VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REKONSTRUKCE DATOVÝCH TYPŮ PŘI ZPĚTNÉM PŘEKLADU KÓDU

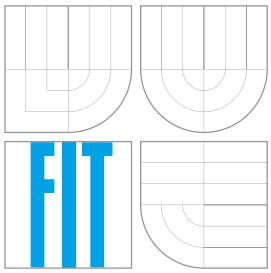
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETER MATULA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REKONSTRUKCE DATOVÝCH TYPŮ PŘI ZPĚTNÉM PŘEKLADU KÓDU

RECONSTRUCTION OF DATA TYPES FOR DECOMPILATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETER MATULA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAKUB KŘOUSTEK

BRNO 2013

Zadání diplomové práce/14787/2012/xmatul01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2012/2013

Zadání diplomové práce

Řešitel: **Matula Peter, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Rekonstrukce datových typů při zpětném překladu kódu**
Reconstruction of Data Types for Decompilation
Kategorie: Překladače

Pokyny:

1. Studujte problematiku zpětného inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se se zpětným překladačem projektu Lissom a jazykem LLVM IR, který je v překladači použit pro vnitřní reprezentaci kódu. Zkoumejte kvalitu produkovaného kódu.
3. Studujte existující metody zpětného překladu zaměřené na rekonstrukci jednoduchých i složitějších datových typů (např. struktury a pole).
4. Navrhněte metody, které umožní efektivní zpětný překlad programů obsahujících datové typy. Řešení musí být nezávislé na konkrétní architektuře a překladači.
5. Po konzultacích s vedoucím metody navržené v předchozím bodě implementujte.
6. Vytvořené řešení důkladně otestujte sadou minimálně dvaceti testů, zohledněte různé procesorové architektury a překladače. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- Křoustek, J.: *Analýza a převod kódů do vyššího programovacího jazyka*, diplomová práce, FIT VUT v Brně, 2009.
- Dolgova, E. N., Chernov, A. V.: *Automatic Reconstruction of Data Types in the Decompilation Problem*, Programming and Computing Software 35, 2009.
- Troshina, K., Derevenets, Y., Chernov, A.: *Reconstruction of Composite Types for Decompilation*, SCAM, 2010.
- E. Eilam: Reversing: *Secrets of Reverse Engineering*, Wiley, 2005, ISBN 978-0764574818.
- Interní dokumentace projektu Lissom.

Při obhajobě semestrální části diplomového projektu je požadováno:

- První tři body zadání, částečně bod čtvrtý.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí: **Křoustek Jakub, Ing.**, UIFS FIT VUT
Datum zadání: 17. září 2012
Datum odevzdání: 22. května 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práce se zabývá popisem metod rekonstrukce datových typů při zpětném překladu. Je definován pojem zpětného inženýrství a představen zpětný překladač vyvíjen v rámci projektu Lissom, pro potřeby kterého tato práce vznikla. Jsou představeny stávající metody rekonstrukce jednoduchých i složených datových typů a podrobně vysvětleny přístupy založené na analýze toku dat a analýze ofsetů paměťových operací. Jádrem práce je návrh nové techniky rekonstrukce jednoduchých a složených datových typů, vhodné pro nasazení v prostředí rekonfigurovatelného zpětného překladače projektu Lissom. Jsou vysvětleny základní principy nového návrhu, jeho implementace a souvisejících změn ve vyvíjeném zpětném překladači a jeho medzikódě. Výsledné řešení je podrobeno řadě testů. V závěru jsou diskutovány dosažené výsledky, nedostatky a směr další práce.

Abstract

This document describes methods for a reconstruction of data types in the decompilation problem. It defines the concept of reverse engineering and introduces decompiler developed by the Lissom project. It presents existing methods of reconstruction of the simple and complex data types, and explains in detail approaches based on data-flow analysis and analysis of the memory operation offsets. The core of this thesis is the design of a new technique of reconstructing simple and complex data types, suitable for deployment in a retargetable decompiler environment of the Lissom project. Basic principles of the new technique, its implementation and the related changes in decompiler and intermediate language are described. The solution is tested and the conclusion discusses the achievements, shortcomings and direction of the further work.

Klíčová slova

Zpětné inženýrství, zpětný překladač, Lissom, LLVM IR, analýza datových typů

Keywords

Reverse engeneering, decompiler, Lissom, LLVM IR, data type analysis

Citace

Peter Matula: Rekonstrukce datových typů při zpětném překladu kódu, diplomová práce, Brno, FIT VUT v Brně, 2013

Rekonstrukce datových typů při zpětném překladu kódu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jakuba Křoustka. Další informace mi poskytl Ing. Lukáš Ďurfina.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Peter Matula
22. května 2013

Poděkování

Na tomto místě bych rád poděkoval mému vedoucímu Ing. Jakubu Křoustkovi za odborné vedení, poskytnuté rady a možnosť zapojit se do projektu Lissom. Dále bych chtěl poděkovat členům projektu za mnoho užitečných podnětů.

Obsah

Zoznam obrázkov

Zoznam skratiek

ADL	Architecture Description Language – jazyk pre popis architektúr
ARM	Architektúra RISC procesorov
ARM Thumb	Rozširujúca sada inštrukcií pre platformu ARM
BFD	Binary File Descriptor library
BIL	Binary Analysis Language
BIR	Back-end IR – medzikód zadnej časti spätného prekladača
COFF	Common Object File Format – formát objektových súborov operačných systémov rodiny UNIX
CPU	Central Processing Unit
DEX	Dalvik Executable – formát objektových súborov operačného systému Android
DNA	Deoxyribonukleová Kyselina
DU reťaze	Definition-Use reťaze
DWARF	Na architektúre a objektovom súbore nezávislý formát ladiacich informácií
E32Image	Formát objektových súborov operačného systému Symbian
ELF	Executable and Linkable Format – formát objektových súborov operačných systémov rodiny UNIX
GPL	GNU General Public License
IR	Intermediate Representation
ISAC	Instruction Set Architecture C
IT	Informačné Technológie
LLVM IR	LLVM Intermediate Representation
Mach-O	Mach object – formát objektových súborov operačného systému Mac OS X

MIPS	Architektúra RISC procesorov
MMX	Sada multimediálnych inštrukcií
PDB	Program Database – formát ladiacich informácií spoločnosti Microsoft
PE	Portable Executable – formát objektových súborov operačných systémov spoločnosti Microsoft Windows
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Language
RTTI	Run-Time Type Information
SSA	Static Single Assignment – vlastnosť medzikódu
SSL	Syntax/Semantic Language
TIE	Type Inference on Executables
x86	Rodina tridsaťdva bivových procesorov spoločnosti Intel
XML	Extensible Markup Language

Kapitola 1

Úvod

V súčasnej dobe sme svedkami masívneho rozšírenia osobných elektronických zariadení. Ľudia už nevlastnia len osobný počítač, ale celú radu inteligentných prístrojov od chytrých telefónov, cez tablety, prehrávače hudby až po moderné televízie a herné konzoly. S narastajúcimi možnosťami a funkciami týchto zariadení narastá aj ich úloha v našich životoch. Tento trend je živnou pôdou pre útočníkov, ktorým sa otvára množstvo nových ciest ako preniknúť k našim osobným údajom. Zároveň je výzvou pre antivírové spoločnosti, ktoré musia byť schopné všetky prístroje ochrániť a efektívne reagovať na nové hrozby.

Kritickým aspektom reakcie na nový škodlivý kód je čas potrebný na aktualizáciu antivírovej aplikácie tak, aby dokázala hrozbe zabrániť. Celý proces sa skladá z niekoľkých krokov, z ktorých je často časovo najnáročnejší rozbor škodlivého programu. Obvykle sa vykonáva pomocou takzvaných disassemblerov prekladajúcich binárne súbory do jazyka symbolických inštrukcií. Výstupom je nízkoúrovňový kód závislý na danej architektúre procesoru a inštrukčnej sade. Ďalšou možnosťou je použitie spätného prekladača – dekompilátoru[1]. Ten dokáže transformovať spustiteľný súbor na vysokoúrovňový kód univerzálny pre všetky architektúry a pre človeka čitateľnejší ako výstup spätného assembleru. Pretože sa ale pri preklade nenávratne stratí veľké množstvo informácií obsiahnutých v zdrojovom súbore, nie je možné dokonale obnoviť pôvodný program. Spätný prekladač sa tak skladá zo série analýz, snažiacich sa čo najlepšie rekonštruovať jednotlivé aspekty aplikácie.

Táto práca vznikla v rámci projektu Lissom, prebiehajúcom na Fakulte informačných technológií Vysokého učení technického v Brně. Jedným z jeho cieľov je v spolupráci so spoločnosťou AVG Technologies CZ, s.r.o. vývoj rekonfigurovateľného spätného prekladača, nezávislého na architektúre, formáte binárneho súboru alebo operačnom systéme. Dekompilátor môže byť využitý pri rozbore škodlivého kódu (anglicky *malware*), verifikácií programov a prekladačov, migrácií kódu a ďalších podobných činnostiach. Cieľom práce je návrh a implementácia analýzy jednoduchých a zložených datových typov, ktorá priradí každému objektu jeho typ, odvodený od spôsobu akým sa s ním v programe pracuje. Zdokonalenie a ďalší rozvoj analýzy je plánovaný v naväzujúcom doktorskom štúdiu.

Práca je organizovaná nasledovne: 2. kapitola predstavuje problematiku reverzného inžinierstva ako analýzy človekom vytvoreného diela. Bližšia pozornosť je venovaná jeho uplatneniu v oblasti informačných technológií a zoznámeniu čitateľa s niektorými existujúcimi spätnými prekladačmi.

[1] Spisovne spätný prekladač (anglicky *decompiler*), pojmy dekompilátor a dekompilácia ale budú v texte používané ako synonymá.

Kapitola 3 ďalej predstavuje spätný prekladač vyvíjaný projektom Lissom, jeho štruktúru, používaný medzikód a úlohu, ktorú zohráva analýza datových typov. Zvýšená pozornosť je venovaná popisu typového systému LLVM medzikódu.

Prvá časť kapitoly 4 je venovaná zdrojom typových informácií v strojovom kóde, definícií jednoduchých datových typov a ich rekonštrukcii. Sú popísané existujúce spôsoby obnovy a podrobne vysvetlená technika založená na analýze toku dat, ktorej modifikácia je implementovaná v spätnom prekladači projektu Lissom. Druhá časť kapitoly sa ďalej zaoberá identifikáciou zložených datových typov. Opäť je predstavených niekoľko existujúcich techník a podrobne vysvetlená tá z nich, ktorá slúži ako základ pre implementovanú analýzu.

Kapitola 5 popisuje návrh analýzy rekonštrukcie jednoduchých aj zložených datových typov. Navrhnuté techniky sú založené na už existujúcich prístupoch predstavených v predchádzajúcej kapitole. Sú však upravené pre použitie v ramci existujúcej infraštruktúry spätného prekladača Lissom.

Nasledujúca 6. kapitola v krátkosti popisuje základné princípy implementácie navrhnutých techník. Okrem samotného jadra analýzy sú vysvetlené aj modifikácie vnútorného medzikódu a spôsob generovania typov do výsledného LLVM kódu.

Kapitola 7 je venovaná experimentom s vytvoreným riešením. Sú tu na príkladoch zobrazené úspešné použitia analýzy a diskutované jej nedostatky.

Záverečná kapitola 8 sumarizuje predstavené technológie, hodnotí dosiahnuté výsledky a vymedzuje ďalší vývoj a možné vylepšenia.

Kapitola 2

Spätné inžinierstvo

„Spätné inžinierstvo je proces získania znalostí o skúmanom objekte.“ [15]

Koncept bol ľudstvu známy dlho pred nástupom počítačov a pôvod sa pravdepodobne datuje do obdobia priemyselnej revolúcie. Spätné inžinierstvo je veľmi podobné vedeckému výskumu, od ktorého sa odlišuje tým, že skúma objekty vyrobené človekom. Cieľom je získanie nedostupnych informácií, vlastností, vnútorných vzťahov, architektúry, designu, dokumentácie a ďalších vedomostí o objekte. Výsledkom sú teda informácie umožňujúce rekonštrukciu skúmaného objektu.

Pôvodne sa jednalo o fyzické skúmanie, rozoberanie produktu a následná výroba podobného alebo vylepšeného diela. Po nástupe modernej elektroniky si reverzné inžinierstvo našlo uplatnenie aj na poli softwarových produktov. Cieľom je v tomto prípade pochopiť fungovanie aplikácie alebo jej časti – funkcie, algoritmu.

Kapitole je spracovaná na základe [15, 20].

2.1 Spätné inžinierstvo mimo informačné technológie

Aj napriek tomu, že je spätné inžinierstvo v súčasnosti spojované hlavne s oblasťou informačných technológií, môžeme nájsť mnoho ďalších priemyselných a vedeckých odborov v ktorých nachádza uplatnenie.

Jedným z tradičných odvetví často využívajúcim (zneužívajúcim) reverzné inžinierstvo je vojenstvo. Jedna strana sa snaží dohnať technologickú vyspelosť protivníka pomocou podrobného skúmania jeho technológií. Tie sú získané špionážou, kúpou alebo zmocnením sa priamo na bojisku a následne podrobené štúdiu v laboratórnych podmienkach. Príkladom môže byť konštrukcia ruského bombardéru Tupolev Tu-4 na základe amerického vzoru B-29. Ochranou pred podobnými praktikami je napríklad zanesenie fatálnych chýb do výrobných plánov, ktoré pri pokuse o zostrojenie objektu nepovolanou osobou spôsobia nefunkčnosť alebo až katastrofické zlyhanie výrobku.

Príkladom kladného využitia reverzného inžinierstva môže byť genetický výskum. Genetika je biologická disciplína zaoberajúca sa génmi, dedičnosťou a premenlivosťou. Základnou jednotkou dedičnosti je gén – úsek DNA (*deoxyribonukleová kyselina*), ktorý má schopnosť vytvoriť svoju identickú kópiu alebo preniesť svoju informáciu do ďalšej generácie. Z mnohých podoborov genetiky využíva spätné inžinierstvo hlavne genetické inžinierstvo, ktoré prenáša, modifikuje alebo vkladá nové úseky DNA za účelom vylepšenia nasledujúcej ge-

nerácie. Príkladom môžu byť geneticky upravené potraviny so zvýšeným tempom rastu, odolnosti proti chorobám alebo pesticídom.

Ďalšími oblasťami využitia je získavanie zložení liečiv vo farmaceutickom priemysle ale tiež modelov v architektúre alebo konštrukčných plánov v letectve a automobilovom priemysle.

2.2 Spätné inžinierstvo v informačných technológiach

> *„Pojem spätného inžinierstva chápeme v informatike ako celú radu disciplín, vedúcich k úplnému pochopeniu vlastností a väzieb skúmaného objektu a ich pretvoreniu do inej formy.“* [20]

Typický postup je zobrazený na obrázku 2.1 a môžeme povedať, že sa jedná o proces inverzný k softwarovému inžinierstvu. V oboch prípadoch vykonávame rovnaké kroky ale v chronologicky opačnom poradí. Reverzné inžinierstvo je v odbore informačných technológií uplatniteľné v celej rade odvetví. Niektoré z typických využití sú:

- *Bezpečnosť* – výrobcovia antivírových aplikácií analyzujú škodlivé programy za účelom pochopenia ich princípu, možného poškodenia a spôsobov ako sa proti nim brániť. Na druhej strane ich tvorcovia používajú reverzné techniky k nájdeniu zraniteľností, ktorých by mohli zneužiť k prevzatiu kontroly nad systémom.

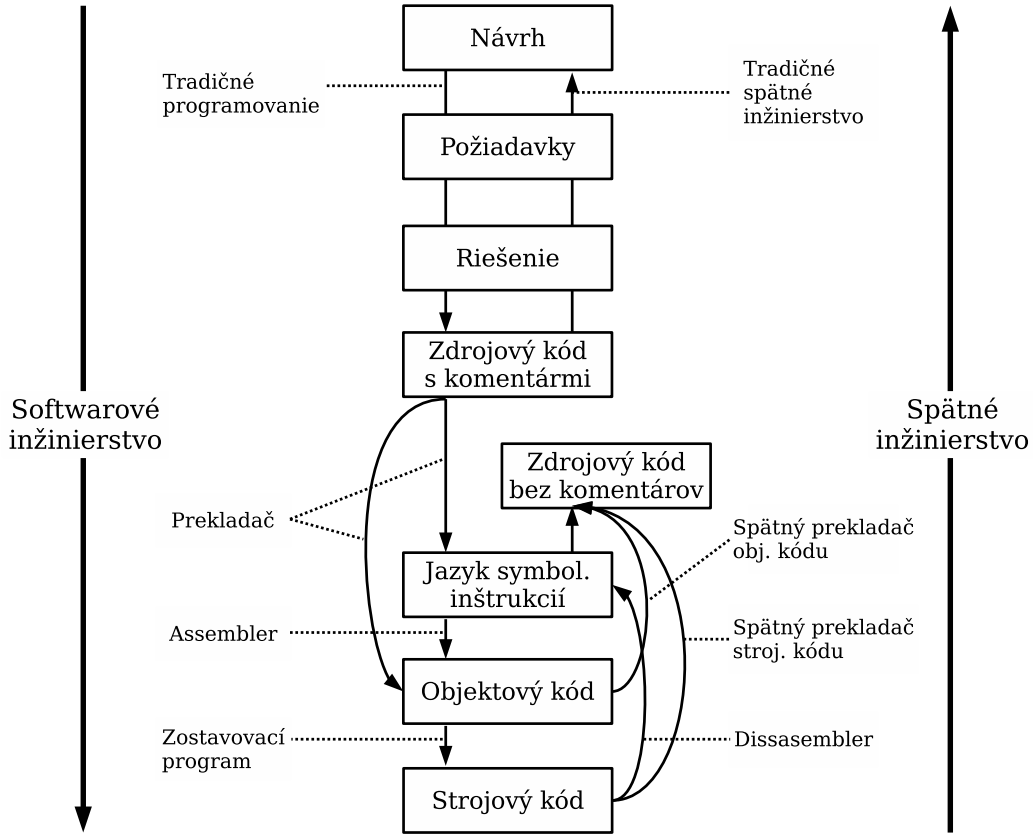
- *Kryptografia* – kryptografické algoritmy môžu byť rozdelené do dvoch skupín: utajované algoritmy a algoritmy založené na kľúčoch. V prvom prípade je kľúčom k šifrovaniu/dešifrovaniu samotný algoritmus, ktorý nie je známy a jeho prezradenie by viedlo k prelomeniu šifry. Reverzné inžinierstvo môže byť použité presne za týmto účelom – pochopiť činnosť algoritmu. V druhom prípade je algoritmus známy a spätná analýza sa skôr hodí na získanie implementačných detailov.

- *Crackovanie* – je snaha útočníka prelomiť a deaktivovať ochranu proprietárneho softwaru tak, aby mohol byť používaný bez zakúpenia platnej licencie. Programátori môžu chrániť svoje programy pomocou rôznych techník ako napríklad obfuskácia, šifrovanie alebo automodifikačný kód.

- *Proprietárny software* – v mnohých prípadoch je programátor nútený používať produkty, ktorých zdrojové kódy alebo dokonca dokumentácia nie sú dostupné. Spätné inžinierstvo môže v tejto situácii slúžiť pre overenie vlastností deklarovaných výrobcom, správnosti implementácie, bezpečnosti alebo pre hľadanie odpovede na otázky nezodpovedané v špecifikácii.

- *Architektúry integrovaných obvodov* – tak ako v prípade využitia reverzného inžinierstva vo vojenstve, aj v tu sa jedná o dobehnutie technologickej vyspelosti protivníka – konkurenta. Jedna strana skúma vlastnosti cudzieho výrobku a zistené informácie využije pri tvorbe produktu vlastného. Príkladom z oblasti výroby procesorov môže byť prevzatie technologických riešení spločnosti Intel ostatnými výrobcami CPU.

- *Testovanie prekladačov* – overenie, že prekladač funguje korektne a naozaj vygeneruje očakávaný kód.



Obrázok 2.1: Spätné inžinierstvo v IT. Obrázok prevzatý z [20] a upravený.

2.3 Právne hľadisko

V súvislosti s predstavenými využitiami reverzného inžinierstva v IT je nutné sa pozastaviť pri ich legálnosti. V mnohých prípadoch je spätné inžinierstvo využívané k plagiátorstvu – napodobeniu diela niekoho iného a vydávanie výsledku za svoj vlastný výtvor. Rovnako protiprávne je aj prelamovanie ochrany softwarových produktov (anglicky *cracking*) a s ním spojené počítačové pirátstvo – zdieľanie programov bez ochrany na internete.

Týmto činnostiam zabraňuje autorský zákon a patentovanie, ktorých cieľom je ochrániť autorovo dielo pred modifikáciou a krádežou. V českej republike sa jedná o zákon č. 121/2000 Sb. definujúci dielo ako vnímatelné dielo, ktoré je výsledkom jedinečnej tvorčej činnosti autora. Patenty vydávané patentovými úradmi slúžia k ochrane vynálezov a zaisteniu exkluzivity ich priemyselného využitia. Na rozdiel od autorského diela nevzniká ochrana automaticky, ale je nutné podať žiadosť, o ktorej oprávnenosti rozhodne patentová komisia. Z hľadiska informačných technológií je dôležité, že sa v Česku patentom nemôžu stať počítačové programy ale len technológie v nich použité.

Záverom teda môžeme povedať, že spätné inžinierstvo samo o sebe nie je protizákonne. Nelegálnym je až jeho zneužitie za účelom modifikácie programu alebo odhaleniu myšlienky alebo technológie patriacej pôvodnému autorovi.

2.4 Nástroje

Pri rozbore softwaru sa typicky používajú nástroje popísané v tejto podkapitole. Môžu analyzovať kód dynamicky – program spustia a sledujú jeho správanie, alebo staticky – transformáciou do inej formy reprezentácie.

2.4.1 Ladiaci nástroj

Ladiaci nástroj (anglicky *debugger*) je program pre identifikáciu chýb v aplikáciach. Základnými funkciami je nastavovanie breakpointov (zastavenie programu za účelom preskúmania určitého stavu), krokovanie programu po inštrukciách a sledovanie stavu výpočtu – aktuálne hodnoty registrov, premenných, zásobníku atď. Pokročilé nástroje umožnujú aj zmenu hodnôt týchto objektov priamo za behu.

Ladiace programy môžu byť zneužité aj pri crackingu pre pochopenie činnosti skúmaného programu a identifikáciu kódu zodpovedného za jeho ochranu. Ďalším možným využitím je verifikácia alebo analýza výkonu aplikácií.

Debuggery sú často zabudované aj priamo v integrovaných vývojových prostrediach, poskytujúcich nadstavbu nad natívnymi nástrojmi ako je tomu v prípade Eclipse CDT a GDB, alebo implementujúcich vlastné riešenia ako Microsoft Visual Studio Debugger.

2.4.2 Spätný assembler

Spätný prekladač jazyka symbolických inštrukcií (anglicky *disassembler*) je program transformujúci strojový kód na inštrukcie assembleru. Vykonáva teda opačnú činnosť ako prekladač symbolických inštrukcií (anglicky *assembler*). Pretože je pri preklade z vysokoúrovňového jazyka stratených množstvo informácií pre procesor nepotrebných, ako napríklad mená premenných, funkcií alebo ich datové typy, je výsledný kód pre človeka ťažko čitateľný.

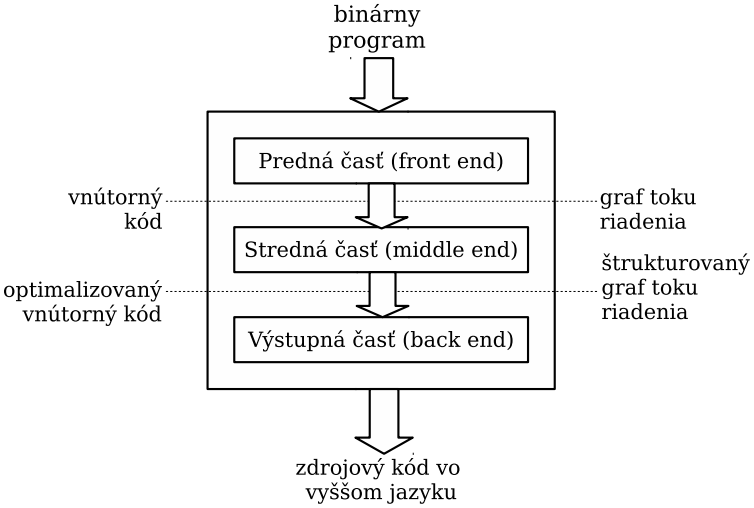
Pretože disassembler pracuje s architektonicky závislým strojovým kódom a inštrukčnou sadou, je obvykle jednoúčelový – nedokáže pracovať s binárnymi súbormi určenými pre rôzne procesory. Ďalšou výzvou je oddelenie kódu od dat, čo je problém ekvivalentný zastaveniu Turingovho stroja. Algoritmy tak môžu prinajlepšom vygenerovať približnú aproximáciu originálneho programu a snahou je zvýšiť ich kvalitu a úspešnosť.

Najznámejším spätným assemblerom je komerčný *IDA Pro* [4], podporujúci radu procesorov a binárnych formátov. Skladá sa z mnohých modulov a jeho súčasťou je napríklad aj ladiaci nástroj alebo spätný prekladač.

2.4.3 Spätný prekladač

> *„Dekompilácia je programová transformácia, ktorá má za cieľ zo vstupného programu vytvoriť kód vo vyššom programovacom jazyku.“* [20]

Jedná sa o opačný proces ku kompilácií vykonávaný nástrojom zvaným spätný prekladač (dekompilátor). Proces (obrázok 2.2) začne podobne ako činnosť disassembleru – strojový kód v binárnom súbore je dekódovaný a je z neho vytvorená interná reprezentácia (IR – *Intermediate Representation*). Tu sa ale skutočná práca len začína. Nad IR je vykonaných množstvo analýz s cieľom získať, alebo odvodiť čo najviac vysokoúrovňových informácií. Výstupom je kód vo vyššom programovacom jazyku, ktorý by mal byť v ideálnom prípade čitateľnejší ako jazyk symbolických inštrukcií. Jedná sa ale o extrémne náročnú úlohu, ktorá zatiaľ nebola zvládnutá na dostatočnej úrovni tak, aby použitie spätných prekladačov hralo hlavnú úlohu pri reverznom inžinierstve softwaru.



Obrázok 2.2: Obecná štruktúra spätného prekladača.
Obrázok prevzatý z [20] a upravený.

Aj napriek tomu ale existuje niekoľko reálne použiteľných spätných prekladačov. Prehľad ich klúčových vlastností je zobrazený v tabuľke 2.3 a nasleduje ich krátky popis spracovaný na základe [30]:

- *dcc* [2] – spätný prekladač bol vyvinutý Cristinou Cifuentes ako súčasť jej doktorskej tézy [13] a je distrubuovaný pod GLP licenciou. Štruktúrou pripomína klasický prekladač, kde predná časť číta strojovo závislé inštrukcie a prekladá ich do vnútornej reprezentácie. Stredná časť vykonáva jadro dekompilácie – analýzy toku dat a toku riadenia. Zadná časť generuje výsledný vysokoúrovňový kód v jazyku C.

- *Boomerang* [1] – je open source projekt pôvodne vytvorený Mike Van Emmerikom. Používa niekoľko algoritmov, ktoré vyhľadávajú vzory v strojovom kóde a prekladajú ich do ekvivalentov v jazyku C. Boomerang podporuje niekoľko architektúr a objektových formátov. Nevýhodou je nerozpoznávanie staticky pripojeného kódu, čo môže spôsobiť preklad veľkého množstva funkcií zo systémových knižníc. Jedná sa pravdepodobne o prvý pokus o rekonfigurovateľný spätný preklad. Pre popis architektúr je používaný jazyk SLED poskytujúci popis syntaxe inštrukcií a spôsobu ich kódovania. Neobsahuje ale ich sémantiku a tak musí byť použitý spolu s RTL (anglicky *Register Transfer Language*) a jazykom SSL. Konečný výsledok sa ukázal ako pomalý, nevhodný pre zložitejšie architektúry a nebol ani plne rekonfigurovateľný. Niektoré, na architektúre zavislé časti, sa museli programovať manuálne.

- *REC Studio* [9] – Reverse Engineering Compiler je freeware ale nie open source interaktívny spätný prekladač. Podporuje niekoľko vstupných formátov a vytvára výstup podobný jazyku C. Pre spresnenie analýzy dokáže využiť prípadné DWARF alebo PDB ladiace informácie.

- *Hex-Rays* [4] – Jedná sa o zásuvný modul komerčného IDA disassembleru. Obsahuje dva manuálne naprogramované spätné prekladače pre architektúry x86 a ARM, pričom autori v súčasnosti pracujú aj na podpore platformy x64. Samotný spätný preklad je veľmi rýchly a orientovaný na detekciu funkcií. Zároveň ale podporuje väčšinu

bežných mechanizmov ako rozlišovanie cyklov, tvorbu zložených podmienok, využívanie ladiacich informácií atď. Súčasťou programu je aj interaktívne grafické uživateľské rozhranie napomáhajúce pochopeniu procesu a lepšej čitateľnosti výsledku.

- *Decompile-it.com* [3] – Projekt je úzko prepojený s nástrojom Valgrind. Valgrind je framework pre vytváranie nástrojov pre dynamickú analýzu ako napríklad detektoru pamäťových alebo vláknových chýb. Decompile-it.com je v súčasnosti dostupný len ako webové rozhranie pre binárne súbory typu x86/Linux. Podľa autora sa nástroj dokáže vysporiadať s rekonštrukciou zložených datových typov, detekciou a transformáciou idiomov, konštrukciou switch, inline funkciami a rozbalenými cyklami.

- *SmartDec* [10] – pôvodne *TyDec* je distribuovaný ako demo aplikácia pod neznámou licenciou. Spätný preklad je zameraný na rekonštrukciu binárnych súborov vytvorených C++ prekladačmi – podporuje špecifické konštrukcie ako virtuálne funkcie, triedy a ich hierarchie, vzťahy dedičnosti, typy členov tried, výnimky a ďalšie.

Aj keď v tabuľke vidíme u väčšiny spätných prekladačov podporu niekoľkých architektúr, nejedná sa o skutočné rekonfigurovateľné riešenia. Ako už bolo spomenuté, o rekonfigurovateľnosť sa snažil len nástroj Boomerang ale konečný výsledok sa neosvedčil. Problematika rekonfigurovateľného spätného prekladu teda nebola zatiaľ úspešne vyriešená a nasadená do praxe.

Spätné prekladače	**dcc**	**Boomerang**	**REC Studio**	**Hex-Rays**	**SmartDec**
Podporované architekury	x86	x86 Sparc PowerPC	x86 Sparc MIPS	x86 ARM	x86
Podporované objektové formáty	MS-DOS formát	PE ELF	PE ELF Mach-O	PE ELF	
Vstup	binárny	binárny	binárny	binárny	asm
Cieľový jazyk	C	C	C	C	C/C++
Distrubúcia	zdroje	zdroje	binárna	binárna	binárna
Licencia	GPL	BFD/GPL	freeware	komerčná	neznáma
Vnútorná reprezentácia	dva typy	jeden typ	**?**	**?**	**?**
DWARF podpora	✗	✗	✓	✓	✗
PDB podpora	✗	✗	čiastočne	✓	✗
Interaktívne rozhranie	✗	✓	✓	✓	✗
Detekuje staticky pripojený kód	✓	✗	✓	✓	**?**

Obrázok 2.3: Porovnanie spätných prekladačov. Prevzaté z [30].

Kapitola 3

Spätný prekladač projektu Lissom

Kapitola predstavuje projekt Lissom, nástroje v rámci neho vyvíjané, použité technológie a podrobne pojednáva o spätnom prekladači, jeho nedostatkoch a úlohe, ktorú táto práca rieši.

3.1 Projekt Lissom

Projekt *Lissom* [5] sa zaoberá vývojom univerzálnych nástrojov pre spoločný návrh hardwaru a softwaru (anglicky *hardware-software codesign*). Cieľom je vytvoriť systém, ktorý na základe modelu procesoru v jazyku pre popis architektúr automatizovane vygeneruje nástroje potrebné pre efektívnu tvorbu programov pre tento procesor.

Pre popis architektúry procesoru bol navrhnutý jazyk ISAC, z ktorého je vygenerovaný XML dokument popisujúci CPU. Z dokumentu sa ďalej automaticky vytvoria assembler, prekladač, ladiaci nástroj, simulátor a disassembler. Takýto prístup výrazne uľahčí a urýchli vývoj nových architektúr. V rámci projektu vzniká aj rekonfigurovateľný spätný prekladač, v súčasnosti podporujúci architektúry MIPS, ARM a Intel x86. Jeho popisu bude venovaná táto kapitola.

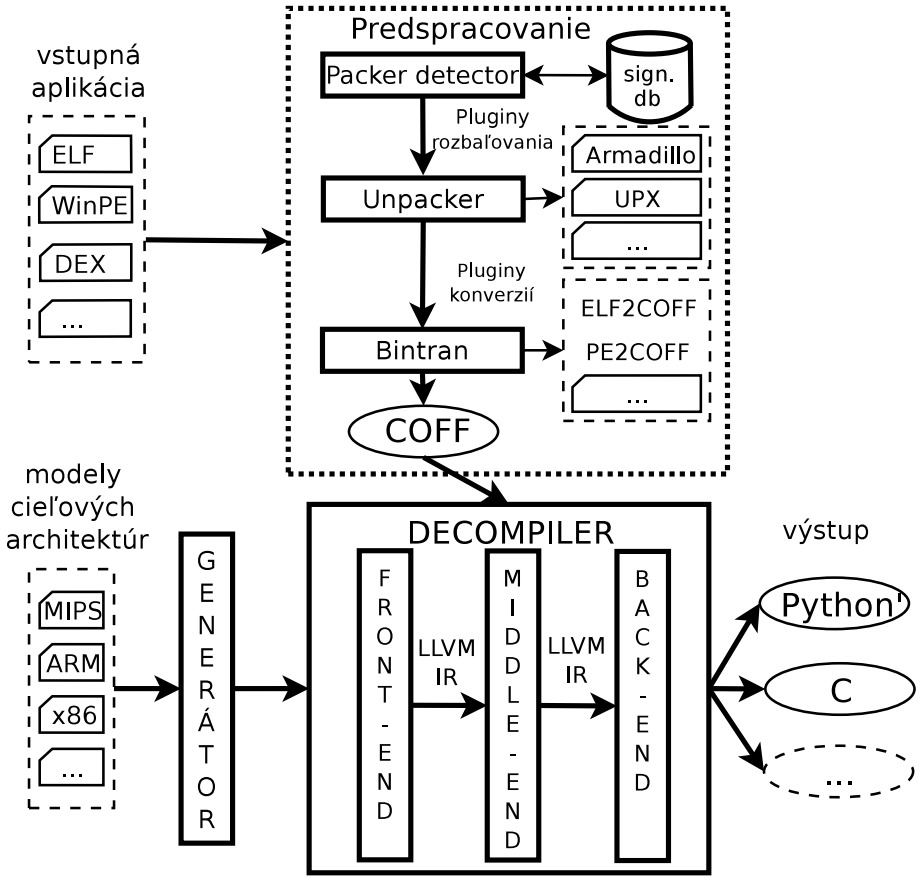
3.2 Infraštruktúra spätného prekladača

Na obrázku 3.1 je zobrazená infraštruktúra dekompilátoru projektu Lissom. Okrem jeho jadra zloženého z prednej, optimalizačnej a zadnej časti sa skladá aj predspracovania vstupného objektového súboru a generátoru pracujúceho s modelom procesoru v jazyku ISAC. Popisom týchto logických častí sú venované nasledujúce podkapitoly.

3.2.1 Predspracovanie

Vlastnosti vstupných súborov sú závislé na cieľovej architektúre, použitom prekladači a objektovom formáte. Vstupy tiež často bývajú komprimované alebo chránené proti akejkoľvek analýze. Z týchto dôvodov je pred samotným spätným prekladom nutné vykonať ich predspracovanie, ktorého cieľom je eliminovať väčšinu špecifických rozdielov.

Predspracovanie začína detekciou objektového formátu súboru. Sú podporované všetky bežné formáty ako Unix ELF, Windows PE, Mac OS X Mach-O, Symbian E32Image alebo Android DEX. Z hlavičky je získaný identifikátor cieľovej architektúry, ktorý bude použitý neskôr.



Obrázok 3.1: Štruktúra spätného prekladača projektu Lissom. Obrázok prevzatý z [31].

Ďalším krokom je detekcia použitého prekladača pomocou jeho podpisu obsiahnutom v niekoľkých počiatočných inštrukciách. Tie sú obvykle špecifické pre každý kompilátor a môžu byť použité ako jeho odtlačok. Analýza musí mať samozrejme k dispozícii databázu známych podpisov. Tie sú zároveň zamerané aj na detekciu prípadnej komprimácie.

Ak bol súbor skutočne zabalený, je nutné ho extrahovať. Tento proces je vykonaný ďalším nástrojom projektu, využívajúcim okrem bežnej dekomprimačnej knižnice aj niekoľko zásuvných modulov implementujúcich rozbalenie špecifických prípadov. Po jeho skončení je vstup opätovne analyzovaný na verziu prekladača. V ojedinelých prípadoch je nutné sekundárne rozbalenie, ale väčšinou je detekcia možná už po prvek dekomprimácii.

Pretože je binárny kód uložený v objektovom súbore závislom na architektúre, je použitý ďalší nástroj nazvaný Bintran (*binary translator*) [23]. Jeho úlohou je obojsmerná konverzia medzi obecne používanými formátmi objektových súborov a interným formátom projektu Lissom označeným ako COFF. Aplikácia je opäť navrhnutá ako systém zásuvných modulov, čo umožnuje jednoduché rozširovanie funkcionality o ďalšie konverzie. V súčasnosti existujú pluginy podporujúce päť najrozšírenejčích formátov vymenovaných na začiatku podkapitoly.

Pomocou informácie o cieľovej architektúre z hlavičky súboru a odpovedajúceho modelu CPU je v ďalšom kroku automaticky vygenerovaný inštrukčný dekóder. Jedná sa o prvú časť prednej časti spätného prekladača, ktorej úlohou je preklad strojových inštrukcií na sémantický popis ich správania. Takto vytvorený súbor môže byť následne spracovaný rekonfigurovateľným dekompilátorom.

3.2.2 Jazyk ISAC

ISAC (*Instruction Set Architecture C*) [22] je jazyk pre popis architektúr procesorov (anglicky *Architecture description language*) vyvinutý projektom Lissom.

Samotný model sa skladá z niekoľkých blokov definujúcich konkrétne aspekty procesoru. Zdroje – registre, pamäti a ich hierarchia sú popísané blokom `RESOURCES`. Blok `OPERATION` obsahuje definície kódovania a správania jednotlivých inštrukcií CPU. Krátky príklad popisu operácie sčítania je zobrazený na zdrojovom kóde 3.2.

```
RESOURCES {
// HW zdroje
      PC REGISTER bit[32] pc;
      REGISTER bit[32] gpregs[32];
      RAM bit[32] memory { ENDIAN(LITTLE); FLAGS(R,W,X); };
}

OPERATION reg REPRESENTS gpregs
      { /* Textový a binárny popis registrov */ }

OPERATION op_add {
// Popis inštrukcie
      INSTANCE gpregs ALIAS { rd, rs, rt };
      ASSEMBLER { "ADD" rd "," rs "," rt };
      CODING { 0b0001 rd rs rt };
// Správanie inštrukcie
      BEHAVIOR { gpregs[rd] = gpregs[rs] + gpregs[rt]; };
}
```

Obrázok 3.2: Ukážka popisu operácie sčítania v jazyku ISAC. Príklad prevzatý z [31]

3.2.3 LLVM IR

Architekturálne zavislé inštrukcie dekompilovaného programu sú pomocou ISAC modelu preložené na sekvenciu mikroinštrukcií LLVM IR [7, 25] kódu, slúžiaceho ako ich vnútorná reprezentácia počas celého procesu spätného prekladu. Jedná sa o jednoduchý, univerzálny trojadresný kód v SSA forme (*Static Single Assignment* [16]) vyvinutý pre potreby projektu LLVM [6]. SSA vraví, že každá premenná môže byť priradená maximálne raz. Pôvodné premenné sú preto typicky rozdelené na niekoľko verzií, ktorých meno sa skladá z originálneho názvu a indexu. Jednoduchý príklad formy je zobrazený na príklade 3.3.

```
y := 1
y := 2
x := y
```

```
y1 := 1
y2 := 2
x1 := y2
```

Obrázok 3.3: Príklad SSA kódu.
Kód naľavo je transformovaný na odpovedajúci kód v SSA forme vpravo.

Aj napriek svojej jednoduchosti obsahuje IR riadkové komentáre (označené „;“), typované globálne (prefix „@“) a lokálne (prefix „%“) premenné, množstvo vstavaných inštrukcií a možnosť rozdeliť kód do funkcií alebo modulov.

Typy LLVM IR môžeme klasifikovať do tried zobrazených v tabuľke 3.1. Primitívne typy sú základné stavebné bloky LLVM systému. Do tejto kategórie patria návestia, typ `void`,

metadata a typy hodnôt MMX registrov. Ďalej tiež typy celých čísiel (integer) `iN`, kde `N` je bitová šírka čísla a desatinné typy (floating point) rôznych veľkostí.

Odvodené typy umožnujú vytváranie zložitejších typov z jednoduchších. Prvým z nich je typ ukazovateľ používaný pre špecifikovanie miesta v pamäti. Jeho typickým využitím je referencovanie uložených objektov. Syntax typu je nasledovná: `<typ>*`. Príkladom je ukazovateľ na desatinné číslo s dvojitou presnosťou: `double*`.

Typ pole ukladá prvky sekvenčne do pamäte. Pole vyžaduje špecifikáciu svojej veľkosti – počtu položiek a typu položiek: `[<# položiek> x <typ položiek>]`. Príkladom môže byť pole štyridsiatich tridsaťdva bitových celých čísiel: `[40 x i32]` alebo aj dvojrozmerné pole desatiných čísiel: `[10 x [10 x float]]`.

Typ štruktúra reprezentuje usporiadanú n-ticu dielčích hodnôt obecne rôznych typov. Štruktúry môžu byť buď definované inline s ostatnými typmi ako: `{ <zoznam typov> }`, alebo deklarované na začiatku programu: `%<meno> = type { <zoznam typov> }`.

Typy funkcií môžu byť chápané ako ich popisy vytvorené z deklarácií. Pozostávajú z návratového typu a zoznamu typov parametrov: `<návrat. typ> (<zoznam typov param.>)`. Príkladom je typ ukazovateľ na funkciu vracajúcu desatinné číslo, prijímajúcu šestnásť bitové celé číslo a ukazovateľ na desatinné číslo: `float (i16, i32*)*`.

Typom `opaque` a vektorovým typom sa nebudeme v tomto dokumente zaoberať.

Trieda first class určuje tie typy, ktorých hodnoty môžu byť produkované inštrukciami programov v IR.

Klasifikácia	**Typy**
integer	i1, i2, ..., i8, ..., i16, ..., i32, ..., i64, ...
floating point	half, float, double, x86_fp80, fp128, ppc_fp128
primitive	label, void, integer, floating point, x86mmx, metadata
derived	array, function, pointer, structure, vector, opaque
first class	integer, floating point, pointer, vector, structure, array, label, metadata

Tabuľka 3.1: Klasifikácia datových typov LLVM IR. Prevzaté zo [7].

K odvodeným datovým typom (polia, štruktúry) v pamäti sa pristupuje jednotlivo po ich položkách. Najprv sa získa ukazovateľ na požadovaný prvok pomocou inštrukcie `getelementptr` a ten je následne použitý k načítaniu alebo uloženiu hodnoty inštrukciami `load` a `store`. Prvým argumentom `getelementptr` je ukazovateľ typu, z ktorého chceme získať položku. Ostatné parametre identifikujú element odkazovaný výsledným ukazovateľom. Ich význam je závislý na zloženom objekte.

Zdrojový kód 3.4 zobrazuje komplexný príklad demonštrujúci väčšinu vyššie popísaných mechanizmov. Na začiatku programu sú vytvorené dva nové datové typy – pomenované štruktúry. Položkou prvej z nich je aj dvojrozmerné pole celých čísiel. Jedným z elementov druhej je celá prvá štruktúra. Ďalej je definovaná funkcia s návratovou hodnotou a parametrom. Funkcia v jednej inštrukcii získa ukazovateľ na položku poľa a v ďalšej vráti jeho hodnotu. Prvý index odkazuje na typ `%struct.ST`, druhý na jeho tretiu položku `%struct.RT`, tretí na pole `[10 x [20 x i32]]`, z ktorého ďalšie dva indexy vyberú štrnásty element na šiestom riadku. Druhá funkcia je funkčne ekvivalentná prvej, ale popísaný postup rozdeľuje na mnoho podkrokov, demonštrujúcich prácu s jednoduchým poľom a štruktúrou.

Zdrojový kód 3.5 zobrazuje LLVM IR kód tak, ako sa typický používa vo vnútry spätného prekladača projektu Lissom. Je vygenerovaný z MIPS inštrukcie sčítania dvoch registrov: `add v0, v0, v1` a v ďalšom texte bude niekoľko krát použitý ako demonštračný príklad. Oba registre sú najprv načítané do pomocných premenných použitých len v rámci tejto postupnosti subinštrukcií. Tieto sú následne sčítané a výsledok je uložený späť do registru `v0`. Pretože sú pomocné premenné generované pre každú MIPS inštrukciu, sú rozlíšené pomocou pridania adresy inštrukcie k svojmu názvu: `_8508`. Keďže je IR typované, musia aj subinštrukcie obsahovať nejaké typy. Ako východzí sa používa celočíselný typ `i32`, čo ale neznamená, že je to aj skutočný typ objektov.

```
struct RT
{
  char A;
  int B[10][20];
  char C;
};

struct ST
{
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s)
{
  return &s[1].Z.B[5][13];
}
```

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo1(%struct.ST* %s) {
    %arrayidx = getelementptr inbounds %struct.ST* %s, i64 1, ↩
         i32 2, i32 1, i64 5, i64 13
    ret i32* %arrayidx
}

define i32* @foo2(%struct.ST* %s) {
    ; %t1 = %struct.ST*
    %t1 = getelementptr %struct.ST* %s, i32 1
    ; %t2 = %struct.RT*
    %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2
    ; %t3 = [10 x [20 x i32]]*
    %t3 = getelementptr %struct.RT* %t2, i32 0, i32 1
    ; %t4 = [20 x i32]*
    %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5
    ; %t5 = i32*
    %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13
    ret i32* %t5
}
```

Obrázok 3.4: Program v jazyku C vľavo je preložený do medzikódu vpravo. Funkcia `foo` je zobrazená v dvoch možných, ekvivalentných variantách. Prevzaté zo [7].

```
; 8508 - add v0, v0, v1
    %u0_8508 = load i32* @gpregs0
    %u1_8508 = load i32* @gpregs1
    %u2_8508 = add i32 %u0_8508, %u1_8508
    store i32 %u2_8508, i32* @gpregs0
```

Obrázok 3.5: Príklad LLVM IR kódu.
Registre `v0, v1` sú tu značené ako globálne premenné `@gpregs0, @gpregs1`.

3.2.4 Predná časť (Front-end)

Úlohou prednej časti je preložiť strojovo závisly kód na univerzálny LLVM IR program. Vstupný súbor v jednom z obecne používaných binárnych formátov je na začiatku transformovaný na špeciálny, interný formát COFF [19] pomocou už predstaveného nástroja bintran.

Inštrukcie takto vytvoreného binárneho súboru sú následne inštrukčným dekóderom preložené na sekvenciu LLVM IR mikroinštrukcií. Preklad je založený na extrahovanej sémantike a binárnom kódovaní operácií získanom z modelu architektúry v jazyku ISAC.

Taktiež je nutné vyrovnať sa so špecifickými vlastnosťami architektúr, ako napríklad oneskorenými skokmi alebo uložením dat v pamäti.

Nasleduje rozpoznanie staticky zostaveného kódu za použitia signatúr knižníc a porovnávania vzorov. Takto pripojený kód nie je nutné analyzovať a dekompilovať, čo sa prejaví na rýchlosti spätného prekladu a čitateľnosti výsledku. Ďalej prebieha vyhľadanie inštrukčných idiomov – sekvencií kódu ktorých kombinovaná sémantika nie je zjavná zo semantiky individuálnych inštrukcií.

Poslednou fázou je vykonanie rady statických analýz za účelom získania čo najväčšieho množstva dodatočných informácií o analyzovanom programe. Analýzy postupne modifikujú a obohacujú vnútornú reprezentáciu, ktorá je na konci transformovaná na výsledný LLVM IR kód.

3.2.5 Optimalizačná časť (Middle-end)

Vstupom je nízkoúrovňová LLVM IR reprezentácia vytvorená prednou časťou spätného prekladača, obsahujúca množstvo mŕtveho, duplicitného alebo inak neefektívneho kódu. Úlohou optimalizátoru je vylepšenie vlastností výstupného medzikódu a jeho príprava na záverečné generovanie vysokoúrovňového programu. Za týmto účelom sú použité vstavané LLVM optimalizácie a optimalizácie navrhnuté pre potreby tohto projektu:

- Odstraňovanie mŕtveho kódu – inštrukcií, ktoré neovplyvňujú výsledok aplikácie. Zníži sa tak veľkosť programu a zlepší jeho čitateľnosť.
- Propagácia konštánt a výrazov, ktorej cieľom je ich zjednodušenie pri zachovaní pôvodného významu.
- Detekcia indukčných premenných, zvyšujúcich alebo znižujúcich sa pri každej iterácii cyklu.

3.2.6 Výstupná časť (Back-end)

Vstupom poslednej fázy je optimalizovaný LLVM IR medzikód a výstupom špecifikovaný cieľový jazyk. V súčasnosti sa jedná o netypovaný jazyk podobný Pythonu a typovaný jazyk C [29]. Ukážky výstupov sú zobrazené v nasledujúcej podkapitole. Pred samotnou konverziou je ale vykonaných niekoľko ďalších krokov:

- Vstupné LLVM IR je preložené na kód zadej časti (BIR – *back-end IR*). Počas konverzie sú detekované a rekonštruované vysokoúrovňové konštrukcie ako cykly a podmienky.
- Medzikód je ďalej optimalizovaný metódami používanými v klasických prekladačoch. Jedná sa napríklad o konverziu globálnych premenných na lokálne, propagáciu konštánt, kopírovanie premenných, zjednodušovanie aritmetických výrazov, preklad cyklov `while` na `for` a ďalšie.
- Ak sú k dispozícii ladiace informácie, sú premenným pridelené ich skutočné mená. V opačnom prípade sú im pridelenné čitateľné názvy ako napríklad druhy ovocia.
- Ak je to požadované, sú zostrojené grafy riadenia toku a volaní.
- Na záver je BIR kód preložený na textovú reprezentáciu vo vybranom jazyku.

3.3 Generovaný kód

Demonštrácia výsledkov spätného prekladu pred použitím typovej analýzy je vykonaná na niekoľkých jednoduchých aplikáciach napísaných v jazyku C, určených pre architektúru MIPS. Táto platforma bola vybraná ako najjednoduchšia z aktuálne podporovaných. Podobné výsledky sú ale dosiahnuté aj na ostatných architektúrach. Programy boli preložené prekladačom *psp-gcc v4.3.2* s vypnutými optimalizáciami -O0. Výstupmi sú opäť programy jazyka C, vygenerované pomocou zadnej časti pre jazyk C. Aby nedošlo k nežiadúcemu odstráneniu kódu, sú premenné inicializované funkciou `rand`. Je dôležité upozorniť, že spätný prekladač dokáže správne odvodiť niektoré jednoduché typy aj bez použitia analýzy datových typov vytvorenej na základe tejto práce.

Prvý program 3.6 vytvorí dve desatinné premenné a ich násobok uloží na pamäťové miesto pridelené ukazovateľu `*fres`. Tento ukazovateľ je následne predaný funkcii `print()`, ktorá vypíše výsledok. Vidíme, že vo výslednom kóde sú aj bez analýzy typov správne určené typy jednoduchých lokálnych premenných. Nepodarilo sa ale rekonštruovať typ alokovanej pamäte ani parametru funkcie.

Ďalší ukážkový program 3.7 pracuje s globálnym poľom celých čísiel `array`. Položky poľa sú najprv inicializované náhodnými číslami a následne sčítané do akumulátoru `sum`. Ten je vypísaný na štandardný výstup a vrátený ako návratová hodnota hlavnej funkcie. Výsledok je funkčne ekvivalentný, ale tentokrát veľmi neprehľadný. Globálne pole je reprezentované adresou `143749976`. K jednotlivým prvkom sa pristupuje pomocou ofsetu a pretypovania. Prístup `array[i]` je nahradený `*(uint32_t *)(4 * apple + 143749976)`. Taktiež môžeme vidieť miernu zmenu pri iterácii cez položky, kde prvý prvok je spracovaný oddelene od ostatných a pôvodný cyklus `for` bol nahradený `while`.

Nasledujúci program 3.8 definuje globálnu štruktúru, ktorú následne inicializuje a použije vo funkcii `print()` (výpis položiek na štandardný výstup). V tomto prípade boli vo výstupnom programe všetky položky štruktúry rozpoznané ako jednoduché globálne premenné, a tak sa s nimi pracuje po jednom.

```c
#include <stdio.h>
#include <stdlib.h>

void print(float *fres) {
   printf("%f\n", *fres);
}

int main() {
   float f1 = (float) rand();
   float f2 = (float) rand();

   float *fres = (float*) malloc(sizeof(float));

   *fres = f1 * f2;

   print(fres);

   return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* ------------- Functions -------------- */

uint32_t print(int16_t a) {
    return printf("%f\n",
        (float64_t)*(float32_t *)(int32_t)a);
}

int main(int argc, char **argv) {
    float32_t apple = rand();
    float32_t banana = rand();
    uint8_t * mem = malloc(4);
    *(float32_t *)mem = apple * banana;
    print((uint16_t)(uint32_t)mem);
    return 0;
}
```

Obrázok 3.6: Príklad spätného prekladu jednoduchého ukazovateľa. Naľavo vstupný program, napravo dekompilovaný výsledok.

```c
#include <stdio.h>
#include <stdlib.h>

int array[20];

int main()
{
   int i;
   int j;

   for (i=0; i<20; i++)
   {
      array[i] = rand();
   }

   int sum;
   for (j=0; j<20; j++)
   {
       sum += array[j];
   }

   printf("%d\n", sum);

   return sum;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    *(uint32_t *)143749976 = rand();
    uint32_t apple = 1;

    uint32_t result;
    uint32_t banana;
    int32_t lemon;
    while (apple < 20) {
        *(uint32_t *)(4 * apple + 143749976) = rand();
        apple = apple + 1;
    }

    lemon = 0;
    result = *(uint32_t *)(4 * lemon + 143749976);
    banana = lemon + 1;

    while (banana < 20) {
        lemon = banana;
        result = result + *(uint32_t *)(4 * lemon + 143749976);
        banana = lemon + 1;
    }

    printf("%d\n", result);
    return result;
}
```

Obrázok 3.7: Príklad spätného prekladu globálneho poľa. Naľavo vstupný program, napravo dekompilovaný výsledok.

```c
#include <stdio.h>
#include <stdlib.h>

struct structure
{
   char c;
   int i;
   float f;
};

struct structure p;

int print()
{
   printf("%c %d %f", p.c, p.i, p.f);
}

int main()
{
   p.c = (char)rand();
   p.i = rand();
   p.f = rand();

   print();

   return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* ---------- Global Variables ---------- */

uint8_t apple = 0;
uint32_t banana = 0;
float32_t lemon = 0.0;

/* ------------- Functions -------------- */

uint32_t print(void) {
    return printf("%c %d %f",
        apple,
        banana,
        (float64_t)lemon);
}

int main(int argc, char **argv) {
    apple = (uint8_t)
            (0x1000000 * rand() / 0x1000000);
    banana = rand();
    lemon = rand();
    print();
    return 0;
}
```

Obrázok 3.8: Príklad spätného prekladu globálnej štruktúry. Naľavo vstupný program, napravo dekompilovaný výsledok.

```c
#include <stdio.h>

struct t {
   int f1;
   int f2;
   int f3;
};
struct s {
   int b;
   char c;
   struct t d[4];
   float f;
   float e[4];
};
struct s x;

int main(void) {
   x.b = rand();
   x.c = (char) rand();
   x.f = rand();
   int i;
   for (i=0; i<4; i++) {
      x.d[i].f1 = rand();
      x.d[i].f2 = rand();
      x.d[i].f3 = rand();
      x.e[i] = rand();
   }
   return 0;
}
```

```c
#include <stdio.h>

uint32_t apple = 0;
uint8_t banana = 0;
float32_t lemon = 0.0;

int main(int argc, char **argv) {
    apple = rand();
    banana = (uint8_t)(0x1000000 * rand() / 0x1000000);
    lemon = rand();
    int32_t cherry = 0;
    *(uint32_t *)143750432 = rand();
    *(uint32_t *)(12 * cherry + 143750436) = rand();
    *(uint32_t *)(12 * cherry + 143750440) = rand();
    *(float32_t *)(4 * (cherry + 14) + 143750428) =
        (float32_t)rand();
    uint32_t grape = cherry + 1;
    cherry = grape;

    while (grape < 4) {
        *(uint32_t *)(12 * grape + 143750432) = rand();
        *(uint32_t *)(12 * cherry + 143750436) = rand();
        *(uint32_t *)(12 * cherry + 143750440) = rand();
        *(float32_t *)(4 * (cherry + 14) + 143750428) =
            (float32_t)rand();
        grape = cherry + 1;
        cherry = grape;
    }
    return 0;
}
```

Obrázok 3.9: Príklad spätného prekladu zložitej globálnej štruktúry. Naľavo vstupný program, napravo dekompilovaný výsledok.

Predchádzajúce príklady boli pomerne jednoduché a výstupy spätného prekladu stále pomerne čitateľné. Posledná demonštrácia ukazuje, čo sa stane ak sú vo vstupnom programe komplikovanejšie zložené typy. Zdrojový kód 3.9 definuje globálnu premennú `x` typu štruktúra, obsahujúcu okrem jednoduchých položiek aj jednoduché pole a pole štruktúr. Výstup je veľmi nekvalitný. Jednoduché elementy sú opäť považované za globálne premenné, ale k poliam sa pristupuje pomocou ich adresy. V takto vyzerajúcom výsledku, je veľmi tažké rozoznať skutočný význam príkazov a celkovú funkciu programu.

Výstupy spätného prekladu po nasadení analýzy datových typov pre tieto, ale aj iné programy sú diskutované v kapitole 7. Konkrétne zdrojové kódy sú porovnané v prílohe A.

3.4 Analýza datových typov

Informácie o datových typoch sú jednou z hlavných vlastností odlišujúcich nízkoúrovňový strojový kód od vysokoúrovňového zdrojového kódu. Typy sú doležité pre vysokoúrovňové vyjadrenie programu: zvyšujú čitateľnosť, zapúzdrujú znalosti, oddeľujú ukazovatele od čísiel a reprezentujú objektovú orientáciu. Na predstavených príkladoch môžeme vidieť, že spätný prekladač v stave pred implementáciou analýzy datových typov nebol vôbec schopný rekonštruovať odvodené typy ako ukazovatele, polia alebo štruktútry. Vďaka jednoduchej analýze objektov, založenej na type použitých operácií sú bezprostredne odvodené len jednoduché typy. Nedochádza ale k ich propagácii a interferencii, čo znižuje presnosť. Cieľom tejto práce je návrh a implementácia takej analýzy, ktorá dokáže presne rekonštruovať väčšinu jednoduchých aj zložených datových typov.

Kapitola 4

Existujúce techniky rekonštrukcie datových typov

> *„Problém analýzy datových typov pri dekompilácii je priradenie vysokoúrovňového datového typu ku každému bloku dat.“* [16]

Keď programátor deklaruje premennú celočíselného datového typu, implikuje tak hodnoty, ktoré môže obsahovať a operácie, ktoré na nej môžu byť vykonané. Súčasťou typu je jeho veľkosť, na základe ktorej typy delia datovú sekciu na množinu objektov so známymi vlastnosťami. Aj napriek tomu, že sú pri preklade datové typy odstránené, môžeme napríklad z aritmetického posunu premennej usúdiť, že sa jedná o celé a nie desatinné číslo. Zároveň je blok dat patriaci premennej odlíšený od ostatných objektov v datovej sekcii. Týmto spôsobom môže spätný prekladač zostrojiť obmedzenia pre každú premennú vždy, keď je použitá v kontexte implikujúcom jej typ.

Zložené datové typy sú na strojovej úrovni väčšinou spracované postupne, jeden prvok za druhým. Komplexné typy možu byť objavené z kontextu elementárnych operácií, napríklad cyklov alebo prístupov na ofsety od známych ukazovateľov.

Tieto základné princípy typovej analýzy budú podrobne popísané v nasledujúcich podkapitolách zaoberajúcich sa existujúcimi technikami rekonštrukcie jednoduchých a zložených datových typov.

4.1 Rekonštrukcia jednoduchých datových typov

> *„Jednoduché (elementárne) datové typy ako celé čísla, ukazovatele alebo desatinné čísla sú odvodené zo sémantiky jednotlivých inštrukcií.“* [16]

Podkapitola pojednáva o zdrojoch typových informácií v strojových jazykoch, existujúcich technikách rekonštrukcie jednoduchých datových typov a podrobne popisuje prístup založený na analýze toku dat.

4.1.1 Zdroje informácií o jednoduchých datových typoch

Ako už bolo spomenuté, strojový jazyk ani jazyk symbolických inštrukcií nie sú typované. Informácie o typoch premenných a ďalších objektoch sú stratené počas prekladu, nakoľko nie sú pre vykonanie programu na procesore potrebné. Ak teda chceme rekonštruovať datové typy, musíme identifikovať zdroje typových informácií v sémantike samotných inštrukcií.

Tieto vlastnosti obmedzujú možné typy operandov každej inštrukcie a môžeme ich rozdeliť do štyroch kategórií:

- *Obmedzenia registrov* – ak je objekt uložený do celočíselného registru, môžeme o ňom prehlásiť, že je celočíselného datového typu alebo ukazovateľom. Na druhej strane, ak je uložený do registra s plávajúcou desatinnou čiarkou, jedná sa o desatinné číslo. Podobne môžeme využiť aj špeciálne registre obsahujúce príznaky.
- *Obmedzenia inštrukcií* – sú hlavným zdrojom typových informácií. Pre každú inštrukciu môžeme určiť všetky prípustné typy jej operandov a výsledku. Ak sú napríklad dva objekty sčítané pomocou inštrukcie celočíselného sčítania, môžeme s určitosťou vylúčiť, že sa jedná o desatinné čísla a predpokladať celočíselný typ alebo ukazovateľ.
- *Obmedzenia príznakov* – doplnková informácia o znamienku celočíselného datového typu odvodená z inštrukcií pracujúcimi s príznakmi.
- *Obmedzenia prostredia* – sú získané zo signatúr knižničných funkcií volaných z analyzovaného programu. Na ich základe môžeme s úplnou presnosťou určiť typy operandov a návratovej hodnoty. V tomto prípade je nutná databáza podpisov a schopnosť spätného prekladača identifikovať volania známych funkcií.

4.1.2 Alan Mycroft

Základy typovej analýzy pri spätnom preklade položil Alan Mycroft v roku 1999 [24], kedy predstavil algoritmus založený na unifikácii typových obmedzení odvodených z inštrukcií programu. Jedná sa o univerzálne riešenie prekladajúce RTL na jazyk C. RTL forma programu je získaná spätným prekladom a rozbalením makier z jazyka symbolických inštrukcií. Algoritmus predpokladá, že bitová veľkosť RTL objektov patrí do množiny $\{8,\ 16,\ 32,\ 64\}$, ktoré sú vo výslednom kóde reprezentované ako $\{char,\ short,\ int,\ long\}$[1] s možnosťou doplnenia o kvalifikátor *unsigned* v prípade, že sa dá odvodiť. Inak sú všetky objekty znamienkové. Ukazovatele sú považované za 32-bitové hodnoty, od celých čísiel rozlíšiteľné len na základe spôsobu ich použitia. Notácia $mem(s)$ reprezentuje typ v pamäti, o ktorom je známe, že obsahuje niekoľko typov na rôznych ofsetoch (štruktúra). Na základe týchto pravidiel teda môžeme zostaviť gramatiku typov zobrazenú na rovniciach 4.1.

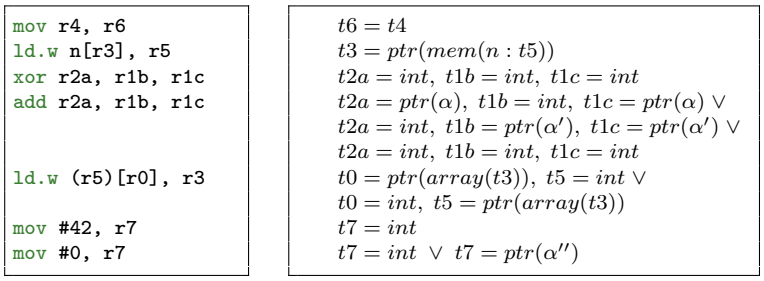
$$\begin{aligned} type &::= char \mid short \mid int \mid ptr(t) \mid array(t) \mid mem(s) \mid union(t1,\ \ldots,\ t_k) \\ struct &::= n_1 : t_1,\ \ldots,\ n_k : t_k \\ reg &::= int \mid ptr(t) \end{aligned} \tag{4.1}$$

Teraz môžeme pre každú inštrukciu generovať typové obmedzenia objektov v nej vystupujúcich. Príklad procesu je zobrazený na obrázku 4.1. Na ľavej strane vidíme symbolické inštrukcie v SSA forme. Na pravej strane odvodené obmedzenia typov. Symbol α označuje nové typové premenné vytvorené behom odvodzovania.

Takto vytvorené obmedzenia (termy) môžeme v záverečnom kroku riešiť. Používa sa takzvaná Herbrandova unifikácia [12] rozšírená o niekoľko ďalších pravidiel použitých v prípade, že základná procedúra zlyhá. Výsledkom je také priradenie typov objektom, ktoré

[1] Aj keď článok nezmieňuje nejakú konkrétnu architektúru, z tohto mapovania vyplýva, že uvažuje tridsaťdva bitové platformy.

zaistí splnenie všetkých vygenerovaných obmedzení. Na ich základe sú nakoniec pre objekty vybrané odpovedajúce typy jazyka C.



Obrázok 4.1: Príklad odvodenia typových obmedzení (vpravo) z Intel x86 inštrukcií v SSA forme (vľavo). Prevzaté z [24].

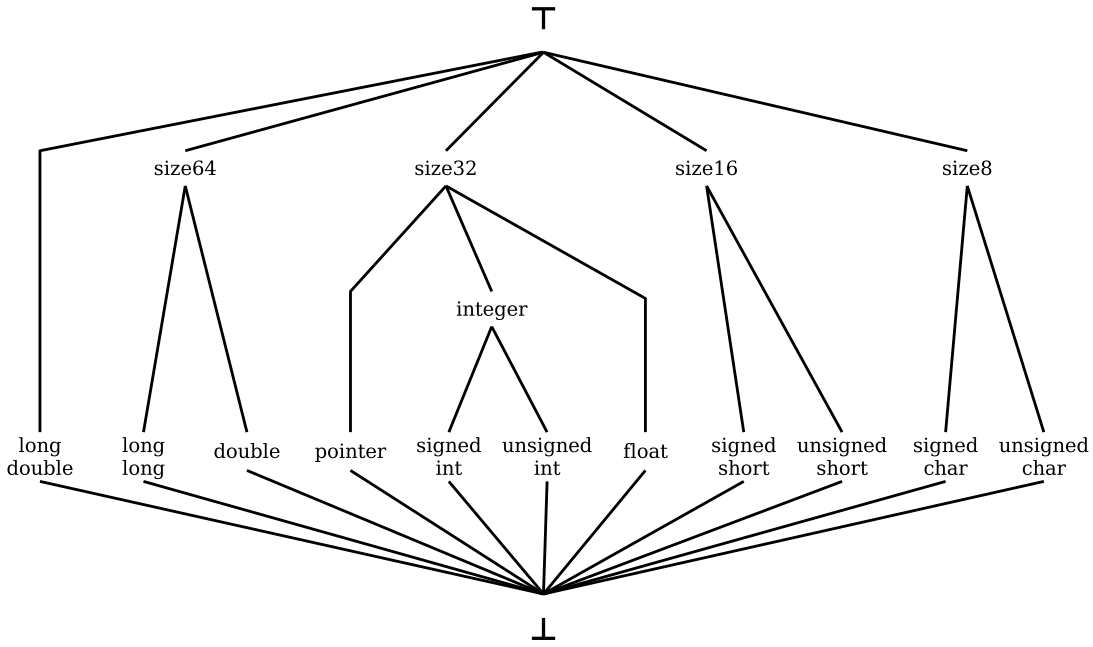
Nevýhodou tohto prístupu je, že často nedokáže rozlišovať znamienka celých čísiel a vôbec sa nezaoberá číslami desatinnými. V prípade, že unifikácia nie je schopná plne vyriešiť systém obmedzení, hovoríme o konflikte – istý objekt sa správa ako keby mal dva a viac typov. Tento stav môže nastať v prípade použitia únií v pôvodnom programe, explicitnom pretypovaní ukazovateľov, alebo ak aplikácia nedodržiava bežné pravidlá pre prácu s typmi. Algoritmus pre tieto objekty produkuje únie a konfliktné termy neodstraňuje z pracovnej množiny. To môže viesť k rýchlemu šíreniu konfliktov. Navyše je samotný algoritmus často divergentný.

4.1.3 Michael Van Emmerik

Ďalším príspevkom k rekonštrukcii datových typov je dizertačná práca Michaela Van Emmerika [16] venovaná SSA dekompilácii, zaoberajúca sa aj problematikou datových typov. Podrobne rozoberá význam typov pre čitateľnosť programu, zdroje typových informácií a rieši celú škálu problémov, ktoré môžu nastať. Základný princíp je opäť rovnaký – odvodenie typov zo sémantiky operácií. Okrem Mycroftovho prístupu unifikácie termov je tu však predstavená aj možnosť riešiť sústavu obmedzení pomocou analýzy toku dat. V tomto prípade sa využívajú relácie medzi typmi k zostaveniu zväzu zaznamenávajúceho ich hierarchiu (obrázok 4.2). Suprémum $\top$ predstavuje úplnú neznalosť typu a infímum $\bot$ typový konflikt. Pomocou analýzy toku dat propagujúcej informácie cez zostavené rovnice sa algoritmus snaží odvodiť čo najšpecifickejší typ každého objektu. Tento spôsob riešenia bude podrobne popísaný v podkapitole 4.1.5.

Navrhovaný algoritmus používa takzvanú riedku reprezentáciu typov. Namiesto udržiavania typovej informácie pre každý živý objekt, sú ukladané len unikátne definície v miestach priradenia objektov. Ak nie je objekt jednoznačne priradený, vytvorí sa preň umelá definícia. Na tieto sa potom odkazujú ďalšie výskyty objektu a všetky odvodenné objekty. Výhodou sú nižšie datové nároky algoritmu, čo sa prejaví hlavne pri analýze programu v SSA forme, ktorá obsahuje veľké množstvo odvodených objektov rovnakého typu. V riedkej reprezentácii nie sú ukladané typy podvýrazov, čo znamená, že musia byť vypočítané na požiadanie. Za efektívnejšie využívanie pamäte teda zaplatí zvýšená časová náročnosť. Taktiež je nutná nie zrovna jednoduchá implementácia typových výrazov a ich on-demand vyhodnocovania.

Algoritmus bol implementovaný ako súčasť spätného prekladača *Boomerang* predstaveného v podkapitole 2.4.3.



Obrázok 4.2: Zväz datových typov. Obrázok prevzatý z [16] a upravený.

4.1.4 TIE: Type Inference on Executables

Jednou z najnovších prác je [21] z roku 2011. Predstavuje vylepšenú verziu Mycroftovho unifikačného algoritmu. Opäť sú budované formule z jednotlivých inštrukcií a tie následne riešené. Kladie sa pritom dôraz na konvergentnosť a striktnosť algoritmu – snaha odvodiť čo najviacej informácií ale nikdy typy nehádať.

Rekonštrukcia je súčasťou robustného generického reverzného nástroja TIE [11]. Jej techniky využívajú informácie o toku riadenia a preto môže byť použitá v statickej aj dynamickej analýze. Vstupný program je najskôr preložený do internej formy v jazyku BIL (*Binary Analysis Language*) a následne je spustená analýza premenných, ktorá odhalí ich lokácie. V tomto momente je spustená rekonštrukcia typov pracujúca rovnako ako predchádzajúce v troch štandardných krokoch: 1) priradenie typových termov k premenným, 2) generovanie obmedzení z inštrukcií, 3) riešenie obmedzení.

Pri rekonštrukcii sa opäť využiva zväz typov a pohyb smerom nadol od supréma $\top$. Tvar odvodzovacích pravidiel je tento krát vyjadrený vo forme zobrazenej na rovnici 4.2. Čitateľ obsahuje zoznam predpokladov, po ktorých splnení môžeme vyvodiť záver v menovateli. Takéto pravidlá sú opakovane aplikované na predpoklady, až kým nie je dosiahnutý ustálený stav – ďalšia aplikácia už nič nezmení. Pri riešení obmedzení nad zväzmi sa využivajú operácie spojenia (join) $\sqcup$ a prieseku (meet) $\sqcap$. Pričom existuje mnoho spôsobov ich aplikácie na základe štruktúry obmedzenia. Pre ich konkrétny tvar viz [21].

$$\frac{P_1 \; P_2 \; \ldots \; P_n}{C} \tag{4.2}$$

4.1.5 Data-flow analýza datových typov

Posledná popísaná technika automatickej rekonštrukcie jednoduchých datových typov bude využívať analýzu toku dat. Bola predstavená v roku 2009 v článku [14] autormi Dolgova a Chernov. Pre architektúru x86 je implementovaná v spätnom prekladači *SmartDec* uvedenom v podkapitole 2.4.3.

Objekty

Pod pojmom *objekt* budeme rozumieť akýkoľvek register, globálnu premennú (adresu), lokálnu premennú (ofset na zásobníku) alebo parameter a návratovú hodnotu funkcie, ktorých datový typ sa analýza bude snažiť rekonštruovať.

Datový typ T bude vytvorený pomocou trojice atribútov *core*, *size*, *sign*, spolu reprezentujúcich tri zložky každého typu. Atribút *core* predstavuje základnú informáciu o type, a teda či sa jedná o celé číslo, ukazovateľ alebo desatinné číslo. Atribút *size* indikuje jeho veľkosť v bitoch a *sign* znamienko. Každý atribút je rekonštruovaný oddelenne a ich výsledná kombinácia udáva všetky možné datové typy, ktoré môže objekt mať.

$$core \in \{integer, pointer, float\} \tag{4.3}$$
$$size \in \{1, 8, 16, 32, 64\} \tag{4.4}$$
$$sign \in \{signed, unsigned\} \tag{4.5}$$
$$T = \left(\tau^{core}, \tau^{size}, \tau^{sign}\right) \tag{4.6}$$

Cieľom analýzy je redukovať počet možností na minimum bez toho, aby ostal atribút *core* prázdny. Takýto stav by znamenal, že je objekt v programe používaný nekonzistentne. Napríklad na jednom mieste ako celé číslo a na inom ako ukazovateľ. Keďže desatinné čísla nemajú znamienko a ukazovateľ je reprezentovaný len ako $(\{pointer\}, \emptyset, \emptyset)$, nemusia byť ostatné dva atribúty vždy neprázdne. Pseudotyp *void* reprezentuje trojica $(\emptyset, \emptyset, \emptyset)$. Takto vytvorená reprezentácia môže byť jednoducho namapovaná na skutočné datové typy jazyka C. Napríklad `unsigned short` = $(\{integer\}, \{16\}, \{unsigned\})$ alebo `char` = $(\{integer\}, \{8\}, \{signed\})$.

Pretože sa môže jeden objekt vyskytovať v programe na viacerých miestach, je priradený každému výskytu vlastný datový typ T_i. Všetky typy objektu sú $T = \{T_1, \ldots, T_n\}$ a výsledný rekonštruovaný typ bude rovný $T = T_1 \sqcup \cdots \sqcup T_n$, kde $\sqcup$ je funkcie spojenia (*join*) charakteristická pre data-flow analýzu definovaná ako prienik (rovnica 4.8) jednotlivých atribútov:

$$T_1 \sqcup T_2 = \left(\tau_1^{core} \sqcup^{core} \tau_2^{core}, \tau_1^{size} \sqcup^{size} \tau_2^{size}, \tau_1^{sign} \sqcup^{sign} \tau_2^{sign}\right) \tag{4.7}$$
$$\sqcup^{core} \equiv \sqcup^{size} \equiv \sqcup^{sign} \equiv \cap \tag{4.8}$$

Inými slovami, datový typ tvorí zväz – na začiatku analýza o objekte nič nevie a jeho typ môže byť akýkoľvek (suprémum $\top$), postupnou aplikáciou join funkcie sa posúva v zväze nižšie a nižšie. Cieľom je dostať sa čo najnižšie bez dosiahnutia úplného dna (infímum $\bot$). To by totiž znamenalo, že je objekt v programe používaný nekonzistentne – na rôznych miestach sa správa ako keby mal rôzne datové typy.

$$\top = (\{integer, pointer, float\}, \{1, 8, 16, 32, 64\}, \{signed, unsigned\}) \tag{4.9}$$
$$\bot = (\emptyset, \emptyset, \emptyset) \tag{4.10}$$

Pretože má každá platforma len obmedzený počet registrov, je jeden a ten istý register často nezávisle používaný na rôznych miestach programu. Rovnaká situácia môže nastať aj pri znovupoužití miesta na zásobníku (anglicky *stack reuse*). Z tohto dôvodu teda nemôžeme o takýchto objektoch prehlásiť, že majú na všetkých miestach aplikácie rovnaký datový typ. Riešením je používanie takzvaných DU reťazí (*Definition-Use Chain*), pozostávajúcich z definície objektu a zoznamu všetkých, bez možnosti redefinície dosiahnuteľných použití. DU grafy sú získané pomocou osobitnej analýzy toku dat a sú často používané aj v prekladačoch za účelom rôznych optimalizácií. Môžeme ich teda využiť pre rozdelenie množiny všetkých výskytov jedného objektu do takých podmnožín, v ktorých sú len výskyty s rovnakým datovým typom.

Systém propagačných rovníc

Pre daný program v jazyku symbolických inštrukcií je zostavená množina *propagačných rovníc*, cez ktoré je možné šírenie typovej informácie. Môžeme ich rozdeliť na tieto kategórie:

- Binárne rovnice tvaru ($T_{dst} \Leftrightarrow T_{op1} \bigcirc T_{op2}$) reprezentujú strojové inštrukcie vykonávajúce operáciu $\bigcirc$ nad operandami a ukladajúce výsledok do cieľového objektu.
- Unárne rovnice tvaru ($T_{dst} \Leftrightarrow \bigcirc T_{op}$) majú rovnaký význam, ale neobsahujú druhý operand.
- Kopírovacie rovnice tvaru ($T_{dst} \Leftrightarrow T_{op1} \rightarrow T_{op2}$) značia presúvanie prvého operandu do výsledku s využitím operandu druhého (obsahujúceho napríklad ofset).

Výsledný systém presne korešponduje s analyzovaným programom a je následne využitý k šíreniu typových informácií v dvoch smeroch: od operandov k výsledku a od výsledku k operandom.

Propagácia od operandov k výsledku

Pri tomto type propagácie je na základe datových typov operandov vyvodený záver o type výsledku. Deje sa tak pomocou *propagačných pravidiel* špecifických pre každú jednu inštrukciu programu – propagačnú rovnicu. Pravidlá su zostavené na základe operácií jazyka C [18], tu využívaného ako typovaný jazyk symbolických inštrukcií. Zároveň ale predpokladáme, že väčšina dekompilovaných programov bola pôvodne napísaná práve v tomto jazyku. Celkovo sa jedná o veľké množstvo pravidiel a tak na tomto mieste budú popísané len skupiny do ktorých ich môžeme zaradiť.

Propagácie cez binárne rovnice budú vysvetlené na príklade inštrukcie sčítania: $T_{dst} \Leftrightarrow T_{op1}\ ADD\ T_{op2}$. Pravidlá pre atribút *core* vyzerajú nasledovne:

$$\begin{array}{l} \dfrac{integer \in \tau_{op1}^{core} \wedge integer \in \tau_{op2}^{core}}{integer \in \tau_{dst}^{core}} \wedge \\ \dfrac{pointer \in \tau_{op1}^{core} \wedge integer \in \tau_{op2}^{core}}{pointer \in \tau_{dst}^{core}} \wedge \\ \dfrac{integer \in \tau_{op1}^{core} \wedge pointer \in \tau_{op2}^{core}}{pointer \in \tau_{dst}^{core}} \end{array} \tag{4.11}$$

Zlomková čiara je tu použitá vo význame implikácie tak, ako sme to videli v podkapitole 4.1.4. V čitateli je zoznam predpokladov, ktorých splnenie implikuje záver v menovateli.

Prvé pravidlo napríklad hovorí, že ak môžu byť oba operandy celočíselného typu, potom aj výsledok môže byť tohto typu. Ostatné pravidlá majú analogický význam. Iné kombinácie, nezachytené v týchto predpisoch, sú považované za chybné a nie je pre ne vykonaná žiadna akcia.

Definujme operáciu vrchného prieniku pre dve množiny celých čísiel nasledovne: ak S_1 a S_2 sú množiny, potom ich vrchný prienik $S = S_1 \sqcap S_2$ je:

$$S \ = \{s \mid s = max(s_1, s_2) \ : \ (s_1, s_2) \in S_1 \times S_2\} \tag{4.12}$$

Kde operácia *max* vyberie takú dvojicu z binárneho kartézskeho súčinu, v ktorej sú oba prvky maximálne a vytvorí z nich dvojprvkovú množinu S.

Napríklad pre množiny $\{1, 2\} \sqcap \{2, 4\} = \{2, 4\}$. Pravidlo pre propagáciu atribútu *size* teraz môžeme zapísať ako: $\tau^{size}_{dst} = \tau^{size}_{op1} \sqcap \tau^{size}_{op2}$.

Pravidlá pre atribút *sign* sú obdobné atribútu *core*:

$$\begin{gathered}
\frac{unsigned \in \tau^{sign}_{op1} \wedge unsigned \in \tau^{sign}_{op2}}{unsigned \in \tau^{sign}_{dst}} \wedge \\
\frac{unsigned \in \tau^{sign}_{op1} \wedge signed \in \tau^{sign}_{op2}}{unsigned \in \tau^{sign}_{dst}} \wedge \\
\frac{signed \in \tau^{sign}_{op1} \wedge unsigned \in \tau^{sign}_{op2}}{unsigned \in \tau^{sign}_{dst}} \wedge \\
\frac{signed \in \tau^{sign}_{op1} \wedge signed \in \tau^{sign}_{op2}}{signed \in \tau^{sign}_{dst}}
\end{gathered} \tag{4.13}$$

Pre unárne rovnice existuje jediné, jednoduché propagačné pravidlo hovoriace, že typ výsledku je rovnaký ako typu operandu: $T_{dst} = T_{op}$

Šírenie informácie cez kopírovacie rovnice tvaru $T_{dst} \Leftrightarrow T_{op1} \rightarrow T_{op2}$ je opäť veľmi jednoduché: $T_{dst} = T_{op1} \sqcup T_{op1}$.

V prípade, že sa jedná o kopírovanie z/do pamäte na adresu určenú jedným z operandov, hovoríme o operácii prístupu do pamäte a tento operand je považovaný za ukazovateľ: $T_{dst} = T_{op1} \sqcup (\{pointer\}, \emptyset, \emptyset)$.

Propagácia od výsledku k operandom

Pri opačnej propagácii sa na základe datového typu výsledku odvodia typy operandov. V mnohých prípadoch je postup rovnaký ako pri predchádzajúcej propagácii, ale existujú inštrukcie spôsobujúce komplikácie. Uvažujme opäť binárnu rovnicu $T_{dst} \Leftrightarrow T_{op1}\ ADD\ T_{op2}$ a predpokladajme, že typ výsledku môže byť ukazovateľ: $pointer \in \tau^{core}_{dst}$. Štandard jazyka C, z ktorého pravidlá vychádzajú vraví, že výsledok sčítania je ukazovateľ v prípade, že jeden z operandov je ukazovateľ a druhý celé číslo. V tomto prípade teda existujú dve možnosti:

$$(pointer \in \tau^{core}_{op1} \ \wedge \ integer \in \tau^{core}_{op2}) \ \vee (integer \in \tau^{core}_{op1} \ \wedge \ pointer \in \tau^{core}_{op2}) \tag{4.14}$$

Pretože je nemožné vybrať len jednu, je treba brať do úvahy obe. Riešením je pridanie takzvaných alternatívnych elementov, zaznamenávajúcich vzťah medzi objektami: $\tau^{class}_p?[q]$.

Kde $class \in \{core, size, sign\}$, p je objekt, ktorého typ obsahuje alternatívny element a q je objekt obsahujúci korešpondujúci alternatívny element. Aspoň jeden z páru musí po operácii *join* patriť do výslednej množiny. V prípade, že sa tak nestane musia byť oba explicitne pridané do oboch množín. Pravidlo pre uvedenú binárnu rovnicu by vyzeralo nasledovne:

$$\frac{ptr \in \tau_{dst}^{core}}{\tau_{op1}^{core} \leftarrow \tau_{op1}^{core} \cup \{pointer_{op1}^{dst}?[op2]\} \ \wedge \ \tau_{op2}^{core} \leftarrow \tau_{op2}^{core} \cup \{pointer_{op2}^{dst}?[op1]\}} \tag{4.15}$$

Hovorí, že v prípade ak môže byť výsledok ukazovateľom, je do *core* atribútov oboch operandov pridaný alternatívny prvok, ktorý ich prepája.

Pravidlá pre unárne operácie a operácie pristupujúce k pamäti sú podobné ako pri propagácii od operandov k výsledku. Pre kopírovacie operácie sú pravidlá nasledovné: $T_{op1} = T_{op1} \sqcup T_{dst}$ a $T_{op2} = T_{op2} \sqcup T_{dst}$.

Algoritmus

Výsledný algoritmus rekonštrukcie jednoduchých datových typov je zobrazený na pseudokóde 4.3. Vstupom je analyzovaný program a množina objektov $OBJ = \{obj_1, \ldots, obj_n\}$. Funkcia `make_tree(program, OBJ)` zostaví systém propagačných rovníc z inštruckcií aplikácie. Funkcia `init(Equation)` inicializuje všetky typy objektov počiatočnou hodnotou *void*. Funkcie `spread_info_lr(Equation)` a `spread_info_rl(Equation)` propagujú typové informácie od operandov k výsledku a od výsledku k operandom. Funkcia `Type(obj)` vráti množinu všetkých výskytov typov objektu *obj* v systéme rovníc. Táto množina je následne spojená funkciami `left_join()`, `right_join()` a `full_join`, vracajúcich *true* ak sa typ aspoň jedného výsktu zmenil. Inak vrátia $false$. Cyklus analýzy toku dat končí v momente, keď množiny výskytov všetkych objekty dosiahnu pevného bodu a ďalšie iterácie by ich už nikdy nezmenili. Nakoniec funkcia `match_type()` priradí objektom odpovedajúce typy jazyka C.

Takýto algoritmus by mal byť schopný perfektne rekonštruovať jednoduché datové typy za predpokladu, že sa vstupný program riadi štandardom na základe ktorého boli vytvorené pravidlá inicializácie a propagácie datových typov. Pretože algoritmus operuje nad konečnými zväzmi pomocou monotónej funkcie spojenia, zaručene v konečnom čase dosiahne pevný bod a skončí.

```
Tree = make_tree(program);
Equation = make_equation(Tree, OBJ);
init(Equation);
set_constraint(Equation);
flag = true;

while (flag)
{
    flag = false;

    forall equation in Equation :
        spread_info_lr(equation);
    forall obj in OBJ :
        flag |= left_join(Type(obj));
    forall equation in Equation :
        spread_info_rl(equation);
    forall obj in OBJ :
        flag |= right_join(Type(obj));
    forall obj in OBJ :
        flag |= full_join(Type(obj));
}

forall obj: OBJ :
    Tobj = match_type(Type(obj));
```

Obrázok 4.3: Algoritmus rekonštrukcie základných datových typov. Prevzaté z [14].

4.2 Rekonštrukcia zložených datových typov

> *„Zložené (agregované) datové typy ako polia, štruktúry a uniony sú kombináciou elementárnych a zložených typov. Agregované typy sú rozpoznané pomocou analýzy ofsetov.“* [16]

Kapitola pojednáva o vlastnostiach zložených datových typov, existujúcich technikách ich rekonštrukcie a podrobne popisuje prístup založený na analýze ofetov.

4.2.1 Polia

Pole je homogénna kolekcia elementov rovnakého typu uložených v pamäti jeden za druhým. K položkám sa pristupuje pomocou indexových výrazov. Napríklad k poľu `int p[32]` a prístupu `p[i]` je vytvorený výraz `m + i * 4`. Použitie multiplikácie je charakteristickou črtou prístupu k poliam. Typ každého takéhoto prístupu je určený typom základovej adresy. Multiplikatívna komponenta obsahuje informáciu o veľkosti jedného elementu. V prípade viacrozmerných polí je výraz komplexnejší a obsahuje niekoľko multiplikácií. V takom prípade je veľkosť elementu rovná najmenšej konštante vo výraze. Problém môže nastať, ak v pôvodnom programe samotný index prístupu obsahuje násobenie konštantou. Napríklad v prípade `p[2*i]` bude číslo 2 nesprávne považované za veľkosť jedného elementu. Riešením je použitie najväčšieho spoločného deliteľa všetkých multiplikatívnych konštánt použitých pre prístup k jednému poľu. Ak všetky prístupy obsahujú rovnakú konštantu, potom je z pohľadu rekonštrukčnej analýzy element poľa štruktúra dvoch položiek, z ktorých prvá sa dá rekonštruovať a druhá nie, keďže do nej neexistujú žiadne prístupy.

Náhodný prístup do poľa pomocou konštantného indexu, napríklad `p[5]` je v strojovom jazyku nerozlíšiteľný od prístupu k položke štruktúry. Zbieraním informácií o pamäťových blokoch v celom programe je často možné túto nejednoznačnosť vyriešiť. Hlavným problémom analýzy je určenie veľkosti poľa – počtu jeho elementov. Jazyk C nekontroluje hranice

polí a assembler tak často neobsahuje žiadne informácie o veľkosti. Jednou z možností je analýza umiestnenia položiek v pamäti. V niektorých prípadoch, ak je známa veľkosť pamäťového bloku je možné určiť aj veľkosť poľa. Konkrétne ak je pole umiestnené medzi ďalšími objektami, napríklad položkami štruktúry alebo premennými, jeho horná hranica môže byť z týchto informácií vypočítaná.

4.2.2 Štruktúry

Štruktúra je heterogénna n-tica elementov rôznych datových typov uložených v pamäti jeden za druhým. Veľkosť štruktúry je rovná veľkosti okupovanej pamäti. Prekladač môže kvôli zarovnaniu pridať voľné bloky medzi jednotlivé položky alebo na ich koniec. V strojovom kóde neexistujú žiadne mená členov. Pristupuje sa k nim pomocou ofsetov z počiatočnej (bázovej) adresy. Položky môžu byť rekonštruované pomocou konštrukcie množiny všetkých ofsetov použitých pre prístup do pamäťovej oblasti, v ktorej sa potenciálne nachádza nejaká štruktúra. Za týmto účelom musia byť sledované typy všetkých bázových ukazovateľov. Rekonštrukcia všetkých členov štruktúry nemôže byť garantovaná, pretože spätný prekladač nemá žiadne informácie o položkách, ku ktorým sa nikdy nepristúpi. Typ položiek môže byť jednoduchý alebo zložený – vnorená štruktúra alebo pole. Prípad vnorenej štruktúry väčšinou nie je rozlíšiteľný od prípadu, keď sú jej položky priamo súčasťou nadradenej štruktúry. K lokálnym premenným a parametrom funkcií umiestným na zásobníku sa typicky pristupuje pomocou ofsetu od ukazovateľa na vrchol zásobníku. Javia sa tak ako jedna veľká štruktúra a automatická rekonštrukcia zložených datových typov na zásobníku nie je môžná.

4.2.3 Únie

Únia je datová štruktúra obsahujúca premennú, ktorá môže mať niekoľko datových typov. Táto vlastnosť je v priamom protiklade k základnému princípu analýzy – objekt má po celú dobu existencie len jeden typ. Z tohto dôvodu spôsobuje pokus o rekonštrukciu únií konflikty a nie je preto v ďalšom texte uvažovaný. Kvalitná detekcia únií je stále otvoreným problémom a nebola úpešne vyriešená ani v jednom z predstavených článkov. Touto otázkou sa nepochybne bude zaoberať budúci výskum.

4.2.4 Existujúce techniky rekonštrukcie

Už pri rekonštrukcii jednoduchých datových typov vychádzali všetky metódy z jedného a toho istého princípu – analýzy sémantiky inštrukcií s ohľadom na implikované typy. Techniky predstavené v podkapitole 4.1 sa od seba líšili reprezentáciou a kvalitou – spôsobom riešenia typových obmedzení zostavenými nad inštrukciami. Nakoľko je technika rozpoznania zložených datových typov vo všetkých článkoch rovnaká a líši sa len zasadením do už popísaných systémov, bude v tejto podkapitole vysvetlený len prístup naväzujúci na metódu analýzy toku dat založený na článku [14] a jeho pokračovaní [28].

4.2.5 Algoritmus

Algoritmus rekonštrukcie zložených datových typov môžeme rozdeliť na tri hlavné kroky:

- Priradenie adresových výrazov k pamäťovým prístupom.
- Konštrukcia množín ekvivalentných návestí a agregovaných množín.

- Rekonštrukcia štruktúr a polí.

Algoritmus je konvergentný, lebo operuje na konečnej množine návestí a všetky operátory sú monotóne. Nedokáže si poradiť s úniami, niekedy nevie zistiť veľkosť polí a nie vždy dokáže rekonštruovať zanorené štruktúry. Jeho jednotlivé kroky sú popísané v nasledujúcom texte.

Adresové výrazy

Adresový výraz t popisuje spôsob vypočítania adresy pamäťového prístupu. Môže byť reprezentovaný nasledovne:

$$\begin{aligned} t &::= \text{addr}(b, o, m) \mid \text{nil} \\ b &::= \text{label} \mid 0 \\ o &::= \text{integer} \\ m &::= 0 \mid f \mid sum(f, m) \\ f &::= mul(\text{integer}) \end{aligned} \tag{4.16}$$

Kde integer predstavuje celé číslo, label návestie, $sum()$ funkciu sčítania dvoch parametrov a $mul()$ násobenie parametrom.

Nad programom je vykonaná analýza dosahujúcich definícií (*reaching definition dataflow analysis*), ktorá každému použitiu registra u priradí množinu inštrukcií definujúcich toto použitie: $Defs(u) = \{r_1, \ldots, r_m\}$. Každej definícii z množiny môžeme priradiť odpovedajúci adresový výraz: $AE(r)$. Na začiatku budú všetky výrazy rovné hodnote nil. Term $\text{addr}(0, 0, mul(1))$ predstavuje nezistiteľnú hodnotu označenú ANY.

S výrazmi pracujeme pomocou dvoch operácií. $Merge(t_1, t_2)$ prijíma dva výrazy a vracia jeden, ktorého zoznam multiplikácií vznikol zlúčením a aplikáciou funkcie najväčšieho spoločného deliteľa na pôvodné zoznamy. Funkcia $Join(t_1, \ldots, t_m)$ nad množinou výrazov je definovaná nasledovne:

- ak pre každé $i, j = 1 \ldots m,\ t_i \neq \text{nil} \wedge t_i = t_j$, potom $Join(t_1, \ldots, t_m) = t_1$.
- ak existuje i, také že $t_i = ANY$, potom $Join(t_1, \ldots, t_m) = ANY$.
- ak existujú indexy i, j, také že $t_i \neq \text{nil} \wedge t_i \neq t_j$, potom $Join(t_1, \ldots, t_m) = ANY$.
- inak $Join(t_1, \ldots, t_m) = \text{nil}$.

Pre množinu definícií registru $\{r1, \ldots, r_m\}$ môžeme definovať funkciu:

$$Join(r_1, \ldots, r_m) = Join(AE(r_1), \ldots, AE(r_m)) \tag{4.17}$$

Pomocou týchto operácií je možné vytvoriť propagačné pravidlá pre adresové výrazy. Pre každú inštrukciu programu, pracujúcu (počítajúcu) hodnotu registru obsahujúceho adresu musí existovať špecifické pravidlo vyčíslujúce výsledný výraz. Nakoľko sa jedná o veľký počet pomerne komplikovaných pravidiel, ktorých presné znenie nie je pre pochopenie princípu algoritmu nutné, nebudú tu uvedené. Pre viac informácií viz [28].

Pravidlá sú aplikované až kým nie je dosiahnutý pevný bod – pre všetky termy platí $AE(r_i) \neq \text{nil}$ a ďalšie použitie pravidiel by už nezmenilo žiadny výraz. Výsledkom je výraz v popísanej forme definujúci hodnotu každého jedného použitia ľubovoľného registru v programe.

Pamäťové prístupy

Výsledky predchádzajúceho kroku vo forme $(b, o, sum(mul(C_1), sum(mul(C_2)\dots)))$ sú použité k zostaveniu výrazov pre pamäťové prístupy:

$$\left(b + o \ + \sum_{j=0}^{n} C_j x_j \right) \tag{4.18}$$

kde b je bázová adresa, o ofset a ostatné multiplikatívna komponenta. C_j konštanta pre ktorú predpokladáme: $0 < C_j < C_{j+1}$.

Ďalej definujme dve význačné multiplikatívne konštanty:

$$m = \min_{j=1}^{n} \ C_j = C_1 \tag{4.19}$$

$$M = \max_{j=1}^{n} \ C_j = C_n \tag{4.20}$$

V programe dodržujúcom štandard jazyka C bude b nejaké návestie l_i rovné konkrétnej adrese v prípade prístupu ku globálnej premennej, alebo identifikátoru symbolu (ukazovateľu) obsahujúceho bázovú adresu. Po tomto kroku sú zachované len tie adresové výrazy, použité pre načítanie alebo ukladanie do pamäte. Všetky ostatné sú z množiny vyradené.

Priradenie návestí

Každý prístup do pamäte má na začiatku unikátne návestie l. Neskôr sú tieto združované do ekvivalenčných tried z ktorých sú zostrojené výsledné typy. Návestie vyzerá nasledovne:

$$l_i : (b_i, o_i, C_{i,1}, \dots, C_{i,n}) \tag{4.21}$$

Návestie vyjadruje možnosť, že označená premenná použitá pre prístup je nejakého, zatiaľ neznámeho typu ukazovateľ. Samozrejme, niekoľko pamäťových prístupov na rôznych miestach programu môže mať rovnaký výsledný typ. Cieľom je zostavenie množín návestí, odpovedajúcich rovnakým vysokoúrovňovým datovým typom. Na začiatku je každé návestie umiestnené do vlastnej ekvivalenčnej triedy, ktoré budú postupne zlučované.

Konštrukcia množiny návestí pre registre a pamäťové lokácie

Pre určenie ekvivalencií návestí je použitý takzvaný interferenčný graf (anglicky *interference graph*) na registroch a jednotlivých pamäťových prístupoch.

Nech je $LS_{in}(obj, n)$ disjunktná množina návestí prislúchajúcich objektu obj v bode priamo pred n-tým riadkom programu. Podobne nech je $LS_{out}(obj, n)$ množina návestí prislúchajúcich objektu v bode priamo za n-tým riadkom. LS_{in} bude ďalej značená skrátene len ako LS. Množina LS je zostrojená pre každý register/pamäťovú lokáciu a riadok programu.

Konštrukcia množiny ekvivalentných návestí

Ekvivalenčné relácie medzi návestiami sú definované nasledovne:

$$\frac{\exists l_1, l_2, obj, n \ : \ \{l_1, l_2\} \subseteq LS(obj, n)}{l_1 \equiv l_2} \tag{4.22}$$

$$\frac{l_1 : (b_1, o_1, C_{1,1}, \ldots, C_{1,n}),\ l2 : (b_2, o_2, C_{2,1}, \ldots, C_{2,n}),\ b_1 \equiv b_2}{l_1 \equiv l_2} \tag{4.23}$$

Konštanty $C_{1,i}$ a $C_{2,i}$ nemusia byť rovnaké. Relácie sú reflexívne, tranzitívne a symetrické, takže návestia sú rozdelené do ekvivalenčných tried. Ľubovoľné návestie v triede reprezentuje datový typ celej triedy.

Prvá definícia 4.22 vraví, že dve návestia l_1, l_2 sú považované za ekvivalentné ak existuje taký riadok programu, pred ktorým je jeden objekt použitý k prístupu k obom návestiam. Druhá definícia 4.23 považuje návestia za ekvivalentné, ak sú ich bázové adresy ekvivalentné.

Konštrukcia agregovaných množín

Nech t_j reprezentuje j-tu ekvivalenčnú triedu. Pre každú operáciu prístupu do pamäte $(b_i, o_i, C_{i,1}, \ldots, C_{i,n})$ môže byť b_i nahradené ľubovoľným zástupcom ekvivalenčnej triedy t_j.

Relácia agregácie polí dvoch návestí označená: $AA(l_1, l_2)$ je definovaná ako:

$$\begin{gathered} l_1 = \text{addr}(t_1, o_1, sum(mul(C), m_1)) \\ l_2 = \text{addr}(t_1, o_2, sum(mul(C), m_2)) \\ \frac{(l_1, l_2)\ |o_1 - o_2| < C}{AA(l_1, l_2)} \end{gathered} \tag{4.24}$$

Relácia AA nemusí byť vždy tranzitívna, teda existujú návestia l_1, l_2 patriace do jednej triedy pre ktoré platí: $|o_1 - o_2| \geq C$. V takomto prípade ju nazývame *konfliktná* a pre odstránenie konfliktov rozdelíme triedu do podtried v ktorých platí $|o_1 - o_2| < C$ a rozdiel minimálnych ofsetov položiek dvoch podtried K_1 a K_2 neni menší ako konštanta C:

$$|\min_{K_1}(o) - \min_{K_2}(o)| > C \tag{4.25}$$

Relácia AA s delením na podtriedy rozkladá množinu návestí na *agregačné množiny*. Nech $K = \{l_1, l_2, \ldots, l_m\}$ je agregačná množina a $o = \min_{j=1}^{m} o_j$. Nech $t' = (b, o)$ je návestie pre poľe zanorených štruktúr, kde (b, o) značí výpočet ofsetu od bázovej adresy bez dereferencovania. Takže ofset o je na začiatku vnorenej štruktúry a ostatné ofsety v množine K sú prepočítané tak, aby boli relatívne k o:

$$l_i : (b, o_i, C_{i,1}, \ldots, C_{i,n}) \in K \tag{4.26}$$

$$l'_i : (b, o_i - o, C_{i,1}, \ldots, C_{i,n}) \tag{4.27}$$

Pretože boli návestia agregované do poľa struktúr, konštanta $C_{1,1}$ je odstránená a množina K' vyzerá nasledovne:

$$l'_i : (t', o_i - o, C_{i,2}, \ldots, C_{i,m}) \tag{4.28}$$

Pretože sa kvoli odstráneniu $C_{1,1}$ zmenila množina návestí, môžeme zostrojiť novú množinu agregačných množín. Proces sa opakuje kým je možné vykonať agregačný krok.

Rekonštrukcia objektov

Pomocou informácií získaných v predchádzajúcich krokoch môžu byť rekonštruované zložené objekty. Nech S je množina všetkých návestí, rozložená do ekvivalenčných tried $S_1, S_2, \ldots, S_k$ indukovaných reláciou: *mať spoločnú bázovú adresu*.

Nech $S_i = \{\text{addr}(b_{i,1}, o_{i,1}, m_{i,1}), \ldots, \text{addr}(b_{i,n}, o_{i,n}, m_{i,n})\}$.

Nech $O_i\{o_{i,1}, \ldots, o_{i,n}\}$ je množina ofsetov od spoločnej báze v S_i. Táto množina bude považovaná za množinu ofsetov na elementy nového zloženého datového typu t_i.

- Ak $O_i = 0$, potom adresový výraz odpovedá dereferencii jednoduchého ukazovateľa alebo prístupu k jednoduchému poľu – poľu jednoduchých datových typov.
- Inak je zostrojený nový datový typ T_i typu štruktúra s prvkami odpovedajúcimi ofsetom z množiny O_i. Typy prvkov zatiaľ nie sú známe, ale príslušné objekty sú pridané do pracovnej množiny analýzy základných datových typov.

Zhodnotenie techniky

Predstavený prístup má niekoľko nedostatkov zhrnutých v nasledujúcom zozname:

- Únie a explicitne pretypované ukazovateľe niesú korektne rekonštruované.
- Prístup k objektom po bajtoch nie je správne zvládnutý.
- Hranice polí sú odvodené len v ojedinelých prípadoch.
- Vnorené štruktúry sú rekonštruované len v prípade, že tvoria polia.

Kapitola 5

Návrh analýzy

Predchádzajúca kapitola predstavila niekoľko existujúcich techník rekonštrukcie jednoduchých a zložených datových typov založených na rovnakých základoch. Okrem rozdielov na abstraktnej úrovni reprezentácie a riešenia typových obmedzení sa od seba odlišovali aj na úrovni praktickej. Každý z nich bol len súčasťou väčšieho dekompilačného systému analyzujúceho súbory pre rôzne vstupné architektúry reprezentované rôznymi vnútornými kódmi. Mali tak k dispozicii rôzne výstupy iných analýz a ďalších komponentov systému.

Cieľom tejto kapitoly je výber najvhodnejšieho prístupu a úprava jeho návrhu tak, aby mohol byť úspešne zasadený do infraštruktúry spätného prekladaču projektu Lissom zo všetkými jeho špecifikami. Zároveň sú tu prezentované niektoré vylepšenia pôvodného frameworku, ktoré pri návrhu vznikli.

5.1 Výber vhodnej techniky

Prvou predstavenou technikou bola typová analýza navrhnutá Alanom Mycroftom pracujúca na princípe unifikácie typových obmedzení odvodených z inštrukcií programu. Táto práca položila základy podobných analýz a všetky nasledujúce prístupy z nej vychádzajú. Ako nové riešenie problému mala ale niekoľko nedostatkov ako napríklad nerozlišovanie znamienok celých čísiel, rýchle šírenie konfliktov a hlavne častú divergentnosť unifikačného algoritmu. Mnohé z nedostatkov sú riešené v novších prácach, ktorých prístupy su tak vhodnejšie na prevzatie.

Michal Van Emmerik sa v svojej dizertačnej práci venuje rekonštrukcii typov skôr prehľadovo – dopodrobna rozoberá rôzne existujúce prístupy a aj keď nakoniec navrhne svoj vlastný algoritmus, jeho popis je stále pomerne abstraktný. Nie je tu presne špecifikovaná reprezentácia typov, rovníc, pravidiel alebo krokov výpočtu. Zároveň sa ešte stále jedná o riešenie staršie ako ďalšie dva príspevky.

Technika TIE je najnovšou z predstavených. Jedná sa o silné riešenie zasadené do robustného reverzného nástroja. Práve tento fakt je ale prekážkou jeho možného prevzatia a úprave. Analýza totiž využíva množstvo ďalších komponentov systému, pracuje nad špeciálne navrhnutou internou reprezentáciou a jej princípy sú snáď až príliš zložité a neprehľadné.

Navrhnutá a implementovaná analýza typov pre projekt Lissom je tak založená na analýze toku dat popísanej v [14]. Jedná sa o pomerne nový článok, definujúci jednoduchý a priamočiary algoritmus rovnakej sily ako TIE. Okrem samotného matematického frameworku sa podrobne zaoberá aj implementáciou reprezentácie typov, obmedzení a algoritmu rieše-

nia. Obsahuje taktiež niekoľko príkladov použitia techniky na reálnom vstupe. Ďalšou výhodou je, že okrem naväzujúceho článku [28] podrobnejšie sa zaoberajúceho zloženými typmi, publikovali autori aj ďalšie príspevky s podobnou tématikou. Publikácia [27] prezentuje niekoľko techník založených na profilovaní dekompilovanej aplikácie pomáhajúcich presnejšej detekcii zložených typov. Jedná sa napríklad o rozlíšenie celých čísiel od ukazovateľov pomocou profilovania hodnôt a identifikáciu zložených typov pomocou profilovania haldy. Článok [17] predstavuje metódu rekonštrukcie hierarchie tried pri spätnom preklade C++ programov. Ak sú v programe prítomné informácie o type za behu (RTTI – *Run-Time Type Information*), hierarchia je odvodená analýzou RTTI štruktúr. V opačnom prípade sa použije technika skúmajúca tabuľky virtuálnych funkcií. Posledný príspevok [26] pojednáva o zlepšení kvality rekonštruovaných datových typov pomocou informácií získaných pri spustení analyzovaného programu. Jedná sa teda o využitie dynamickej analýzy pri spätnom preklade.

5.2 Rekonštrukcia jednoduchých datových typov

Technika popísaná v podkapitole 4.1.5 bola zjednodušená a upravená pre nasadenie v prostredí projektu Lissom. Medzi úpravy patrí využívanie informácií získaných ostatnými analýzami, nasadenie algoritmu na LLVM IR medzikóde, počiatočná nad-aproximácia typov a následný pohyb len v smere k infímu typového zväzu, zavedenie lenivých pravidiel a ďalšie. Tieto zmeny sú prezentované v nasledujúcich podkapitolách.

5.2.1 Zdroje informácií o jednoduchých datových typoch

Zo štyroch druhov obmedzení implikujúcich datové typy sú použité len tri: obmedzenia inštrukcií sú hlavným, obmedzenia príznakov doplnkovým a prípadne obmedzenia prostredia veľmi presným zdrojom nutných informácií. Keďže má spätný prekladač projektu Lissom ambíciu byť skutočne rekonfigurovateľný – schopný dekompilovať programy pre ľubovoľnú architektúru, ktorej model je k dispozícii, nemôže sa spoliehať na obmedzenia registrov. Niektoré architektúry totiž na podobné pravidlá neberú ohľad a veľmi ľahko sa môže stať, že bude celé číslo uložené do desatinného registru a naopak. Pri striktných obmedzeniach by v takýchto prípadoch nastalo množstvo konfliktov a efektívna propagácia by nebola možná.

5.2.2 Spoločný základ

Pôvodná aj novo navrhnutá analýza majú rovnaký spoločný základ. Cieľom je každému objektu (register, globálna/lokálna premenná, návratová hodnota a parametre funkcie) priradiť datový typ T čo najbližší skutočnému. Typ je reprezentovaný trojicou atribútov *core, size, sign* a manipuluje sa s ním pomocou operácie *join*. Objekt o počas analýzy nemá len jeden typ, ale každý jeho výskyt dostane typ vlastný: $T_o = \{T_{o1}, \ldots, T_{oN}\}$. Výskyty rôznych objektov sú medzi sebou previazané propagačnými rovnicami vytvorenými na základe inštrukcií programu. S každým druhom rovnice je asociovaná množina pravidiel použitá k riešeniu systému – propagácii informácií. Po dosiahnutí pevného bodu algoritmus objektom vyberie odpovedajúce výsledné datové typy jazyka C a ukončí svoju činnosť.

5.2.3 Rozdelenie programu na funkcie

Analýza datových typov je súčasťou prednej časti spatného prekladača. Je zaradená takmer na jej koniec a v čase jej behu už bolo nad LLVM IR vykonaných množstvo analýz, ktorých

výsledky môže využiť. Jedná sa napríklad o informácie o riadiacom toku v programe, statickom kóde, mŕtvom kóde, lokálnych a globálnych premennách alebo funkciách. Niektoré z nich len zbierajú informácie o vstupnej aplikácii, iné aj modifikujú medzikód.

Detekcia a analýza funkcií a ich volaní je jedna z tých, ktoré upravujú priebežné LLVM IR. Po jej skončení sú do IR pridané definície detekovaných funkcií a sú s nimi asociované odpovedajúce inštrukcie. Zároveň sú niektoré skoky nahradené inštrukciami reprezentujúcimi volania funkcií a iné inštrukciami návratu. Prípadné parametre sú prepojené s registrami a k funkciám sú pomocou analýzy zásobníku priradené ich lokálne premenné. Výsledné IR je tak bohatšie ako jednoduchý strojový kód. Typová analýza môže na jednej strane dodatočné informácie využiť k svojmu prospechu, na strane druhej musí rekonštruovať aj typy funkcií a parametrov. Príklad LLVM IR s ktorým sa musí rekonštrukcia vysporiadať je zobrazený na zdrojovom kóde 5.1. Významné objekty vyžadujúce ďalšie spracovanie sú zvýraznené červenou farbou.

```
define i32 @fnc1(i32 %arg1)
    store i32 %arg1, i32* @gpregs1
...
    %arg0 = load i32* @gpregs1
    ...
    %argN = load i32* @gpregsN
    %res3 = call i32 @fnc2(i32 %arg0, ..., i32 %argN)
    store i32 %res3, i32* @gpregs2
...
    %res_1 = load i32* @gpregs1
    ret i32 %res_1
    ...
    %res_2 = load i32* @gpregs2
    ret i32 %res_2
```

Obrázok 5.1: Príklad LLVM IR kódu po vykonaní analýzy funkcií.

Rekonštrukcia berie do úvahy a ďalej rieši len zo vstupného bodu dosiahnuteľné a nestaticky (z detekovaných štandardných knižníc) pripojené funkcie. Množina všetkých inštrukcií je rozdelená na podmnožiny náležiace príslušným funkciám. Analýza teda nebude bežať naraz na celom programe, ale vždy len v rámci príslušnej procedúry. Okrem propagačných rovníc vytvorených neskôr z inštrukcií sú vytvorené aj rovnice zaznamenávajúce vzťahy medzi funkciami a ich objektami:

- Rovnice spájajúce argumenty funkcií s prvým použitím objektu v ktorom sa nachádzajú. Jedná sa o takzvané rovnice rovnosti – neobsahujú operáciu, len reprezentujú ekvivalenciu typov:

$$T_{fnc1Arg1} \Leftrightarrow T_{@gpregs1} \tag{5.1}$$

- Rovnice zapájajúce do propagácie návratové hodnoty:

$$T_{\%res\_1} \Leftrightarrow T_{fnc1RetObj} \tag{5.2}$$

$$T_{\%res\_2} \Leftrightarrow T_{fnc1RetObj} \tag{5.3}$$

- Rovnice reprezentujúce volania funkcií pomocou operácie CALL:

$$T_{@gpregs2} \Leftrightarrow CALL \;\; T_{@gpregs1} \;\; \ldots T_{@gpregsN} \tag{5.4}$$

V prípade, že algoritmus pri propagácii od operandov k výsledku narazí na rovnicu s operáciou CALL vykoná nasledujúce kroky:

1. Spojenie objektov v ktorých sa predávajú argumenty so skutočnými objektami reprezentujúcimi argumenty funkcie:

$$(T_{@gpregs1} \sqcup T_{fnc2Arg1}) \ldots (T_{@gpregsN} \sqcup T_{fnc2ArgN}) \tag{5.5}$$

2. V prípade, že spojenie zmenilo typ aspoň jedného argumentu existuje potenciál aj na zmenu ostatných objektov vo volanej funkcii. Vtedy je na ňu algoritmus spustený a funkcia preriešená.

3. Spojenie výsledku funkcie s odpovedajúcim objektom: $T_{@gpregs2} \sqcup T_{fnc2RetObj}$.

Pri opačnej propagácií od výsledku k operandom je poradie krokov obrátené:

1. Spojenie výsledku funkcie s odpovedajúcim objektom: $T_{@gpregs2} \sqcup T_{fnc2RetObj}$.

2. V prípade zmeny návratového typu preriešenie volanej funkcie.

3. Spojenie objektov v ktorých sa predávajú argumenty so skutočnými objektami reprezentujúcimi argumenty funkcie:

$$(T_{@gpregs1} \sqcup T_{fnc2Arg1}) \ldots (T_{@gpregsN} \sqcup T_{fnc2ArgN}) \tag{5.6}$$

Takto upravený algoritmus rekonštruuje aj návratový typ a všetky parametre dosiahnuteľných funkcií. Je navyše efektívnejší ako pôvodná verzia. Nielenže nestráca čas s riešením nedosiahnuteľných a staticky pripojených funkcií, ale aj samotné spracovanie programu po častiach je výhodnejšie. Tam kde by originálna technika pri akejkoľvek zmene riešila všetky inštrukcie znova, je tu ovplyvnený len istý blok.

S volaniami suvisia aj obmedzenia prostredia. Pri narazení na inštrukciu volania procedúry algoritmus skontroluje, či sa nejedná o známu knižničnú funkciu. Ak áno, využije databázu deklarácií takýchto funkcií k presnému určeniu datových typov parametrov a návratovej hodnoty. V tomto prípade nie sú vytvorené propagačné rovnice, ale len priamo odvodené typy príslušných objektov.

5.2.4 Prvý priechod

V nasledujúcom kroku je na na funkciach a ich inštrukciach vykonaný takzvaný prvý priechod. Ten sekvenčne iteruje nad množinou príkazov a vytvára objekty a propagačné rovnice. Každá inštrukcia je preskúmaná a v prípade, že pracuje so zatiaľ neznámym objektom je preň vytvorená nová štruktúra obsahujúca jeho hlavný typ a zároveň inicializovaný prvý výskyt. Ak je objekt už známy, len sa do štruktúry pridá ďalší výskyt. Táto inicializácia sa od originálu odlišuje v dvoch bodoch: 1) Propagačná rovnica je vytvorená len pre tie inštrukcie, cez ktoré je naozaj možné šírenie typovej informácie – na základe typov operandov sa dá vyvodiť záver o type výsledku a naopak. 2) Počiatočná inicializácia typov výskytov je špecifická pre každý typ inštrukcie. Datový typ výskytu je nadaproximovaný na základe sémantiky operácie na všetky prípustné možnosti. Ďalší priebeh analýzy už bude aproximáciu len zužovať odstraňovaním niektorých atribútov, nikdy sa už ale do odhadu nič nepridá. Na zväze datových typov si to môžeme predstaviť tak, že počiatočný odhad sa

nachádza niekde v hornej časti okolo supréma, pričom presná pozícia je pre každú inštrukciu špecifická. Cieľom ďalšieho postupu je priblíženie odhadu čo najviac k infímu, ale bez jeho skutočného dosiahnutia. To by totiž znamenalo konflikt.

Výstupom priechodu je teda zoznam objektov, kde sa každý skladá zo zoznamu datových typov svojich výskytov a zoznam rovníc, prepájajúcich objekty medzi sebou. Takéto zoznamy sú vytvorené pre každú spracovávanú funkciu, ktorá sa stane ich vlastníkom. Každá procedúra ma naviac prístup k spoločnému zoznamu globálnych objektov. Celý proces bude demonštrovaný na príklade spracovania inštrukcie celočíselného sčítania v architektúre MIPS (zdrojový kód 5.2):

```
; add v0, v0, 1234
    %u1 = add i32 1234, 0
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1, %u2
    store i32 %u3, i32* @gpregs0
```

Obrázok 5.2: Príklad LLVM IR kódu pre inštrukciu celočíselného sčítania. Registre `v0, v1` sú tu značené ako globálne premenné `@gpregs0, @gpregs1`.

- `%u1 = add i32 1234, 0`
  Inštrukcia vypočíta výraz (1234 + 0) a výsledok uloží do pomocnej premennej `%u1`. Pretože sa jedná o jej prvý výskyt, je vytvorená nová štruktúra reprezentujúca objekt. Cez túto inštrukciu nie je možná typová propagácia, nakoľko sa na ľavej strane nenachádza žiadny objekt. Datový typ použitia je teda možné odvodiť priamo. V nasledujúcich inicializáciach a rovniciach sú typy objektov indexované poradím ich výskytov. Napríklad T_{u1_0} znamená datový typ prvého použitia objektu `%u1`.

$$T_{u1\_0} \leftarrow (\{int\}, \{32\}, \{signed, unsigned\}) \tag{5.7}$$

- `%u2 = load i32* @gpregs0`
  Inštrukcia načíta obsah registra `@gpregs0` do dočasnej premennej `%u2`. Predpokladajme, že sa v oboch prípadoch jedná o prvý výskyt týchto objektov. Sú teda vytvorené dve nove štruktúry a do nich priradené typy prvých výskytov odvodené z inštrukcie. Keďže je obsah registra len prenesený do premennej, môžeme prehlásiť, že sú ich datové typy rovnaké. Pretože z inštrukcie môžeme odvodiť len veľkosť jej operandov, sú ostatné atribúty nastavené na všetky možnosti. Vytvorená propagačná rovnica implikuje zhodnosť typov oboch výskytov týchto objektov.

$$T_{u2\_0} \leftarrow T_{gpregs0\_0} \leftarrow (\{integer, pointer, float\}, \{32\}, \{signed, unsigned\}) \tag{5.8}$$

$$T_{u2\_0} \Leftrightarrow T_{gpregs0\_0} \tag{5.9}$$

- `%u3 = add i32 %u1,  %u2`
  Inštrukcia celočíselne sčíta dočasné premenne `%u1`, `%u2` a výsledok uloží do `%u3`. Predpokladajme, že sa jedná o druhé výskyty operandov a prvý výskyt výsledku. Pre `%u3` sa teda vytvorí nová štruktúra. Pre `%u1`, `%u2` sa len pridajú ďalšie použitia. Z inštrukcie môžeme odvodiť, že sa jedná o 32 bitové sčítanie vykonané na celých číslach alebo ukazovateľoch. Je vytvorená propagačná rovnica zaznamenávajúca vzťah medzi objektami.

$$T_{u1\_1} \leftarrow T_{u2\_1} \leftarrow T_{u3\_0} \leftarrow (\{int, pointer\}, \{32\}, \{signed, unsigned\}) \tag{5.10}$$

$$T_{u3\_0} \Leftrightarrow T_{u1\_1}\ ADD\ T_{u2\_1} \tag{5.11}$$

- `store i32 %u3, i32* @gpregs0`
  Inštrukcia uloží obsah pomocnej premennej do registra. Jej spracovanie je analogické k načítaniu z registra. O typoch môžeme opať povedať len to, že majú tridsaťdva bitov a sú ekvivalentné.

$$T_{u3\_1} \leftarrow T_{gpregs0\_1} \leftarrow (\{integer, pointer, float\}, \{32\}, \{signed, unsigned\}) \tag{5.12}$$

$$T_{u3\_1} \Leftrightarrow T_{gpregs0\_1} \tag{5.13}$$

5.2.5 Propagačné pravidlá

Potom čo prvý priechod vytvoril zoznamy objektov a propagačných rovníc, môžeme pristúpiť k ich riešeniu. To je založené na rovnakom princípe ako v pôvodnom návrhu – riešenie v dvoch smeroch pomocou špecifických pravidiel pre každú jednu operáciu. Pravidlá sú opäť zostavené na základe jazyka C a špecifikácie LLVM IR [7].

Pretože ale prvý priechod nadaproximoval datové typy všetkých výskytov a analýza pracuje na princípe postupného spresňovania toto odhadu, je nutné aby pravidlá z množín atribúty odstraňovali a nie pridávali. Pravidlá sú z tohto dôvodu upravené z aditívnej do subtraktívnej podoby. Na rovniciach 5.14 môžeme vidieť porovnanie starej (5.14a) a novej (5.14b) formy pre operáciu celočíselného sčítania. Prvé nové pravidlo hovoriace, že ak prvý alebo druhý operand nemôže byť celé číslo, potom ani výsledok nemôže byť celé číslo je ekvivalentné s podmienkou, že výsledok je celočíselný len ak sú oba operandy celočíselné. Analogicky druhé pravidlo odstráni ukazovateľ z atribútov výsledku v prípade, ak sa nenachádza v množine atribútov ani jedného operandu.

$$\frac{int \in \tau_{op1}^{core} \wedge int \in \tau_{op2}^{core}}{int \in \tau_{dst}^{core}}, \frac{ptr \in \tau_{op1}^{core} \wedge int \in \tau_{op2}^{core}}{ptr \in \tau_{dst}^{core}}, \frac{int \in \tau_{op1}^{core} \wedge ptr \in \tau_{op2}^{core}}{ptr \in \tau_{dst}^{core}} \tag{5.14a}$$

$$\frac{int \notin \tau_{op1}^{core} \vee int \notin \tau_{op2}^{core}}{int \notin \tau_{dst}^{core}}, \frac{ptr \notin \tau_{op1}^{core} \wedge ptr \notin \tau_{op2}^{core}}{ptr \notin \tau_{dst}^{core}} \tag{5.14b}$$

Podobne ako pre atribút *core* sú upravené aj pravidlá pre *sign*. V pôvodnom návrhu bol nový atribút veľkosti počítaný pomocou vrchného prieniku dvoch *size* množín. Pretože ale všetky[1] LLVM IR inštrukcie operujú na objektoch rovnakých veľkostí, nie je toto naďalej nutné. Propagácia bitových veľkostí objektov teda nie je v novom návrhu vôbec riešená.

5.2.6 Lenivé propagačné pravidlá

Výrazné zjednodušenie propagácie typov od výsledku k operandom bolo dosiahnuté pomocou takzvaných *lenivých pravidiel* (anglicky *lazy rules*), nahradzujúcich alternatívne atribúty používané v originálnom článku. Rovnica 5.15a pripomína navrhované riešenie pre operáciu celočíselného sčítania. Je tu zaznamenaná relácia medzi objektami, z ktorých práve jeden môže byť ukazovateľom. Aspoň jeden z páru musí po operácií *join* patriť do výslednej množiny. V opačnom prípade sú oba explicitne pridané do oboch množín.

[1] Výnimkou sú inštrukcie explicitného zmenšovania alebo zväčšovania bitovej šírky operandov, v ktorých je ale veľkosť operandov jasne zadaná a môže byť priamo odvodená.

Tento prístup je zbytočne náročný a plne nevyužíva možnosti analýzy toku dat. Podobné zaznamenanie vzťahu objektov by bolo nutné v prípade použitia techniky zostavujúcej z inštrukcií množinu formulí, ktoré následne rieši unifikačný algoritmus bez prístupu k pôvodným operáciam. Naproti tomu ale algoritmus data-flow analýzy iteruje cez propagačné rovnice znova a znova a vždy vie, o aký druh inštrukcie sa jedná. Môžeme teda vytvoriť lenivé pravidlá, spustené len v prípade, že môže byť práve jedno z nich aplikované (sú skúšané v určitom poradí). Takto zdržíme rozhodnutie o typoch objektov až do okamihu, v ktorom je možné vyvodiť určitý záver. Je dôležité si uvedomiť, že analýza nestratí na svojej sile. Alternatívne elementy len vyjadrovali vzťah medzi objektami a v prípade, že nebolo možné v aktuálnom kroku vykonať rozhodnutie boli opätovne uvažované obe varianty.

Príklad nových pravidiel pre operáciu sčítania je zobrazený na rovnici 5.15b. Vidíme, že alternatívne elementy nie sú potrebné a vystačíme s už zavedenou logikou. Prvé hovorí, že v prípade, ak je typ výsledku jedine ukazovateľ a prvý operand určite nie je ukazovateľ, potom je ukazovateľom určite druhý operand. Ostatné pravidlá sú analogické.

$$\frac{ptr \in \tau^{core}_{dst}}{\tau^{core}_{op1} \leftarrow \tau^{core}_{op1} \cup \{pointer^{dst}_{op1}?[op2]\} \ \wedge \ \tau^{core}_{op2} \leftarrow \tau^{core}_{op2} \cup \{pointer^{dst}_{op2}?[op1]\}} \tag{5.15a}$$

$$\frac{\{ptr\} = \tau^{core}_{dst} \ \wedge \ ptr \notin \tau^{core}_{op1}}{\tau^{core}_{op2} \leftarrow \{ptr\}}, \frac{\{ptr\} = \tau^{core}_{dst} \ \wedge \ ptr \notin \tau^{core}_{op2}}{\tau^{core}_{op1} \leftarrow \{ptr\}}, \frac{\{ptr\} = \tau^{core}_{dst}}{\tau^{core}_{op1} \leftarrow \{int\} \ \wedge \ \tau^{core}_{op2} \leftarrow \{int\}} \tag{5.15b}$$

5.2.7 Riešenie konfliktov

Počas aplikácie pravidiel na propagačné rovnice môže nastať situácia, kedy by po odstránení atribútu alebo vykonaní funkcie *join* ostala množina atribútov prázdna. Ako už bolo spomenuté, jedná sa o konflikt pravdepodobne spôsobený nekonzistentným používaním objektu v programe. Je nutné sa s touto situáciou vhodne vysporiadať, pretože jej neriešenie spôsobí lavínové šírenie konfliktov. Riešením je zabrániť operácii vo vytvorení prázdnej množiny a vyradenie rovnice z ďalšieho spracovania. Po každej operácii s typom ľubovoľného objektu algoritmus skontroluje, či nie je prázdna nejaká významná množina. Ak áno sú atribúty vrátené do stavu pred operáciou a rovnica je označená príznakom signalizujúcim, že už ďalej nemá byť používaná. Množina *core* je významná vždy a musí obsahovať aspoň jeden prvok. Množiny *size* a *sign* môžu byť za určitých okolností prázdne – desatinné čísla nemajú znamienko a ukazovateľ je reprezentovaný len ako $(\{pointer\}, \emptyset, \emptyset)$.

5.2.8 Náhrada DU reťazí

V predchádzajúcich príkladoch sme uvažovali, že objekt unikátne identifikovaný menom má rovnaký datový typ na všetkých miestach programu. Toto tvrdenie platí pre návratové hodnoty a parametre funkcií ako aj pre pomocné premenné (označované ako `%uN`), vďaka ktorým je LLVM IR v SSA forme. Ako už bolo naznačené v kapitole 3.2.3 sú ale mená pomocných premenných vo všetkých mikroinštrukciách vytvorených z pôvodných inštrukcií generované rovnako, bez ich ďalšieho rozlíšenia by v programe existovalo množstvo rôznych objektov so zhodným názvom. Z tohto dôvodu sú v skutočnosti už na úrovni IR od seba odlíšené pomocou sufixu zloženého s adresy inštrukcie v ktorej sa nachádzajú. Tento fakt nebol doteraz uvažovaný z dôvodu zjednodušenia príkladov propagačných rovníc a pravidiel.

Už predstavený kód inštrukcie sčítania (obrázok 5.3 vľavo) teda vyzerá ako na obrázku 5.3 vpravo.

```
; add v0, v0, 1234
   %u1 = add i32 1234, 0
   %u2 = load i32* @gpregs0
   %u3 = add i32 %u1, %u2
   store i32 %u3, i32* @gpregs0
```

```
; add v0, v0, 1234
   %u1_8500 = add i32 1234, 0
   %u2_8500 = load i32* @gpregs0
   %u3_8500 = add i32 %u1_8500, %u2_8500
   store i32 %u3_8500, i32* @gpregs0
```

Obrázok 5.3: Príklad LLVM IR kódu pre inštrukciu celočíselného sčítania `add v0, v0, 1234` na adrese `8500`. Registre `v0, v1` sú tu značené ako globálne premenné `@gpregs0, @gpregs1`. Vľavo je doteraz uvažovaná zjednodušená verzia kódu. Vpravo skutočný kód, v ktorom sú mená pomocných premenných medzi rôznymi inštrukciami odlíšené adresou (zvýraznená červenou farbou).

Opačný prípad nastáva pri registroch, ktoré sú v IR definované ako globálne premenné. O všetkých použitiach registru `regX` môžeme povedať, že sa jedná o jeden a ten istý objekt. Ale nemôžeme tvrdiť, že má vždu rovnaký datový typ. Táto komplikácia bola v pôvodnom návrhu riešená pomocou takzvaných DU reťazí (viz 4.1.5), rozdeľujúcich množinu všetkých použití na podmnožiny s rovnakým datovým typom. V tomto návrhu analýzy pre projekt Lissom je problém riešený v dvoch krokoch. Oba sú však aplikované len na úrovni reprezentácie objektov v analýze datových typov a na skutočnom medzikóde sa neprejavia. Prvý krok rozlíši všetky použitia všetkých registrov spôsobom rovnakým ako u pomocných premenných – pridaním sufixu adresy. Príklad situácie je zobrazený na obrázku 5.4. Na začiatku je register `@gpregs0` použitý na adrese 8000. Potom je o mnoho inštrukcií ďalej do neho zapísaná hodnota, ktorá je neskôr načítaná a použitá.

```
; 8000 - mv v0, v5
   %u0_8000 = load i32* @gpregs5
   store i32 %u0_8000, i32* @gpregs0
...
; 8500 - lw v0, 4(sp)
   %u0_8500 = load i32* %stack_var_4
   store i32 %u0_8500, i32* @gpregs0

; 8504 - lw v1, 8(sp)
   %u0_8504 = load i32* %stack_var_8
   store i32 %u0_8504, i32* @gpregs1

; 8508 - add v0, v0, v1
   %u0_8508 = load i32* @gpregs0
   %u1_8508 = load i32* @gpregs1
   %u2_8508 = add i32 %u0_8508, %u1_8508
   store i32 %u2_8508, i32* @gpregs0
```

```
; 8000 - mv v0, v5
   %u0_8000 = load i32* @gpregs5_8000
   store i32 %u0_8000, i32* @gpregs0_8000
...
; 8500 - lw v0, 4(sp)
   %u0_8500 = load i32* %stack_var_4
   store i32 %u0_8500, i32* @gpregs0_8500

; 8504 - lw v1, 8(sp)
   %u0_8504 = load i32* %stack_var_8
   store i32 %u0_8504, i32* @gpregs1_8504

; 8508 - add v0, v0, v1
   %u0_8508 = load i32* @gpregs0_8508
   %u1_8508 = load i32* @gpregs1_8508
   %u2_8508 = add i32 %u0_8508, %u1_8508
   store i32 %u2_8508, i32* @gpregs0_8508
```

Obrázok 5.4: Príklad problému použitia niektorých objektov na viacerých miestach. Na ľavej strane sú červenou farbou zvýraznené výskyty registru, ktoré nemusia nutne mať rovnaký datový typ. Na pravej strane je kód po aplikácií riešenia problému – pridané adresy sú zvýraznené modrou farbou.

Prvý krok rozlíšil všetky použitia, ktoré nemusia mať rovnaký typ. Zároveň ale porušil propagáciu informácií medzi inštrukciami fungujúcu v originálnom kóde. Ak sa bližšie pozrieme na kód na obrázku 5.5, vidíme že po uložení hodnoty do registru `@gpregs0` na adrese

8500 nasleduje jej bezprostredné načítanie a použitie na 8508. Objekty `@gpregs0_8500` a `@gpregs0_8508` teda majú rovnaký datový typ a musia byť nejakým spôsobom previazané. Riešením je nájsť všetky takéto objekty a spojiť ich pomocou kopírovacích rovníc značiacich rovnosť typov. Hľadanie je vykonané jednoduchým algoritmom:

1. Iteruj cez všetky inštrukcie programu.
2. Ak inštrukcia zapisuje do registra, nerob nič, pretože jeho typ mohol byť zmenený.
3. Ak inštrukcia číta z registra, nájdi jeho posledné použitie a spoj ich pomocou kopírovacej rovnice.

Pri spätnom prechode inštrukcií v treťom kroku, môže nastať situácia, v ktorej algoritmus narazí na začiatok základného bloku (anglicky *basic block*). Znamená to, že nie je možné jednoznačne určiť predchodcu inštrukcie – môže ich byť niekoľko. V súčasnosti analýza preruší prehľadávanie a spojenie nevykoná. Aj napriek tomu, že je táto jej časť obmedzená len na základové bloky, samotná propagácia funguje aj medzi nimi pomocou lokálnych a globálnych premenných a parametrov a návratovej hodnoty funkcií. Testy vykonané v kapitole 7 ukázali, že aj takéto riešenie je funkčné a schopné presnej rekonštrukcie. Možnosti ako prehľadávať niekoľko vetví programu za účelom nájdenia použitia určitého objektu budú preskúmané pri ďalšej práci na zlepšeniach analýzy.

```
; 8000 - mv v0, v5
    %u0_8000 = load i32* @gpregs5_8000
    store i32 %u0_8000, i32* @gpregs0_8000
...
; 8500 - lw v0, 4(sp)
    %u0_8500 = load i32* %stack_var_4
    store i32 %u0_8500, i32* @gpregs0_8500

; 8504 - lw v1, 8(sp)
    %u0_8504 = load i32* %stack_var_8
    store i32 %u0_8504, i32* @gpregs1_8504

; 8508 - add v0, v0, v1
    %u0_8508 = load i32* @gpregs0_8508
    %u1_8508 = load i32* @gpregs1_8508
    %u2_8508 = add i32 %u0_8508, %u1_8508
    store i32 %u2_8508, i32* @gpregs0_8508
```

$$T_{@gpregs0\_8508} \Leftrightarrow T_{@gpregs0\_8500}$$

$$T_{@gpregs1\_8508} \Leftrightarrow T_{@gpregs1\_8504}$$

Obrázok 5.5: Naľavo príklad problému porušenej propagácie medzi červeno vyznačenými registrami. Napravo riešenie pomocou prepojenia objektov rovnicami.

Posledným detailom je rozlíšenie výskytov objektu v rámci jednej inštrukcie – niekoľkých mikroinštrukcií. V kóde na obrázku 5.5 sa jedná o odlíšenie zápisu do registru `@gpregs0_8508` (zvýraznený zeleným textom) od jeho predchádzajúceho načítania. Ani v tomto prípade totižto nemôžeme povedať, že majú rovnaký datový typ, nakoľko jeho hodnota bola prepísaná. Rozlíšenie je opäť založené na princípe pridania sufixu `-s` (store) každému objektu, do ktorého sa zapisuje. Explicitné prepojenie nie je nutné, pretože je zaistené rovnicami vygenerovanými z mikroinštrukcií.

5.2.9 Znovupoužitie miest na zásobníku

Práve popísaný problém registrov môže postihnúť aj lokálne premenné. Jedná sa o takzvané znovupoužitie miest na zásobníku (anglicky *stack reuse*), pri ktorom je jeden ofset od ukazo-

vateľa na vrchol zásobníku použitý pre niekoľko rôznych premenných. Riešenie je analogické ako pri registroch, ale s tým rozdielom, že na konci prebieha agregácia objektov vytvorených na základe jedného ofsetu. Všetky výskyty s rovnakým rekonštruovaným typom sú zlúčené do jedného tak, aby nedošlo k zbytočnému zvýšeniu počtu lokálnych premenných vo výslednom IR.

5.2.10 Rozhodnutie konečných typov

Po dosiahnutí pevného bodu algoritmus ukončí riešenie systému propagačných rovníc a prejde k premietnutiu výsledkov do medzikódu. Keďže sú rekonštruované objekty aj propagačné rovnice prepojené so skutočnými objektami a mikroinštrukciami, nie je zanesenie zmien na všetky potrebné miesta problematické. Otázkou ale ostáva čo v prípade, že sa nepodarilo presne určiť datový typ istého objektu. Jednou z možností je tvorba únií so všetkými prípustnými variantami typov. Takéto riešenie je ale neprehľadné a pre čitateľnost výsledku skôr škodlivé. Ideálnym riešením by bolo, keby dal spätný prekladač uživateľovi na výber a ten by posúdil, ktorá varianta je najlepšia. Na takúto možnosť ale Lissom dekompilátor nie je momentálne pripravený.

Preto je použité riešenie využívajúce nadradenosť niektorých atribútov. Datový typ *integer* je považovaný za východzí a je priradený objektu v prípade, že sa analýza nevie jednoznačne rozhodnúť. Ak množina *size* obsahuje niekoľko možných veľkostí, je ako finálna vybraná najväčšia z nich. V množine *sign* je nadradený atribút *signed*.

5.2.11 Nový algoritmus

Výsledný algoritmus rekonštrukcie jednoduchých datových typov zobrazený na zdrojovom kóde 5.6 verne kopíruje popísané princípy. Analýza začína iteráciou cez všetky detekované funkcie a v prípade, že je aktuálna funkcia dosiahnuteľná a nepripojená staticky z niektorej štandardnej knižnice, je pre ňu vytvorený objekt v zozname rekonštruovaných funkcií. Ďalej sú jej priradené inštrukcie v nej použité a inicializované argumenty a návratová hodnota. Nasleduje prvý priechod inštrukciami, po ktorom funkcia obsahuje zoznamy objektov a propagačných rovníc. Keďže prvý priechod rozlíšil všetky výskyty objektov, je zavolaná metóda *mergeObjects()* identifikujúca použitia s rovnakými typmi, pre ktoré vytvorí rovnice rovnosti. Na konci sú spojené všetky výskyty každého objektu pomocou úvodného volania metódy *joinAllObjects()*. Tá postupne aplikuje *join* funkciu na výskyty objektu a jeho hlavný typ, slúžiaci ako akumulátor. Jeho výsledná hodnota je nakoniec uložená naspäť do výskytov. Po jej zavolaní sú teda všetky výskyty rovné prieniku svojich hodnôt pred volaním:

$$T_{x_{main}} = \{T_{x_1} \sqcup \cdots \sqcup T_{x_n}\} \tag{5.16}$$

$$T_{x_1} \leftarrow \cdots \leftarrow T_{x_n} \leftarrow T_{x_{main}} \tag{5.17}$$

Po inicializácii funkcií cez ne algoritmus opäť iteruje a postupne ich rieši. Samotné jadro analýzy toku dat je takmer totožné s originálnym návrhom. Systém propagačných rovníc aktuálnej funkcie je v cykle riešený, až kým nie je dosiahnutý pevný bod. Opať je použitá metóda *joinAllObjects()* a následne vykonaná kontrola jej návratovej hodnoty indikujúcej zmenu aspoň jedného typu u aspoň jedného objektu. Ako už bolo spomenuté, samotné propagácie od operandov k výsledku a od výsledku k operandom môžu spustiť opätovné preriešenie ľubovoľnej funkcie.

Posledná iterácia rozhodne výsledné typy a upraví priebežný medzikód.

```
forall f in functions do:
    if (!f.reachable() || f.static())
        continue;
    TypeFnc &tf = typeFunctions.push_back(TypeFnc(f));
    forall i in instructions do:
        if f.belong(i) then tf.addInstr(i)
    tf.initializeArgumentsAndReturn();
    tf.firstPass();
    tf.mergeObjects();
    tf.joinAllObjects();

bool flag = true;
forall tf in typeFunctions do:
    while (flag) do:
        flag = false;
        tf.equations.opsToDstPropagation();
        flag |= tf.joinAllObjects();
        tf.equations.dstToOpsPropagation();
        flag |= tf.joinAllObjects();

forall tf in typeFunctions do:
    tf.objects.decideFinalType();
    tf.objects.toLLVMIR();
```

Obrázok 5.6: Nový návrh algoritmu rekonštrukcie jednoduchých datových typov pre spätný prekladač projektu Lissom.

5.3 Rekonštrukcia zložených datových typov

Podkapitola pojednáva o návrhu rekonštrukcie zložených datových typov pre spätný prekladač projektu Lissom. Predstavená technika je založená na princípoch popísaných v sekcií 4.2.5. Postup kopíruje tri hlavné kroky pôvodného algoritmu: 1) priradenie adresových výrazov pamäťovým prístupom, 2) konštrukcia množín ekvivalentných návestí a agregácia, 3) rekonštrukcia vyšších typov a ich zanesenie do výstupného medzikódu. Zmeny sa dotkli hlavne postupu tvorby pamäťových výrazov a samozrejme spôsobu zaznamenania výsledku do IR. Reprezentácia pamäťových prístupov a metódy agregácie sú na druhej strane takmer totožné s originálnym návrhom.

Navrhnuté techniky v nasledujúcich podkapitolách budú ilustrované na príklade zobrazenom na zdrojovom kóde 5.7. Vidíme deklaráciu dvoch štruktúr, z ktorých prvá je pomerne jednoduchá – obsahuje len tri celočíselné premenné. Členmi druhej sú tri jednoduché elementy rôznych typov, jedno pole desatinných čísiel a pole štruktúr. V programe sú deklarované dve premenné: 1) globálna premenná typu štruktúra, 2) lokálny ukazovateľ na štruktúru. Na rekonštrukciu datových typov týchto dvoch objektov budú úspešne použité metódy navrhnuté v zbytku tejto podkapitoly.

5.3.1 Súčasné a budúce možnosti navrhovanej analýzy

Ako už bolo mnoho krát spomenuté, pri preklade zdrojového súboru do jazyka symbolických inštrukcií je stratených množstvo vysokoúrovňových údajov. Medzi nimi aj informácie o zložených datových typoch, s ktorými sa na úrovni procesoru pracuje oddelene, jeden prvok za druhým. Podkapitola 4.2 medzi iným pojednávala o charakteristikách zložených typov a teoretických možnostiach a obmedzeniach ich rekonštrukcie. Nasledujúci text vymedzuje súčasné ciele analýzy, ktorých dosiahnutie si vytýčila táto diplomová práca. Taktiež sa zaoberá budúcimi rozšíreniami, realizácia ktorých bude náplňou ďalšej práce na projekte

```
struct t
{
    int f1;
    int f2;
    int f3;
};

struct s
{
    int b;
    char c;
    struct t d[4];
    float f;
    float e[4];
};

// Globálna premenná typu štruktúra.
struct sglobal;

int main(void)
{
    // Lokálny ukazovateľ na štruktúru.
    struct s*local = malloc(sizeof(struct s));

    /* Použitie oboch objektov. */

    return 0;
}
```

Obrázok 5.7: Ilustračný príklad pre rekonštrukciu zložených datových typov.

v rámci autorovej dizertačnej práce. Diskutuje ale aj pravdepodobne principiálne nevyriešiteľné obmedzenia analýzy.

Keďže sa k lokálnym premenným na zásobníku pristupuje pomocou ofsetu od ukazovateľa na jeho vrchol, správajú sa všetky ako jedna veľká štruktúra. Tak ako v pôvodnom návrhu, ani tu sa teda nebudeme pokúšať priamo rekonštruovať lokálne štruktúry. To ale neznamená, že sa nedajú odvodiť žiadnym iným spôsobom. V prípade detekcie volania známej funkcie využijeme jej podpis nad odvodenie asociovaných typov. Ak sa jedná o procedúru vracajúcu alebo prijímajúcu štruktúru, je možné túto informáciu využiť k určeniu typu ľubovoľného objektu. Ďalšia možnosť sa naskytne, ak je lokálna štruktúra predávaná medzi funkciami ako ukazovateľ. Znamená to totiž, že niekde v programe budú existovať pamäťové prístupy využívajúce predaný odkaz. Tieto už môžu byť rozpoznané, ukazovateľ identifikovaný ako odkaz na štruktúru a informácia propagovaná až k pôvodnému zásobníkovému objektu. Poslednou možnosťou je využitie prípadných ladiacich informácií. Lokálne polia sú na druhej strane teoreticky priamo identifikovateľné, pomocou techniky podobnej ako pri poliach globálnych. V súčasnosti sa o to ale analýza taktiež nepokúša a zameriava sa na presnú rekonštrukciu založenú na pamäťových prístupoch. V budúcnosti by ale nemalo byť komplikované rozšíriť riešenie aj o takúto funkcionalitu.

Hlavným cieľom práce je rekonštrukcia zložených datových typov pomocou analýzy pamäťových prístupov (load/store operácií). Jedná sa teda o dva základné prípady: 1) prístupy ku globálnym premenným umiestneným v pamäti na konkrétnych adresách, 2) prístupy do dynamicky alokovanej pamäte pomocou ofsetov od hodnoty určitej premennej – ukazovateľa na zložený typ. Keďže je aj k položkám zanorených štruktúr pristupované pomocou ofsetu od začiatku nadradenej štruktúry, nedajú sa takéto konštrukcie bez heuristiky rekonštruo-

vať. Analýza sa o to momentálne ani nebude pokúšať. Vďaka prístupu k poliam pomocou multiplikatívnych operácií, je teoreticky možné rozpoznanie ich neobmedzeného zanorenia. Ak sú položkami polí štruktúry, je dokonca možná aj ich rekonštrukcia. Adresové výrazy prístupov do viacdimenzionálnych polí sú ale výrazne zložitejšie a pre úplnu obecnosť ich rozpoznania bude nutná ďalšia práca. Súčasným cieľom analýzy bude rozpoznanie jednorozmerných polí a ich kombinácií so štruktúrami – maximálne uvažované zanorenie bude pole štruktúr v štruktúre. Zároveň musí byť ale návrh natoľko kvalitný, že umožní budúce zvýšenie sily analýzy prostým rozšírením funkcionality niektorých jej časti – nebude treba jej úplné preprogramovanie.

S dynamicky alokovanými objektami a prístupom k nim úzko súvisia ukazovatele. Práca sa bude primárne zaoberať základnými ukazovateľmi na jednoduché a zložené datové typy a ich propagáciou. Viacúrovňové ukazovatele a rekurzívne štruktúry budú predmetom ďalšieho skúmania a rozšírení.

5.3.2 Adresové výrazy

Analýza zložených datových typov nie je striktne oddelená od analýzy typov jednoduchých, ale pracuje popri nej. Jej prvá fáza je pridružená k prvému priechodu a je spustená pri spracovaní operácií čítania (load) a zapisovania (store) do pamäte (zdrojový kód 5.8). Cieľom prvej fázy je vytvorenie adresového výrazu pre každý prístup. Výraz reprezentuje spôsob výpočtu adresy uloženej v premennej `%ptr` a je ekvivalentný pamäťovému prístupu:

$$\left(b + o \ + \sum_{j=0}^{n} C_j x_j \right) \tag{5.18}$$

```
%val = load <type>* %ptr
store <type> %val, <type>* %ptr
```

Obrázok 5.8: Syntax operácií načítania (load) a zápisu (store) do pamäte.

Výsledná štruktúra výrazu je rovnaká ako v originálnom návrhu: $l_i : (b_i, o_i, C_{i,1}, \ldots, C_{i,n})$ ale spôsob jeho získania je úplne odlišný. Návrh počítal s vykonaním analýzy dosahujúcich definícií, kde každá dostala svoj vlastný výraz. Pomocou operácií nad výrazmi boli vytvorené propagačné pravidlá pre inštrukcie programu. Tie boli aplikované kým nebol dosiahnutý pevný bod. Nakonci bola drvivá väčšina výrazov zahodená a použité boli len tie, ktoré pristupovali k pamäti. Tento postup vyžaduje implementáciu analýzy dosahujúcich definícií, propagačných pravidiel a algoritmu ich riešenia. Keďže sa väčšina výsledkov nepoužije, nie je veľmi efektívny.

Nový postup je podobný spájaniu objektov s rovnakým typom zo sekcie 5.2.8. Aj v tomto prípade budeme postupovať proti smeru inštrukcií, ale namiesto prostého prepojenia dvoch rovnomenných objektov budeme budovať binárny strom reprezentujúci výraz. Strom sa skladá z dvoch typov uzlov: 1) binárnych reprezentujúcich aritmetickú operáciu na podstromoch, 2) unárnych reprezentujúcich konkrétne číslo (adresa, offset, konštanta násobenia) alebo meno symbolu. Pretože sú adresové výrazy typicky počítané len pomocou operácií

sčítania a násobenia, sú toto jediné dve operácie stromom povolené[2]. Na začiatku je vytvorený strom obsahujucúci len jeden uzol – koreň, nesúci meno premennej použitej ako ukazovateľ. Nasleduje spätná iterácia cez inštrukcie, až kým nie je strom v požadovanej podobe, alebo sa nenarazí na začiatok základného bloku[3]. Za požadovanú podobu sa považuje strom, ktorého unárne uzly obsahujú len čísla, symboly lokálnych/globálnych premenných, parametre funkcií alebo dočasné premenné. Registre nie sú žiadúce. Pri iterácií sa pre každú inštrukciu definujúcu premennú overí, či sa táto nenachádza v strome. Ak áno, je odpovedajúci symbolický uzol nahradený stromom vytvoreným z pravej strany definujúcej inštrukcie. Tento proces využíva služby interpretu, ktorého sa zakaždým opýta, či nedokáže vyhodnotiť hodnotu práve pridávanej premennej. Ak je to možné, symbolický uzol je nahradený konkrétnym číslom.

Vytvorené binárne stromy pre prístupy do premenných `global.e[]` a `local->e[]` sú zobrazené na príklade 5.9. Každý riadok tu reprezentuje celý strom v určitom kroku iterácie. Výsledok je zvýraznený na poslednom riadku. Vidíme, že v prvom strome pre prístup do globálnej premennej sa nachádza výrazne veľké číslo značiace bázovú adresu štruktúry v pamäti. V druhom strome je toto nahradené lokálnou premennou `%stack_var_-28` obsahujúcou adresu pamäte pridelenej pri alokácii. Pretože sa v oboch prípadoch jedná o prístup do poľa, je prítomná operácia násobenia.

```
%u3_89005a8
%u1_89005a8 + 4
@gpregs2 + 4
%u2_89005a4 + 4
(%u1_89005a4 + 143750424) + 4
(@gpregs2 + 143750424) + 4
(%u3_89005a0 + 143750424) + 4
((%u1_89005a0 * 4) + 143750424) + 4
((@gpregs3 * 4) + 143750424) + 4
((%u3_8900598 * 4) + 143750424) + 4
(((%u1_8900598 + 14) * 4) + 143750424) + 4
(((@gpregs16 + 14) * 4) + 143750424) + 4
(((%u4_890057c + 14) * 4) + 143750424) + 4




==========================================
(((%stack_var_-32 + 14) * 4) + 143750424) + 4
```

```
%u3_890058c
%u1_890058c + 4
@gpregs2 + 4
%u2_8900588 + 4
(%u1_8900588 + %u0_8900588) + 4
(@gpregs2 + %u0_8900588) + 4
(@gpregs2 + @gpregs3) + 4
(%u3_8900584 + @gpregs3) + 4
((%u1_8900584 * 4) + @gpregs3) + 4
((@gpregs2 * 4) + @gpregs3) + 4
((%u3_8900580 * 4) + @gpregs3) + 4
(((%u1_8900580 + 14) * 4) + @gpregs3) + 4
(((@gpregs16 + 14) * 4) + @gpregs3) + 4
((((@gpregs16 + 14) * 4) + %u4_890057c) + 4
(((@gpregs16 + 14) * 4) + %stack_var_-28) + 4
(((%u4_8900564 + 14) * 4) + %stack_var_-28) + 4
==============================================
(((%stack_var_-32 + 14) * 4) + %stack_var_-28) + 4
```

Obrázok 5.9: Naľavo vytvorenie stromu pre prístup do `global.e[]`.
Napravo pre prístup do `local->e[]`.

Strom prejde po vytvorení radou modifikácií, ktorých cieľom je upraviť ho do rozpoznateľnej podoby. V súčasnosti sa jedná o presunutie unárnych podstromov binárnych uzlov doľava a následné roznásobenie. Cieľom je presunúť multiplikatívne operácie priamo k symbolom a zbaviť sa násobenia dvoch konštánt, ktoré môže byť vyhodnotené. Po úprave nasleduje pokus o rozpoznanie vzoru prístupu. Ak je úspešný je na základe stromu zostavený adresový výraz. O neúspechu sa vypíše varovanie. V budúcnosti bude možné zvýšiť silu riešenia implementáciou nových modifikačných alebo rozpoznávacích metód.

[2]Niektoré ďalšie operácie ako napríklad aritmetický posuv sú na ne transformované. Ak by sa narazilo na doposiaľ nepodporovanú operáciu, je o tom vypísané varovanie upozorňujúce, že je treba rozšíriť funkcionalitu.

[3]Táto časť analýzy je teda momentálne obmedzená len na výrazy vypočítané v jednom základnom bloku. Je ale teoreticky možné, že sa bude výpočet nachádzať v niekoľkých blokoch.

Vytvorené adresové výrazy pre prístupy do premenných `global.e[]` a `local->e[]` sú zobrazené na príklade 5.10. Prvý riadok je výsledok predchádzajúceho kroku – binárny strom. Druhý reprezentuje strom po modifikáciach a tretí výsledný adresový výraz. Ako už bolo naznačené, návestím je v prvom prípade adresa a v druhom symbol – lokálna premenná. Ofset od báze a multiplikatívna komponenta sú v oboch prípadoch rovnaké.

```
(((%stack_var_-32 + 14) * 4) + 143750424) + 4
=============================================
4 + (143750424 + (56 + (4 * %stack_var_-32)))
=============================================
(143750424, 60, 4*%stack_var_-32)
```

```
(((%stack_var_-32 + 14) * 4) + %stack_var_-28) + 4
==================================================
4 + (%stack_var_-28 + (56 + (4 * %stack_var_-32)))
==================================================
(%stack_var_-28, 60, 4*%stack_var_-32)
```

Obrázok 5.10: Naľavo vytvorenie adresového výrazu pre prístup do `global.e[]`. Napravo pre prístup do `local->e[]`.

Keď má algoritmus k dispozícii adresový výraz, môže pristúpiť k tvorbe zložených objektov. Tie sú reprezentované ako množiny adresových výrazov a sú uložené v kontajneroch. Globálne premenné v spoločnom kontajnery mapujúcom bázové adresy na objekty. Lokálne v kontajnery aktuálnej funkcie mapujúcom symboly obsahujúce adresy na objekty. Výraz je vložený do odpovedajúceho kontajneru a v prípade, že sa jedná o prvý výskyt kľúča (adresa/premenná) je vytvorený nový objekt. Inak sa existujúci objekt rozšíri. Rozšírenie môže byť buď o úplne nový výraz, alebo ak rovnaký výraz už existuje tak o ich spojenie. Vkladanie do kontajnerov teda zároveň konštruuje množiny ekvivalentných návestí podľa definície 5.19. Metóda pridania výrazu však v každom prípade vráti jednoduchý typ s výrazom asociovaný. Ten je použitý pre vytvorenie propagačnej rovnice spájajúcej typ elementu zloženého typu s typom cieľového/zdrojového objektu load/store operácie (rovnica 5.20). Objektu obsahujúcemu adresu prístupu je priradený typ ukazovateľ (rovnica 5.21). Týmto spôsobom bude propagácia nad jednoduchými typmi zároveň pojednom riešiť aj typy prvkov kompozitných objektov.

$$\frac{l_1 : (b_1, o_1, C_{1,1}, \ldots, C_{1,n}),\ l2 : (b_2, o_2, C_{2,1}, \ldots, C_{2,n}),\ b_1 \equiv b_2}{l_1 \equiv l_2} \tag{5.19}$$

$$T_{\%val} \Leftrightarrow T_{compositeElement} \tag{5.20}$$

$$T_{\%ptr} \leftarrow (\{pointer\},\ \emptyset,\ \emptyset) \tag{5.21}$$

Výsledné zložené objekty pre globálnu a lokálnu premennú sú zobrazené na príklade 5.11. V hlavičke je klúč, pod ktorým je záznam namapovaný v príslušnom kontajnery. Každý ďalší riadok predstavuje jeden adresový výraz – jeden prvok zloženého typu – jeden jednoduchý typ. Môžeme si všimnúť, že prístupy do polí naľavo majú viac ako jednu multiplikatívnu komponentu. Vznikli totiž spojením dvoch adresových výrazov líšiacich sa len premennou použitou ako index prístupu.

Týmto končí fáza rekonštrukcie zložených datových typov vykonaná počas prvého priechodu. Z pohľadu pôvodného algoritmu boli vykonané kroky vytvorenia adresových výrazov, spracovania pamäťových prístupov, priradenia návestí a konštrukcie množín ekvivalentných návestí. Zároveň bola zaistená rekonštrukcia základných datových typov jednotlivých elementov.

```
******************** 143750424 ********************
(143750424, 0)
(143750424, 4)
(143750424, 8, 12*%stack_var_-32, 12*%stack_var_-16)
(143750424, 12, 12*%stack_var_-32, 12*%stack_var_-16)
(143750424, 16, 12*%stack_var_-32, 12*%stack_var_-16)
(143750424, 56)
(143750424, 60, 4*%stack_var_-32, 4*%stack_var_-16)
```

```
************ %stack_var_-16 ************
(%stack_var_-16, 0)
(%stack_var_-16, 4)
(%stack_var_-16, 8, 12*%stack_var_-24)
(%stack_var_-16, 12, 12*%stack_var_-24)
(%stack_var_-16, 16, 12*%stack_var_-24)
(%stack_var_-16, 56)
(%stack_var_-16, 60, 4*%stack_var_-24)
```

Obrázok 5.11: Naľavo zložený objekt pre premennú `global`. Napravo pre premennú `local`.

5.3.3 Agregácia pamäťových výrazov

Pri porovnaní rekonštruovaných objektov na príklade 5.11 s pôvodnou deklaráciou štruktúry na zdrojovom kóde 5.7 vidíme, že sa položky štruktúry vnorenej v poli javia ako samostatné polia nadradenej štruktúry. Detekované ofsety navyše nesúhlasia s multiplikatívnymi konštantami. Ofset elementu `global.d[?].f1` je 8 a elementu `global.d[?].f2` 12. Multiplikatívna konštanta indikujúca veľkosť jednej položky poľa je ale vo všetkých troch prípadoch 12. To naznačuje, že element prvého poľa v skutočnosti zahrňa aj ďalšie dva adresové výrazy. Práve tento prípad rieši pravidlo agregácie návestí dvoch polí $AA(l_1, l_2)$:

$$\begin{aligned} l_1 &= \text{addr}(t_1, o_1, sum(mul(C), m_1)) \\ l_2 &= \text{addr}(t_1, o_2, sum(mul(C), m_2)) \\ &\frac{(l_1, l_2)\ |o_1 - o_2| < C}{AA(l_1, l_2)} \end{aligned} \tag{5.22}$$

Po dosadení návestí `global.d[?].f1` a `global.d[?].f2` do pravidla:

$$\begin{aligned} l_1 &= \text{addr}(143750424, 8, 12 * \dots) \\ l_2 &= \text{addr}(143750424, 12, 12 * \dots) \\ &\frac{(l_1, l_2)\ |8 - 12| < 12}{AA(l_1, l_2)} \end{aligned} \tag{5.23}$$

Vidíme, že návestia spĺňajú podmienu a môžu byť agregované. Orignálny návrh počítal s vytvorením nového návestia, identifikujúceho vnorený objekt a prepočítaním ofsetov vzhľadom k tejto báze. Nový návrh pracuje podobne. Keďže sú už adresové výrazy zatriedené v objektoch, nie je nutné zachovať ich aktuálnu bázu. Tá je vymazaná a nahradená novou. V prípade jednoduchých elementov štruktury alebo priamo zanorených štruktúr je rovná nule a ofset udáva vzdialenosť od začiatku celého objektu. Ak ale došlo k agregácii, nová báza je rovná pôvodnému ofsetu prvého agregovaného prvku. Ofsety ďalších elementov sú potom relatívne práve k tomuto návestiu, a nie k skutočnému začiatku premennej v pamäti.

Výsledok agregácie pre globálny, respektíve lokálny objekt sú na príklade 5.12. Vidíme, že po tejto fáze sa ich adresové výrazy od seba nijak nelíšia a rozpoznanie globánosti/lokálnosti je vykonané len na základe príslušnosti do kontajneru. Zároveň môžeme povedať, že táto forma umožnuje priamu rekonštrukciu vyšších typov.

S poľami je spojený aj nie vždy riešiteľný problém rekonštrukcie ich veľkosti. Pozrime sa opäť na príklad 5.12. V rámci jednej štruktúry môžeme rekonštruovať hranice polí v prípade,

```
**************** 143750424 ****************
(0, 0)
(0, 4)
(8, 0, 12*%stack_var_-32, 12*%stack_var_-16)
(8, 4, 12*%stack_var_-32, 12*%stack_var_-16)
(8, 8, 12*%stack_var_-32, 12*%stack_var_-16)
(0, 56)
(60, 0, 4*%stack_var_-32, 4*%stack_var_-16)
```

```
************** %stack_var_-16 **************
(0, 0)
(0, 4)
(8, 0, 12*%stack_var_-24)
(8, 4, 12*%stack_var_-24)
(8, 8, 12*%stack_var_-24)
(0, 56)
(60, 0, 4*%stack_var_-24)
```

Obrázok 5.12: Naľavo zložený objekt po agregácii pre premennú `global`. Napravo pre premennú `local`.

ak nie sú zaradené na koniec objektu. Vieme napríklad, že pole `struct t d[4]` začína na ofsete 8 a nasledujúca položka štruktúry na ofsete 56. Z multiplikatívnej konštanty odvodíme veľkosť jednej položky poľa na 12 bajtov. Počet položiek poľa X je teda rovný: $X = (56 - 8)/12 = 4$.

Ak je pole na umiestnené na konci, môžeme sa pokúsiť odhadnúť jeho veľkosť len v prípade, ak sa nachádza v globálnej pamäti. V tej nájdeme nasledujúci objekt umiestnený za aktuálnou premennou a využijeme jeho bázovú adresu ako vrchný odhad hranice poľa. Nie je zaručené, že celá pamäť medzi dvoma adresami skutočne patrí rekonštruovanému poľu alebo dokonca, že sa nám taký objekt vôbec podarí nájsť. Tento postup nie je možné využiť pri objektoch odkazovaných ukazovateľmi, nakoľko nemôžeme určiť ich skutočnú bázovú adresu.

```
(0, 0)                    -> { integer } ; { 32 } ; { signed }
(0, 4)                    -> { integer } ; {  8 } ; { signed }
(8, 0, 12*%stack_var_-32) -> { integer } ; { 32 } ; { signed }
(8, 4, 12*%stack_var_-32) -> { integer } ; { 32 } ; { signed }
(8, 8, 12*%stack_var_-32) -> { integer } ; { 32 } ; { signed }
(0, 56)                   -> {   float } ; { 32 } ; { }
(60, 0, 4*%stack_var_-32) -> {   float } ; { 32 } ; { }
```

Obrázok 5.13: Mapovanie adresových výrazov na jednoduché typy.

5.3.4 Vytvorenie zložených datových typov a objektov

Tvorba typov nasleduje po analýze jednoduchých datových typov previazanej s jednotlivými elementami zložených objektov. Stav pre ilustračný objekt je zobrazený na príklade 5.13. Generovanie typov pre zložené objekty je závislé na ich štruktúre a riadi sa nasledujúcimi pravidlami:

- Globálne objekty:
  - Jednoduchý objekt obsahuje len jeden adresový výraz bez multiplikatívnej komponenty. Tieto prípady boli už rozpoznané analýzou globálnych premenných, ktorá pre ne vytvorila deklarácie a špeciálne inštrukcie prístupu. Rekonštrukcia jednoduchých datových typov im priradí odvodený typ a ich ďalšie spracovanie nie je nutné.
  - Objekt typu pole sa skladá len z výrazov so spoločnou bázou, z ktorých každý obsahuje multiplikatívnu komponentu.

    - Jednoduché pole obsahuje len jeden adresový výraz. Pole je zostavené nad jednoduchým typom.
    - Pole štruktúr obsahuje niekoľko výrazov. Najprv je zostavený typ štruktúry a potom nad ním typ poľa.

    Pretože globálne polia neboli rozpoznané analýzou globálnych premenných, je pre ne vytvorená nová premenná práve zostaveného typu. Typ pole je v LLVM IR generovaný inline – v každej deklarácii premennej a preto nie je nutná agregácia rovnakých finálnych typov.
  - Objekt je typu štruktúra ak nie je jednoduchý ani pole. Je vytvorený nový prázdny typ štruktúra, do ktorého sa postupne pridávajú položky založené na adresových výrazoch. Ak sa narazí na výraz, ktorého báza nie je rovná nule, znamená to, že sa jedná o vnorené pole. Na základe množiny všetkých adresových výrazov so spoločnou bázou je vytvorený typ poľa a ten je následne zaradený ako položka štruktúry.

    Jednoduché položky štruktúr boli analýzou globálnych premenných identifikované ako samostatné premenné. Teraz sa namiesto nich vytvorí jediná globálna premenná práve zostaveného štrukturovaného typu.

    Pri generovaní výsledného IR budú na začiatok programu umiestnené pomenované deklarácie štruktúr. Ich názvy budú ďalej použité pri deklarácii premenných. Pretože si každý objekt typu štruktúra vytvorí nový datový typ, môže sa stať, že medzi deklaráciami bude niekoľko úplne rovnakých štruktúr. Aby sa tomuto predišlo, je pri pridávaní nových štruktúr kontrolovaná zhoda so všetkými už existujúcimi. V prípade nájdenia ekvivalentného typu sa zabráni vytvoreniu nového a použije sa typ starý.

- Lokálne objekty:
  - Jednoduché objekty obsahujúce len jeden výraz bez multiplikácie sú ukazovatele na jednoduchý datový typ.

    Napríklad v: `(%stack_var_-24, 0) -> { float } ; { 32 } ; { }` považujeme lokálnu premennú za ukazovateľ na desatinné číslo. Premenná už v LLVM IR existuje, takže sa len upraví jej typ.
  - Pre lokálny objekt pole sa zostaví datový typ rovnako ako pre globálnu premennú. Rozdiel je v tom, že nie je vytvorený nový objekt tohto typu, ale len aktualizovaná lokálna premenná. Tá je považovaná za ukazovateľ na pole.
  - Pre lokálnu štruktúru je opäť zostavený typ a aktualizovaná zásobníkova premenná.

5.3.5 Úprava LLVM medzikódu

Adresové výrazy, z ktorých sa zložené objekty skladajú boli vytvorené počas prvého priechodu inštrukciami programu. Ako súčasť ich inicializácie teda nebol problém zaznamenať odpovedajúce load/store operácie. Práve tieto sú v poslednej fáze rekonštrukcie zložených datových typov nahradené novými inštrukciami navrhnutými za týmto účelom.

Ako už bolo popísané pri predstavení jazyka LLVM IR v podkapitole 3.2.3, k položkám zložených typov sa pristupuje pomocou spoločnej inštrukcie `getelementptr`. Z tohto dôvodu sa o načítanie/zapisovanie do polí aj štruktúr starajú dve spoločné operácie:

`op_composite_read` a `op_composite_write`. Do oboch sa zaznamená meno načítanej/ukladanej dočasnej premennej, zložený objekt obsahujúci pristupovaný element a množina indexov použitá pre prístup. Nasledujúci text sa bude zaoberať transformáciou týchto operácií na výstupný LLVM IR kód.

Šablóna operácie `op_composite_read` je zobrazená na zdrojovom kóde 5.14. Na začiatku sú načítané všetky symbolické indexy (typicky premenné použité k iterácii cez pole) do dočasných premenných. Ich počet nie je obmedzený a operácia tak môže reprezentovať prístup do ľubovoľne komplexného typu. Nasleduje získanie ukazovateľa na požadovaný prvok pomocou operácie `getelementptr`. Jej parametrami sú zaradom typ zloženého objektu, jeho meno a zoznam indexov prístupu. Ten sa skladá z načítaných dočasných premenných a numerických konštánt (prístup ku konkrétnym prvkom štruktúr). Posledný riadok vykoná načítanie vybraného prvku do cieľovej premennej.

Skutočný vygenerovaný kód pre operáciu načítania prvku `global.d[%stack_var_-16].f2` z ilustračného príkladu 5.7 je zobrazený na zdrojovom kóde 5.15.

```
<sym_idx_1>      = load <sym_idx_1_type>* <symbol_1>
...
<sym_idx_N>      = load <sym_idx_N_type>* <symbol_N>
<composite_elem> = getelementptr inbound <composite_type>* <composite_name>, <index_list>
<dst_var>        = load <dstType>* <compposite_elem>
```

Obrázok 5.14: Šablóna LLVM IR kódu operácie `op_composite_read`.

```
%index_0 = load i32* %stack_var_-16
%struct_elem = getelementptr inbounds %struct.0* @global_struct, i32 0, i32 2, i32 %index_0, i32 1
%u4 = load i32* %struct_elem
```

Obrázok 5.15: Príklad načítania elementu `global.d[%stack_var_-16].f2`.

Šablóna operácie `op_composite_write` je zobrazená na zdrojovom kóde 5.16. Jej prvá časť je totožná s predchádzajúcou. Odlišujú sa len posledným riadkom, na ktorom je namiesto načítania, do vybraného elementu vykonaný zápis obsahu zdrojovej premennej.

Zdrojový kód 5.17 opäť zobrazuje reálny LLVM IR vygenerovaný pomocou predstavenej šablóny.

```
<sym_idx_1>      = load <sym_idx_1_type>* <symbol_1>
...
<sym_idx_N>      = load <sym_idx_N_type>* <symbol_N>
<composite_elem> = getelementptr inbound <composite_type>* <composite_name>, <index_list>
store <src_var_type> <src_var>, <src_var_type>* <composite_elem>
```

Obrázok 5.16: Šablóna LLVM IR kódu operácie `op_composite_write`.

Obe operácie môžu byť použité pre priamy prístup ku globálnym premenným alebo pre nepriamy prístup cez lokálny ukazovateľ. Vyššie predstavené varianty pristupovali ku globálnym objektom. Pri nepriamom prístupe je potrebné najskôr ukazovateľ načítať do dočasnej premennej a tú použiť ako parameter `getelementptr` operácie. Zmena relevantnej časti šablóny je zobrazená na zdrojovom kóde 5.18.

```
%index_0 = load i32* %stack_var_-32
%struct_elem = getelementptr inbounds %struct.0* @global_struct, i32 0, i32 2, i32 %index_0, i32 1
store i32 %u1, i32* %struct_elem
```

Obrázok 5.17: Príklad zápisu elementu `global.d[%stack_var_-32].f2`.

```
<composite_deref> = load <composite_type>* <composite_name>
<composite_elem>  = getelementptr inbound <composite_type> <composite_deref>, <index_list>
```

Obrázok 5.18: Časť šablóny nepriameho prístupu cez ukazovateľ.

Podobne ako pre prístupy do objektov zložených typov, boli navrhnuté a implementované špecifické operácie pre načítanie a ukladanie premenných odkazovaných cez ukazovatele. Jedná sa o dvojicu podobných inštrukcií, ktoré najprv dereferencujú ukazovateľ do pomocnej premennej, a tú následne využijú pre prečítanie alebo uloženie požadovanej hodnoty. Šablóny oboch operácií sú zobrazené na zdrojových kódoch 5.19 a 5.20.

```
<ptr_var_deref> = load <dst_type>** <ptr_var>
<dst_var>       = load <dst_type>* <ptr_var_deref>
```

Obrázok 5.19: Šablóna LLVM IR kódu operácie `op_pointer_read`.

```
<ptr_var_deref> = load <src_type>** <ptr_var>
store <src_type> <src_var>, <src_type>* <ptr_var_deref>
```

Obrázok 5.20: Šablóna LLVM IR kódu operácie `op_pointer_write`.

Kapitola 6

Implementácia

Navrhnutá analýza jednoduchých a zložených datových typov je implementovaná ako súčasť prednej časti spätného prekladača. Je umiestnená takmer na koniec radu vykonávaných analýz tak, aby mala v čase behu prístup k ich výsledkom. Využívané sú hlavne informácie o detekovaných funkciách, toku riadenia a zásobnikových a globálnych premenných. Tak ako celá predná časť, aj rekonštrukcia typov je naprogramovaná v jazyku *C++*, podporujúcom objektovo orientované programovanie. Nakoľko boli funkčné mechanizmy riešenia podrobne vysvetlené v prechádzajúcej kapitole, je nasledujúci text primárne zameraný na jeho zasadenie do objektového návrhu aplikácie.

Poznámka: Predstavené objektové modely tried sú zjednodušené tak, aby zachytávali len podstatné atribúty a metódy. V žiadnom prípade sa nejedná o modely úplné.

6.1 Jadro analýzy

Objektový model rekonštrukcie typov je zobrazený na obrázku 6.1. Celá analýza je reprezentovaná triedou *TypeRecoveryAnalysis*. Tak ako aj ostatné analýzy prednej časti, je aj táto vytvorená pomocou návrhového vzoru singleton. Ten zaistí, že v celej aplikácii bude len jedna jej inštancia, ku ktorej sa bude dať jednoducho pristúpiť pomocou metódy *getInstance()*. Trieda vlastní zoznam dosiahnuteľných definovaných funkcií, zoznam globálnych premenných a zložených objektov.

Metóda *doAnalysis()*, vytvorená podľa algoritmu na zdrojovom kóde 5.6 vykoná celú analýzu. Na začiatku vyberie funkcie na spracovanie, vytvorí pre ne inštancie triedy *RecoveryFunction*, pridá ich do zoznamu, zavolá inicializačné metódy, vyrieši funkcie a prenesie výsledky do IR.

Metóda *initialize()* vytvorí objekty pre návratový typ a argumenty funkcie.

Metóda *firstPass()* prejde inštrukcie a zostaví zoznam objektov a propagačných rovníc. Zároveň sú pri prvom priechode vytvorené aj kontajnery zložených objektov a zoznam globálnych premenných:

- Do *globalVars* sú pridané všetky globálne premenné, s ktorými sa pracuje pomocou špecifických operácií vytvorených pri ich detekcii.
- Do *globalCompositeObjs* sú pridané všetky zložené globálne objekty. Ako kľúč sú použité ich adresy v pamäti.
- Do *localCompositeObjs* sú pridané objekty, na ktoré ukazujú lokálne premenné. Kľúčom sú ich mená.

Trieda *ObjectContainer* reprezentuje zoznam objektov typu *Object*. Okrem bežných operácií nad zoznamom, je možné vykonať nad každým objektom spojenie jeho výskytov *joinAll()*, prepojenie s predchádzajúcim výskytom objektu rovnakého typu *merge()* a jeho konverziu do medzikódu *typesToIR()*.

Objekt má svoje meno, typ (globálna/lokálna premenná, register, návratová hodnota, parameter), odkaz na IR objekt z ktorého vznikol, hlavný datový typ a zoznam typov svojich výskytov. Metódy implementujú spájanie výskytov objektu, rozhodovanie výsledného typu a jeho konverziu do IR.

Trieda *SimpleType* reprezentuje datový typ presne tak, ako bol navrhnutý. Jej kľúčovými členmi sú množiny *core, size, sign* a naviac obsahuje príznak *corrupted* indikujúci, že nad typom bola vykonaná operácia, ktorá by spôsobila konflikt. Takejto operácii bolo zabránené zaniesť zmenu a typ nebude ďalej použitý pri propagácii. Týmto je zabránené lavínovému šíreniu konfliktov. Trieda implementuje veľké množstvo operácií pre jednoduchú modifikáciu množín na mnoho rôznych spôsobov.

Trieda *EquationContainer* poskytuje zapúzdrenie zoznamu propagačných rovníc. Rovnice je možné vytvárať a vykonať nad všetkými propagáciu od operandov k výsledku a od výsledku k operandom.

Samotná propagačná rovnica je reprezentovaná triedou *PropagationEquation*. Obsahuje odkazy na datové typy výskytov, nad ktorými pracuje a odkaz na pôvodnú IR inštrukciu, slúžiacu pre určenie aplikovaného propagačného pravidla. Ako bolo vysvetlené, nie všetky rovnice sú vytvorené na základe inštrukcií – niektoré len prepájajú dva objekty. Im je nastavený príznak *equality* značiaci, že sú si typy operandov a výsledku rovné. Rovnicu je ďalej možné označiť príznakom *solved*, čo zabráni jej ďalšiemu používaniu.

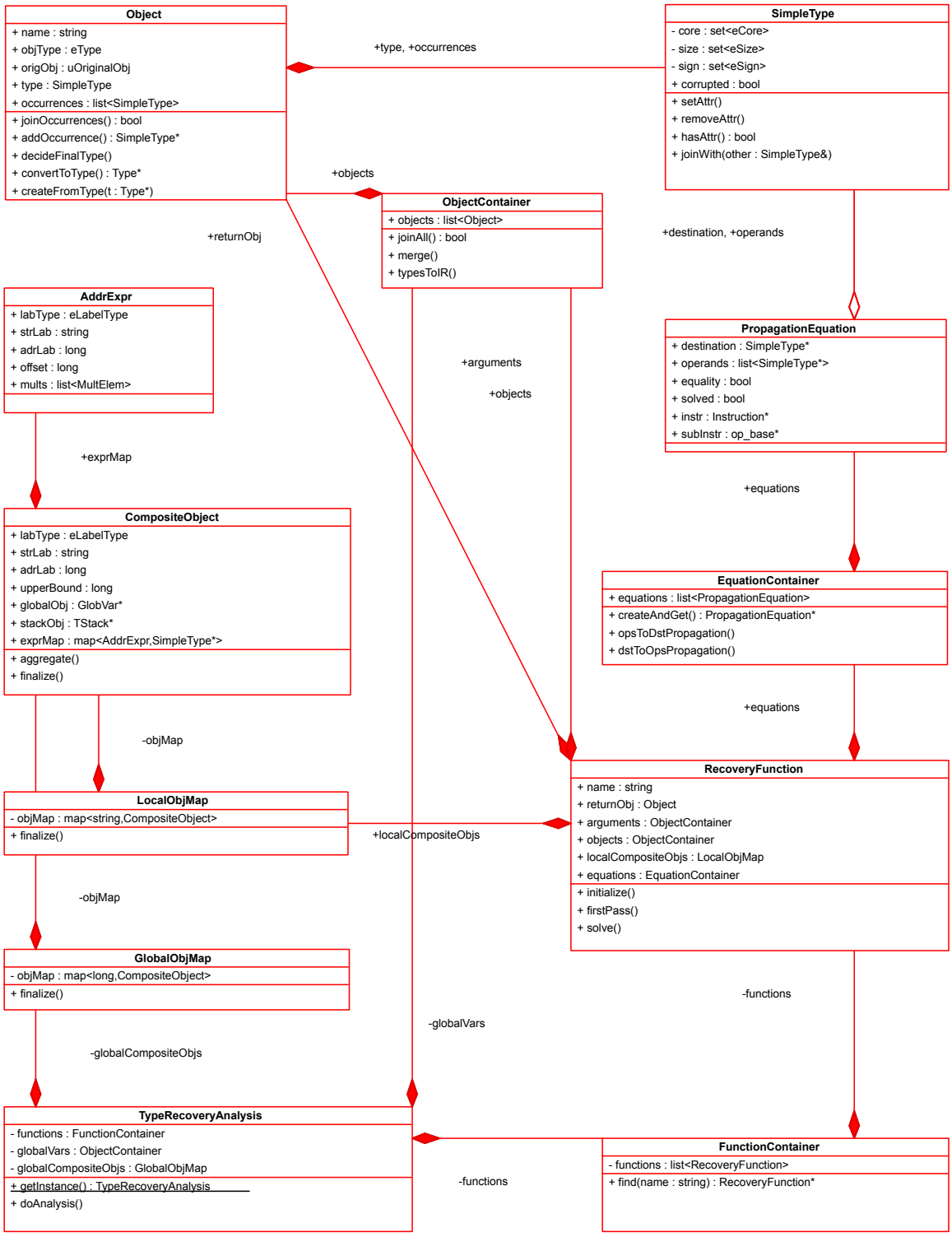
Zložené objekty sú reprezentované triedou *CompositeObject*. Jej jadrom je mapovanie adresových výrazov na jednoduché datové typy. Ďalej ale obsahuje informácie o svojom návestí (typ, symbol/adresa), umiestnení nasledujúceho objektu a odkaz na asociovanú globálnu alebo lokálnu premennú.

Samotný adresový výraz predstavuje trieda *AddrExpr*. Atribútami sú opať informácie o návestí, ofsete a zoznam multiplikatívnych komponentov. Tie môžu byť konštantné alebo symbolické. S každým výrazom sú spojené inštrukcie, na základe ktorých vznikol a ktoré musia byť zmenené pri zanášaní výsledkov do medzikódu.

Po zostavení všetkých týchto tried, je na každú položku zoznamu funkcií zavolaná metóda *solve()*. Jej implementácia odpovedá telu druhého cyklu na zdrojovom kóde 5.6.

Po vyriešení systému propagačných rovníc nasleduje fáza finalizácie typov a prenesenia výsledkov do IR. Metódou *decideFinalType()* sú rozhodnuté typy všetkých jednoduchých objektov a metódou *convertTyType()* sú vytvorené ich IR ekvivalenty. Tie môžu byť následne zapísané do asociovaných IR objektov. Pri jednoduchých typoch nie je treba modifikovať inštrukcie.

Metóda *finalize()* zloženého objektu vykoná agregáciu, určenie hraníc polí a zostaví odpovedajúce komplexné datové typy v IR. Tie sú potom použité pri vytvorení alebo modifikácií odpovedajúcich objektov. Všetky inštrukcie poznamenané v adresových výrazoch sú nahradené novými podľa návrhu v podkapitole 5.3.5.



Obrázok 6.1: Model tried analýzy jednoduchých a zložených datových typov.

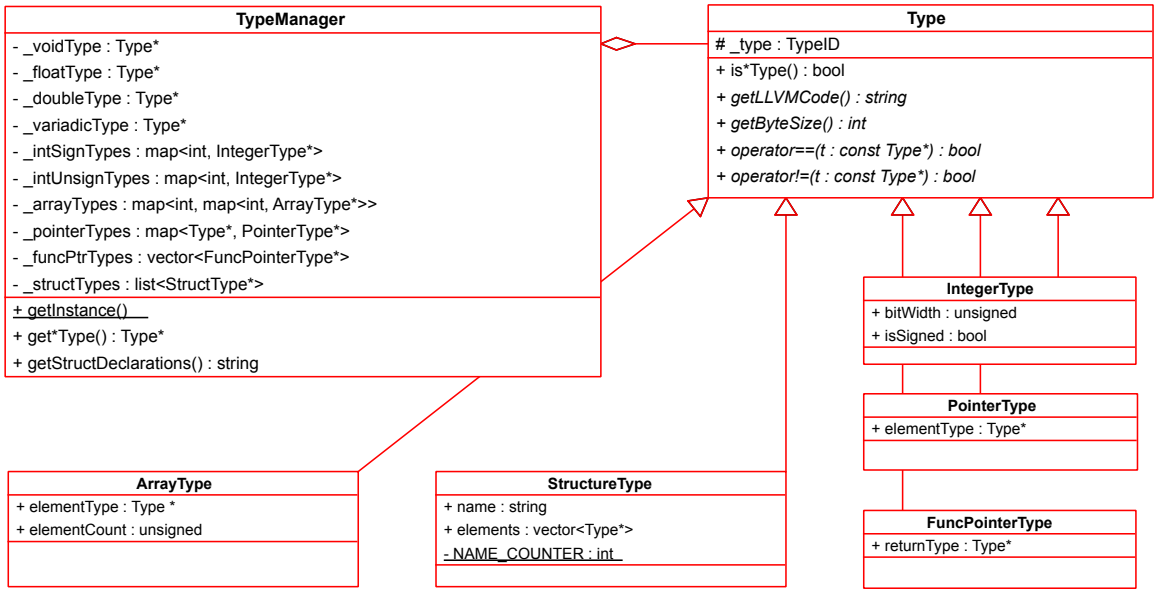
6.2 Typový systém prednej časti

Doposiaľ predstavená reprezentácia datových typov bola navrhnutá za účelom ich postupnej rekonštrukcie a nie je vhodná pre nasadenie v celej prednej časti spätného prekladača. Táto podkapitola predstavuje typový systém používaný inštrukciami a objektami front-endu, do ktorého musia byť rekonštruované typy konvertované.

Poznámka: Autor tejto práce nie je výlučným autorom implementácie typového systému. Jeho veľká časť existovala už pred začiatkom práce na analýze datových typov a v jej rámci bola len mierne rozšírená a upravená.

Keď niektorá z ostatných analýz detekuje nový objekt, ako napríklad globálnu/lokálnu premennú alebo parameter funkcie, vytvorí preň inštanciu odpovedajúcej triedy a upraví inštrukcie s ním pracujúce. Jedným z členov takejto triedy je aj ukazovateľ na typ objektu *Type**. Keďže v tomto čase typicky nie sú dostupné žiadne informácie o jeho skutočnom datovom type, inicializuje sa na typ východzí – celé číslo. Potom čo práve analýza datových typov odvodí pravdepodobný typ, vytvorí jeho reprezentáciu a priradí ju objektu.

Typ je teda reprezentovaný triedou *Type* obsahujúcou identifikátor určujúci, či sa jedná o celé/desatinné číslo, ukazovateľ, pole, štruktúru atď. Desatinné čísla sa reprezentované len na tejto úrovni ako dva rôzne identifikátory *float* a *double*. Pre ostatné možnosti sú z *Type* odvodené nové triedy obsahujúce špecifické informácie o reprezentovaných typoch. Pre celé čísla sa jedná len o bitovú veľkosť a znamienko. Ukazovatele zase obsahujú odkaz na odkazovaný typ. Pole sa skladá z počtu položiek a ich typu. Štruktúra z jej názvu a vektoru odkazov na typy položiek. Ak teda identifikátor určí, že je inštancia *Type* niektorou zo špecifických možností, je nutné ju vhodne pretypovať. Nech sa už jedná o ľubovoľnú triedu, musí implementovať metódy preťažených operátorov rovnosti/nerovnosti a dvojicu *getLLVMCode()*/*getByteSize()* vracajúcu LLVM IR reprezentáciu typu a jeho veľkosť v bajtoch.

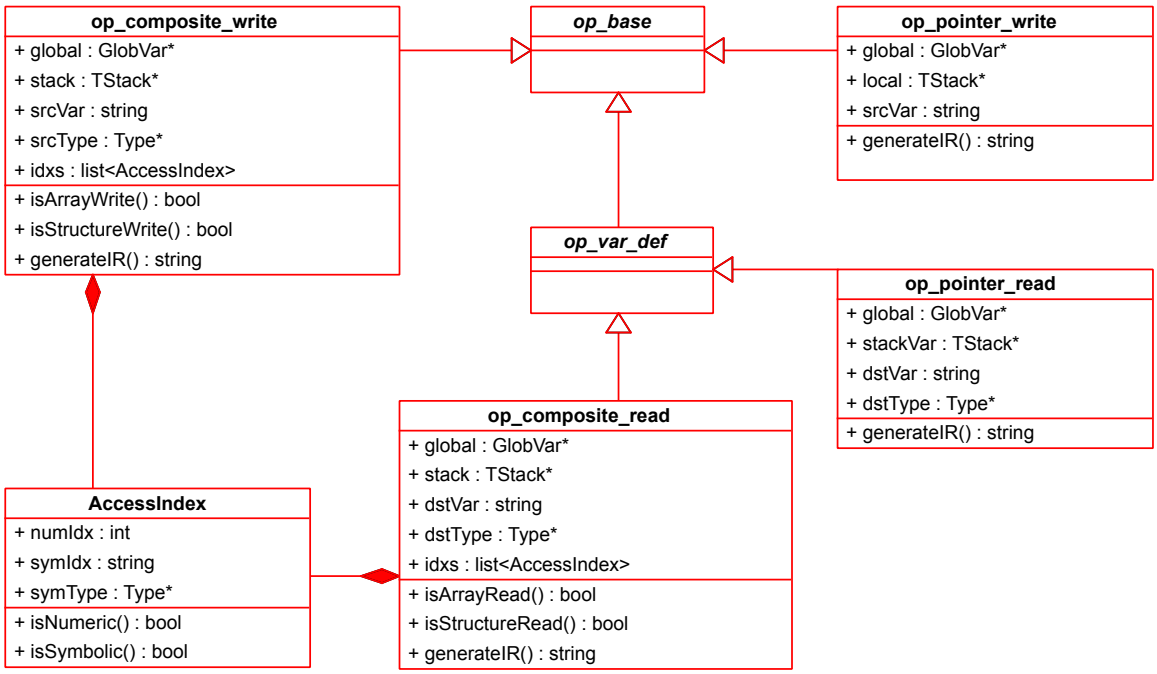


Obrázok 6.2: Model tried typového systému prednej časti spätného prekladača.

Ukazovateľ na požadovanú inštanciu *Type* je získaný pomocou singleton triedy *TypeManager*. Jej cieľom je zaistiť, že bude vytvorená len jedna inštancia z každého použitého datového typu. Za týmto účelom udržiava odkazy na všetky typy namapované v interných kontajneroch, ktorých klúče ich unikátne identifikujú. Klúčom pre znamienkové celé čísla je napríklad ich veľkost, pre ukazovateľe odkazovaný typ, atď. Trieda ďalej poskytuje množinu metód *get*Type()* vracajúcich požadovaný typ. Hviezdička predstavuje niektorú z udržiavaných kategorií. V súčasnosti sa z tohto princípu vymykájú typy štruktúr, ktoré sú udržiavané v jednoduchom zozname a nie je zaistená ich unikátnosť. So štruktúrami suvisí aj metóda *getStructDeclarations()*, vracajúca reťazec obsahujúci LLVM IR deklarácie všetkých existujúcich štrukturovaných typov.

6.3 Nové operácie medzikódu

Poslednou časťou je implementácia operácií medzikódu predstavených v podkapitole 5.3.5. Jedná sa o dve dvojice operácií pre načítanie/zápis z/do zložených objektov alebo ukazovateľov. Dvojica, ktorá zapisuje nedefinuje novú pomocnú premennú a preto je odvodená od základnej triedy operácií *op_base*. Načíta sa vždy do pomocnej premennej a tak je druhá dvojica odvodená od *op_var_def*. Všetky štyri triedy obsahujú ukazovatele na globálne alebo lokálne premenné, s ktorými pracujú – vždy samozrejme len s jednou. Ďalej meno zdrojovej/cieľovej dočasnej premennej a jej typ. Inštrukcie pre zložené objekty majú navyše zoznam indexov triedy *AccessIndex*, použitý ako parameter inštrukcie *getelementptr*. Všetky triedy implementujú metódu *generateIR* podľa šablón z podkapitoly 5.3.5.



Obrázok 6.3: Model triedy nových operácií prístupu k zloženým typom a ukazovateľom.

Kapitola 7

Experimenty

Experimentálne overenie funkčnosti riešenia bolo vykonané ako na jednoduchých príkladoch, tak aj na reálnych programoch. Jednoduché príklady slúžia pre otestovanie konkrétnych schopností analýzy. Reálne pre preverenie jej časovej náročnosti.

Keďže spätný prekladač projektu Lissom má ambíciu byť rekonfigurovateľný, boli experimenty vykonané na celej rade rôznych architektúr, prekladačov a úrovni optimalizácií.

7.1 Experimenty s jednoduchými programami

Jednoduché programy boli navrhnuté za účelom testovania špecifických vlastností rekonštrukcie. Testujú ale len také vlastnosti, ktoré je súčasná analýza principiálne schopná zvládnuť. Neobsahujú teda viacrozmerné polia, priamo zanorené štruktúry, viacúrovňové ukazovatele alebo zložené objekty umiestnené na zásobníku. Pretože sú prezentované výsledky celej dekompilácie na úrovni jazyka C, sú testy zamerané hlavne na rekonštrukciu zložených datových typov a ich propagácie. Rekonštrukcia jednoduchých typov sa viditeľne prejaví len na výstupe prednej časti v LLVM IR kóde. Ako už bolo spomenuté, zadná časť je do určitej miery schopná odvodiť typy aj bez tu predstavenej analýzy. Jej výsledky by teda neboli dobre rozoznateľné. Päť základných príkladov preložených s vypnutými optimalizáciami pre architektúru MIPS je zobrazených v prílohe A. Ostatné sú vytvorené ich postupným upravovaním a rozširovaním. Na obrázkoch v prílohe je vždy na ľavej strane pôvodný kód programu, v strede výstup spätného prekladu bez rekonštrukcie datových typov a napravo po nasadení analýzy. Pretože príklady slúžia ako základ, budú ich výsledky zhodnotené v ďalšom texte.

Príklad A.1 testuje detekciu jednoduchého ukazovateľa a jeho propagáciu do funkcie. Vidíme, že propagácia do parametru a tela funkcie dopadla úspešne. Nepodarilo sa ale na úrovni jazyka C rekonštruovať ukazovateľ v mieste jeho alokácie a to aj napriek tomu, že asociovaná premenná bola v LLVM IR označená ako ukazovateľ na desatinné číslo. Pri optimalizácii bola ale odstránená a ďalej sa pracuje len s premennou obsahujúcou výstup funkcie *malloc()*. Momentálne nie je jasné, či je problém na strane prednej alebo optimalizačnej časti. Situácia bude podrobená ďalšiemu skúmaniu.

Výstup pre príklad A.2 je až na určenie hraníc poľa v poriadku. V pamäti sa nenachádzajú žiadne iné objekty a tak nie je možné určiť jeho maximálnu veľkosť. Testovanie jej určenia v prípade existencie ďalších objektov bude predmetom odvodeného testu. Taktiež bude otestované odvodenie hraníc v prípade, že je pole súčasťou štruktúry, v ktorej sa za ním nachádza nejaká položká.

Príklady A.3 a A.4 sú rekonšturované úplne presne[1].

Posledný príklad A.5 trpí rovnakým nedostatkom ako prvý. Aj napriek presnej rekonštrukcii zložených datových typov a nastaveniu príslušnej premennej sa pracuje s výsledkom alokácie. Na druhej strane je ale vidieť, že analýza nebola zbytočná. Objekt `mem` je pri prístupoch správne pretypovaný a výsledný kód oveľa kvalitnejší ako pôvodný. Takto vyzerajúce výstupy budú teda pri testoch považované za úspešné.

Všetky vykonané jednoduché testy sú zhrnuté v tabuľke 7.1 a výsledky na obrázku 7.2. Každému testu je priradený identifikátor, podľa ktorého sa dá dohľadať na priloženom CD. Nasleduje popis experimentu špecifikujúci, aká vlastnosť bola testovaná a čo by malo byť dosiahnuté. Posledné stĺpce zaznamenávajú úspech (✓) alebo neúspech (✗) pre testované architektúry a úrovne optimalizácií. Ako môžeme vidieť, experimenty boli vykonané pre dve úrovne optimalizácií (*O0, O1*) na týchto platformách:

- MIPS (MIPS I Architecture) – prekladač *psp-gcc 4.3.5* generujúci 32-bitové programy vo formáte ELF.
- ARM – prekladač *arm-elf-gcc 4.1.1* generujúci 32-bitové programy vo formáte ELF.
- x86 – prekladač *gcc 4.7.2* generujúci 32-bitové programy vo formáte ELF.

Z výsledkov vidíme, že je analýza najúspešnejšia na programoch preložených bez optimalizácií pre najjednoduchšiu architektúru MIPS. S väčšou komplexnosťou architektúry a optimalizáciami úspešnosť klesá. Dôvody neúspechu niektorých testov budú krátko popísané v zostávajúcej časti tejto podkapitoly.

Na architektúre ARM sa nepodarilo rekonštruovať typy premenných deklarovaných v pôvodnom súbore ako čísla s plávajúcou desatinnou čiarkou. V použitom modely procesoru totiž chýbajú príslušné registre a operácie. Násobenie dvoch desatinných čísiel je tak preložené na obyčajnú celočíselnú multiplikáciu. Vzhľadom k tomu, že sémantika operácií je primárnym zdrojom informácií pre analýzu, nedá sa v tomto prípade nič robiť.

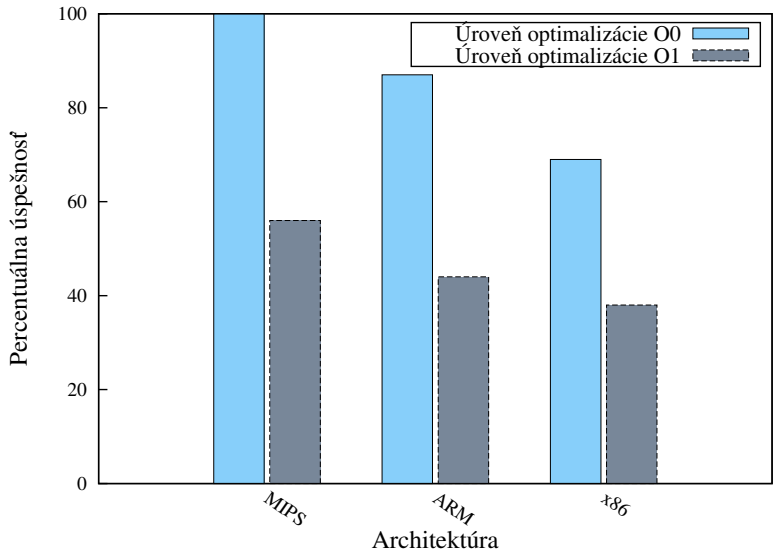
Ďalším architektonicky závislým javom je reprezentácia jednoduchých globálnych štruktúr na architektúre x86. Namiesto prístupov vo formáte $(adresa, ofset)$ totiž dochádza pri preklade k prepočítaniu na priamu adresu. To spôsobí neuspokojivý výsledok analýzy na teste `struct_base` a ďalších založených na globálnych štruktúrach. Pretože sa pri preklade nedá vypočitať adresa dynamicky alokovanej štruktúry, sú v týchto prípadoch adresové prístupy v správnom formáte. To sa prejavilo na úspešnom teste `malloc_struct`.

Najväčší podiel chybných výsledkov majú na svedomí príliš komplikované adresové výrazy, ktoré zatiaľ nie sú rozpoznané alebo ani úplne zostavené. Pri vyšších úrovniach optimalizácie totiž dochádza k výpočtu presahujúcemu aktuálny základný blok a tak nie sú príslušné binárne stromy plne zostavené. Toto postihuje najmä prístupy do polí v cykloch. Tie dokonca často nevyužívajú ani charakteristickú operáciu násobenia a ako iterátor slúži register postupne zvyšovaný pričítaním konštanty. Výsledkom sú zlyhania analýzy na všetkých optimalizovaných vstupoch obsahujúcich polia. Oblasť výpočtu adresových výrazov na rôznych architektúrach tak bude musieť byť podrobená ďalšiemu skúmaniu.

[1] Až na hranicu posledného poľa v A.4, tento nedostatok bol už ale spomenutý.

ID testu	Popis cieľa	Architektúra					
		MIPS		ARM		x86	
		O0	O1	O0	O1	O0	O1
`ptr_base`	Jednoduchá propagácia ukazovateľa. Príklad A.1.	✓	✓	✓	✓	✗	✗
`ptr_ret`	Funkcia vracia ukazovateľ.	✓	✓	✓	✗	✓	✓
`ptr_arg_3x`	Funkcia prijíma tri rôzne ukazovatele.	✓	✓	✓	✓	✓	✓
`array_base`	Jednoduché globálne pole. Príklad A.2.	✓	✗	✓	✗	✓	✗
`array_2x_it`	Jednoduché globálne pole. Niekoľko iterátorov. Test spájania adresových výrazov.	✓	✗	✓	✗	✓	✗
`array_bound_glob`	Dve globálne polia. Test určenia hornej hranice prvého poľa na základe druhého.	✓	✗	✓	✗	✓	✗
`struct_base`	Jednoduchá globálna štruktúra. Príklad A.3.	✓	✓	✓	✓	✗	✗
`struct_2x_glob`	Dva globálne objekty rovnakého štrukturovaného typu. Test spájania deklarácií.	✓	✓	✓	✓	✗	✗
`struct_2x_malloc`	Dva ukazovatele na rovnaký štrukturovaný typ. Test spájania deklarácií.	✓	✓	✓	✓	✓	✓
`struct_from_fnc`	Rozpoznanie lokálnej štruktúry z volania známej knižničnej funkcie.	✓	✓	✓	✓	✓	✓
`complex_glob`	Zložitá globálna štruktúra. Príklad A.4.	✓	✗	✗	✗	✗	✗
`malloc_array`	Ukazovateľ na jednoduché pole.	✓	✗	✓	✗	✓	✗
`malloc_array_bound`	Určenie veľkosti poľa na základe ofsetu nasledujúcej položky v alokovanej štruktúre.	✓	✗	✓	✗	✓	✗
`malloc_struct`	Ukazovateľ na jednoduchú štruktúru.	✓	✓	✓	✓	✓	✓
`complex_malloc`	Ukazovateľ na zložitú štruktúru. Príklad A.5.	✓	✗	✗	✗	✗	✗
`struct_return`	Ukazovateľ na štruktúru ako parameter aj návratová hodnota funkcie.	✓	✓	✓	✗	✓	✓
Percentuálna úspešnosť [%]		100	56	87	44	69	38

Obrázok 7.1: Výsledky jednoduchých testov.



Obrázok 7.2: Úspešnosť jednoduchých testov podľa architektúr.

7.2 Časová náročnosť analýzy

Nájdenie uspokojivého riešenia systému obmedzení nad konečnou doménou je obecne NP-úplný problém. Situácia je o to horšia, že analýza datových typov pracuje nad kódom v SSA forme obsahujúcom niekoľkonásobne väčšie množstvo objektov ako pôvodný kód v jazyku symbolických inštrukcií. Okrem rozdelenia inštrukcií na funkcie a priameho odvodenia typov niektorých objektov neboli implementované žiadne špeciálne optimalizácie. Je tak na mieste otázka časovej náročnosti vytvorenej analýzy.

Za týmto účelom bola vykonaná rada experimentov nad desiatimi reálnymi programami, porovnávajúcich čas behu spätného prekladu pred a po nasadení analýzy datových typov. Testovacie programy väčšinou pochádzajú z projektu *MiBench* [8] poskytujúcom aplikácie určené pre testovanie vstavaných systémov. Jedná sa o konzolové programy napísané v jazyku C implementujúce známe algoritmy ako *md5sum, sha, sha2 aes, dijkstra, quicksort, stringsearch* a ďalšie. Experimenty boli vykonané pre už predstavené architektúry MIPS, ARM, x86, ku ktorým bola pridaná aj pomerne komplexná architektúra ARM Thumb. Programy boli preložené postupne so všetkymi úrovňami optimalizácií, tri krát spustené a výsledné časi spriemerované. Porovnanie prebiehalo no osobnom počítači so šesťdesiatštyri bitovým operačným systémom *ARCH Linux 3.9.2-1*, procesorom *Intel Core i7-2670QM* 2,20 GHz, obsahujúcom štyri jadrá s hyper-threadingom a pamäť RAM o veľkosti 8 GB. Spätný prekladač bol preložený programom *gcc 4.8.1* s úrovňou optimalizácie O2. Výsledky pre jednotlivé platformy sú zobrazené na obrázkoch 7.3 až 7.6.

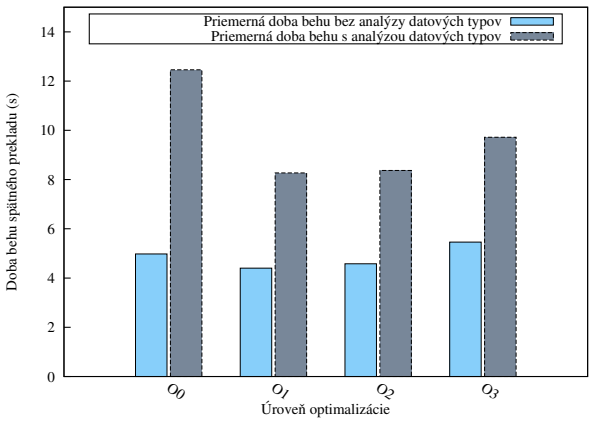
Vidíme, že po zapnutí rekonštrukcie stúpla doba spätného prekladu pre niektoré architektúry takmer na dvojnásobok. Najvýraznejší nárast bol zaznamenaný na neoptimalizovaných programoch, ale v zásade sa výsledky moc nelíšia. Keďže sa nejednalo o nejak výrazne rozsiahlé programy, nie je priemerné zhoršenie neúnosné. Na druhej strane polovica času strávená len v jednej z mnohých analýz nie je zanedbateľná. V budúcnosti teda bude potrebné vykonať podrobné profilovanie a optimalizáciu celého procesu.

Nakoľko nie je v súčasnosti k dispozicií automatizovaný nástroj hodnotenia úspešnosti

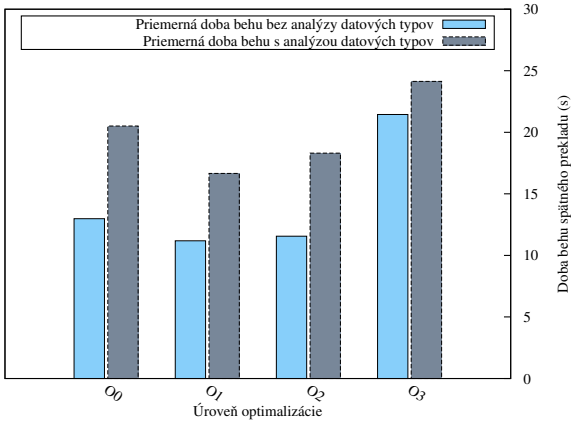
rekonštrukcie, je len veľmi tažké určiť jej presnosť na čo i len trochu komplikovanejších vstupoch. Aj bežným prezretím výstupov môžeme ale zistiť, že minimálne v programoch bez optimalizácií sa podarilo odvodiť mnoho doteraz nedetekovaných zložených typov.

7.3 Budúce experimenty

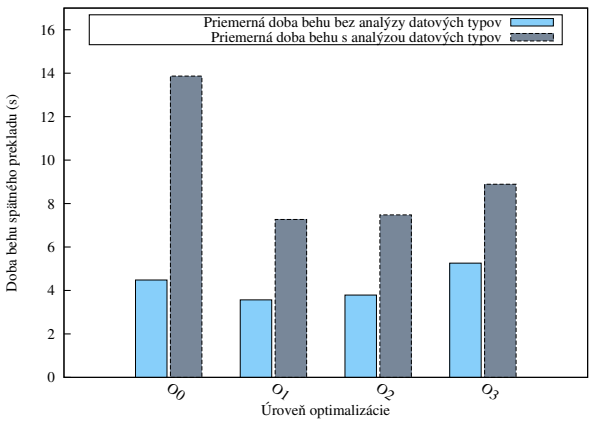
Okrem práce na rozšírení a skvalitnení analýzy bude v budúcnosti vhodné implementovať aj automatizovaný systém hodnotenia vygenerovaných výsledkov. Ten by mohol využívať ladiace informácie vo formátoch DWARF a Microsoft PDB, ktoré sú už v súčasnoti načítané a spracovávané spätným prekladačom. Tieto maximálne presné dáta by bolo možné namapovať na rekonštruované objekty a porovnať ich typy. S pomocou vhodnej metriky by analýza dokázala ohodnotiť svoju úspešnosť a upozorniť na oblasti vyžadujúce zvýšenú pozornosť.



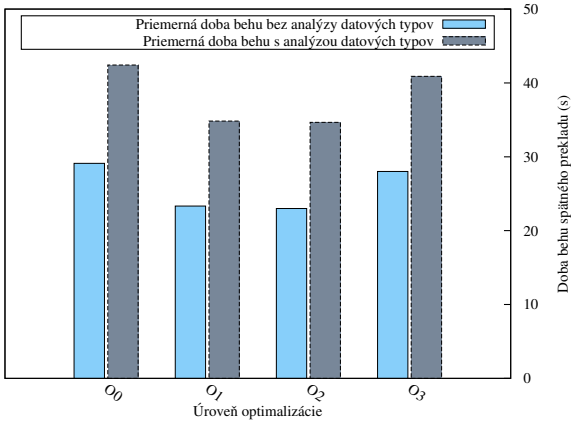
Obrázok 7.3: Porovnanie časov behu spätného prekladu pre architektúru MIPS.



Obrázok 7.4: Porovnanie časov behu spätného prekladu pre architektúru ARM.



Obrázok 7.5: Porovnanie časov behu spätného prekladu pre architektúru x86.



Obrázok 7.6: Porovnanie časov behu spätného prekladu pre architektúru ARM Thumb.

Kapitola 8

Záver

Cieľom diplomovej práce bolo priblížiť tématiku rekonštrukcie jednoduchých a zložených datových typov pri spätnom preklade binárnych súborov. Za týmto účelom bol definovaný pojem spätného inžinierstva a jeho významu v informačných technológiách. Boli vymenované existujúce nástroje a ich schopnosti a predstavený spätný prekladač vyvíjaný projektom Lissom na Fakulte informačných technológií Vysokého učení technického v Brně. V rámci ktorého bude nasadený výstup tejto práce. Práca predstavila niekoľko existujúcich metód odvodzovania typov, diskutovala ich vlastnosti, výhody a nevýhody a ako svoj základ vybrala techniky založené na analýze toku dat a analýze pamäťových operácií.

Jadrom bol návrh a implementácia vlastnej rekonštrukcie datových typov vhodnej pre nasadenie v prostredí Lissom dekompilátoru. Podrobne boli vysvetlené úpravy data-flow analýzy tak, aby efektívne a pokiaľ možno čo najjednoduchšie pracovala nad LLVM IR kódom a využívala výsledky už vykonaných analýz. Významným zjednodušením je použitie lenivých pravidiel, odstraňujúcich pomerne komplikované aternatívne elementy. Ďalej boli využité informácie o detekovaných procedúrach a deklarácie známych funkcií. V návrhu rekonštrukcie zložených typov boli detailne vysvetlené a na príkladoch demonštrované jednotlivé kroky. Táto metóda takmer úplne odpovedá pôvodnému návrhu, s tým rozdielom, že pre zostavenie adresových výrazov používa binárny strom.

Navrhnuté techniky boli implementované a otestované radou jednoduchých aj zložitejších testov. Taktiež bola experimentálne overená ich časová náročnosť na sade reálnych programov implementujúcich známe algoritmy. Ukázalo sa, že riešenie je schopné rozpoznať aj zložitejšie datové typy, aj keď zatiaľ len v jednoduchších programoch bez optimalizácií. Nedá sa ale očakávať plne funkčná rekonfigurovateľná analýza datových typov v priebehu jedného roka určeného na tvorbu diplomového projektu. Práca tak poslúži ako dobrý základ pre ďalší vývoji v naväzujúcom doktorskom štúdiu. Z prezentovaných výsledkov a nedostatkov môžeme vytýčiť niekoľko úloh, ktoré bude treba vyriešiť:

- Hlavným nedostatkom a obmedzením je tvorba adresových výrazov len z aktuálneho základného bloku. Ukázalo sa, že takéto riešenie je nedostatočné pre optimalizované programy a bude ho nutné rozšíriť.
- Rozšírením výrazov a ich rozpoznania sa dosiahne aj rekonštrukcia doteraz neuvažovaných viacrozmerných polí, viacnásobných ukazovateľov alebo rekurzívnych štruktúr.
- Princíp detekcií polí založený na rozpoznaní iterácií s multiplikáciou bude možné použiť aj na lokálne polia umiestnené na zásobníku. Ako už bolo vysvetlené, lokálne štruktúry bude možné rozpoznať nepriamo.

- Ďalej bude treba odhaliť príčinu zahodenia rekonštruovaného typu pri dynamickej alokácii objektov.
- Výslednú analýzu bude vhodné profilovať a optimalizovať, za účelom zníženia potrebného času. Ten by mohol v rozsiahlych programoch prekročiť únosnú hranicu.
- Pre automatizované vyhodnotenie výsledkov bude implementovaná vhodná metrika využívajúca ladiace informácie.

V širšom časovom horizonte bude možné implementovať aj skvalitnenie rekonštrukcie s využitím profilovania, dynamickej analýzy alebo dokonca doplniť analýzu o detekciu C++ tried podľa článkov [27, 26, 17].

Literatura

[1] Boomerang decompiler. [cit. 22. května 2013].
URL http://sourceforge.net/projects/boomerang/

[2] The dcc Decompiler. [cit. 22. května 2013].
URL http://itee.uq.edu.au/~cristina/dcc.html

[3] Decompile-It.com – Online C Decompiler. [cit. 22. května 2013].
URL http://decompile-it.com/

[4] Interactive DisAssembler. [cit. 22. května 2013].
URL http://www.hex-rays.com/

[5] Lissom projekt. [cit. 22. května 2013].
URL http://www.fit.vutbr.cz/research/groups/lissom/

[6] The LLVM Compiler Infrastructure Project. [cit. 22. května 2013].
URL http://llvm.org/

[7] The LLVM Intermediate Representation. [cit. 22. května 2013].
URL http://llvm.org/docs/LangRef.html

[8] MiBench. [cit. 22. května 2013].
URL http://www.eecs.umich.edu/mibench/

[9] REC Studio 4 – Reverse Engineering Compiler. [cit. 22. května 2013].
URL http://www.backerstreet.com/rec/rec.htm

[10] SmartDec. [cit. 22. května 2013].
URL http://smartdec.ru/en/tools

[11] TIE: Type Inference on Executables. [cit. 22. května 2013].
URL http://security.ece.cmu.edu/tie/index.html

[12] Baader, F.; Snyder, W.: Unification Theory. In *Handbook of Automated Reasoning*, editace J. A. Robinson; A. Voronkov, Elsevier and MIT Press, 2001, ISBN 0-444-50813-9, 0-262-18223-8, s. 445–532.

[13] Cifuentes, C.; of Technology. School of Computing Science, Q. U.: *Reverse Compilation Techniques*. Queensland University of Technology, Brisbane, 1994.
URL http://books.google.cz/books?id=DWEFNQAACAAJ

[14] Dolgova, E. N.; Chernov, A. V.: Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.*, ročník 35, č. 2, Březen 2009: s. 105–119, ISSN 0361-7688, doi:10.1134/S0361768809020066.
URL http://dx.doi.org/10.1134/S0361768809020066

[15] Eilam, E.: *Reversing: secrets of reverse engineering*. Wiley, 2005, ISBN 978-0764574818.

[16] Emmerik, M. V.: *Static single assignment for decompilation*. 2007.
URL http://espace.library.uq.edu.au/view/UQ:158682

[17] Fokin, A.; Troshina, K.; Chernov, A.: Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, Washington, DC, USA: IEEE Computer Society, 2010, ISBN 978-0-7695-4321-5, s. 240–243, doi:10.1109/CSMR.2010.43.
URL http://dx.doi.org/10.1109/CSMR.2010.43

[18] ISO/IEC 9899-1999: *Programming Language–C*. Geneva: ISO, 1999.

[19] Kolář, D., Husár, A.: *Návrh výstupního formátu pro assembler a linker*. Interní dokumentace. Brno, FIT VUT v Brně, 2010.

[20] Křoustek, J.: *Analýza a převod kódu do vyššího programovacího jazyka*. diplomová práce, Brno, FIT VUT v Brně, 2009.

[21] Lee, J.; Avgerinos, T.; Brumley, D.: TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS*, The Internet Society, 2011.
URL http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#LeeAB11

[22] Masařík, K.: *Systém pro souběžný návrh technického a programového vybavení počítačů*. VUTIUM, Faculty of Information Technology BUT, první vydání, 2008, ISBN 978-80-214-3863-7, 156 s.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=8915

[23] Matula, P.: *Nástroje pro konverzi formátů spustitelných souborů*. bakalářská práce, Brno, FIT VUT v Brně, 2011.

[24] Mycroft, A.: Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *Programming Languages and Systems, 8th European Symposium on Programming, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, *Lecture Notes in Computer Science*, ročník 1576, Springer, 1999, ISBN 3-540-65699-5, s. 208–223.
URL
http://link.springer.de/link/service/series/0558/bibs/1576/15760208.htm

[25] Trmač, M.; Husár, A.; Hranáč, J.; aj.: Instructor Selector Generation from Architecture Description. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, Masaryk University, 2010, ISBN 978-80-87342-10-7, s. 167–174.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9518

[26] Troshina, E. N.; Chernov, A. V.: Using information obtained in the course of program execution for improving the quality of data type reconstruction in decompilation. *Programming and Computer Software*, 2010: s. 343–362.

[27] Troshina, K.; Chernov, A.; Fokin, A.: Profile-based type reconstruction for decompilation. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, 2009, ISSN 1092-8138, s. 263–267, doi:10.1109/ICPC.2009.5090054.

[28] Troshina, K.; Derevenets, Y.; Chernov, A.: Reconstruction of Composite Types for Decompilation. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, Washington, DC, USA: IEEE Computer Society, 2010, ISBN 978-0-7695-4178-5, s. 179–188, doi:10.1109/SCAM.2010.24.
URL http://dx.doi.org/10.1109/SCAM.2010.24

[29] Urban, M.: *Zadní část zpětného překladače produkující kód v jazyce C*. bakalářská práce, Brno, FIT VUT v Brně, 2012.

[30] Ďurfina, L.; Křoustek, J.: *Machine-Code Decompilation*. správa TAČR, 2013.

[31] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. In *The 5th International Conference on Information Security and Assurance*, Communications in Computer and Information Science, Volume 200, Springer Verlag, 2011, ISBN 978-3-642-23140-7, s. 72–86.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9582

Príloha A

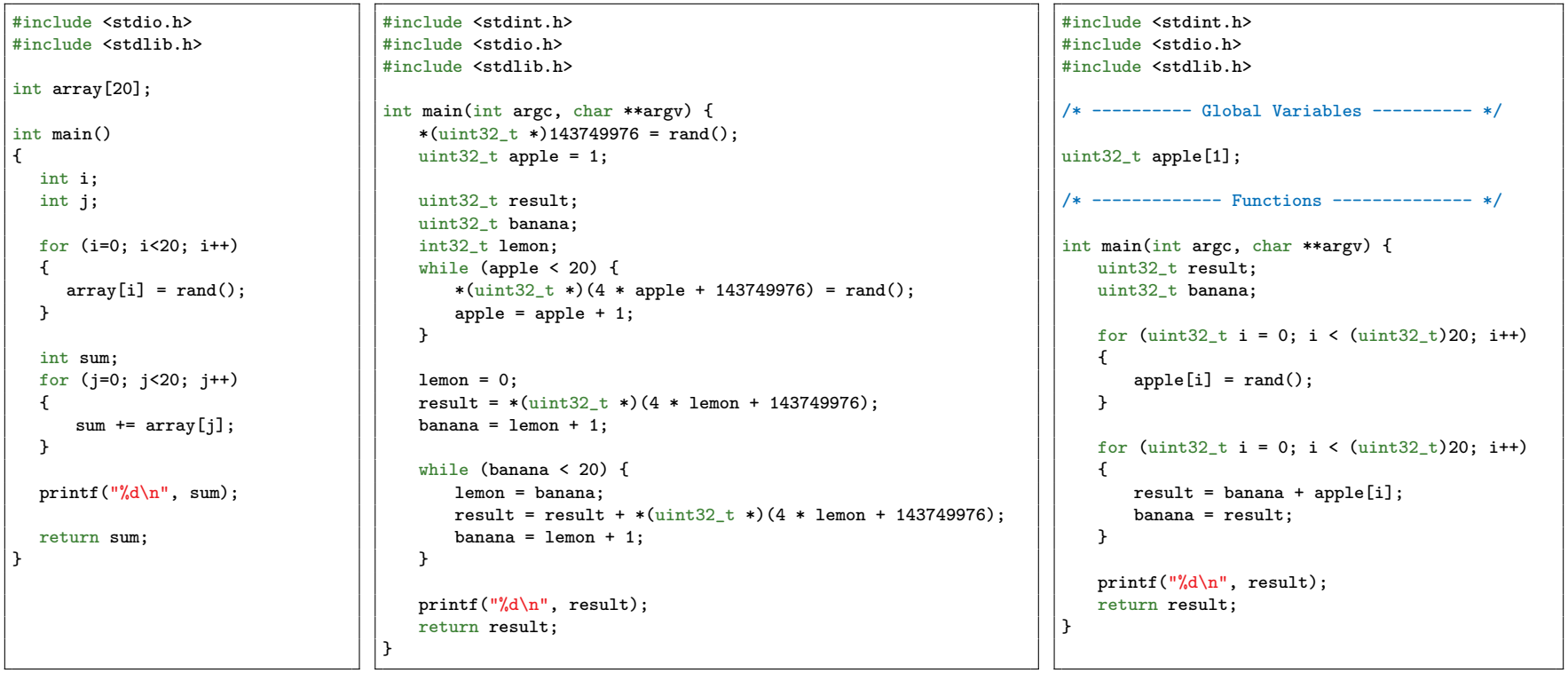
Testovacie príklady


```c
#include <stdio.h>
#include <stdlib.h>

void print(float *fres) {
   printf("%f\n", *fres);
}

int main() {
   float f1 = (float) rand();
   float f2 = (float) rand();

   float *fres = (float*) malloc(sizeof(float));
   *fres = f1 * f2;
   print(fres);

   return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t print(int16_t a) {
    return printf("%f\n",
        (float64_t)*(float32_t *)(int32_t)a);
}

int main(int argc, char **argv) {
    float32_t apple = rand();
    float32_t banana = rand();
    uint8_t * mem = malloc(4);
    *(float32_t *)mem = apple * banana;
    print((uint16_t)(uint32_t)mem);
    return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

uint32_t print(float32_t * a) {
    return printf("%f\n", (float64_t)*a);
}

int main(int argc, char **argv) {
    float32_t apple = rand();
    float32_t banana = rand();
    uint8_t * mem = malloc(4);
    *(float32_t *)mem = apple * banana;
    print((float32_t *)mem);
    return 0;
}
```

Obrázok A.1: Príklad spätného prekladu jednoduchého ukazovateľa.
Naľavo vstupný program, v strede pôvodný výstup, napravo výstup po nasadení analýzy datových typov.



```c
#include <stdio.h>
#include <stdlib.h>

int array[20];

int main()
{
   int i;
   int j;

   for (i=0; i<20; i++)
   {
      array[i] = rand();
   }

   int sum;
   for (j=0; j<20; j++)
   {
       sum += array[j];
   }

   printf("%d\n", sum);

   return sum;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    *(uint32_t *)143749976 = rand();
    uint32_t apple = 1;

    uint32_t result;
    uint32_t banana;
    int32_t lemon;
    while (apple < 20) {
        *(uint32_t *)(4 * apple + 143749976) = rand();
        apple = apple + 1;
    }

    lemon = 0;
    result = *(uint32_t *)(4 * lemon + 143749976);
    banana = lemon + 1;

    while (banana < 20) {
        lemon = banana;
        result = result + *(uint32_t *)(4 * lemon + 143749976);
        banana = lemon + 1;
    }

    printf("%d\n", result);
    return result;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* ---------- Global Variables ---------- */

uint32_t apple[1];

/* ------------- Functions -------------- */

int main(int argc, char **argv) {
    uint32_t result;
    uint32_t banana;

    for (uint32_t i = 0; i < (uint32_t)20; i++)
    {
        apple[i] = rand();
    }

    for (uint32_t i = 0; i < (uint32_t)20; i++)
    {
        result = banana + apple[i];
        banana = result;
    }

    printf("%d\n", result);
    return result;
}
```

Obrázok A.2: Príklad spätného prekladu jednoduchého globálneho poľa.
Naľavo vstupný program, v strede pôvodný výstup, napravo výstup po nasadení analýzy datových typov.

```c
#include <stdio.h>
#include <stdlib.h>

struct structure
{
   char c;
   int i;
   float f;
};

struct structure p;

int print()
{
   printf("%c %d %f", p.c, p.i, p.f);
}

int main()
{
   p.c = (char)rand();
   p.i = rand();
   p.f = rand();

   print();

   return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* ------- Float Types Definitions ------ */

typedef float float32_t;
typedef double float64_t;

/* ---------- Global Variables ---------- */

uint8_t apple = 0;
uint32_t banana = 0;
float32_t lemon = 0.0;









/* ------------- Functions -------------- */

uint32_t print(void) {
    return printf("%c %d %f",
        apple,
        banana,
        (float64_t)lemon);
}

int main(int argc, char **argv) {
    apple = (uint8_t)
            (0x1000000 * rand() / 0x1000000);
    banana = rand();
    lemon = rand();
    print();
    return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* ------- Float Types Definitions ------ */

typedef float float32_t;
typedef double float64_t;

/* -------- Struct Declarations --------- */

struct struct1 {
    uint8_t e0;
    uint32_t e1;
    float32_t e2;
};

/* ---------- Global Variables ---------- */

struct struct1 apple = {.e0=0, .e1=0, .e2=0.0};

/* ------------- Functions -------------- */

uint32_t print(void) {
    return printf("%c %d %f",
        apple.e0,
        apple.e1,
        (float64_t)apple.e2);
}

int main(int argc, char **argv) {
    apple.e0 = (uint8_t)
               (0x1000000 * rand() / 0x1000000);
    apple.e1 = rand();
    apple.e2 = (float32_t)rand();
    print();
    return 0;
}
```

Obrázok A.3: Príklad spätného prekladu jednoduchej globálnej štruktúry.
Naľavo vstupný program, v strede pôvodný výstup, napravo výstup po nasadení analýzy datových typov.

```c
#include <stdio.h>
#include <stdlib.h>

struct t {
   int f1;
   int f2;
   int f3;
};

struct s {
   int b;
   char c;
   struct t d[4];
   float f;
   float e[4];
};

struct s x;

int main(void)
{
   x.b = rand();
   x.c = (char) rand();
   x.f = rand();
   int i;
   for (i=0; i<4; i++)
   {
      x.d[i].f1 = rand();
      x.d[i].f2 = rand();
      x.d[i].f3 = rand();

      x.e[i] = rand();
   }

   return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* ---------- Global Variables ---------- */

uint32_t apple = 0;
uint8_t banana = 0;
float32_t lemon = 0.0;

/* ------------- Functions -------------- */

int main(int argc, char **argv) {
    apple = rand();
    banana = (uint8_t)(0x1000000 * rand() / 0x1000000);
    lemon = rand();
    int32_t cherry = 0;
    *(uint32_t *)143750432 = rand();
    *(uint32_t *)(12 * cherry + 143750436) = rand();
    *(uint32_t *)(12 * cherry + 143750440) = rand();
    *(float32_t *)(4 * (cherry + 14) + 143750428) =
        (float32_t)rand();
    uint32_t grape = cherry + 1;
    cherry = grape;

    while (grape < 4) {
        *(uint32_t *)(12 * grape + 143750432) = rand();
        *(uint32_t *)(12 * cherry + 143750436) = rand();
        *(uint32_t *)(12 * cherry + 143750440) = rand();
        *(float32_t *)(4 * (cherry + 14) + 143750428) =
            (float32_t)rand();
        grape = cherry + 1;
        cherry = grape;
    }

    return 0;
}
```

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* -------- Struct Declarations --------- */

struct struct1 {
    uint32_t e0;
    uint32_t e1;
    uint32_t e2;
};

struct struct2 {
    uint32_t e0;
    uint8_t e1;
    struct struct1 e2[4];
    float32_t e3;
    float32_t e4[1];
};

/* ---------- Global Variables ---------- */

struct struct2 apple = {.e0=0, .e1=0, .e3=0.0,};

/* ------------- Functions -------------- */

int main(int argc, char **argv) {
    apple.e0 = rand();
    apple.e1 = (uint8_t)
                (0x1000000 * rand() / 0x1000000);
    apple.e3 = (float32_t)rand();

    for (uint32_t i = 0; i < 4; i++) {
        apple.e2[i].e0 = rand();
        apple.e2[i].e1 = rand();
        apple.e2[i].e2 = rand();
        apple.e4[i] = (float32_t)rand();
    }

    return 0;
}
```

Obrázok A.4: Príklad spätného prekladu zloženej globálnej štruktúry.
Naľavo vstupný program, v strede pôvodný výstup, napravo výstup po nasadení analýzy datových typov.

```c
#include <stdio.h>
#include <stdlib.h>

struct t {
   int f1;
   int f2;
   int f3;
};

struct s {
   int b;
   char c;
   struct t d[4];
   float f;
   float e[4];
};

int main(void)
{
   struct s *x = (struct s*)
     malloc(sizeof(struct s));

   x->b = rand();
   x->c = (char) rand();
   x->f = rand();

   for (int i=0; i<4; i++)
   {
      x->d[i].f1 = rand();
      x->d[i].f2 = rand();
      x->d[i].f3 = rand();
      x->e[i] = rand();
   }

   return 0;
}
```

```c
#include <stdint.h>
#include <stdlib.h>

/* ------------- Functions -------------- */

int main(int argc, char **argv) {
    uint8_t * mem = malloc(76);
    uint32_t apple = (uint32_t)mem;
    *(uint32_t *)mem = rand();
    *(uint8_t *)(apple + 4) = (uint8_t)
        (0x1000000 * rand() / 0x1000000);
    *(float32_t *)(apple + 56) = (float32_t)rand();
    int32_t banana = 0;
    *(uint32_t *)(apple + 8) = rand();
    *(uint32_t *)(12 * banana + apple + 12) = rand();
    *(uint32_t *)(12 * banana + apple + 16) = rand();
    *(float32_t *)(apple + 56 + 4 * banana + 4) =
        (float32_t)rand();
    uint32_t lemon = banana + 1;
    banana = lemon;

    while (lemon < 4) {
        *(uint32_t *)(12 * lemon + apple + 8) = rand();
        *(uint32_t *)(12 * banana + apple + 12) = rand();
        *(uint32_t *)(12 * banana + apple + 16) = rand();
        *(float32_t *)(apple + 56 + 4 * banana + 4) =
            (float32_t)rand();
        lemon = banana + 1;
        banana = lemon;
    }

    return 0;
}
```

```c
#include <stdint.h>
#include <stdlib.h>

/* -------- Struct Declarations --------- */

struct struct1 {
    uint32_t e0;
    uint32_t e1;
    uint32_t e2;
};

struct struct2 {
    uint32_t e0;
    uint8_t e1;
    struct struct1 e2[4];
    float32_t e3;
    float32_t e4[1];
};

/* ------------- Functions -------------- */

int main(int argc, char **argv) {
    uint8_t * mem = malloc(76);
    *(uint32_t *)mem = rand();
    mem[4] = (uint8_t)(0x1000000 * rand() / 0x1000000);
    *(float32_t *)&mem[56] = (float32_t)rand();

    for (int32_t i = 0; i < 4; i++) {
        (struct struct2 *)mem.e2[i].e0 = rand();
        (struct struct2 *)mem.e2[i].e1 = rand();
        (struct struct2 *)mem.e2[i].e2 = rand();
        (struct struct2 *)mem.e4[i] = (float32_t)rand();
    }

    return 0;
}
```

Obrázok A.5: Príklad spätného prekladu zloženej lokálnej štruktúry.
Naľavo vstupný program, v strede pôvodný výstup, napravo výstup po nasadení analýzy datových typov.

Příloha B

Obsah CD

Adresár	**Popis**
./README.txt	Informácie o CD.
./Makefile	Súbor pre automatizovaný preklad a testovanie.
./doc	Elektronická verzia tohto dokumentu.
./decfront/operations/composite	Zdrojové kódy operácií nad zloženými datovými typmi.
./decfront/operations/pointers	Zdrojové kódy operácií nad ukazovateľmi.
./decfront/types	Zdrojové kódy rekonštrukcie datových typov.
./decfront/linux64	Preložené objekty a knižnice.
./decfront/*	Hlavičkové súbory nutné pre preklad
./decompiler/backend	Knižnice a spustiteľné súbory zadnej časti.
./decompiler/share	Zdroje nutné pre preklad z predstavených architektúr.
./decompiler/bin	Programy pre spúštanie spätného prekladača.
./decompiler/tests	Testy.
./sdkcodasip	Knižnice, programy a hlavičky Codasip SDK.
./compilers/clang	Prekladač clang.
./compilers/gcc-x86-elf	Prekladač jazyka C pre Intel x86, formát ELF.
./compilers/gnuarm	Prekladač jazyka C pre ARM, formát ELF.
./compilers/pspsdk	Prekladač jazyka C pre MIPS, formát ELF.