezpečnost algoritmu a i tak dával příznivé výsledky za jeho poměrně nízkou cenu. Následující sekce se bude věnovat důvodu, proč se blíže věnovat proměnným, které jsou někde v programu odkazovány.

5.1 Motivační příklad

Důvod proč vlastně vzniklo zadání diplomové práce je, že rozhraní Code Listener využívají v současné době nástroje pro analýzu programové haldy a na ní uložených dynamicky vázaných datových struktur. Pro tyto nástroje je doména ukazatelů důležitá. Prakticky, i když se podaří zabít jednu jedinou proměnnou navíc oproti původní verzi analýzy živosti, může dojít k výraznému zrychlení analýzy ve využívajícím nástroji.

Například, představte si pod takovou „cennou“ proměnnou kořen dynamicky vázané struktury typu strom. Proměnná je na první pohled už z definice živosti bez pochyby (myšleno z pohledu člověka, nikoliv stroje) mrtvá, ale nemohla být odstraněna pouze z důvodu, že na ni byla vzata adresa a uložena do ukazatele. To může způsobit zbytečně vynaložený čas zpracování takové proměnné nástroji, které si nějakým způsobem modelují paměť programu a uchovávají si informace o datových strukturách. Tím, že by se odstranil kořen stromu, mohl by se analyzátor zbavit bez problému celého stromu a ušetřil by tím jak prostor, tak především čas, který by vynakládal na jeho analýzu.

Důkazem této situace bude následující útržek kódu, který je převzat z příkladu z testovací sady nástroje Predator. Na kompletní zdrojový kód se můžete podívat v jeho zdrojových kódech v souboru `test-0207.c` a nebo v příloze A.

```
1  void seq_sort(struct list **ptr_to_seq)
2  {
3      struct list *tmp = *ptr_to_seq;
4  #ifndef HAVE_ADVANCED_VAR_KILLER
5      *ptr_to_seq = NULL;
6  #endif
7
8      // do O(log N) iterations
9      while (tmp->next)
10         tmp = seq_sort_core(tmp);
11
12     *ptr_to_seq = tmp;
13 }
14
15 int main()
16 {
17     struct list *seq = NULL;
18
19     /** vytrzeno -- naplneni seznamu "data" **/
20
21     // puvodni hodnota promenne 'seq' po provedeni instrukce
22     // seq_sort() neni potreba -- obsah promenne je totiz behem
23     // provadeni funkce prepsan.
24     seq_sort(&seq);
25
26     // zde uz je ctena a dereferencovana jina hodnota 'seq'
27     struct node *node = seq->slist;
28
29     /** vytrzeno .. **/
30
31     return 0;
32 }
```

Pokusím se teď vysvětlit, co je na tomto příkladu zvláštního, a kde může vylepšené zabíjení proměnných za pomoci PT analýzy pomoci. Zaměřme se tedy na uvedený zdrojový kód (zatím bez ohledu na to, co skutečně dělá).

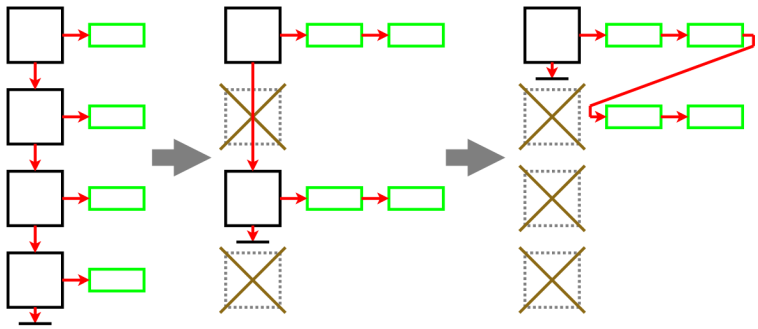
Nejzajímavější část jsou řádky 4–6 Pokud není v době překladu definováno makro `HAVE_ADVANCED_VAR_KILLER`, provede se přiřazení na 5. řádku → `*ptr_to_seq = NULL` (uvedené makro nemá nějaký speciální význam – slouží pouze pro jednoduché vyloučení řádku 5 ze zdrojového kódu v případě potřeby). Přiřazení z řádku 5. ani po bližším prozkoumaní nemá žádný praktický efekt z hlediska činnosti zdrojového kódu. Nicméně z hlediska doby běhu překladu pomocí `gcc` se zavedeným modulem Predatoru ale docházelo (před implementací mého rozšíření) k výraznému zrychlení (téměř pětinásobnému), pokud instrukce na řádku 5. nebyla odstraněna. Můžete si prohlédnout výstup jednoduchého skriptu `testOfSpeedup.sh`, jenž byl spuštěn před aplikací mého rozšíření[1]:

```
$ ./testOfSpeedup.sh
Running predator without -DHAVE_ADVANCED_VAR_KILLER
sl/memdebug.cc:126: note: peak memory usage: 1.92 MB
cl/cl_easy.cc:77: note: clEasyRun() took 4.170 s
Running predator *with* -DHAVE_ADVANCED_VAR_KILLER
sl/memdebug.cc:126: note: peak memory usage: 3.42 MB
cl/cl_easy.cc:77: note: clEasyRun() took 20.350 s
```

Všimněte si onoho téměř 5-násobného zpomalení a nezanedbatelného zvýšení nároků na paměť v případě běhu s aktivní definicí `HAVE_ADVANCED_VAR_KILLER`.

Důvod, proč je druhý běh analýzy Predatorem o tolik pomalejší je, že neexistující přiřazení → `*ptr_to_seq = NULL` neprovede explicitní odstranění proměnné `seq`, která sídlí na adrese shodné s hodnotou proměnné `ptr_to_seq`. Přiřazení hodnoty `NULL` do ukazatele je z pohledu Predatoru ekvivalentní operaci zabití proměnné na základě obsahu množiny `killPerTarget` (viz. odstavec 5). No a právě ani ne tak samotný `seq` ukazatel, ale všechna paměť, která je přes tento ukazatel přístupná, způsobuje Predatoru práci navíc.

Důležité je nyní říci, že při vhodné implementaci analýzy živých proměnných je tuto situaci možné automaticky detekovat. Lze totiž zjistit, že cíl proměnné `ptr_to_seq` je naposledy použit na řádku 101, za nímž může být zabit, protože už nikdy nedojde k jeho čtení – dojde pouze k jeho přepisu na řádku 110. Více o návrhu řešení v kapitole 6.3.



Obrázek 5.1: Příklad výpočtu algoritmu Merge Sort motivačního příkladu.

K samotnému nezjednodušenému zdrojovému kódu. Příklad implementuje řadící algoritmus Merge-sort a také jej rovnou využívá. Na vstupu je algoritmu předána adresa proměnné typu typu `struct list` (řádek 138, viz. příloha A). Tato struktura je jednosměrně vázaným seznamem, jehož prvky mohou obsahovat další pod-seznamy typu `struct node` (libovolné délky).

[1] Zdrojový kód naleznete v příloze A nebo v přiložených zdrojových kódech na CD

Na začátku je však proměnná `seq` (po naplnění v cyklu na řádce 116) v lineárním tvaru, který je vhodný právě pro Merge sort – každý prvek seznamu `seq` ukazuje na podseznam délky 1. Procedura `seq_sort` iteruje nad seznamem `tmp` (řádek 108), dokud jej funkce `seq_sort_core()` nezredukuje tak, že obsahuje pouze jeden prvek. Všechny prvky napříč seznamem `seq` byly setříděny podle definované relace uspořádání a zavěšeny na první podseznam proměnné `tmp`. Příklad výpočtu tohoto algoritmu pro čtyři prvky je vyobrazen na obrázku 5.1.

Kapitola 6

Návrh a realizace rozšíření

V této sekci popíšu postup návrhu rozšíření, zmíním se o pár úpravách algoritmu analýzy živých proměnných, které předcházely implementaci mého rozšíření a o úpravách rozhraní Code Listener, ke kterým řešení této práce vedlo.

6.1 Oprava chyb v analýze živosti

Ještě před zahájením návrhu rozšíření jsem se snažil podrobně pochopit jednak referenční algoritmus k analýze živosti z [12], ale také implementovaný algoritmus v rozhraní Code Litener. Z počátku jsem měl problém s pochopením jeho kroků. Částečnou příčinou byla chyba v algoritmu, na kterou se tím přišlo.

První chyba v algoritmu analýzy živosti. Původně, při 3. fázi analýzy, docházelo k tomu, že byly všechny zdrojové i cílové proměnné daných instrukcí přidávány do seznamu `live` (viz. 5). Pro proměnnou obsaženou v tomto seznamu to znamená, že je v daném místě CFG živá. Ovšem, jakmile se proměnná takto stala živou, nikdy už z tohoto seznamu nebyla odstraněna a tudíž nebyla zabíjena při následující situaci:

```
1  ; Mnozina varsToKill ~~>   #spravne#     #pred opravou#
2  %mF1705:x := 1          ;  { x }         { }
3  %mF1705:x := 2          ;  { x }         { }
4  %mF1705:x := 3          ;  { }           { }
5  %mF1706:z := %mF1705:x  ;  { z, x }      { z, x }
6  ret
```

Uvedený problém v pravém sloupci příkladu je důsledkem toho, že (při průchodu instrukcemi ve 3. fázi – tedy odzadu) instrukce na řádku 5 při čtení proměnné `x` nastavila tuto proměnnou jako živou a při té příležitosti ji také přidala jako cíl zabíjení do množiny `varsToKill`. Nikdy již ale nebyla z množiny živých proměnných `live` odstraněna – tudíž nemohla být znova (v kódu „dříve“) zabita. Tuto chybu opravuje příspěvek v oficiálním git repozitáři `152fbf6840ed53`.

Druhá chyba v analýze živosti. Další chyby si všiml vedoucí práce nedlouho po opravě první a byla oproti ní poněkud záludnější. Souvisela s tím, že pokud proměnná, která byla zařazena do množiny `gen` při výpočtu pevného bodu (viz. 5) pouze pouze kvůli tomu, že měla zůstat živá jen pro některý cílový základní blok *a zároveň* nebyla *vůbec použita* v rámci aktuálně analyzovaného bloku, nedošlo k jejímu zabíjení pomocí `killPerTarget` vektoru. Tuto chybu opravuje můj příspěvek v git repozitáři `80b0968e4b1e`.

Platforma pro automatické testy algoritmu

Vzhledem k tomu, že právě kvůli podobným opravám chyb (které byly mimo jiné spíše vylepšením, jelikož opravené chyby nebyly nebezpečné) hrozí vždy riziko dalšího zanášení problémů do kódu, vznikla potřeba pokrýt výpočet tohoto algoritmu regresními testy (ponechávám v sekci oprav chyb kvůli přímé návaznosti).

Bohužel, pro tento typ testů doposud neexistoval v CL žádný podpůrný mechanizmus, proto vedoucí práce vytvořil příspěvkem `9ea8779087c7ba` separátní modul `chk_var_killer`. Tento modul by sice již (po implementaci mého rozšíření) potřeboval mírně upravit, aby pokryl veškeré potřebné situace, jež jsou zapotřebí nyní testovat, nicméně, dovede i tak — nezávisle na modulech Predator nebo Forester — detekovat, pokud dojde úpravou algoritmu k nějakému problému při zabíjení proměnných.

Právě tento testovací modul mi velmi pomohl při dalším ladění projektu a především při návrhu dalšího testovacího modulu (viz. 6.4).

6.2 Implementace grafu volání

Už dopředu nám bylo jasné, že bude potřeba k implementaci mého rozšíření graf volání, který doposud nebyl v CL implementován. Během mojí práce na konstrukci PT grafu proto vzniklo paralelně interní rozhraní poskytující vystavěný graf volání analyzovaného programu. Vznik grafu volání a především znovupoužitelného rozhraní má na svědomí vedoucí práce.

Zdrojové kódy jsou rozděleny na dvě části (1) první, deklarační část je, jako zbytek rozhraní `CodeStorage` zahrnut v rámci souboru `include/cl/storage.hh` a (2) druhá část, tedy implementační dostala svůj vlastní soubor `cl/callgraph.cc`. Příspěvky do git repositáře, které vytvořily rozhraní a kód jsou `1fce3fec7dc62` a `dd0561b7404273`.

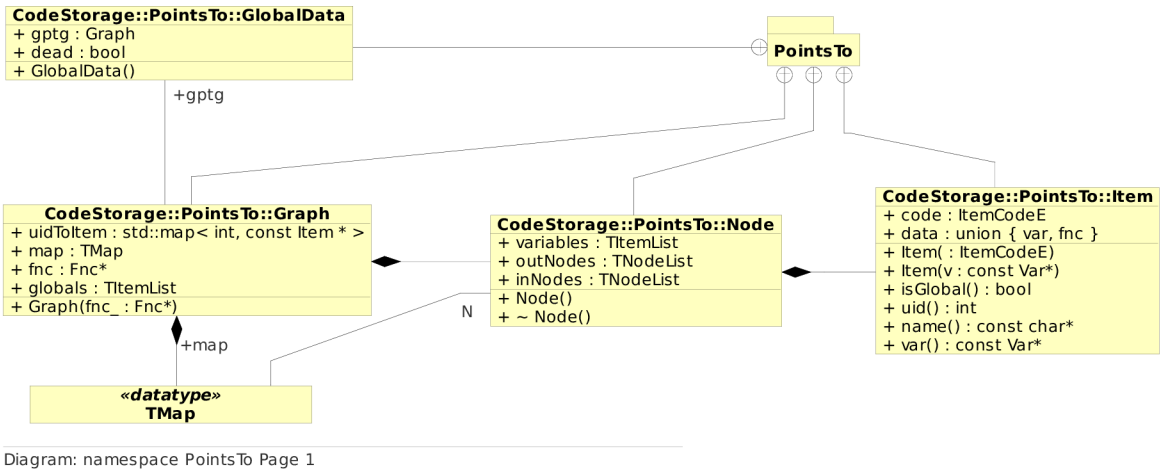
Časem jsem pro účely hledání chyb přidal kousek kódu pro vykreslení grafické podoby grafu volání. V sekci 2.4 jsem uvedl příklad grafu volání na obrázku 2.2, který je jeho výstupem. Jak vygenerovat tento graf ještě řeknu blíže v sekci 4.2.

6.3 Návrh rozšíření

Při návrhu formátu uložení PT grafu jsem se snažil, aby byl co možná nejvíce znovupoužitelný – a to jednak z pohledu uživatele knihovny CL, ale jednak také z hlediska rozvoje CL. Snažil jsem se vytvořit strukturu grafu, uzlů i hran tak, aby postihla potřeby známých algoritmů, jako je Steensgaardův, Andersenův, apod. Už teď je jasné, že pro tokově senzitivní algoritmy podobná struktura v žádném případě nebude dostatečná (implementace PT grafu takové analýzy musí využívat řídkých datových struktur, jinak by docházelo k velkým nárokům na čas i paměť, bude potřebovat zcela jiný typ obslužných metod, atd.), nicméně pro reprezentaci běžných tokově insenzitivních algoritmů poslouží návrh dobře – co je důležité, poslouží také algoritmu FICS.

Návrh struktur týkajících se points-to výpočtu

Jako globální sklad informací týkajících se Points-to slouží třída `GlobalData`. Její instance je vložena jako součást celé struktury `CodeStorage`, takže je uživateli dostupná. V tuto chvíli je uvnitř struktury uložena adresa globálního PT grafu (informace o PT relacích,



Obrázek 6.1: Návrh rozhraní pro PT analýzu.

které jsou platné pro celý program) a informace o tom, zda-li je graf v pořádku vystavěn – tedy zda nedošlo při konstrukci k chybě (položka `dead`).

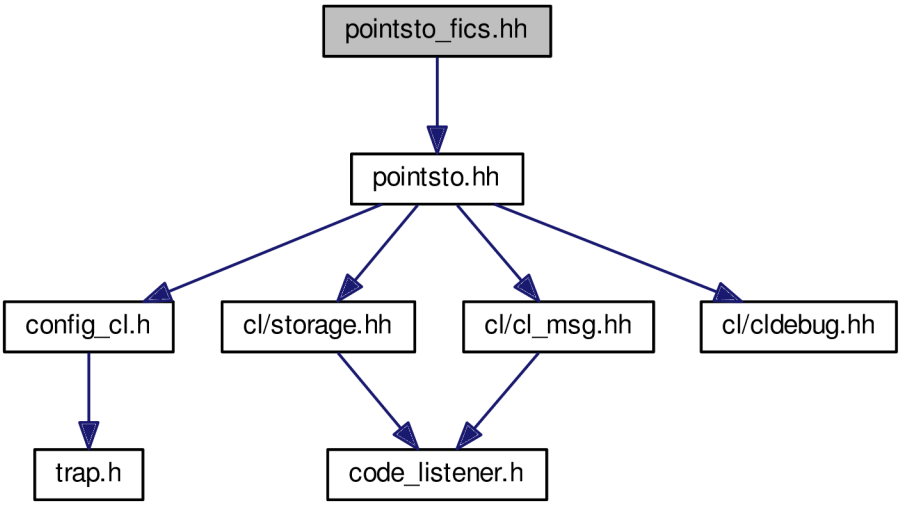
Základní strukturou návrhu je třída `CodeStorage::PointsTo::Graph`. Instance této třídy reprezentují PT grafy pro každou funkci programu, ale také již zmiňovaný globální PT-graf (`GlobalData::gptg`). Obsahuje položku `map`, která je typu `TMap` a slouží jako rychlý vyhledávač uzlu grafu na základě identifikátoru proměnné. Dále, seznam odkazů na položky globálních proměnných `globals` – tato struktura zjednodušuje implementaci algoritmu FICS nad daným grafem.

Uzel grafu je definován jako `CodeStorage::PointsTo::Node`, jeho instance vždy náleží pouze jednomu objektu typu `Graph`. Položky typu `Node` jsou především `variables`, což je množina prvků typu `Item`.

Položka uzlu reprezentující (nejen) proměnnou. Třída `Item` slouží pro reprezentaci třech typů možných objektů v rámci jednoho uzlu, a to (1) běžné proměnné, ať už pouze lokální, nebo globální, (2) návratové hodnoty funkcí a (3) alokovaná místa (např. pomocí funkce `malloc`). Druhý typ položky existuje v rámci uzlu proto, že je potřeba při výpočtu kontextově senzitivních PT algoritmů počítat někdy (např. v algoritmu FICS) s návratovými hodnotami funkcí. Návratová hodnota je brána jako lokální parametr funkce – tedy běžná lokální proměnná. Třetí typ, tedy alokovaná místa, budou reprezentovat paměť, která se stává přístupnou programu po provedení nějaké externí funkce. Nejlepším příkladem je funkce `malloc()`. Po provedení funkce vzniká na haldě objekt, na něž vede ukazatel, jež je vracen jako návratová hodnota tohoto volání. I když se při každém volání této funkce vrátí ukazatel na jiné reálné místo v paměti, lze nad nimi vytvořit abstrakci a zahrnout je všechny pod jednu položku. Tedy – položka tohoto typu abstrahuje právě jedno místo volání dané funkce. V pozdější implementaci to bude znamenat, že pokud přijde na zpracování instrukce volání

$$int \;\; x = malloc(4)$$

vytvoří se dva uzly. Jeden standardně bude reprezentovat uzel s položkou x, a druhý bude obsahovat alokovanou paměť, řekněme h_1. Index ve jméně bude unikátní pro každé místo volání podobné funkce. Vztah těchto uzlů bude takový, že první uzel bude spojen výstupní hranou s uzlem druhým na znamení, že x *může* ukazovat na h_1. Dále, při výpočtech



Obrázek 6.2: Graf vkládání s počátkem v `pointsto_fics.hh`

ve FICS se tento typ položky bere, jako by se jednalo o globální parametr (viz. metoda `Item::isGlob()`).

Shrnutí. Diagram tříd jmenného prostoru `CodeStorage::PointsTo` je uveden na obrázku 6.1 – obsahuje pouze nejdůležitější struktury a třídy, podrobnější informace jsou ve zdrojových kódech. Podrobnější informace o implementaci bude v následující sekci 6.4.

6.4 Praktická implementace mého rozšíření

V této sekci popíšu podrobněji hierarchii nových souborů, popíšu implementační detaily přidaného FICS algoritmu v návaznosti na kapitolu 3.2. Předvedu zde ladící grafický výstup z FICS algoritmu – tedy grafickou podobu PT grafu. Konečně, popíšu také potřebné úpravy algoritmu analýzy živých proměnných, která s využitím vystavěného PT grafu dovede zabít více proměnných, než tomu bylo doposud.

Hierarchie přidaných zdrojových souborů

Na obrázku 6.2 je vyobrazena závislost jednotlivých hlavičkových souborů. Dle této hierarchie jsem se snažil segmentovat obslužné funkce tak, aby obecné datové struktury byly dle konvencí Code Listeneru v souboru `include/cl/storage.hh`. Tento soubor se zdá být držen v co nejkompaktnější formě, a proto také interní metdoy mají své rozhraní v souboru `cl/pointsto.hh`. Oba soubory můžou být využity koncovým analyzátorem, ale také interně při konstrukci či zpracování výsledného grafu.

Na souboru `pointsto.hh` závisí další popisovaný soubor `pointsto_fics.hh`, který slouží jako rozhraní obsahující spouštěcí funkci algoritmu FICS. Motivace k takové implementaci řetězce závislostí je zřejmý. Chtěl jsem rozumně separovat obecné PT funkce od funkcí, které implementují konkrétní algoritmus – ty zůstávají jako statické funkce v `cl/pointsto_fics.cc`.

Konkrétní implementace všech obslužných funkcí poté sídlí v stejně se jmenujících souborech s příponou `*.cc`.

Obslužné zdrojové kódy pro regresní testy.

V souboru `cl/tests/chk_pt.cc` byl vytvořen nový zásuvný modul `gcc` pro umožnění automatického testování algoritmu pro výpočet points-to relací. Tento modul při spuštění prochází celý CFG programu a hledá instrukce volání vestavěné funkce `PT_ASSERT()`, které posléze zpracovává. Pro účely testování dále vznikl soubor `cl/tests/data/include/pt.h`. Ten obsahuje obslužná makra pro jednodušší testování požadovaných vlastností PT grafu. Testy pomocí přeloženého modulu `chk_pt.so` pracují tak, že se spustí `gcc` s tímto modulem a předá se mu jako parametr testovací zdrojový kód.

Pseudo-funkce PT_ASSERT. Pro účely regresních testů vkládáme v místech, kde je to žádoucí, volání pseudo-funkce `PT_ASSERT`. Definice této funkce ve skutečnosti ani neexistuje – jedná se jen o způsob, jak efektivně komunikovat z prostředí testovacího zdrojového kódu s modulem `chk_pt.so`. Toto volání by mělo být prakticky bez vedlejších efektů (tedy by nemělo mít vliv na výsledný kód, který který je generován), vyjma přidání dané instrukce volání). Obsahuje variabilní počet parametrů a relativně mnoho možností, jak jej použít. Důležitý je především parametr `type` datového typu `enum PTAssertType`. Ten určuje, jaký předpoklad na výsledný PT graf je kladen. Pro příklad uvedu, že může jít o test, zda-li některá proměnná ukazuje v rámci celého programu na jinou konkrétní proměnnou. Pokud bychom chtěli provést tady tento test, vložíme v místě testovacího souboru toto makro s prvním parametrem `PT_ASSERT_MAY_POINT` tak, jako je tomu v následujícím testu na řádku 14:

```
1  /* potrebne inkluze pro funkcnost maker */
2  #include "include/pt.h"
3
4  int main(int argc, char *argv)
5  {
6      int   *   p;
7      int       d;
8
9      // vynutime, aby promenna 'p' ukazovala na 'd'
10     p = &d;
11
12     // dan predpoklad, ze promenna 'p' musi "nekde" v ramci celeho
13     // programu ukazovat na promennou 'd'
14     PT_ASSERT(PT_ASSERT_MAY_POINT, &p, &d);
15
16     // alternativni jednodussi zpusob zapisu
17     ___cl_pt_points_glob_y(p, d);
18
19     // opacny typ predpokladu
20     ___cl_pt_points_glob_n(d, p);
21 }
```

Všimněte si operátoru `&`, který předchází oběma testovaným proměnným. Zjistili jsme, že pokud tento operátor v rámci volání `PT_ASSERT` u proměnných není zadán, `gcc` bohužel vygeneruje (samozřejmě mimo jedné instrukce volání této funkce) jednu instrukci přetypování navíc. Tato instrukce navíc je z hlediska PT volena ale tak nešťastně, že pokud by makro bylo užito tímto způsobem, docházelo by jednak k velkým změnám PT relací, ale co víc, proměnná vygenerovaná jako parametr volání v interním kódu `gcc` by ani nebyla originální. Jednalo by se o její dočasnou přetypovanou kopii. Viz. následující příklad:

```
1  #include "include/pt.h"
2
3  #define MP PT_ASSERT_MAY_POINT
4
5  int main()
6  {
7    int d = 10;                | %mF1691:d := 10
8    int *p = &d;               | %mF1692:p := &%mF1691:d
9    /* instrukce navic => */   | %mF1694:d.0 := %mF1691:d
10   PT_ASSERT(MP, p, d);       | PT_ASSERT(0, %mF1692:p, %mF1694:d.0)
11                              | ret
12 }
```

Zde si všimněte, že proměnná předaná testovacímu modulu (řádek 10.) ve skutečnosti není `d`, ale dočasná proměnná `d.0`. Nutno podotknout, že použití znaménka `&` sice způsobí, že funkci skutečně klademe podmínku, jakou chceme, nicméně stejně neřeší vygenerování instrukci přetypování navíc. To ovšem až tak nevadí – vznikne jeden nikým neodkazovaný uzel v PT grafu s jedinou (netečnou) proměnnou vevnitř (ta odpovídá zmiňované dočasné proměnné).

Pomocné ladící skripty. Testy tohoto typu naleznete v cestě `cl/tests/data/pt-*.c` – po aplikaci změn v rámci diplomové práce jsou spuštěny při klasických testech při spuštění `make check`. Uvedené postupy lze mj. jednoduše vyzkoušet tak, že si svoje testovací zdrojové kódy zkontrolujete pomocí skriptů `cl/tests/ptgccv`, případně `cl/tests/ptgdb`, které jsem (inspirován existujícími ladícími skripty v CL a Predatoru) také vytvořil. V případě, že zdrojový kód nesplňuje některý z předpokladů zadaných pomocí funkce `PT_ASSERT`, modul `chk_pt.so` vrátí chybový kód 1 a vypíše například podobné hlášení na standardním chybovém výstupu:

```
ck_pt.cc:128: error: points-to expect variable "d" should follow "p" in
function main
```

Výstupem jsou v tomto případě mj. také soubory `pointsto-0000.dot` a `callgraph-0000.dot`, o kterých bude ještě řeč v ukázkách průběhu výstavby PT grafu v následující sekci.

6.4.1 FICS algoritmus

Vstupním bodem algoritmu je v souboru `cl/pointsto.cc` spuštění funkce `runFICS()`, která je definována v souboru `cl/pointsto_fics.cc`. implementace algoritmu byla prováděna se snahou co možná nejlépe následovat pseudokód uvedený v [10]. Názvy metod jsou tedy voleny velice podobně.

Na uvedení implementace je potřeba říct, že algoritmus pracuje, nicméně není pokryta celá škála příkazů jazyka C, která může nastat. Z tohoto důvodu také existuje proměnná `PointsTo::GlobalData::dead`, která jak název napovídá, značí, zda-li byl graf vystavěn správně a nenarazilo se při konstrukci na žádný problém. Pokud se tedy narazí na příkaz nebo typ konstrukce (v tuto chvíli například instrukce nepřímého volání funkce), se kterým si stávající algoritmus zatím neumí poradit, analýza neproběhne. Na tuto situaci musí dále reagovat závislé algoritmy jako je analýza živosti, která ač méně efektivně, i tak musí proběhnout (tak, jak tomu bylo před mou úpravou). V rámci zmiňovaného souboru tedy můžete několikrát narazit na konstrukci `FALLBACK(error_message)`, která je použita právě v těchto situacích – ukončuje výstavbu grafu, vypisuje na standardní chybový výstup důvod

přerušení a vynutí nastavení příznaku `dead`. Dále, z velmi jednoduché funkce `runFICS` je vidět, že všechny tři fáze algoritmu musí proběhnout v pořádku (vrátit log. hodnotu `true`), aby mohla být výsledná sada PT grafů brána za korektně sestavenou. Nyní k postupu.

Fáze první – `ficsPhase1()`. Oproti referenčnímu pseudokódu zde není téměř žádný rozdíl. Pro případ definovaných funkcí se v této fázi postupně prochází jednotlivé instrukce, o které se stará funkce `phase1handleInsn` – ta přeskakuje vestavěná volání (např. zmiňovaný `PT_ASSERT`) a instrukce, které nejsou prozatím pro FICS zajímavé (tedy vše vyjma instrukcí `return` a přiřazení do ukazatele).

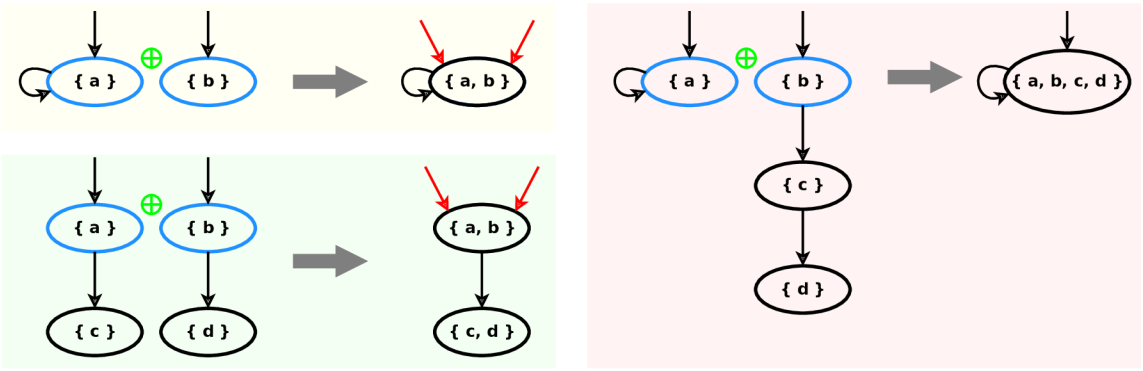
Fáze druhá. Tato fáze již nepracuje se všemi instrukcemi. Nyní už se prochází pouze selektivně instrukcemi volání typu `CL_INSN_CALL` a to na základě grafu volání. Cílem této fáze je protlačit PT relace z volaných funkcí do volajících (funkce `bind()`) a vystavět globální graf (funkce `bindGlobal()`).

V rámci fáze dvě dochází poprvé k využití šablony `FixPoint`, která také vznikla v rámci mých úprav (jako inspirace velmi jednoduché šablony `WorkList` z `cl/worklist.hh`). Šablona slouží (pro tento příklad) jako zásobník funkcí, které se mají zpracovat. Jednoduše dovede hlídat „nekonečný cyklus“ a zastavit jej, jakmile je tento zásobník prázdný (viz. soubor `cl/fixpoint.hh`). Tento zásobník je postupně doplňován, pokud dojde uvnitř volání `bind()` či `bindGlobal` k nějaké úpravě cílového grafu. V takovém případě se na základě informací, jež jsou poskytnuty uzlem grafu volání aktuálně zpracovávané funkce, naplánuje znova zpracování všech funkcí, které tato funkce volá (metoda `FixPoint.schedule()`). Na závěr každého kroku výpočtu pevného bodu také dochází k doplňování globálního grafu pomocí `bindLocationsGlob()`, což je první použitá obálka nad funkcí `bindLocations()`. Tato funkce vyžaduje dva grafy – zdrojový a cílový, a také očekává vektor prvků, které jednoznačně identifikují dvojice uzlů, na kterých startuje propagace vzhůru (struktura `TBindLocData`). Následující postup se několikrát v FICS opakuje, proto si jej pojmenujme podobně jako v originálním článku *propagace do hloubky*. Na základě oněch dvojic si funkce postupně vezme vždy uzel ze zdrojového grafu a jemu odpovídající uzel z cílového grafu a nakopíruje položky ze zdroje do cíle.

Pokud daná položka už v cílovém grafu existuje (a dané místo není cílový uzel), musí provést sjednocení těchto uzlů v cílovém grafu[1]. Další postup je takový, že se vezme následující **`nenavštívený`** uzel ve zdrojovém grafu (dokud nějaký ještě existuje) a následující uzel v cílovém grafu a provede se s ním to samé, jako se startujícím párem – tedy překopírování proměnných, případně spojení dvou cílových uzlů. O operaci spojení uzlů bude pár slov ještě níže.

Fáze třetí. Před zahájením poslední fáze již máme kompletně vypočtený globální PT graf. Máme také téměř vypočteny lokální PT grafy všech funkcí. Jediné, co se v tuto chvíli může změnit je, že se může propagovat nějaká PT relace směrem z volající funkce do volané, kde může jít například i o lokální proměnnou. Opačný směr propagace lokálních proměnných (tedy k volajícímu) nedává smysl – jakmile je totiž volaná funkce u konce, proměnné zanikají a odkaz z volajícího by neměl žádný význam.

[1] Tohle je jedna z částí algoritmu, která není příliš přehledná a kterou se mi vlastně ani nepovedlo na základě referenčního materiálu [10] pochopit. Na korektní implementaci jsem vlastně postupně přicházel iterativně v cyklu neustálých (1) úprav zdrojového kódu a následného (2) psaní regresních testů, které opět způsobovaly nečekaný výstup algoritmu.



Obrázek 6.3: Operace spojení uzlů

Tedy, výpočet této fáze využívá také `FixPoint` strukturu. Zpracovává všechny funkce minimálně jednou a to tím způsobem, že volá pro každou funkci dva typy `bindLocation()`. První je `bindLocationsGlob`, jež nastartuje propagaci do hloubky (1) počínaje uzly, jež odpovídají globálním proměnným globálního PT grafu a to (2) z globálního PT grafu do grafu zpracované funkce. Další propagace do hloubky – `bindLocationsArgs` – je provedena pro každé volání funkce z aktuálně zpracovávané. Zde je potřeba vzít v úvahu to, že návratová hodnota nelze propagovat směrem dolů. Tato propagace směrem dolů tedy (a) začíná na uzlech, které odpovídají operandům, jež jsou ukazateli a (2) informace o PT relacích se kopírují do PT grafu volající funkce (včetně lokálních proměnných).

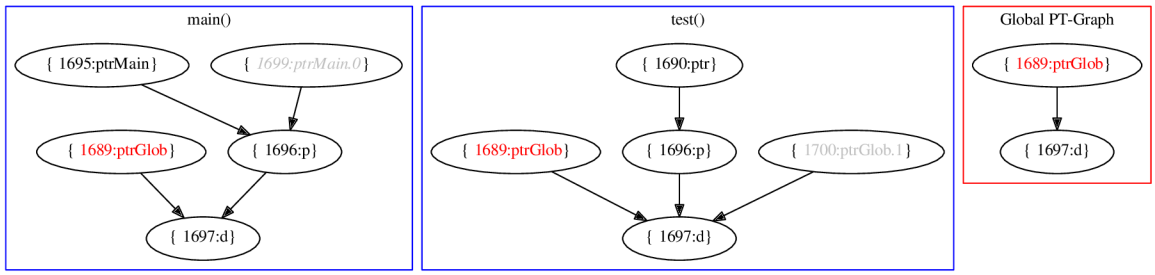
Sjednocení proměnných dvou uzlů. Prozatím jsem nepopisoval žádnou z funkcí definovaných v souboru `cl/pointsto.hh`. Není ani moc co popisovat, jelikož se v pro většinu z nich jedná o elementární operaci. Nicméně jedna z nich (respektive dvě) si to, už kvůli předchozímu popisu FICS algoritmu, jenž operoval s propagací do hloubky, zaslouží. Jde o funkci `joinFixPointS` (v kombinaci s `joinNodeS()`). Tato funkce se stará o ono zdánlivě snadné spojení dvou uzlů v jeden. Při této operaci však může dojít k několika situacím, které je proto potřeba řešit dokonce výpočtem pevného bodu.

Na obrázku 6.3 jsou tři běžné příklady operace spojení. Všimněte si, že příklad na žlutém pozadí je běžná situace, dojde k zániku jednoho uzlu, jehož proměnné jsou přesunuty do uzlu druhého. Nový uzel také obsahuje všechny vstupní hrany, které před tím obsahovaly oba dva uzly. Příklady na zeleném a červeném pozadí jsou komplikovanější. Pokud by došlo pouze ke spojení uzlů tak, jako ve žlutém příkladu, ztratila by se informace o následovnících daných uzlů. Proto algoritmus probíhá zmiňovaným postupem výpočtu pevného bodu – jakmile dojde ke spojení, naplánují se následující uzly také ke spojení. A to i když následovníkem uzlu je ten samý uzel (červený příklad).

Grafické reprezentace konstruovaného grafu lze vygenerovat pomocí dříve uvedených ladících skriptů. Jakmile proběhne PT analýza daným skriptem v pořádku, objeví se v daném adresáři dva soubory (jeden pro graf volání a druhý pro points-to graf, oba s příponou `*.dot`), které lze pomocí překladače jazyka *dot*[2] převést na mnoho podporovaných grafických formátů. Pro výstp v podobě `pdf` je vhodný následující příklad použití:

```
$ dot -Tpdf pointsto-0000.dot > pt-graf.pdf
```

[2] http://en.wikipedia.org/wiki/DOT_language



Obrázek 6.4: Grafická reprezentace poinst-to grafu – modře orámovaný podgraf je grafem funkce, červeně orámovaný je globální graf. Červenou barvou psané proměnné jsou globální proměnné, černé lokální a šedé jsou dočasné proměnné.

Příklad výstupu je na obrázku 6.4. Tento graf byl sestaven pro jeden z regresních testů (`data/pt-0804.c`) pro PT analýzu.

Dále, např. kvůli ladění výstavby grafu, se může hodit vykreslit PT graf po každé provedené změně v tomto grafu. K tomu slouží argument `plot-changes`, který se jako proměnná prostředí předává generujícím skriptům. Příklad použití:

```
ARGS=plot-changes ./ptgccv fileToAnalyse.c
```

Podpora pro tento grafický výstup jsem implementoval uvnitř souboru `cl/clplot.cc`.

6.4.2 Úprava dosavadní analýzy živých proměnných

Analýza živosti bude už pracovat s PT grafem. To znamená, že výpočet tohoto grafu musí být udělán minimálně bezprostředně před zahájením. Volání obou analýz tedy naleznete v souboru `cl/cl_easy.cc` těsně vedle sebe:

```
1  /** utrzek ze souboru  cl/cl_easy.cc */
2  void run(CodeStorage::Storage &stor) {
3      /** .. vytrzeno .. **/
4
5      CL_DEBUG("building points-to graph...");
6      pointsToAnalyse(stor, configString_);
7
8      CL_DEBUG("killing local variables...");
9      killLocalVariables(stor);
10
11     /** .. vytrzeno .. **/
12 }
```

Situace, kdy je bezpečné zabít něco navíc. Jako cíl mých úprav bylo postihnout situaci v motivačním příkladu ze sekce 5.1. Jde o jasnou situaci, kdy člověk dovede pohledem na kód bez problémů usoudit, že proměnná `seq` je mrtvá. Důvod pro tento úsudek je zřejmý – v rámci kódu funkce `main()` dojde k zavolání `seq_sort()`, a v rámci kódu volané funkce dojde k přiřazení do cíle ukazatele `ptr_to_seq`. Člověk ví, že tedy cíl – paměť `*ptr_to_seq` může být považována během průběhu `seq_sort()` za mrtvou. Tento úsudek by ale nebylo možné provést v případě, že by funkce `seq_sort()` byla v rámci analyzovaného programu volána vícekrát. Nastával by problém v tom, že by nebylo naprosto zřejmé při pohledu na funkci `seq_sort()`, jaká proměnná je na ukazateli `ptr_to_seq` zavěšena.

Pokud máme nalézt místo v CFG dané funkce, které má zabíjet cíl ukazatele, musíme si být jisti, že hodnoty všech proměnných tohoto místa nebudou nikdy více čteny. Vypočtená PT analýza nám dovede říct množinu proměnných, jež se může objevit jako cíl daného ukazatele. Z hlediska automatizace lze bezpečně usoudit, že (1) pokud je na daném ukazateli právě jedna možná proměnná a (2) pokud je tato funkce volána v rámci programu pouze jednou, (3) ta konkrétní jedna proměnná je lokální proměnnou volající funkce a konečně, (4) odkazovaná proměnná není nikde jinde v PT grafu odkazována[3], máme jistotu, že na počátku analýzy volané funkce můžeme prohlásit daný cíl ukazatele jako alias pro tuto proměnnou. Teoreticky ještě může jít o ukazatel s hodnotou `NULL`, nicméně jakákoliv operace vedoucí k zabití cíle takového ukazatele by byla nekorektní operací.

Použitá úprava analýzy živosti. Na základě předchozích úvah jsem provedl také implementaci. Analýza je proto nastavena tak, že rozšířené zabíjení nastane pouze tehdy, pokud jsou splněny výše uvedené podmínky. To zajišťuje, že algoritmus zůstává bezpečný pro libovolný program. Algoritmu se nyní, za předem specifikovaných podmínek, podaří zabít více proměnných. Pokud ale nastane nepodporovaná situace, provádí se původní analýza.

Důležitou funkci v tomto ohledu hraje funkce `VarKiller::alias()`. Ta je použita ze začátku analýzy k sestavení všech bezpečných aliasů typu *ukazatel* → *cílová proměnná*. Jakmile dojde k operaci „čtení z“ nebo „zápisu do“ cíle takového ukazatele, provede se funkce `VarKiller::scanVar()` jak pro daný ukazatel, tak pro cíl ukazatele – a to v tomto pořadí. Co je zde jediné navíc, je prozkoumání cílové proměnné. Proto se dá říct, že původní algoritmus zůstal téměř nezměněn. Pro podrobnější informace lze nahlédnout do zdrojových kódů.

Alternativní úprava rozvojem CFG. Další možností, jak počítat interprocedurální analýzu živosti, by bylo provádět výpočet pevného bodu napříč celým programem – tedy ne pouze lokálně pro dané funkce. Tento postup by byl i efektivnější, protože při zanedbání vedlejších efektů externích funkcí bychom nemuseli rozlišovat globální a lokální proměnné, tedy existovala by možnost zabíjet i proměnné globální (což nyní není). Tento postup by však vyžadoval rozvoj CFG tak, jako je tomu pokud použijeme před funkcí klíčové slovo `inline`. Program by byl tedy brán jako jedna velká funkce, pro niž by byl výpočet živosti podobný, jako je tomu doposud u jednotlivých funkcí. Nevýhoda tohoto řešení je, že by ovšem nefungovalo pro programy s rekurzivním grafem volání. CFG takového programu nelze rozvinout. Další možnost by byla provádět dva běhy analýzy živosti – první tak, jako tomu bylo před mými úpravami a druhý, který by zkusil, zda-li je možné CFG rozvinout, a případně provedl analýzu podruhé nad grafem rozvinutým. Tento postup jsem nezvolil z důvodu, že jsem se snažil minimalizovat zásah do algoritmu analýzy živosti.

[3] Tuto podmínku lze relaxovat při použití rozvoje CFG, jak je uvedeno níže.

Kapitola 7

Zhodnocení výsledků

V této kapitole shrnu, jaký přínos má moje úprava pro nástroj Code Listener, jaký přínos má rozšíření pro nástroje, které jej používají, a shrnu některé úpravy, které jsem provedl v rámci práce a můžou mít vliv na další vývoj projektu. Posléze se podívám na související problémy, na které jsem během řešení diplomové práce narazil, a jejichž řešení přesahovalo požadavky stanovené v zadání diplomové práce.

7.1 Přínos práce

Cíle stanovené v zadání práce byly splněny. To znamená, že byl navržen, a také implementován algoritmus, který pro některé zdrojové kódy dovede provést analýzu živých proměnných efektivněji, než tomu bylo doposud. Tato skutečnost může vést k nezanedbatelnému zrychlení analýzy jisté třídy programů a bylo jí dosaženo za následujících podmínek:

- Základní pravidlo, kterého jsem se po celou dobu vývoje držel, bylo implementovat rozšíření tak, aby moje rozšíření nezpůsobilo problémy v existujících regresních testech nástrojů Predator a Forester. Tedy, jinými slovy, aby nedošlo k omezení množiny analyzovatelných programů.
- Další cíl byl implementovat PT-analýzu, která bude dostatečně efektivní na to, aby čas vynaložený na její výpočet cílového uživatele neobtěžoval.
- Nejdůležitější předpoklad, jež jsem kladl na výslednou implementaci, byl zachování bezpečnosti algoritmu.

Za těchto okolností lze prohlásit, že moje cíle byly splněny. Nyní následuje zhodnocení praktických přínosů mého rozšíření.

Očekávané zrychlení motivačního příkladu. Důkazem přidané hodnoty rozšíření je běh testovacího skriptu `testOfSpeedup.sh`, který jsem již jednou v této dokumentaci spouštěl v kapitole 5.1 – to proto, abych demonstroval cenu výpočtu analýzy před mými úpravami. Pokud si jej spustíme po aplikaci mých úprav, výsledkem bude následující výstup[1]:

```
$ ./testOfSpeedup.sh
Running gcc without -DHAVE_ADVANCED_VAR_KILLER
```

[1] Návod, jak kterýkoliv z uvedených postupů reprodukovat, je v souboru README v kořenovém adresáři přiloženého CD

Soubor testu	Počet PT zabití proměnných	Celkem zabito
test-0034.c	1	25
test-0035.c	1	24
test-0058.c	1	25
test-0059.c	1	37
test-0093.c	1	49
test-0124.c	1	108
test-0155.c	1	41
test-0167.c	1	65
test-0176.c	1	15
test-0207.c	2	100

Tabulka 7.1: Statistiky počtu zabití s pomocí PT analýzy.

```
sl/memdebug.cc:126: note: peak memory usage: 1.92 MB
cl/cl_easy.cc:77: note: clEasyRun() took 4.120 s
Running gcc *with* -DHAVE_ADVANCED_VAR_KILLER
sl/memdebug.cc:126: note: peak memory usage: 1.92 MB
cl/cl_easy.cc:77: note: clEasyRun() took 4.090 s
```

Tedy, jak lze vidět z paměťových a časových nároků ve výpisu, nároky na verifikaci pomocí Predatoru nyní skutečně nerozlišují situace, kdy „motivační kód“ obsahuje zmiňovanou instrukci `*ptrToSeq = NULL` a kdy ne. Znamená to, že dochází tak jako tak k implicitnímu zabití proměnné `seq` ve funkci `main()`, nezávisle na zmiňované instrukci. Tato skutečnost vede na zhruba pětinásobné zrychlení celkové analýzy tohoto zdrojového kódu (pro případ, kdy není tato instrukce součástí kódu).

Ovlivnění existujících testů. Kromě výše uvedeného ukázkového příkladu vylepšení příznivě ovlivnilo verifikaci také dalších testů distribuovaných spolu s nástrojem Predator.

Prvně bych rád prezentoval množinu regresních testů, v nichž moje rozšíření dosáhlo nějakého úspěchu a zabilo nějaké proměnné navíc oproti předchozí implementaci. Množina testů byla zjištěna pomocí skriptu `testOfPTKiller.sh`. Běh skriptu provádí (1) záměnu informačního hlášení o počtu „navíc“ zabitých proměnných za úplné ukončení běhu analýzy, (2) provede překlad zdrojových kódů, a (3) následně spustí regresní testy. Tato posloupnost akcí způsobí, že testy, ve kterých dochází k zabíjení proměnných na základě mého rozšíření, havarují. Tabulku 7.1 jsem sestavil na základě výstupu tohoto skriptu.

Za druhé bych rád uvedl statistické výsledky skriptu `countStats.sh`, který má za úkol zjistit (pomocí časovače programu `ctest`), zda-li nedochází k markantnímu zpomalení některých regresních testů po aplikaci mého rozšíření. Výstupem skriptu je pěti-řádkový CSV soubor (jeden řádek pro jedno měření) s časy běhů jednotlivých testů (podrobné výsledky, pro něž je zde uvedeno shrnutí, jsou přiloženy na CD ve složce `stats`). Z těchto naměřených výsledků je zřejmé, že začlenění rozšíření *nezpůsobilo* výrazné zpomalení.

Prováděl jsem pomocí tohoto skriptu celkem dva typy měření – (a) napřed pro původní verzi analýzy živosti, a (b) posléze pro analýzu živosti, rozšířenou o moje úpravy. Podívejme se tedy na výsledky. Celkem 141 testů bylo přidáním mého rozšíření zrychleno a 157 naopak zpomaleno. Počty zrychlení a zpomalení je však velmi zavádějící – maximálního zrychlení

(16%) i zpomalení (−33%) totiž bylo dosaženo u testů, které trvaly celkem méně než 0.06s – tak vysoká procenta jsou tedy pravděpodobně způsobena chybu měření. Tyto hodnoty zde uvádím proto, abych prezentoval, že nebylo zaznamenané žádné zpomalení způsobené zavedením mého rozšíření.

To, že nebylo naměřeno výrazné zpomalení, také kopíruje již zmiňovanou teoretickou složitost algoritmu FICS, která by měla být pro běžné programy téměř lineární [10] (testy jsou obvkle kratšího rozsahu). Dalším důvodem je, že jakmile FICS narazí na na něco, s čím si neumí poradit, okamžitě končí a předává řízení programu následující analýze. I velmi rozsáhlý zdrojový kód tak může být hned z počátku přeskočen.

Pokrytí regresními testy. Přínosem práce je také nová množina celkem 29 regresních testů, z nichž některé jsou už i v oficiálním git repozitáři (ty, které se týkají algoritmu analýzy živých proměnných). Některé z testů vznikly jen z potřeby ujištění, zda-li algoritmy (ať už analýza živosti či PT analýza) fungují správně, jiné jsme psali, protože se v kódu objevila chyba a chtěli jsme mít jistotu, že případné znovu-zanesení téže chyby bude co nejrychleji odhaleno. Pro nahlédnutí na testy, o kterých jsem teď mluvil, se čtenář může podívat do složky přiložených zdrojových kódů Predatoru `cl/tests/data`. Hodnota regresních testů je v softwarových projektech obvykle vysoká. Proto věřím, že i zde pomůžou v budoucnu Code Listeneru včas odhalit nejeden problém.

Poins-to analýza je dostupná uživateli. V neposlední řadě lze jako přínos označit vůbec samotný výpočet PT analýzy pro analyzovaný program. Výsledky tohoto algoritmu (ve formě PT-grafů) má uživatel CL od teď kdykoliv k dispozici, což znamená, že můžou být v budoucnosti využity k nejrůznějším účelům.

7.2 Prostor ke zlepšení

Přesto, že jsem splnil zadání diplomové práce, jistě zůstává prostor pro další zlepšování do budoucna. Jako očekávaný další vývoj považuji postupné zvětšování současné množiny programů, pro něž algoritmus FICS v rámci CL proběhne úspěšně. Současná verze vylepšeného algoritmu například nepodporuje volání externích funkcí, nepodporuje operátor `pointer_plus` (tedy posun ukazatele o daný offset, což je velmi důležité například u vázaných listů linuxového jádra [14]). Dále není podporováno zmiňované použití ukazatele na funkci – se kterými je ale v [10] počítáno. Dále, existuje možnost zpřesnit výpočet PT relací o podporu prvků struktur, tak jak bylo naznačeno v kapitole 3.2. Samozřejmě, alternativně lze do CL implementovat paralelně podporu pro jiné PT algoritmy, které můžou být vhodné k jiným účelům, než právě FICS.

I stávající analýza živých proměnných nabízí prostor ke zlepšení – doposud si například neporadí s globálními proměnnými, nebo některými odkazovanými proměnnými, které jsem mým rozšířením nepokryl. Tady by mohl pomoci již diskutovaný rozvojem CFG.

Kapitola 8

Závěr

Zadáním diplomové práce bylo upravit existující algoritmus analýzy živých proměnných v rozhraní Code Listener. Výsledkem této úpravy mělo být zefektivnění tohoto algoritmu – konkrétně pro speciální situaci v analyzované funkci, kdy je možno zabít proměnnou, k níž je přístup pomocí ukazatele. Tento cíl se mi podařilo splnit. Rozšířil jsem rozhraní Code Listeneru o potřebnou points-to analýzu, s jejíž pomocí bylo možné zadanou situaci úspěšně detekovat. Úprava samotného algoritmu analýzy živých proměnných poté nemusela být ani příliš ivazivní, což bylo jedním z mých dílčích cílů.

Během práce na rozšíření se narazilo na několik problémů ve stávajícím algoritmu analýzy živých proměnných, které jsem opravil. Tyto chyby vedly na vznik platformy pro automatické testování tohoto algoritmu, což lze také považovat za přínos diplomové práce. V rámci diplomové práce jsem poté přidal několik regresních testů – ty budou v budoucnu pomáhat s detekcí chyb v tohoto algoritmu v případě jeho úprav.

Přínosem pro nástroj Predator je, že bylo dosaženo očekávaného zrychlení verifikace při existenci některých konstrukcí jazyka C. Toto zrychlení bylo téměř 5-násobné.

Zadání práce vedlo na přidání podpory grafu volání do rozhraní Code Listener. Uživatel Code Listeneru má nyní k dispozici graf toku řízení, jenž je doplněn o graf volání a také points-to graf, který je produktem ponts-to analýzy. To, že byly přidány zmiňované grafy, bude mít v budoucnu pozitivní vliv na praktickou použitelnost rozhraní.

Vzhledem k tomu, že konstrukce points-to grafu není úplně triviální, přidal jsem do práce mechanizmus pro jeho testování. Spolu s ním jsem přidal celkem 30 automatických testů, jež mají na starosti hlídání korektní konstrukce poinst-to grafu.

Literatura

[1] Andersen, L. O.: Program Analysis and Specialization for the C Programming Language. Technická zpráva, 1994.

[2] Choi, J.-D.; Burke, M.; Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, New York, NY, USA, 1993, ISBN 0-89791-560-7, s. 232–245.

[3] Comon, H.; Dauchet, M.; Gilleron, R.; aj.: Tree Automata Techniques and Applications. 2007, release October, 12th 2007.
URL http://www.grappa.univ-lille3.fr/tata

[4] Dudka, K.; Peringer, P.; Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proceedings of the 13th International Conference on Computer Aided Systems Theory*, The Universidad de Las Palmas de Gran Canaria, 2011, ISBN 978-84-693-9560-8, s. 328–329.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9822

[5] Dudka, K.; Peringer, P.; Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. Technická zpráva, 2011.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9723

[6] Dudka, K.; Peringer, P.; Vojnar, T.: Code Listener (stránka projektu). Naposledy kontrolováno: 6.1.2012.
URL
http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/

[7] Habermehl, P.; Holik, L.; Rogalewicz, A.; aj.: Forester, Tool for Verification of Programs with Pointers. Naposledy kontrolováno: 6.1.2012.
URL http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/

[8] Hind, M.; Burke, M.; Carini, P.; aj.: Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, July 1999.

[9] Hind, M.; Pioli, A.: Which pointer analysis should I use. In *In Proc. of the 2000 Int. Symp. on Soft. Testing and Analysis*, 2000, s. 113–123.

[10] Liang, D.; Harrold, M. J.: Efficient points-to analysis for whole-program analysis. *SIGSOFT Softw. Eng. Notes*, ročník 24, October 1999: s. 199–215, ISSN 0163-5948.
URL http://doi.acm.org/10.1145/318774.318943

[11] Marlowe, T. J.; Ryder, B. G.; Burke, M. G.: Defining Flow Sensitivity in Data Flow Problems. Technická zpráva, IBM T. J. Watson Research Center, 1995.

[12] Nielson, F.; Nielson, H. R.; Hankin, C.: *Principles of program analysis*. Springer, 1999, ISBN 9783540654100.
URL http://books.google.cz/books?id=RLjt0xSj8DcC

[13] Ramalingam, G.: The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, ročník 16, September 1994: s. 1467–1471, ISSN 0164-0925.
URL http://doi.acm.org/10.1145/186025.186041

[14] Shanmugasundaram, K.: Linux Kernel Linked List Explained. Naposledy kontrolováno: 23.5.2012.
URL http://isis.poly.edu/kulesh/stuff/src/klist/

[15] Shapiro, M.; Horwitz, S.: Fast and Accurate Flow-Insensitive Points-To Analysis. In *In Symposium on Principles of Programming Languages*, 1997, s. 1–14.

[16] Steensgaard, B.: Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, New York, NY, USA, 1996, ISBN 0-89791-769-3, s. 32–41.
URL ftp://ftp.research.microsoft.com/users/rusa/popl96.ps

[17] Vojnar, T.: Informace k předmětu Formální analýza a verifikace. poslední návštěva 2. ledna 2012.
URL http://www.fit.vutbr.cz/study/courses/FAV/public/.cs

[18] Vojnar, T.: Materiály k přednášce předmětu Formální analýza a verfikace: Úvod, základní pojmy. poslední návštěva 2. ledna 2012.
URL https://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-01.pdf

[19] Wikipedia.org: List of tools for static code analysis. číslo revize: 469456461.
URL http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

[20] Wikipedia.org: Control Flow Graph. číslo revize: 492353392.
URL http://en.wikipedia.org/wiki/Control_flow_graph

Příloha A

Přiložené zdrojové kódy

Příklad ladícího výstupu ve „verbose“ módu v CL

```
main():
                goto L1

        L1:
                %mF1228:i := rand()
                %mF1229:odd := 0
                %mF1230:even := 0
                %mF1983:i.0 := %mF1228:i
                %r1984 := (%mF1983:i.0 & 1)
                %r1048577 := (%r1984 == 0)
                if (%r1048577)
                        goto L2
                else
                        goto L3

        L2:
                %mF1230:even := (%mF1230:even + %mF1228:i)
                goto L4

        L3:
                %mF1229:odd := (%mF1229:odd + %mF1228:i)
                goto L4

        L4:
                %r1988 := %mF1230:even
                ret %r1988
```

Úplný zdrojový kód motivačního příkladu

```
 1 #include <verifier-builtins.h>
 2
 3 #include <stdlib.h>
 4
 5 struct node {
 6     struct node     *next;
 7     int             value;
 8 };
 9
10 struct list {
```

```
11      struct node     *slist;
12      struct list     *next;
13  };
14
15  static void inspect_before(struct list *shape)
16  {
17      // we should get a list of sub-lists of length exactly one
18      ___SL_ASSERT(shape);
19
20      for (; shape->next; shape = shape->next) {
21          ___SL_ASSERT(shape);
22          ___SL_ASSERT(shape->next);
23          ___SL_ASSERT(shape->slist);
24          ___SL_ASSERT(shape->slist->next == NULL);
25      }
26
27      // check the last node separately to make the exercising more fun
28      ___SL_ASSERT(shape);
29      ___SL_ASSERT(shape->next == NULL);
30      ___SL_ASSERT(shape->slist);
31      ___SL_ASSERT(shape->slist->next == NULL);
32  }
33
34  static void inspect_after(struct list *shape)
35  {
36      // we should get exactly one node at the top level and one nested list
37      ___SL_ASSERT(shape);
38      ___SL_ASSERT(shape->next == NULL);
39      ___SL_ASSERT(shape->slist != NULL);
40
41      // the nested list should be zero terminated (iterator back by one
            node)
42      struct node *pos;
43      for (pos = shape->slist; pos->next; pos = pos->next);
44      ___SL_ASSERT(!pos->next);
45  }
46
47  static void merge_single_node(struct node ***ppdst,
48                                struct node **psrc)
49  {
50      // pick up the current item and jump to the next one
51      struct node *node = *psrc;
52      *psrc = node->next;
53      node->next = NULL;
54
55      // insert the item into dst and move cursor
56      **ppdst = node;
57      *ppdst = &node->next;
58  }
59
60  static void merge_pair(struct node **pdst,
61                         struct node *sub1,
62                         struct node *sub2)
63  {
64      // merge two sorted sub-lists into one
65      while (sub1 || sub2) {
66          if (!sub2 || (sub1 && sub1->value < sub2->value))
67              merge_single_node(&pdst, &sub1);
68          else
```

```
69              merge_single_node(&pdst, &sub2);
70      }
71  }
72
73  static struct list* seq_sort_core(struct list *data)
74  {
75      struct list *dst = NULL;
76
77      while (data) {
78          struct list *next = data->next;
79          if (!next) {
80              // take any odd/even padding as it is
81              data->next = dst;
82              dst = data;
83              break;
84          }
85
86          // take the current sub-list and the next one and merge them into
                one
87          merge_pair(&data->slist, data->slist, next->slist);
88          data->next = dst;
89          dst = data;
90
91          // free the just processed sub-list and jump to the next pair
92          data = next->next;
93          free(next);
94      }
95
96      return dst;
97  }
98
99  void seq_sort(struct list **ptr_to_seq)
100 {
101     struct list *tmp = *ptr_to_seq;
102 #ifndef HAVE_ADVANCED_VAR_KILLER
103     *ptr_to_seq = NULL;
104 #endif
105
106     // do O(log N) iterations
107     while (tmp->next)
108         tmp = seq_sort_core(tmp);
109
110     *ptr_to_seq = tmp;
111 }
112
113 int main()
114 {
115     struct list *seq = NULL;
116     while (___sl_get_nondet_int()) {
117         struct node *node = malloc(sizeof *node);
118         if (!node)
119             abort();
120
121         node->next = NULL;
122         node->value = ___sl_get_nondet_int();
123
124         struct list *item = malloc(sizeof *item);
125         if (!item)
126             abort();
```

```
127
128         item->slist = node;
129         item->next = seq;
130         seq = item;
131     }
132
133     if (!seq)
134         return EXIT_SUCCESS;
135
136     inspect_before(seq);
137
138     seq_sort(&seq);
139
140     inspect_after(seq);
141
142     struct node *node = seq->slist;
143     free(seq);
144
145     while (node) {
146         struct node *snext = node->next;
147         free(node);
148         node = snext;
149     }
150
151     return EXIT_SUCCESS;
152 }
```

Skript pro měření zrychlení analýzy po úpravě v rámci práce

```
1  #!/bin/bash
2
3  # Tento skript musi byt spousten z korenoveho adresare
4  # git-repozitare projektu Predator.
5
6  echo "Running gcc without -DHAVE_ADVANCED_VAR_KILLER"
7
8  ./gcc-install/bin/gcc                          \
9          -fplugin=./sl_build/libsl.so           \
10         -Iinclude/predator-builtins            \
11         tests/predator-regre/test-0207.c
12
13 echo "Running gcc *with* -DHAVE_ADVANCED_VAR_KILLER"
14
15 ./gcc-install/bin/gcc                          \
16         -fplugin=./sl_build/libsl.so           \
17         -Iinclude/predator-builtins            \
18         -DHAVE_ADVANCED_VAR_KILLER             \
19         tests/predator-regre/test-0207.c
```