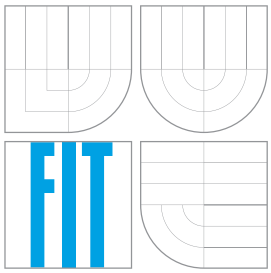
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MIGRACE ZDROJOVÝCH KÓDŮ POMOCÍ DEKOMPILACE

SOURCE-CODE MIGRATION USING DECOMPILATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ KOREC

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR ZEMEK

BRNO 2014

Abstrakt

Tato práca sa zaoberá migráciou zdrojových kódov vysokoúrovňových programovacích jazykov za pomoci dekompilácie. Migračný nástroj vyvinutý v rámci práce je postavený na prostrednej a zadnej časti dekompilátoru projektu Lissom. V práci je rozobraných niekoľko prekladačov, ktoré zo vstupného jazyka generujú kód v LLVM IR. Vhodné prekladače boli vybrané pre integráciu do migračného nástroja. Kód preložený do LLVM IR je vstupom prostrednej optimalizačnej časti dekompilátoru. Výstupom migračného nástroja je kód v jazyku C alebo v jazyku podobnom Pythonu generovaný zadnou časťou dekompilátoru. Vstupnými jazykmi sú Fortran a jeho dialekty, C/C++/Objective-C/Objective-C++ a D. V práci sú popísane problémy spojené s migráciou týchto jazykov, ich riešenie a spôsoby ako zlepšiť kvalitu a čitateľnosť výsledného kódu.

Abstract

This thesis deals with source-code migration of high-level programming languages using decompilation. A migration tool developed within the thesis is built on top of the middle-end and back-end parts of Lissom project decompiler. Several compilers generating LLVM IR code from input languages are discussed. Compilers suitable for integration to the migration tool were chosen. Compiled LLVM IR code is an input of the decompiler's optimizing middle-end. The output from the migration tool is a code in the C language or Python-like language generated by the back-end of the decompiler. The input languages are Fortran and its dialects, C/C++/Objective-C/Objective-C++, and D. The thesis describes problems connected with migration of these languages, their solutions, and ways to improve quality and readability of the produced source code.

Klíčová slova

migrace, zdrojový kód, dekompilace, Lissom, LLVM IR, Fortran, C/C++, D, Objective C

Keywords

migration, source code, decompilation, Lissom, LLVM IR, Fortran, C/C++, D, Objective C

Citace

Tomáš Korec: Migrace zdrojových kódů pomocí dekompilace, diplomová práce, Brno, FIT VUT v Brně, 2014

Migrace zdrojových kódů pomocí dekompilace

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Petra Zemka. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Tomáš Korec
26. mája 2014

Poděkování

Ďakujem vedúcemu diplomovej práce Ing. Petrovi Zemkovi a konzultantovi Ing. Jakubovi Křoustkovi za odbornú pomoc a smerovanie pri vypracovávaní práce. Ďalej by som rád poďakoval mojej priateľke Lucii za občasné rady pri písaní technickej správy a mojej rodine za podporu počas celého štúdia.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

V dnešnej dobe ešte stále existuje množstvo projektov, ktorých hlavné časti sú implementované v zastaralých jazykoch alebo ich dialektoch. Udržovanie týchto častí je náročné a nákladné, potrebné nástroje, ako prekladač a ladiaci nástroj, nemusia byť ďalej vyvíjané a podporované. Navyše odborníci na zastaralé jazyky postupom času ubúdajú. Migrácia zdrojových kódov z jedného vysoko úrovňového programovacieho jazyka do druhého je často jediným riešením (viď [15, 20]).

Ručná migrácia zdrojových kódov je pomerne náročný proces, ktorý je časovo veľmi náročný a generuje množstvo nových chýb. Preto vznikajú snahy vytvoriť poloautomatické alebo automatické nástroje na uľahčenie tohto procesu. Ako príklad môžeme zmieniť poloautomatický migračný nástroj z jazyka PL/IX do jazyka C++ [18] a automatický nástroj na migrovanie zdrojových kódov z jazyka Fortran do jazyka C [17].

Dosiaľ vyvinuté nástroje často nie sú úplne automatické [18], ale ich hlavnou nevýhodou je obmedzená podpora vstupných a výstupných jazykov. Väčšinou ide o nástroje podporujúce jeden vstupný jazyk alebo určitý dialekt jazyka a jeden výstupný jazyk [17]. Cieľom tejto práce je vytvoriť univerzálnejší nástroj podporujúci viacero vstupných a viacero výstupných jazykov. To je dosiahnuté s využitím existujúceho dekompilátoru projektu Lissom, ktorý slúži ako nástroj na spätný preklad [19]. Vyvinutý migračný nástroj integruje prostrednú optimalizačnú a zadnú časť dekompilátoru s prekladačmi rôznych jazykov. Prekladače zo vstupného zdrojového kódu generujú kód v LLVM IR, ktorý je vstupom prostrednej optimalizačnej časti dekompilátoru. Zadná časť dekompilátoru potom produkuje výstup momentálne v dvoch vysoko úrovňových jazykoch: C a modifikovanej verzii jazyka Python.

Práca je rozdelená nasledovne. Po tejto úvodnej kapitole nasleduje oboznámenie s problémom migrácie kódu, o existujúcich možnostiach a nástrojoch pre migráciu. V kapitole 3 je predstavený projekt Lissom, v krátkosti reverzné inžinierstvo a dekompilátor ako nástroj reverzného inžinierstva. Pred predstavením dekompilátoru projektu Lissom je v samostatnej sekcii predstavený systém LLVM, na ktorom je dekompilátor postavený. V nasledujúcej kapitole je predstavený návrh migračného nástroja, po ktorom nasleduje prehľad prekladačov generujúcich LLVM IR a jazykov, ktoré prekladajú. Vhodné prekladače boli vybrané na integráciu do nástroja. Aby boli výstupy implementovaného nástroja použiteľné, bolo nutné zvýšiť ich kvalitu. Návrh metód, ktorými to bolo dosiahnuté sa nachádza v kapitole 6. V kapitole 7 je popísaná implementácia migračného nástroja, integrácia vybraných prekladačov a implementácia metód vylepšujúcich výstup migrácie. Ďalej nasleduje kapitola s popisom spôsobu testovania a s príkladmi migrovaných zdrojových kódov. Posledná kapitola je záverom práce. Práca obsahuje taktiež prílohy s ukážkami migrácie.

Kapitola 2

Migrácia kódu

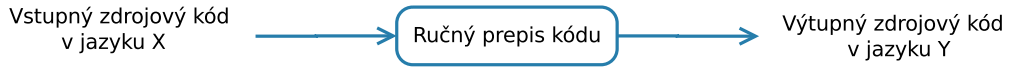
Dlhotrvajúce projekty sa musia potýkať s problémami spojenými s ich implementačným jazykom alebo prekladačom tohto jazyka. Mnoho aplikácií implementovaných pred niekoľkými desaťročiami v zastaralých jazykoch je stále dôležitou súčasťou existujúcich projektov. Udržovanie takýchto aplikácií je veľmi náročné a nákladné. Implementačný jazyk je často zastaralý a neefektívny. Dostupnosť ladiacich a testovacích nástrojov môže byť obmedzená. Rozšírenie alebo portovanie na novšie platformy je zvyčajne nemožné, pretože implementačný jazyk nepodporuje dnešné programovacie techniky ako modulárny dizajn, objektovo orientované programovanie, viac-vláknové programovanie a prekladač nemusí podporovať nové platformy alebo nové vlastnosti existujúcich platforiem. Prekladač daného zastaralého jazyka už často nie je vyvíjaný a jeho podpora skončila. Preto je migrácia zdrojových kódov do moderného jazyka často jediným riešením [25].

Migrácia zvyčajne prináša redukciu množstva kódu (počet riadkov), prináša lepšiu prenositeľnosť a novú funkcionalitu. Vývojové cykli po migrácii sú často kratšie, migrovaná aplikácia dosahuje vyšší výkon. Taktiež je potrebné zvážiť ubúdanie odborníkov na zastaralý programovací jazyk a potenciál pribúdania odborníkov na moderný cieľový programovací jazyk. Príkladom môže byť systém AGPS (Aero Grid and Paneling System) spoločnosti Boeing používaný viac ako dvadsať rokov. Výsledná aplikácia po migrácii umožňuje nové možnosti, priniesla vyšší výkon, je jednoduchšia na udržovanie a portovanie, priniesla modernizáciu uživateľského rozhrania a počet riadkov zdrojového kódu bol zredukovaný o 50%. Vývojový cyklus bol skrátený z pôvodných 12 až 24 mesiacov na 2 až 3 mesiace [15].

Táto kapitola je rozdelená nasledovne. Možnosti migrácie zdrojových kódov sú popísané v sekcii 2.1. Sekcia 2.2 popisuje existujúce migračné nástroje.

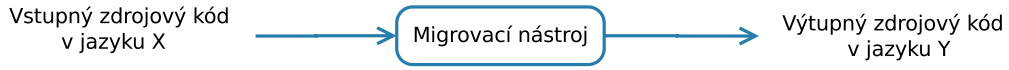
2.1 Možnosti migrácie

Jedným zo spôsobov, ako zdrojové kódy migrovať je ich kompletné prepísanie do moderného programovacieho jazyka, schematicky znázornené na obrázku 2.1, kde jazyk X je vstupný jazyk a jazyk Y je výstupný jazyk. Tento postup je však časovo veľmi náročný a pri jeho použití navyše vzniká množstvo chýb. To si vyžaduje ďalší čas na ladenie a dôsledné testovanie. Ladenie a testovanie je často časovo náročnejšie než samotný prepis zdrojových kódov. Príkladom nevyhnutnej migrácie, ktorá bola uskutočnená týmto spôsobom je migrácia systému AGPS, spomínaná v úvode kapitoly. Ide o 3D modelovací nástroj pôvodne implementovaný v jazyku Fortran a čiastočne v jazyku C (spolu viac ako 300 000 riadkov kódu). Zdrojové kódy boli migrované do jazyka Java [15].



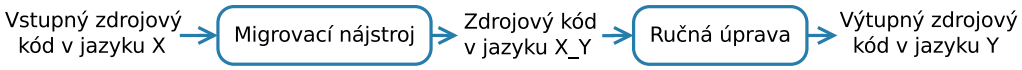
Obr. 2.1: Ručný prepis kódu

Ďalšou možnosťou je vytvorenie nástroja, ktorý spracuje (angl. parsing) vstupný zdrojový kód a na výstup zapíše jeho ekvivalent v cieľovom programovacom jazyku. Ak je nástroj kvalitný, prináša značnú výhodu v podobe automatizácie a znovupoužiteľnosti. Schematické znázornenie nástroja je zobrazené na obrázku 2.2, kde jazyk X je vstupný jazyk a jazyk Y výstupný jazyk. Nevýhodou tohto prístupu je obmedzenie daných nástrojov na určitý vstupný a výstupný jazyk, resp. na ich dialekty. Príkladom takéhoto nástroja je *f2c – A Fortran to C converter* [17]. Ako už samotný názov napovedá, ide o automatický konvertor z jazyka Fortran do jazyka C. Tento nástroj však podporuje len jeden dialekt – FORTRAN77.



Obr. 2.2: Automatický migrovací nástroj

Ďalším príkladom je nástroj popísany v [18]. Ide o poloautomatický nástroj, ktorý migruje kód v jazyku PL/IX (dialekt jazyka PL/I[1]) do jazyka C++. Tento nástroj nie je úplne automatický, a podobne ako predchádzajúci nástroj, migruje len jeden špecifický dialekt jazyka PL/I. Schematické zobrazenie poloautomatického nástroja je zobrazené na obrázku 2.3, kde jazyk X je vstupný jazyk a jazyk Y je výstupný jazyk. Jazyk X_Y predstavuje medzivýstup migračného procesu, ktorý vyžaduje ručnú úpravu, aby nadobudol podobu jazyka Y.



Obr. 2.3: Poloautomatický migrovací nástroj

Ak nezoberieme do úvahy ručnú migráciu kódu, existujúce nástroje nie sú vždy úplne automatické a na vstupe podporujú len určitý jeden jazyk alebo jeho dialekt. To isté platí pre výstup. Cieľom tejto práce je implementovať také riešenie, ktoré odstráni nevýhody predchádzajúcich popísaných riešení. Riešenie by malo byť plne automatické a univerzálne do takej miery, aby podporovalo viacero vstupných aj viacero výstupných jazykov. Toto riešenie je schématicky znázornené na obrázku 2.4. Jazyky X, Y, Z predstavujú vstupné jazyky a jazyky A, B a C predstavujú výstupné jazyky. Zároveň by malo byť jednoducho rozšíriteľné o ďalšie, ako vstupné, tak výstupné jazyky. Návrh tohto riešenia je detailne popísaný v kapitole 4.

[1] http://publibfp.boulder.ibm.com/epubs/pdf/ibm4lr03.pdf

Obr. 2.4: Univerzálny automatický migrovací nástroj

2.2 Existujúce migračné nástroje

V tejto sekcii spomenieme niektoré existujúce migračné nástroje. Keďže existuje veľa jazykov, existuje aj veľa migračných nástrojov medzi rôznymi jazykmi. Väčšinou ide o priamy prevod zdrojového kódu z jedného jazyka do druhého, pričom je podporovaný len jeden vstupný a jeden výstupný jazyk. Rozšíriteľnosť o ďalšie jazyky je pri priamej konverzii obmedzená. Migračné nástroje tak majú prístup k informáciám ako sú názvy premenných a funkcií, komentáre a pod. Tieto informácie zachovávajú aj vo svojom výstupe, čo je ich výhodou. Príklady migračných nástrojov:

- *f2c*[2] – A Fortran to C converter. Jeho nevýhodou je podpora jedného konkrétneho dialektu jazyka Fortran – FORTRAN77 na vstupe a jedného jazyka – C na výstupe.
- *fable*[3] – Konverter z jazyka fortran do C++. Plne podporuje FORTRAN77 a čiastočne Fortran90 na vstupe a na výstupe C++.
- *Incomplete Fortran to C/C++ converter*[4] – Webová služba na konverziu zdrojových kódov v dialekte FORTRAN77 do C/C++. Ako už názov napovedá, nie je možné pomocou nej konvertovať zdrojové kódy úplne.
- *Objexx F2C++*[5] – Komerčný produkt na konverziu z jazyka Fortran do jazyka C++. Na stránkach produktu nie sú uvedené podporované dialekty, z príkladu je však zjavné, že nástroj podporuje minimálne dialekt FORTRAN77.
- *On-Line Fortran F77 - F90 Converter*[6] – Webová služba na konverziu dialektu FORTRAN77 do dialektu Fortran90. Žiadne iné jazyky nie sú podporované.
- *FOR_C*[7] – Komerčný produkt na konverziu z dialektu FORTRAN77, ktorý však podporuje aj Fortran90 do jazyka C. Po zadaní kontaktných údajov je možné stiahnuť demo verziu nástroja.
- *F2CL*[8] – Nástroj na konverziu dialektu FORTRAN77 s niektorými rozšíreniami (ďalšie dialekty) do jazyka Common Lisp.

[2] http://www.netlib.org/f2c/
[3] http://cci.lbl.gov/fable/
[4] http://simulationcorner.net/index.php?page=if2c
[5] http://objexx.com/Fortran_to_Cpp.html
[6] http://www.polyhedron.com/plusfortonline.php
[7] http://www.cobalt-blue.com/fc/fcmain.htm
[8] http://trac.common-lisp.net/f2cl/

- *TDC*[9] – Trivial D Compiler. Nástroj prevádzajúci kód v jazyku D do jazyka C. Jeho vývoj skončil pred šiestimi rokmi a nikdy nebola vydaná oficiálna verzia (angl. release). Zdrojové kódy sú však k dispozícii.

- *j2c*[10] – Java to C++ converter. Nástroj vo forme pluginu do vývojového prostredia *Eclipse*.

- *Tangible Software Solutions*[11] – Komerčné nástroje na konverziu zdrojových kódov medzi jazykmi Java, C++, C# a Visual Basic. Nástroje majú aj varianty, ktoré sú zdarma, sú však obmedzené tak, že konvertujú len určitý maximálny počet riadkov vstupného súboru.

- *java2python*[12] – Nástroj na konverziu zdrojových kódov z jazyka Java do jazyka Python.

- *Sharpen*[13] – Pluginu do vývojového prostredia *Eclipse* na konverziu zdrojových kódov z jazyka Java do jazyka C#.

- *java2haxe*[14] – Pluginu do vývojového prostredia *Eclipse* na konverziu zdrojových kódov z jazyka Java do jazyka Haxe[15] – multiplatformný open-source jazyk preložiteľný (s obmedzeniami) do ďalších jazykov. Vývoj pluginu bol zrejme ukončený bez toho, aby bola implementovaná podpora niektorých základných konštrukcií jazyka Java.

- *dax*[16] – Nástroj na konverziu jazyka D do jazyka Haxe.

- *Tarwins AS3 to Haxe conversion script*[17] – Skript na konverziu zdrojových kódov z jazyka Action Script 3.0 (AS3) do jazyka Haxe.

- *CS2HX*[18] – Nástroj na konverziu zdrojových kódov v jazyku C# do jazyka Haxe.

- *TypeScript to Haxe Convertor*[19] – Skript na konverziu zdrojových kódov z jazyka TypeScript do jazyka Haxe.

Vyššie uvedené nástroje často nie sú úplne automatické a migrácia je len čiastočná. Príkladom môže byť konvertor *C++ to Java Converter* od *Tangible Software Solutions*. Java nepodporuje príkaz `goto`, avšak výstupný zdrojový kód nástroja tieto príkazy obsahuje, ak sa nachádzajú vo vstupnom zdrojovom kóde. Niektoré nástroje existujú len vo forme pluginu do vývojového prostredia, a teda sú na tomto prostredí závislé. Zaujímavú skupinu tvorí posledných päť nástrojov, ktoré konvertujú vstupný zdrojový kód do jazyka Haxe. Kód v tomto jazyku môže byť s určitými obmedzeniami preložený (angl. source-to-source compilation) do ďalších jazykov – JavaScript, C++, Flash, NodeJS, PHP, NekoVM,

[9] http://dsource.org/projects/tdc
[10] https://code.google.com/a/eclipselabs.org/p/j2c/
[11] http://www.tangiblesoftwaresolutions.com/Product_Details/Products.html
[12] https://code.google.com/p/java2python/
[13] http://community.versant.com/Documentation/Reference/db4o-7.12/java/reference/html/Content/sharpen.html
[14] https://github.com/Danielku15/java2haxe
[15] http://haxe.org/
[16] http://dsource.org/projects/dax
[17] http://haxe.org/doc/flash/usingas3classes/tarwins_as3_to_haxe_conversion_script
[18] https://cs2hx.codeplex.com/
[19] https://github.com/Ezelia/ts2haxe

C# a Java. Okrem prekladu z Haxe do vymenovaných jazykov vždy ide o jednoúčelové nástroje – preklad z iba jedného vstupného jazyka do iba jedného výstupného. Neexistuje nástroj, ktorý by zároveň podporoval viacej vstupných a viacej výstupných jazykov (teoreticky by mohol byť vytvorený integráciou nástrojov a skriptov na preklad do jazyka Haxe s prekladačom jazyka Haxe).

Existencia veľkého množstva nástrojov dokladá, že migrácia zdrojových kódov je aktuálny a riešený problém.

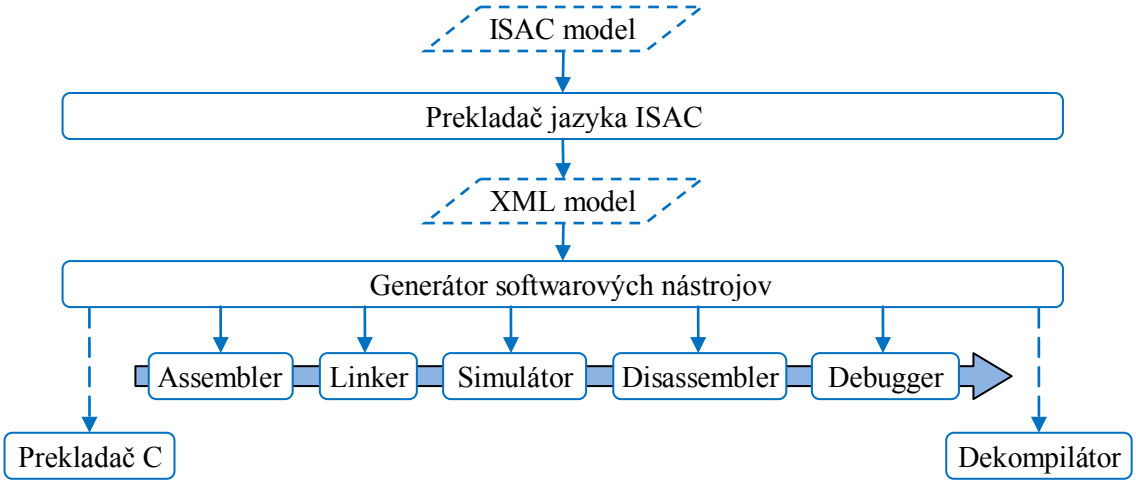
Kapitola 3

Projekt Lissom, reverzné inžinierstvo a dekompilácia

Táto kapitola sa venuje projektu Lissom (sekcia 3.1) a projektu LLVM, na ktorom je postavený dekompilátor projektu Lissom (sekcia 3.4). Sekcia 3.2 približuje, čo je to reverzné inžinierstvo a sekcia 3.3 popisuje obecný dekompilátor ako nástroj reverzného inžinierstva. V poslednej sekcii 3.5 je ďalej bližšie rozobraný dekompilátor projektu Lissom. Prostredná a zadná časť dekompilátoru sú súčasťou návrhu nástroja, ktorý je vyvíjaný v rámci tejto práce.

3.1 Projekt Lissom

Cieľom projektu Lissom[1] je vytvoriť a implementovať jazyk na popis architektúry procesorov. Pre dobrú použiteľnosť jazyka je taktiež nevyhnutné vytvoriť vývojové prostredie, ktoré umožňuje vývoj ako softvérového vybavenia, tak hardvérovej architektúry.



Obr. 3.1: Nástroje projektu Lissom [19]

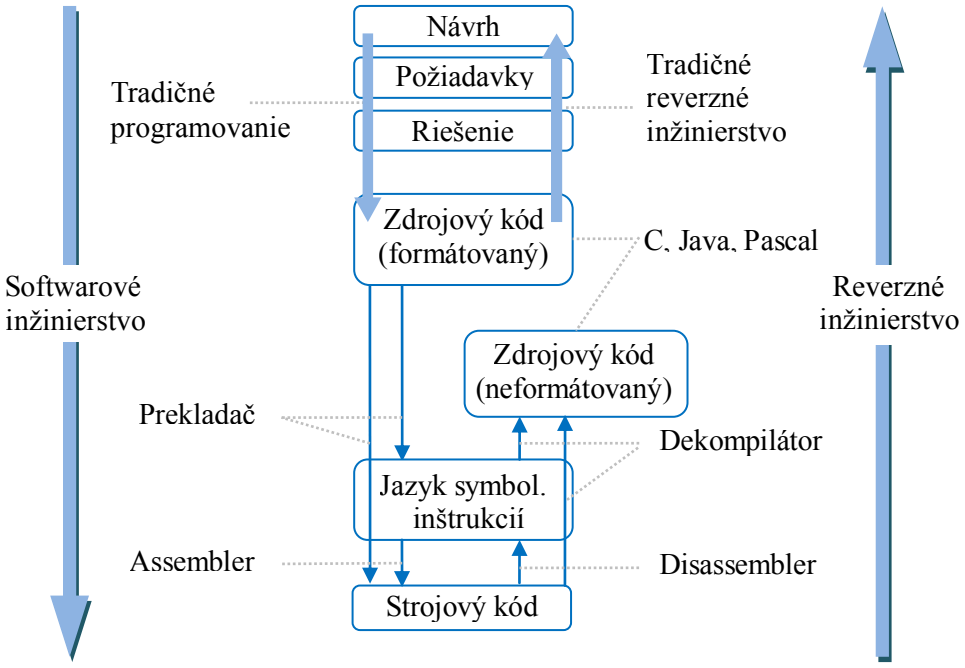
[1] http://www.fit.vutbr.cz/research/groups/lissom/project.html

Popisným jazykom je zmiešaný jazyk ISAC, ktorý dokáže popísať ako architektúru, tak inštrukčnú sadu. Je nadstavbou nad jazykom ANSI C. Návrh procesoru v jazyku ISAC je prekladačom prevedený na XML súbor špecifikujúci celú architektúru. Z tohto súboru sú následne automaticky generované nástroje: assembler, disassembler, simulátor, debugger, prekladač jazyka C a dekompilátor [19]. Obrázok 3.1, prevzatý z [19], ilustruje generovanie nástrojov z modelu v jazyku ISAC.

Vďaka súčasnému vývoju softvéru aj hardvéru (hardware / software co-desing), ktorý umožňujú nástroje projektu Lissom, je možné podstatne skrátiť vývojové cykly aj celkový čas vývoja aplikačne špecifických procesorov ASIP (angl. Application Specific Instruction-set Processors). To následne znamená zníženie nákladov na vývoj. Dekompilátor projektu je taktiež možné použiť za účelom analýzy škodlivého softwaru [24].

3.2 Reverzné inžinierstvo

Reverzné inžinierstvo je proces, pri ktorom sa snažíme odhaliť vnútorné detaily ako návrh, architektúru a princíp fungovania skúmaného predmetu. V informatike ide o proces analýzy predmetného systému s cieľom identifikovať jeho komponenty a vzťahy medzi nimi alebo vytvoriť jeho reprezentáciu na vyššej úrovni abstrakcie, alebo v inej forme [16]. Reverzné inžinierstvo v informatike si môžeme predstaviť ako proces inverzný k softvérovému inžinierstvu [19]. Obrázok 3.2 prevzatý z [19] tento vzťah ilustruje.



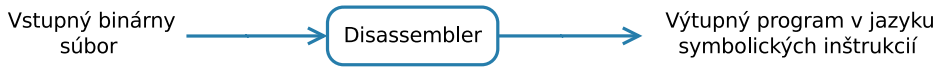
Obr. 3.2: Vzťah softwarového a reverzného inžinierstva [19]

Metódy analýzy softvéru sa delia do dvoch kategórií podľa spôsobu jej prevádzania:

- dynamická analýza,
- statická analýza.

Dynamickou analýzou rozumieme zbieranie informácií o aplikáciách sledovaním ich vykonávania pomocou ladiacich nástrojov (angl. debuggers) a sledovaním (angl. tracing). Ladiace nástroje a nástroje na sledovanie boli spomenuté iba pre úplnosť. Ďalej sa nimi nebudeme zaoberať, pretože sa pohybujeme na inej úrovni a z hľadiska práce nie sú dôležité.

Pri statickej analýze skúmame binárne súbory bez ich vykonania. Medzi nástroje statickej analýzy patria disassembler a dekompilátor. Disassembler, alebo spätný assembler, je nástroj, ktorý prevedie celý vstupný binárny súbor, alebo len jeho časť na zdrojový kód v jazyku symbolických inštrukcií [16]. Funkcia disassembleru je znázornená na obrázku 3.3. Dekompilátor je popísaný v nasledujúcej samostatnej sekcii.



Obr. 3.3: Funkcia disassembleru

3.3 Dekompilátor a jeho časti

Dekompilátor, alebo spätný prekladač, je po disassembleri ďalší krok k vyššej forme abstrakcie. Vstupom dekompilátoru je taktiež binárny súbor, ale jeho výstupom je kód vo vysoko úrovňovom programovacom jazyku. Podstatou dekompilátoru je pokúsiť sa reverzovať kompilačný proces (proces prekladu) a získať originálny zdrojový kód alebo kód podobný originálnemu zdrojovému kódu. Na väčšine platforiem nie je možné úplné obnovenie pôvodného zdrojového kódu. Niektoré črty vysoko úrovňových programovacích jazykov sú vynechané (stratené) pri kompilácii a nie je možné ich obnoviť. Ide napríklad o komentáre, názvy premenných, pôvodné typy (štruktúry), atď. Napriek tomu je dekompilátor schopný z binárneho súboru rekonštruovať do dobre čitateľnej podoby veľkú časť zdrojového kódu [16].

Dekompilátor je často navrhnutý tak, že sa skladá z prednej časti, prostrednej časti a zadnej časti. Ide o jeden z možných prístupov, ktorý si ďalej priblížime. Schematicky je dekompilátor s touto architektúrou znázornený na obrázku 3.4.

Obr. 3.4: Dekompilátor a jeho časti

3.3.1 Predná časť

Predná časť (angl. front-end), ktorá spracúva (angl. parsing) zdrojový kód v prekladači, v dekompilátore dekóduje inštrukcie nízko úrovňového jazyka a prekladá ich do svojej vnútornej reprezentácie [16].

3.3.2 Prostredná časť

V prostrednej časti (angl. middle-end) prekladača prebiehajú optimalizácie. V prostrednej časti dekompilátoru tomu nie je inak, aj keď v skutočnosti nejde o vylepšenie niektorých vlastností programu (rýchlosť, veľkosť výstupného kódu, ...), tak ako tomu je pri prekladači, ale o jeho úpravu do podoby vhodnejšej pre proces transformovania do cieľového vysoko úrovňového jazyka. Z veľkej časti ide o zrušenie / návrat zmien optimalizátora prekladača a elimináciu nepodstatných detailov týkajúcich sa architektúry (napríklad odstránenie použitia dočasných registrov za pomoci analýzy propagácie dát) [16].

3.3.3 Zadná časť

Zadná časť dekompilátoru (angl. back-end) je zodpovedná za produkovanie výstupu vo forme vysoko úrovňového jazyka z výstupu strednej časti dekompilátoru. Zadná časť je špecifická pre daný výstupný jazyk. Tak isto ako je zameniteľná zadná časť prekladača pre jednoduchú podporu viacero výstupných architektúr, je zameniteľná aj zadná časť dekompilátoru pre podporu viacero výstupných jazykov [16].

3.4 Systém LLVM

Táto sekcia z veľkej časti vychádza z [14] a [9]. Dekompilátor vyvíjaný v rámci projektu Lissom je postavený na systéme LLVM[2]. LLVM je názov projektu, ktorý zastrešuje viacero ďalších projektov. V rámci týchto projektov sú vyvíjané nízkoúrovňové nástroje ako assemblery, prekladače, ladiace nástroje a pod. Projekt LLVM je známy vďaka niektorým nástrojom, ako napríklad *Clang* – prekladač C/C++/Objective-C/Objective-C++, ktorý prináša množstvo výhod oproti prekladaču GCC. Avšak hlavnou vlastnosťou LLVM, ktorá projekt odlišuje od ostatných nástrojov, je jeho návrh a vnútorná reprezentácia LLVM IR (LLVM Intermediate Representation).

3.4.1 Typický návrh prekladača

Populárny dizajn tradičného prekladača je zobrazený na obrázku 3.5. Predná časť spracúva (angl. parsing) zdrojový kód, kontroluje jeho syntaktickú správnosť a buduje jazykovo špecifický abstraktný syntaktický strom. Optimalizátor (prostredná časť) je zodpovedný za široké spektrum transformácií, ktorými sa pokúša vylepšiť čas behu kódu, napríklad odstránením nadbytočných výpočtov. Je väčšinou nezávislý od jazyka a cieľovej architektúry. Zadná časť, tiež nazývaná generátor kódu, potom mapuje vnútornú reprezentáciu na cieľovú inštrukčnú sadu. Tento model platí taktiež pre interprety a JIT (Just In Time) prekladače. Java Virtual Machine (JVM) je tiež implementáciou tohto modelu, ktorá používa Java bytecode ako rozhranie medzi prednou časťou a optimalizátorom.

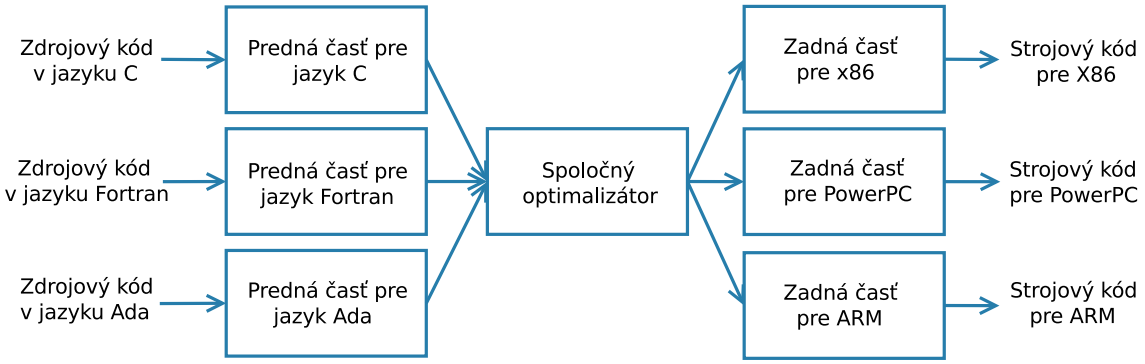
Obr. 3.5: Typická architektúra prekladača, prevzaté z [14]

[2] http://llvm.org/

Najväčšou výhodou tohto dizajnu je jeho jednoduchá rozšíriteľnosť v podobe pridania vstupného jazyka alebo výstupnej cieľovej architektúry. Ak prekladač používa vhodnú vnútornú reprezentáciu, môže byť vytvorená:

- predná časť pre ľubovolný jazyk, ktorý je možné preložiť do tejto reprezentácie,
- zadná časť pre ľubovolnú architektúru do ktorej je možné vnútornú reprezentáciu preložiť,

pričom optimalizátor je spoločný, ako ukazuje obrázok 3.6.



Obr. 3.6: Výhody popísaného designu prekladača, prevzaté z [14]

Úspešnou implementáciou tohto návrhu je napríklad prekladač GCC[3]. GCC podporuje mnoho predných a zadných častí, avšak jeho použitie je limitované, pretože je navrhnuté ako monolitická aplikácia. Napríklad, nie je reálne možné zabudovať GCC do ďalších aplikácií, použiť GCC ako runtime/JIT prekladač, alebo extrahovať a znovu použiť časti GCC bez toho, aby musela byť prítomná väčšina z kódu prekladača. Ak chce napríklad niekto použiť prednú časť GCC pre C++ na generovanie dokumentácie, indoxovanie kódu, refaktoring a statickú analýzu, musí použiť GCC ako monolitickú aplikáciu, ktorá generuje informácie v XML[4] alebo musí naimplementovať vlastný plugin, ktorý vloží cudzí kód do GCC. Dôvodov, prečo nie je možné použiť časti GCC ako knižnice je viacero, napríklad časté používanie globálnych premenných alebo použitie makier, ktoré zabraňujú kompilácii výslednej aplikácie do takej podoby, aby zároveň podporovala viacej ako jeden pár predná časť/zadná časť. Ďalej napríklad zadná časť prechádza abstraktný syntaktický strom prednej časti pri generovaní ladiacich informácií, predná časť generuje dátové štruktúry pre zadnú časť. Spomínané neduhy môžu byť ďalším vývojom odstránené.

LLVM spomenutými problémami netrpí vďaka návrhu a vnútornej reprezentácii LLVM IR. Jednotlivé časti LLVM nie sú na sebe navzájom závislé, tak ako je tomu pri GCC. Je teda možné ich použiť zvlášť v iných projektoch ako knižnice, tak ako tomu je v dekompilátore projektu Lissom.

3.4.2 LLVM IR

Ako už bolo povedané, jednou z najdôležitejších častí návrhu LLVM je jeho vnútorná reprezentácia LLVM IR (LLVM Intermediate Representation), ktorá sa používa na reprezentáciu

[3] http://gcc.gnu.org/
[4] Viac informácií možno nájsť na http://gccxml.github.io/HTML/Index.html

kódu v prekladači. Ide o reprezentáciu, ktorá poskytuje typovú bezpečnosť, nízko úrovňové operácie, flexibilitu a schopnosť čisto reprezentovať vysokoúrovňové jazyky. Je založená na Static Single Assignment (SSA), čo znamená, že každej premennej je priradená hodnota presne jeden krát a premenné sú rozdelené do verzií [22]. To ilustruje obrázok 3.7, kde na pravej strane je kód z ľavej strany vo forme SSA. Prekladač tak ihneď spozná, že premenná `a1` z pravej strany obrázka nie je nikde použitá a môže tak kód efektívnejšie optimalizovať oproti kódu na ľavej strane obrázka [16].

```
a = 1;
a = 3;
b = a;
```

```
a1 = 1;
a2 = 3;
b1 = a2;
```

Obr. 3.7: Bežný kód a kód vo forme Static Single Assignment

LLVM IR je spoločnou reprezentáciou kódu používanou vo všetkých fázach prekladu v LLVM. Je navrhnutá tak, aby nad ňou bolo možné vykonávať analýzu a transformácie, ktoré sa vykonávajú v optimalizačnej časti prekladača. Pri návrhu bolo myslené na mnoho špecifických cieľov zahŕňajúcich podporu odľahčenej optimalizácie za behu, optimalizácie medzi funkciami (angl. cross-function / interprocedural), analýzu celého programu, agresívne reštrukturalizačné transformácie atď. LLVM IR sa snaží byť odľahčená, nízko úrovňová typovaná reprezentácia s vyjadrovacou silou a dobrou rozšíriteľnosťou zároveň. V ukážke na obrázku 3.8 sú v hornej polovici v jazyku C dve rôzne implementácie funkcie, ktorá sčíta dve čísla a v dolnej časti LLVM IR kód korešpondujúci k týmto funkciám. Je vidieť, že ide o inštrukčnú sadu typu RISC s inštrukciami v troj-adresnej forme. Oproti väčšine inštrukčných sád typu RISC je silne typovaná s jednoduchým typovým systémom (napríklad `i32` značí 32-bitový celo číselný typ, `i32**` je ukazovateľ na ukazovateľ na 32-bitový celo číselný typ) a niektoré strojové detaily sú abstrahované. Napríklad volacia konvencia je abstrahovaná pomocou inštrukcií `call` a `ret` a ich explicitnými argumentami. Ďalším podstatným rozdielom od strojového kódu je, že LLVM IR nepoužíva fixnú množinu pomenovaných registrov, ale potenciálne nekonečnú množinu dočasných premenných s menom začínajúcim znakom `%`.

Okrem toho, že je LLVM IR implementovaná ako jazyk, je definovaná v troch izomorfných formách:

- textová forma čitateľná pre človeka (spodná časť obrázku 3.8),
- dátová štruktúra v pamäti (pracuje s ňou a modifikuje ju prekladač pri samotných optimalizáciách),
- efektívny bitcode v súbore na disku (vhodné napríklad pre rýchle načítanie JIT prekladačom).

Tieto tri formy sú ekvivalentné. Vďaka nim je možné vykonávať efektívne transformácie a analýzu prekladačom, zatiaľ čo poskytujú prirodzený prostriedok pre ladenie a vizualizáciu transformácií. Projekt LLVM poskytuje nástroje na konverziu medzi textovou formou a bitcode formou. Assembler *llvm-as* prekladá textový *.ll* súbor do súboru *.bc* obsahujúceho bitcode reprezentáciu a disassembler *llvm-dis* prevádza naopak *.bc* súbor na *.ll* súbor.

Funkcie v jazyku C

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

Preklad do LLVM IR

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

Obr. 3.8: Ukážka jazyka LLVM IR, prevzaté z [14]

3.4.3 Optimalizácie LLVM IR

Dôležitou časťou LLVM systému je *LLVM Pass Framework*[5]. Framework poskytuje množstvo optimalizácií (nazývaných *Passes*) implementovaných ako triedy, ktoré (nepriamo) dedia od triedy `Pass` implementujúce funkcionalitu prekrytím (angl. override) virtuálnych metód. *Passes* môžu vykonávať transformácie a optimalizácie alebo vykonávať analýzy, ktorých výsledky sú použité v týchto transformáciách. Podľa toho ako fungujú sa delia do niekoľkých kategórií:

- `ModulePass` – pracuje nad celým programom,

[5] http://llvm.org/docs/WritingAnLLVMPass.html

- `CallGraphSCCPass` – prechádza graf volaní (angl. call graph) odspodu hore,
- `FunctionPass` – je vykonaný nad každou funkciou,
- `LoopPass` – je vykonaný nad každým cyklom,
- `RegionPass` – podobný predchádzajúcemu, je vykonaný nad každým blokom s jedným vstupom a jedným výstupom (angl. single entry, single exit region),
- `BasicBlockPass` – vykonaný nad každým základným blokom (angl. basic block), čo je zoznam inštrukcií vykonaných sekvenčne (bez skokov a pod.).

V prípade implementácie vlastnej transformácie, bude táto transformácia spadať do jednej z vyššie vymenovaných kategórii. Názov kategórie bude predstavovať aj názov triedy, ktorej bude naša vlastná trieda priamym potomkom. Výberom správnej triedy napovieme systému, čo naša transformácia s kódom robí a ako môže byť skombinovaná s ostatnými pri použití. Takto implementovaná transformácia môže byť použitá rovnako ako vstavané optimalizácie.

Optimalizácie a analýzy, vstavané aj vlastnoručne implementované, sú vykonávané modulárnym optimalizátorom *opt*. Ten prijíma na vstupe zdrojový kód v LLVM IR a vykonáva optimalizácie a analýzy špecifikované parametrom. Na svojom výstupe produkuje buď optimalizovaný kód alebo výsledky analýzy.

3.5 Dekompilátor projektu Lissom

V predchádzajúcich sekciách boli predstavené potrebné informácie pre popis a pochopenie dekompilátoru projektu Lissom. Táto sekcia sa zaoberá samotným dekompilátorom. Väčšina informácií v nej vychádza z [24]. Tento rekonfigurovateľný (angl. retargetable) dekompilátor si kladie za cieľ byť nezávislý na akejkoľvek cieľovej architektúre, operačnom systéme alebo formáte súboru. Na obrázku 3.9 je možné vidieť, že pozostáva z dvoch hlavných častí – časť pre predspracovanie a jadro dekompilátoru.

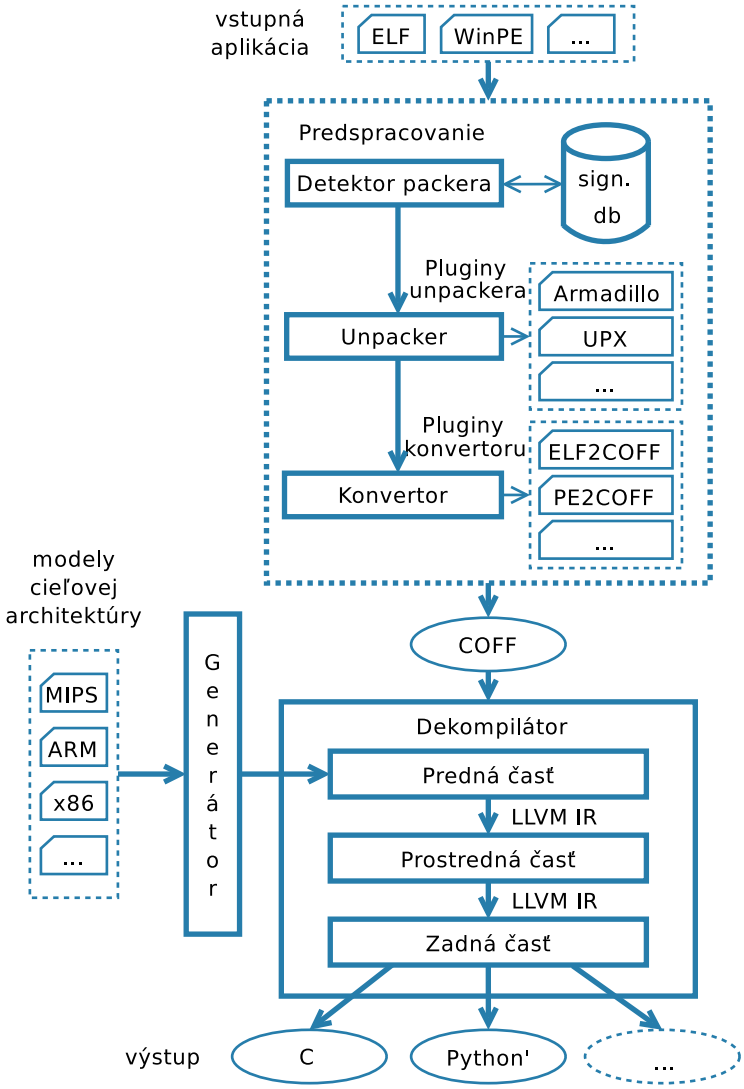
3.5.1 Predspracovanie

Časť pre predspracovanie vykonáva analýzu vstupnej aplikácie, počas ktorej zisťuje aký bol použitý formát binárneho súboru, aký bol použitý prekladač, či bol súbor zbalený (angl. packed) a aký bol použitý nástroj na jeho zbalenie. Po vstupnej analýze súbor rozbalí (ak je to potrebné) a prevedie aplikáciu do interného formátu založenom na formáte COFF (Common Object File Format). Konverzia je podporovaná z Windows PE, Unix ELF, Apple Mach-O a ďalších formátov. Podpora neštandardných formátov môže byť doplnená implementovaním zásuvného modulu. To isté platí pre nástroje na zbalenie (angl. packing) binárneho súboru. Po predspracovaní je výsledný súbor v internom formáte ďalej spracovávaný jadrom dekompilátoru.

3.5.2 Jadro dekompilátoru

Jadro dekompilátoru je postavené na systéme LLVM popísanom v sekcii 3.4, pričom návrh odpovedá návrhu prezentovanom v sekcii 3.3. Pre vnútornú reprezentáciu kódu teda používa LLVM IR a skladá sa z troh častí – prednej, prostrednej optimalizačnej a zadnej [19].

Predná časť je jediná platformne špecifická časť, pretože jej inštrukčný dekodér je automaticky generovaný z modelu architektúry v jazyku ISAC. Dekodér prekladá strojový kód



Obr. 3.9: Koncept dekompilátoru projektu Lissom, prevzaté z [24]

aplikácie na sekvenciu LLVM IR inštrukcií, ktoré sú už platformne nezávislé. V prednej časti ďalej prebieha statická analýza, ktorá je zodpovedná za elimináciu staticky linkovaného kódu, detekciu použitého ABI – aplikačné binárne rozhranie (angl. application binary interface, obsahuje napríklad popis volacích konvencií, práce so zásobníkom atď.), obnovenie funkcií atď.

Prostredná časť prijíma výstup v LLVM IR z prednej časti, ktorý často obsahuje mnoho mŕtveho a neefektívneho kódu. Pre lepší a čitateľnejší výsledok dekompilácie je v tejto časti optimalizovaný. Na optimalizáciu využíva spomenutý optimalizátor *opt*, mnoho vstavaných optimalizácií systému LLVM a ďalšie optimalizácie vyvinuté v rámci projektu. Ide napríklad o odstránenie mŕtveho kódu, optimalizáciu cyklov, propagáciu konštánt, zjednodušenie grafu toku riadenia (angl. control flow graph), atď.

Posledná zadná časť konvertuje optimalizovanú vnútornú reprezentáciu do cieľového vysoko úrovňového jazyka. Konverzia je vykonávaná v niekoľkých krokoch. Najskôr je vstupný kód v LLVM IR konvertovaný do vlastnej vnútornej reprezentácie zadnej časti dekompi-

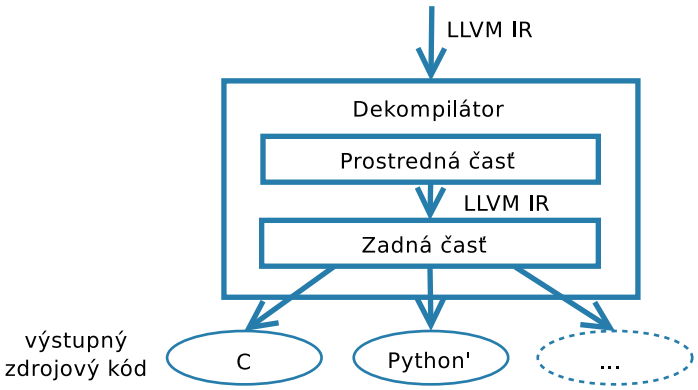
látoru BIR (Back-end Intermediate Representation). Počas tejto konverzie sú rozpoznané a rekonštruované vysoko úrovňové konštrukcie ako cykly a podmienené príkazy. Následne je kód v BIR optimalizovaný a zapísaný na výstup v podobe cieľového vysokoúrovňového jazyka. V čase písania tejto práce sú podporované jazyky C a jazyk podobný jazyku Python (jazyk Python obohatený o konštrukcie jazyka C v prípadoch, keď v jazyku Python pre danú dekompilovanú konštrukciu nie je podpora).

Dekompilátor okrem kódu v cieľovom jazyku produkuje graf volaní dekompilovanej aplikácie, graf toku riadenia pre všetky funkcie a aplikáciu v LLVM IR kóde.

Kapitola 4

Návrh migračného nástroja

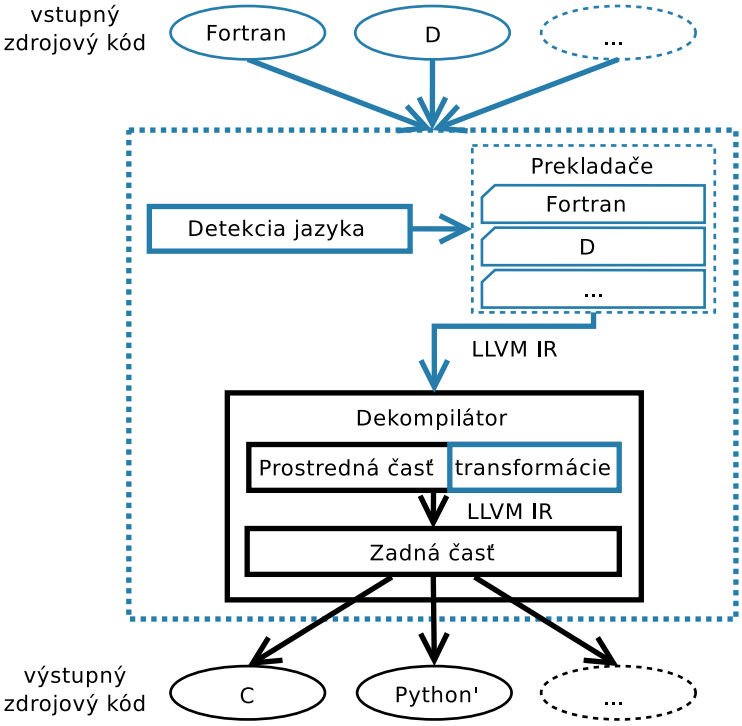
Cieľom tejto práce je vytvoriť nástroj na migrovanie zdrojových kódov a integrovať ho do projektu Lissom. Vďaka návrhu dekompilátoru je možné použiť jednotlivé jeho časti oddelene. Ako už bolo spomenuté, dekompilátor používa LLVM IR ako svoju vlastnú vnútornú reprezentáciu, čo znamená, že vstupom prostrednej časti je kód v LLVM IR. Je teda možné použiť prostrednú a zadnú časť na dekompiláciu ľubovolného vstupného kódu v LLVM IR (výnimkou sú zdrojové kódy obsahujúce exotické konštrukcie, ktoré nie sú podporované v zadnej časti dekompilátoru). Ilustruje to obrázok 4.1, na ktorom je upravený pôvodný obrázok 3.9 prevzatý z [24].



Obr. 4.1: Časť dekompilátoru projektu Lissom

Vďaka existencii projektu LLVM existuje niekoľko prekladačov rôznych jazykov, ktoré produkujú kód v LLVM IR. Idea migračného nástroja teda je nájsť vhodné prekladače produkujúce kód v LLVM IR a integrovať ich s časťou dekompilátoru zobrazenou na obrázku 4.1. Ďalej bude nutné rozšíriť prostrednú a zadnú časť o prípadné nové nepodporované konštrukcie produkované prekladačmi vstupných jazykov. Vznikne teda plne automatický nástroj, ktorý podporuje viacero vstupných a viacero výstupných jazykov a zároveň je jednoducho rozšíriteľný o ďalšie vstupné a výstupné jazyky. Schéma nástroja je zobrazená na obrázku 4.2 (rozšírenie obrázku 4.1). Modro zvýraznená časť je predmetom tejto práce.

Keďže cieľom práce nie je rozširovať podporu výstupných jazykov, výstupnými jazykmi migračného nástroja zostávajú spomenuté jazyky C a obohatený jazyk Python, ktoré sú implementované ako výstupné jazyky zadnej časti dekompilátoru.



Obr. 4.2: Schéma migračného nástroja

Ďalšou vlastnosťou migrovacieho nástroja, ako napovedá obrázok 4.2, by mala byť schopnosť zistiť, o aký vstupný jazyk ide a automaticky použiť správny prekladač. Okrem toho, tak ako na všetky nástroje projektu Lissom, aj na migračný nástroj je kladená požiadavka multiplatformnosti. Znamená to, že všetky prekladače vstupných jazykov musia byť multiplatformné (požadované platformy sú Windows, GNU/Linux, ako 32-bitové tak aj 64-bitové).

Kapitola 5

Výber vstupných jazykov a prekladačov

V tejto kapitole rozoberieme existujúce možnosti generovania LLVM IR z rôznych jazykov. Ide teda o nájdenie vhodných prekladačov, ktorých predná časť je schopná generovať na výstupe LLVM IR. Ďalším požiadavkom na prekladač je jeho multiplatformnosť v podobe možnosti skompilovať prekladač pre linuxové operačné systémy aj Windows, alebo aspoň existencie binárnych súborov pre obe platformy.

Kapitola je rozdelená do sekcií podľa prekladačov, pričom názov sekcie obsahuje jazyky, ktoré daný prekladač podporuje. Na začiatku každej sekcie je stručný informatívny popis jazyka. V poslednej sekcii 5.6 je krátke zhrnutie vybraných jazykov a nástrojov.

5.1 Pascal

Jazyk Pascal je vplyvný imperatívny a procedurálny programovací jazyk vyvinutý v rokoch 1968 – 1969 a publikovaný v roku 1970 Niklausom Wirthom ako jednoduchý a výkonný jazyk s cieľom podporiť dobré programovacie návyky použitím štruktúrovaného programovania a dátových štruktúr. Jeho derivát známy ako Object Pascal navrhnutý pre objektovo orientované programovanie bol vyvinutý v roku 1985 [13].

Pre jazyk Pascal existujú dvaja kandidáti. Prvým je projekt *llvm-pascal*[1] a druhým je implementácia free pascal komunity[2]. Oba projekty mali za cieľ implementovať prekladač jazyka Pascal s využitím LLVM systému, a teda LLVM IR, na vnútornú reprezentáciu programu. Obidva však už dlhšiu dobu nejavia známky života. *llvm-pascal* bol univerzitný projekt vyvíjaný v jazyku Pascal v rámci dizertačnej práce. Posledné vydanie (angl. release) z roku 2010 nieslo označenie „pre-alpha“ a bolo preložené len pre platformu windows. Zdrojové kódy navyše neobsahujú *Makefile* – inštrukcie pre program *Make* na preklad zdrojových súborov. Na linuxovom operačnom systéme sa mi nepodarilo zdrojové súbory preložiť. Druhý z projektov bol výsledkom práce komunity a nebol nikdy oficiálne vydaný. Podľa logu svn repozitára už od roku 2010 nie je ďalej vyvíjaný. Z týchto dôvodov nebol ani jeden z kandidátov vybraný pre ďalšiu integráciu do migračného nástroja.

[1] https://code.google.com/p/llvm-pascal/

[2] http://community.freepascal.org/bboards/message?message_id=320545&forum_id=24105

5.2 D

Jazyk D je objektovo orientovaný, imperatívny, multi-paradigmový programovací jazyk vytvorený Walterom Brightom. Pôvodne vychádzal z jazyka C++ prerobením jeho hlavných častí, ale pri jeho tvorbe sa autor inšpiroval aj ďalšími jazykmi, konkrétne Java, Python, Ruby, C# a jazyk Eiffel. Cieľom návrhu jazyka bolo skombinovať výkon kompilovaných jazykov s bezpečnosťou a vyjadrovacou silou dynamických jazykov. Kód v jazyku D je často rovnako rýchly ako jeho ekvivalent v jazyku C++, zatiaľ čo je kratší a bezpečný, čo sa týka práce s pamäťou. Typové odvodzovanie, automatická správa pamäte a „syntaktický cukor" pre často používané typy napomáhajú k rýchlejšiemu vývoju, zatiaľ čo kontrola hraníc, design by contract (povinnosť špecifikovať jasné rozhrania komponent systému) vlastnosti a konkurentný (angl. concurrency-aware) typový systém pomáhajú redukovať výskyt chýb v kóde [5]. Podľa domovskej stránky jazyka D[3] je to jazyk so syntaxou podobnou jazyku C a so statickým typovaním. Kombinuje efektivitu, kontrolu a modelovaciu silu s bezpečnosťou a produktivitou programátora.

V rámci projektu LDC[4] je vyvíjaný prekladač jazyka D, ktorý používa systém LLVM a LLVM IR pre vnútornú reprezentáciu programu. Vývoj na projekte je aktívny, nie je problém preložiť zdrojové kódy a v rámci projektu sú pravidelne vydávané binárne súbory ako pre linuxové operačné systémy, tak pre Windows. Prekladač projektu LDC je teda ideálny kandidát pre integráciu do migračného nástroja.

5.3 Lua

Lua je odľahčený multi-paradigmový programovací jazyk navrhnutý ako skriptovací jazyk s rozšíriteľnou sémantikou. Bol vytvorený v roku 1993 Robertom Ierusalimschym, Luizom Henrique de Figueiredom a Waldemarom Celesom. Lua je multiplatformný jazyk vďaka tomu, že je naprogramovaný v ANSI C. Lua je relatívne jednoduché API jazyka C, je špeciálne užitočná pre poskytovanie jednoduchého spôsobu koncovým užívateľom ako programovať chovanie softvérového produktu bez nutnosti prílišného zachádzania do detailov. Vďaka tomu je široko používaná vo video hrách, ako napríklad World of Warcraft, v ktorej si môžu užívatelia prispôsobiť užívateľské rozhranie, animácie postáv a vzhľad sveta práve v jazyku Lua [11].

Lua je ďalším kandidátom na vstupný jazyk migračného nástroja vďaka projektu *llvm-lua*[5]. Podobne ako pri predchádzajúcich projektoch, v rámci projektu *llvm-lua* je cieľ vytvoriť prekladač, tento raz jazyka Lua, ktorý využíva LLVM ako svoju zadnú časť. Prekladaný program je teda vnútorne reprezentovaný pomocou LLVM IR. Tento projekt taktiež vyzerá ako ďalej nevyvíjaný. Posledné vydanie je z roku 2010 a aj keď sú posledné zmeny v repozitáry z roku 2012, nie je možné prekladač preložiť s aktuálnou verziou LLVM. Keďže LLVM je súčasťou dekompilátoru projektu Lissom, na ktorom neustále prebieha aktívny vývoj, je tiež pravidelne aktualizované. Aj keby sa teda podarilo preložiť prekladač projektu *llvm-lua* so staršou verziou LLVM, nebol by vhodný pre ďalšiu integráciu do migračného nástroja, pretože LLVM IR nie je stabilné a nezaručuje spätnú kompatibilitu [10].

[3] http://dlang.org/
[4] http://wiki.dlang.org/LDC
[5] https://code.google.com/p/llvm-lua/

5.4 C, C++, Objective-C a Objective-C++

Jazyk C je univerzálny jazyk spočiatku vyvíjaný Dennisom Ritchiem medzi rokmi 1969 a 1973 v AT&T Bell Labs. Tak ako mnoho imperatívnych jazykov v tradícii ALGOLu, C má prostriedky pre štruktúrované programovanie, umožňuje lexikálnu viditeľnosť premenných a rekurziu, zatiaľ čo staticky typový systém zabraňuje mnohým nechceným operáciam. Jeho dizajn poskytuje konštrukcie, ktoré sa efektívne mapujú na strojový kód. C je jeden z najviac rozšírených programovacích jazykov [4].

Pridaním objektovo orientovaných vlastností ako napríklad triedy a ďalších vylepšení od jazyka C vznikol jazyk C++ (pôvodne pomenovaný ako C with Classes – C s triedami). Bol vyvíjaný Bjarneom Stroustrupom od roku 1979 v Bell Labs. Taktiež ide o univerzálny, staticky typovaný, kompilovaný jazyk s podporou viacerých paradigmatov [2].

Objective C je taktiež univerzálny, objektovo orientovaný programovací jazyk, ktorý pridáva Smalltalkový štýl predávania správ do programovacieho jazyka C. Bol vyvinutý začiatkom osemdesiatych rokov devätnásteho storočia. Kombináciou C++ a Objective-C dostávame jazyk označovaný ako Objective-C++. Považuje sa za variantu jazyka Objective-C. Objective-C++ prináša do jazyka C++ rovnaké rozšírenia ako prináša Objective-C do jazyka C. Avšak sémantika za rôznymi vlastnosťami jazykov nie je unifikovaná, a tak v jazyku vznikajú určité obmedzenia (napríklad trieda jazyka C++ nemôže dediť od triedy jazyka Objective-C a naopak) [12].

Vďaka existencii prekladača *Clang*[6], ktorý generuje LLVM IR zo všetkých týchto štyroch jazykov, sú všetky ďalšími uvažovanými vstupnými jazykmi. *Clang* je projekt priamo zastrešovaný LLVM s aktívnym vývojom. Je teda vhodným kandidátom na integráciu do migračného nástroja.

5.5 Fortran, jeho dialekty a ADA

Fortran, kedysi FORTRAN (názov je odvodený od The IBM Mathematical **For**mula **Tran**slating System), je univerzálny, imperatívny programovací jazyk, ktorý je vhodný na numerické a vedecké výpočty. Pôvodne bol vyvinutý firmou IBM na ich kampuse v San Jose v Kalifornii v päťdesiatych rokoch devätnásteho storočia pre vedecké a technické aplikácie. Fortran začal rýchlo dominovať v tejto oblasti programovania a bol nepretržite používaný viac ako pol storočia vo výpočtovo náročných oblastiach ako napríklad numerická predpoveď počasia, metóda konečných prvkov, výpočtová dynamika kvapalín, výpočtová fyzika a výpočtová chémia. Fortran zahŕňa celý rodokmeň verzií (dialektov), z ktorých každá rozšírila jazyk a zvyčajne aj udržala spätnú kompatibilitu s predchádzajúcou verziou [6]. Úspešné dialekty:

- FORTRAN77 – pridaná podpora štruktúrovaného programovania a spracovania dát založených na znakoch,
- Fortran90 – pridaná podpora polí, modulárneho programovania a generického programovania,
- Fortran95 – tzv. High Performance Fortran, čo znamená podpora paralelných výpočtov (dátový paralelizmus),
- Fortran 2003 – pridaná podpora objektovo orientovaného programovania,

[6] http://clang.llvm.org/

- Fortran 2008 – pridaná podpora paralelného programovania.

Jazyk Ada je štruktúrovaný staticky topovaný imperatívny objektovo orientovaný jazyk dizajnovaný ako nízko úrovňový a zároveň vysoko úrovňový. Vychádza z jazyka Pascal a ďalších jazykov. Má vstavanú podporu pre paralelizmus, synchrónne predávanie správ, chránených objektov a nedeterminizmus. Pôvodne bol vyvinutý tímom vedeným Jeanom Ichbiahom z CII Honeywell Bull na základe zmluvy s Ministerstvom obrany Spojených štátov amerických (United States Department of Defense – DoD) medzi rokmi 1977 a 1983 za účelom nahradiť stovky programovacích jazykov používaných DoD. Jazyk Ada je pomenovaný po Auguste Ade Kingovej, grófke Lovelace, ktorá je považovaná za prvého počítačového programátora [1].

Fortran, jeho dialekty a ADA sú ďalšie zvažované jazyky. Neexistuje síce prekladač postavený na LLVM, ktorého predná časť by na vstupe pracovala s týmito jazykmi, ale v rámci LLVM sa pracuje na projekte s názvom *Dragonegg*[7]. Výstupom projektu je plugin do prekladača GCC, vďaka ktorému je možné použiť prednú časť z GCC, pričom optimalizátor a zadná časť je nahradená tým z LLVM projektu. Plugin *Dragonegg* aj prekladač GCC sú aktívne vyvíjané a prinášajú podporu viacerých jazykov. Oba predstavujú ďalšieho kandidáta na integráciu do migračného nástroja. GCC pokrýva všetky zmienené dialekty jazyka Fortran, vďaka čomu je možné zaistiť podporu všetkých v migračnom nástroji (A Fortran to C converter [17] podporuje iba dialekt FORTRAN77).

Plugin *Dragonegg* podporuje jazyky Go, Java, Objective-C a Objective-C++ len čiastočne, preto nie sú uvažované, aj keď existuje predná časť GCC podporujúca tieto jazyky.

5.6 Zhrnutie

Po preskúmaní všetkých objavených možností boli vybrané jazyky D (prekladač *ldc2*), C, C++, Objective-C a Objective-C++ (prekladač *Clang*), Fortran, jeho dialekty a ADA (prekladač GCC + plugin *Dragonegg*). Najčastejším dôvodom vyradenia jazyka bolo ukončenie vývoja prekladača.

[7] http://dragonegg.llvm.org/

Kapitola 6

Návrh metód vylepšujúcich výstup migrácie

Implementovať migračný nástroj integrovaním vybraných prekladačov, prostrednej a zadnej časti dekompilátoru do jedného celku pre uspokojivé výsledky nestačí. Výstup vykazoval množstvo nedokonalostí, často bol zle čitateľný a v mnohých prípadoch nebolo možné výsledný kód v LLVM IR dekompilovať. Bolo potrebné navrhnúť metódy na odstránenie chýb pri dekompilácii a na zlepšenie kvality a čitateľnosti kódu. Táto kapitola sa zaoberá návrhom týchto metód. Ak nie je pri jednotlivých metódach uvedené inak, implementácia je realizovaná vo forme transformácií pre optimalizačnú časť dekompilátoru.

Kapitola je rozdelená do sekcií. Prvá sekcia sa venuje obecným problémom, ktoré sa vyskytovali vo výstupoch migrácie bez ohľadu na vstupný jazyk. Sekcia 6.2 sa venuje problémom vyskytujúcich sa pri migrácii z jazyka C++, sekcia 6.3 je venovaná jazyku Fortran a posledná sekcia 6.4 jazyku D.

6.1 Obecné problémy

Táto sekcia sa zaoberá problémami, ktoré sa pri migrácii vyskytovali obecne bez ohľadu na migrovaný vstupný jazyk.

6.1.1 Volanie `llvm_lifetime_end()`

Volanie `llvm_lifetime_end()` sa objavovalo vo výsledných zdrojových kódoch nezávisle na vstupných jazykoch. Ukážka tohto volania je na obrázku 6.1. Toto volanie vzniklo dekompiláciou inštrukcie `llvm.lifetime.end` z LLVM IR. Inštrukcia špecifikuje ukončenie existencie objektu v pamäti. Vo výslednom kóde sa nachádza za každou premennou v časti kódu, kde končí jej existencia. To robí výsledný kód značne neprehľadným. Pretože dekompilované volanie `llvm_lifetime_end()` nie je nikde definované navyše spôsobuje chybu pri preklade výstupného zdrojového kódu. Do zdrojového kódu je taktiež pridaná deklarácia funkcie. Keďže v cieľových jazykoch neexistuje a ani nie je potrebný žiaden ekvivalent k tomuto volaniu, môže byť volanie spolu s deklaráciou odstránené.

```
void llvm_lifetime_end(int64_t var1, int8_t *var2);

void func(void) {
    float32_t v1 = 1.6;
    llvm_lifetime_end(4, (int8_t *)&v1);
}
```

Obr. 6.1: Ukážka volania `llvm_lifetime_end()`

6.1.2 Spracovávanie výnimiek

Pri migrácii jazykov D a C++ sa vyskytol problém. LLVM IR podporuje inštrukcie pre prácu s výnimkami (`invoke`), ktoré však nie sú podporované v zadnej časti dekompilátoru. Kód s touto inštrukciou je pomerne často produkovaný prekladom zdrojových kódov v jazyku D a C++. Dekompilátor by pri dekompilovaní LLVM IR súboru s takouto inštrukciou skončil s chybou. Toto je nutné ošetriť pri optimalizácií. Je potrebné vyvinúť transformáciu, ktorá buď inštrukcie a celú prácu s výnimkami úplne odstráni, alebo nahradí kódom, ktorý dekompilátor dokáže spracovať.

6.1.3 Transformácia `InstCombine`

`InstCombine` je transformácia, ktorá je súčasťou LLVM a skladá sa z viacerých podoptimalizácií. Testovaním bolo zistené, že pri jej použití sa výrazne zlepšila čitateľnosť výsledkov migrácie takmer všetkých testovacích zdrojových kódov (napríklad odstránením zbytočných pretopovaní a zjednodušením niektorých častí kódu). Avšak niektorá podoptimalizácia zhoršovala výsledok pri niektorých testovacích zdrojových kódoch. Obrázok 6.2 ukazuje výstup toho istého migrovaného kódu s použitím a bez použitia `InstCombine`. V hornom podobrázku je čitateľnejšia časť, kde sa pracuje s reťazcami a zle čitateľná časť, kde sa vracia hodnota funkcie (časti sú oddelené komentárom). V dolnom podobrázku reťazce úplne zmizli a nie je jasné, čo daný kód vykonáva. Naopak vrátenie hodnoty je na prvý pohľad čitateľnejšie.

Je žiadúce, aby transformácie, ktoré robia kód čitateľnejším, boli zachované. Naopak transformácie podobné tej s prácou s reťazcami z obrázka nie sú obecne v dekompilátore žiadúce (táto transformácia sa používa aj pri bežnej dekompilácii). Nie je možné podoptimalizácie deaktivovať jednotlivo. Preto je vhodné vyhľadať časť kódu, ktorá sa o ňu stará a tento kód v transformácii `InstCombine` deaktivovať. Takto zostanú žiadúce účinky transformácie zachované a tie nežiadúce budú potlačené.

Bez transformácie `InstCombine`

```
int8_t g1[9] = "gt       ";
int8_t g2[9] = "lt or eq";

int32_t fun_(int8_t (*a1)[8], int32_t *a2, int32_t *a3) {
    int8_t v1[8];
    if (*a3 > 10) {
        memcpy(v1, g1, 8);
    } else {
        memcpy(v1, g2, 8);
    }
    memmove(a1, v1, 8);
/* ------------------------------------------------- */
    int32_t v2 = *a2;
    return (int32_t)(v2 > 10 ? v2 > 10 : false);
}
```

S transformáciou `InstCombine`

```
int32_t fun_(int8_t (*a1)[8], int32_t *a2, int32_t *a3) {
    *(int64_t *)a1 = *a3 > 10 ? 0x2020202020207467 :
       ↪       0x716520726f20746c;
/* ------------------------------------------------- */
    return *a2 > 10;
}
```

Obr. 6.2: Ukážka transformácie `InstCombine`

6.2 Jazyk C++

Problémov, ktoré by sa dali riešiť transformáciou v optimalizačnej časti dekompilátoru nebolo mnoho. Okrem problémov spomínaných v sekcii 6.1 bolo na základe testov odhalených niekoľko konštrukcií, ktoré po migrácii neboli preložiteľné. Všetky problémy sa vyskytli pri migrácii virtuálnych alebo čisto virtuálnych metód v triedach. Prvým problémom sú nedefinované premenné. Je spôsobený vynechávaním globálnych premenných bez inicializácie zadnou časťou dekompilátoru. Takéto premenné predná časť negeneruje, a tak ich zadná časť nespracovávala. Druhým problémom je nekorektné generovanie bezmenných štruktúr. Tretím problémom je výskyt funkcií, ktoré sú v LLVM IR kóde iba deklarované a nie sú definované v žiadnom hlavičkovom súbore (napr. `__cxa_pure_virtual()`). Ďalším problémom bolo nesprávne zoraďovanie globálnych premenných podľa závislosti a posledným problémom boli chýbajúce zátvorky okolo volaného výrazu pri prítomnosti dereferencie a pretypovania zároveň. Všetky spomenuté problémy boli predané programátorovi zodpovednému za vývoj zadnej časti dekompilátoru, ktorý ich odstránil. Neboli teda riešené v rámci diplomovej práce, avšak boli vďaka nej odhalené.

6.3 Jazyk Fortran

Táto sekcia popisuje návrh riešení problémov, ktoré sa vyskytli pri migrácii jazyka Fortran. Väčšinou ide o výskyt volaní funkcií z knižnice *libgfortran*. Použitý prekladač *gfortran* prekladá niektoré príkazy a funkcie dostupné v jazyku Fortran na volania funkcií z tejto knižnice, ktorú potom pri preklade prilinkuje. Všetky tieto volania sa následne vyskytujú aj v dekompilovanom kóde, ktorý je výstupom migrácie. Volania zhoršujú čitateľnosť kódu a navyše je pri preklade výstupného zdrojového súboru nutné opäť linkovať knižnicu *libgfortran*.

V sekcii sú rozobrané jednotlivé problémy izolovane, avšak vo výsledných zdrojových kódoch sa vyskytovali súčasne v rôznych kombináciách, čo ich robilo značne neprehľadnými. Tieto volania je pre zlepšenie čitateľnosti vhodné previesť na ich ekvivalenty zo štandardných knižníc jazyka C. V jednotlivých podsekciách je popísané, akým spôsobom boli problematické časti transformované.

6.3.1 Podpora reťazcov

Po migrácii zdrojových kódov, ktoré obsahovali textové reťazce, boli tieto reťazce transformované na pole čísel. Na funkcionalitu to síce vplyv nemá, ale pre človeka sa takto reťazce migráciou stali nečitateľnými a nerozoznateľnými od skutočných polí. V prípade, že ich bolo v migrovanom zdrojovom kóde viacero, problém bol citeľnejší. Vďaka diplomovej práci bol tento problém iba odhalený, vyriešený bol programátorom zadnej časti dekompilátoru. Problém demonštruje obrázok 6.3. Na obrázku sú tri polia. Prvé dve uchovávajú reťazce a tretie je číselné. Horná časť zobrazuje pôvodnú podobu polí a spodná časť zobrazuje ich podobu po úprave v zadnej časti dekompilátoru. Rozdiel v čitateľnosti je značný.

Migrované reťazce pred úpravou v zadnej časti dekompilátoru

```
int8_t g1[14] = {104, 101, 108, 108, 111, 95, 119, 111,
   ↪ 114, 108, 100, 46, 102, 0};
int8_t g2[12] = {72, 101, 108, 108, 111, 32, 87, 111,
   ↪ 114, 108, 100, 33};
int32_t g3[8] = {68, 1023, 0, 0, 1, 1, 0, 1};
```

Migrované reťazce po úprave v zadnej časti dekompilátoru

```
int8_t g1[14] = "hello_world.f\x00";
int8_t g2[12] = "Hello World!";
int32_t g3[8] = {68, 1023, 0, 0, 1, 1, 0, 1};
```

Obr. 6.3: Problém s migráciou reťazcov

6.3.2 Volanie `_gfortran_set_options()`

Volanie `_gfortran_set_options()` sa po migrácii zdrojových kódov v jazyku Fortran nachádza v každom výstupnom zdrojovom kóde vo funkcii `main()`. Ukážka volania je na obrázku 6.4. Funkcia nastavuje niekoľko nastavení vzťahujúcich sa k Fortran štandardu.

Ide napríklad o zapnutie/vypnutie spätného trasovania (angl. backtracking), nastavenie varovaní pri výnimkách pri práci s pohyblivou desatinnou čiarkou, nastavenie použitého štandardu [8]. Všetky nastavenia možno nájsť v [8]. Vo výstupných zdrojových kódoch sa taktiež nachádza pole hodnôt, ktoré predstavujú hodnoty týchto nastavení. Vždy sú nastavené hodnoty, ktoré sú predvolené. Nastavenie je teda zbytočné a navyše ide o nastavenia, ktoré v cieľových jazykoch nemajú svoj ekvivalent. Z toho dôvodu môže byť volanie spolu s polom hodnôt nastavení odstránené.

```
void _gfortran_set_options(int32_t var8, int32_t *var9);

int32_t g1[8] = {68, 1023, 0, 0, 1, 1, 0, 1};

int main(int a1, char **a2) {
    _gfortran_set_options(8, g1);
    return 0;
}
```

Obr. 6.4: Ukážka volania `_gfortran_set_options()`

6.3.3 Volanie `_gfortran_iargc()`

Volanie `_gfortran_iargc()` vracia počet parametrov programu. Ukážka sa nachádza na obrázku 6.5. V jazyku C je k dispozícii premenná funkcie `main()`, typicky nazvaná `argc`. Naskytuje sa možnosť volanie `_gfortran_iargc()` nahradiť touto premennou. Volanie funkcie jazyka Fortran `iargc()` môže však byť vo vstupnom kóde prítomné takmer kdekoľvek. Jeho migrovaný ekvivalent z knižnice gfortran `_gfortran_iargc()` sa potom môže vyskytnúť hlboko vnorený (z pohľadu funkcie `main()`) v rôznych funkciách. Dodatočná propagácia premennej `argc` do týchto funkcií by bola značne problematická. Čitateľnosť kódu by sa týmto spôsobom nezlepšila, dokonca by mohlo dôjsť k jej zhoršeniu. Vhodnejšie riešenie je teda vytvoriť globálnu premennú, ktorá bude na začiatku funkcie `main()` inicializovaná hodnotou `argc` a touto premennou jednoducho nahradiť všetky volania `_gfortran_iargc()`.

6.3.4 Volanie `_gfortran_getarg_i4()`

Volanie `_gfortran_getarg_i4()` vracia požadovaný argument programu. Prijíma tri parametre: poradie požadovaného argumentu, pole, do ktorého bude argument skopírovaný a veľkosť tohto pola. Na ukážke na obrázku 6.6 môžeme vidieť, že pri volaní je nutné pretypovanie funkcie, čo ho robí značne nečitateľným. Ďalším problémom je deklarácia funkcie, pretože štandard jazyka C vyžaduje, aby pri funkciách s premenným počtom argumentov bol aspoň jeden fixný pomenovaný argument. Zdrojový kód je teda nepreložiteľný, a preto musí byť volanie odstránené.

V jazyku C je k dispozícii pole argumentov `argv`. Volanie `_gfortran_getarg_i4()` môže byť vnorené podobne ako volanie `_gfortran_iargc()`. Z rovnakého dôvodu teda nie je vhodné `argv` propagovať. Lepším riešením je opäť využitie globálnej premennej, ktorá bude

```
int32_t _gfortran_iargc(void);

int32_t fun_(int32_t *a) {
    return *a + _gfortran_iargc();
}

int main(int a1, char **a2) {
    int32_t x = 1;
    int32_t y = fun_(&x);
    return y;
}
```

Obr. 6.5: Ukážka volania `_gfortran_iargc()`

na začiatku funkcie `main()` inicializovaná hodnotou `argv`. Volania `_gfortran_getarg_i4()` budú potom nahradené volaním funkcie `memcpy()`.

```
void _gfortran_getarg_i4(...);

int main(int a1, char **a2) {
    int8_t v1[32];
    int32_t v2 = 1;
    ((void (*)(int32_t *, int8_t (*)[32], int32_t))
        _gfortran_getarg_i4)(&v2, &v1, 32);
}
```

Obr. 6.6: Ukážka volania `_gfortran_getarg_i4()`

6.3.5 Volanie `_gfortran_set_args()`

Volanie `_gfortran_set_args()` sa podobne ako volanie `_gfortran_set_options()` nachádza v každom migrovanom zdrojovom kóde vo funkcii `main()`. Ukážka je na obrázku 6.7. Funkcia je potrebná ako inicializácia pred použitím `_gfortran_iargc()` a `_gfortran_getarg_i4()`. Ak budú volania týchto funkcií nahradené, je možné volanie `_gfortran_set_args()` odstrániť.

6.3.6 Volanie `_gfortran_exit_i4()`

Volanie `_gfortran_exit_i4()` sa po migrácii nachádza tam, kde sa pôvodný program v jazyku Fortran ukončoval so špecifikovaným návratovým kódom. Na obrázku 6.8 je možné vidieť, že volanie je značne nečitateľné kvôli pretypovaniu. Výstupný zdrojový kód je taktiež nepreložiteľný kvôli problému s deklaráciou a premenným počtom parametrov, ako tomu je pri `_gfortran_getarg_i4()`. Volanie môže byť nahradené volaním `exit()` zo štandardnej knižnice jazyka C.

```
void _gfortran_set_args(int32_t var1, int8_t **var2);

int main(int a1, char **a2) {
    _gfortran_set_args(a1, a2);
    return 0;
}
```

Obr. 6.7: Ukážka volania `_gfortran_set_args()`

```
void _gfortran_exit_i4(...);

int main(int a1, char **a2) {
    int32_t v1 = 0;
    ((void (*)(int32_t *))_gfortran_exit_i4)(&v1);
}
```

Obr. 6.8: Ukážka volania `_gfortran_exit_i4()`

6.3.7 Volania funkcií na výpočet mocniny

Vo výstupe migrácie boli použité tri rôzne funkcie na výpočet mocniny. Ide o funkcie `_gfortran_pow_i4_i4()`, `llvm_powi_f32()` a `llvm_powi_f64()`. Všetky vracajú n-tú mocninu čísla m, kde m je prvý parameter funkcie a n druhý parameter. V pôvodnom zdrojovom kóde v jazyku Fortran: `m ** n`. Ukážka je na obrázku 6.9. Prvé volanie sa objaví tam, kde mali m aj n pôvodný typ `integer`. Druhé volanie ja objaví tam, kde m bol typ `real` a n typ `integer` a tretie tam, kde m bol typ `double precision` a n typ `integer`. Pri ostatných kombináciách problém nebol. Všetky tieto tri volania môžu byť nahradené volaním `pow()` zo štandardnej knižnice jazyka C.

```
int32_t _gfortran_pow_i4_i4(int32_t var1, int32_t var2);

int main(int a1, char **a2) {
    int32_t v1 = 256;
    return _gfortran_pow_i4_i4(v1, v1 + 1);
}
```

Obr. 6.9: Ukážka volania `_gfortran_pow_i4_i4()`

6.3.8 Vstupno-výstupné funkcie

Jedným z najväčších problémov je migrácia vstupno-výstupných (skrátene IO, z anglického input-output) funkcií. Na obrázku 6.10 je zdrojový kód s dvomi príkazmi `WRITE`. Výstup

po migrácii je vidieť na obrázkoch 6.11 a 6.12. Výstup je oproti pôvodnému zdrojovému kódu niekoľko násobne dlhší a podstatne horšie čitateľnejší. Výstup bol navyše ručne skrátený naformátovaním definícií štruktúr do troch stĺpcov. Rovnako je tomu aj pri migrovaní vstupných príkazov.

```
program test
    integer :: a
    real :: b
    a = 47
    b = 3.14
    WRITE (*,*) "Hello", a
    WRITE (*,*) "World", b
end program test
```

Obr. 6.10: Problém pri migrácii IO funkcií, vstup v jazyku Fortran

Odkaz na dokumentáciu ku knižnici *libgfortran* počas písania diplomovej práce nefungoval. Nepomohlo ani problém nahlásiť, a tak bolo nutné pristúpiť k analýze s využitím reverzného inžinierstva. Po dôkladnom analyzovaní problému bolo zistené, že ako v prípade práce so štandardným výstupom, štandardným chybovým výstupom aj práce so súbormi sú použité rovnaké funkcie. Parametre vstupno-výstupných funkcií sú nastavené pomocou štruktúr `struct__st_parameter_common` a `struct__st_parameter_dt`.

Prvá položka štruktúry `struct__st_parameter_dt` je štruktúra `struct__st_parameter_common`, ktorej druhá položka je číslo, ktoré identifikuje výstupný prúd. Význačné hodnoty sú:

- 0 – štandardný chybový výstup,
- 5 – štandardný vstup,
- 6 – štandardný výstup.

Ďalej sa tu môže vyskytnúť číslo, ktoré identifikuje otvorený súbor. Toto číslo je v pôvodnom zdrojovom kóde predávané ako prvý parameter funkcie `open()` z jazyka Fortran. Tento identifikátor si programátor volí sám.

Ďalšia podstatná položka zo štruktúr je šiesta položka štruktúry `struct__st_parameter_dt`. Táto položka obsahuje ukazovateľ na formátovací reťazec. Formátovací reťazec nemusí byť použitý. V takom prípade táto položka po inicializácii hodnotou `NULL` už nie je ďalej nastavovaná.

Keďže možnosti formátovania v jazyku Fortran sú pomerne rozsiahle [21], bolo by náročné transformovať ich na formátovacie reťazce štandardných funkcií jazyka C. Z toho dôvodu boli v práci zvolené na ďalší prevod len funkcie bez použitého formátovacieho reťazca.

Druhé obmedzenie zamerania práce je na transformáciu funkcií zapisujúcich len na štandardný výstup alebo čítajúcich zo štandardného vstupu. Je to z toho dôvodu, že v LLVM IR je nutné uvádzať typy, čo spolu s typom `FILE` vytvára problém. Typ `FILE` je nutné použiť vždy, keď chceme špecifikovať vstup a výstup iný, než štandardný vstup a štandardný výstup. V LLVM IR je teda nutné definovať štruktúru `FILE`. Po dekompilácii však na výstupe

zostane definícia tejto štruktúry, a tak je typ `FILE` na výstupe odlišný od typu `FILE` zo štandardnej knižnice jazyka C.

```
struct struct___st_parameter_common {
    int32_t e0;   int32_t e1;   int8_t *e2;
    int32_t e3;   int32_t e4;   int8_t *e5;
    int32_t *e6;
};

struct struct___st_parameter_dt {
    struct struct___st_parameter_common e0;
    int64_t e1;       int64_t *e2;    int64_t *e3;
    int8_t *e4;       int8_t *e5;     int32_t e6;
    int32_t e7;       int8_t *e8;     int8_t *e9;
    int32_t e10;      int32_t e11;    int8_t *e12;
    int8_t e13[256]; int32_t *e14;   int64_t e15;
    int8_t *e16;      int32_t e17;    int32_t e18;
    int8_t *e19;      int8_t *e20;    int32_t e21;
    int32_t e22;      int8_t *e23;    int8_t *e24;
    int32_t e25;      int32_t e26;    int8_t *e27;
    int8_t *e28;      int32_t e29;    int8_t e30[4];
};

void _gfortran_st_write(struct struct___st_parameter_dt *
   ↪ var1);
void _gfortran_transfer_character_write(struct
   ↪ struct___st_parameter_dt *var2, int8_t *var3,
   ↪ int32_t var4);
void _gfortran_transfer_integer_write(struct
   ↪ struct___st_parameter_dt *var5, int8_t *var6,
   ↪ int32_t var7);
void _gfortran_transfer_real_write(struct
   ↪ struct___st_parameter_dt *var8, int8_t *var9,
   ↪ int32_t var11);
void _gfortran_st_write_done(struct
   ↪ struct___st_parameter_dt *var10);

int8_t g1[92] = "write.f\x00";
int8_t g2[5] = "Hello";
int8_t g3[5] = "World";

int main(int a1, char **a2) {
    struct struct___st_parameter_dt v1;
    struct struct___st_parameter_dt v2;
```

Obr. 6.11: Problém pri migrácii IO funkcií, výstup v jazyku C, prvá časť

```
    int32_t v3 = 47;
    float32_t v4 = 3.1400001;
    v1 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v1.e0.e2 = &g1[0];
    v1.e0.e3 = 6;
    v1.e0.e0 = 128;
    v1.e0.e1 = 6;
    _gfortran_st_write(&v1);
    _gfortran_transfer_character_write(&v1, g2, 5);
    _gfortran_transfer_integer_write(&v1, (int8_t *)&v3,
       ↪ 4);
    _gfortran_st_write_done(&v1);
    v2 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v2.e0.e2 = &g1[0];
    v2.e0.e3 = 7;
    v2.e0.e0 = 128;
    v2.e0.e1 = 6;
    _gfortran_st_write(&v2);
    _gfortran_transfer_character_write(&v2, g3, 5);
    _gfortran_transfer_real_write(&v2, (int8_t *)&v4, 4);
    _gfortran_st_write_done(&v2);
}
```

Obr. 6.12: Problém pri migrácii IO funkcií, výstup v jazyku C, druhá časť

Z obrázkov 6.10, 6.11 a 6.12 je vidieť, že pre každý príkaz pracujúci so vstupom alebo výstupom v pôvodnom zdrojovom kóde je použitá jedna inštancia štruktúry `struct___st_parameter_dt` a niekoľko volaní, ktoré sa starajú o samotný výpis. Prvému príkazu `WRITE` z obrázku 6.10 odpovedá na obrázku 6.12 štruktúra `v1`. Priradenie `v1.e0.e1 = 6` nám ho-

vorí o použití štandardného výstupu a keďže `v1.e5` nebola okrem inicializácie nijak inak nastavená, nebol použitý formátovací reťazec. K štruktúre sú ďalej viazané funkcie: inicializačná funkcia `_gfortran_st_write()`, ukončovacia funkcia `_gfortran_st_write_done()` a funkcia na samotný výpis `_gfortran_transfer_character_write()` a `_gfortran_transfer_integer_write()`. `_gfortran_transfer_character_write()` vypisuje reťazec a `_gfortran_transfer_integer_write()` vypisuje celé číslo.

Z príkladu popísaného vyššie vyplýva, že celá transformácia môže byť založená na vyhľadaní deklarácie štruktúry `struct___st_parameter_dt`, skontrolovaní, či sa nastavuje formátovací reťazec a zistení, či sa pracuje so štandardným vstupom alebo výstupom. Ak sú podmienky splnené, stačí prejsť všetky použitia štruktúry a transformovať všetky volania, ktoré sa starajú o samotné načítanie alebo výpis, na funkcie `scanf()` alebo `printf()`. Na záver, ak zostanú len inicializačné a ukončovacie volanie (u zápisu je to `_gfortran_st_write()` a `_gfortran_st_write_done()`, pri čítaní `_gfortran_st_read()` a `_gfortran_st_read_done()`), je možné ich obe zmazať. Rovnako môže byť zmazaná celá štruktúra a s ňou aj jej inicializácia, ktorá predstavuje veľkú časť kódu. Ak budú takto spracované a zmazané všetky štruktúry, môžeme zmazať aj deklaráciu. Z kódu, ktorý zaberá dva obrázky 6.11 a 6.12, potom zostane len funkcia `main()` a v nej štyri volania `printf()` (ak nepočítame definíciu vypisovaných premenných). Dôjde tak k podstatnej redukcii kódu a značnému zlepšeniu čitateľnosti.

Transformované budú funkcie `_gfortran_transfer_character_write()`, `_gfortran_transfer_integer_write()`, `_gfortran_transfer_real_write()`, `_gfortran_transfer_logical_write()`. Funkcia `_gfortran_transfer_array_write()` bude kvôli zložitosti transformácie vynechaná. Logické hodnoty sú vypisované ako `F` a `T`. Výpis `0` a `1` sú postačujúca alternatíva, a tak bude pri volaní `printf()` použitý formátovací reťazec `"%d"`. Taký istý reťazec bude použitý aj pri nahradzovaní volania `_gfortran_transfer_integer_write()`. Pri volaní `_gfortran_transfer_real_write()` to bude `"%f"`.

Pri volaní `_gfortran_transfer_character_write()` je menší problém v tom, že migrované reťazce z jazyka Fortran na výstupe neobsahujú ukončovací znak `'\0'`. Pri použití formátovacieho reťazca `"%s"` by funkcia `printf()` vytlačila obsah pamäte za reťazcom až po prvý znak `'\0'` nachádzajúci sa náhodne v pamäti. Riešenie, keď by sa do reťazca doplňoval znak `'\0'` by bolo zložité. Riešenie použiť funkciu `fwrite()`, kde máme možnosť špecifikovať dĺžku je neprehľadné. Problém rieši úprava formátovacieho reťazca do tvaru `"%.*s"`. Potom sa funkcii `printf()` ako druhý parameter predá dĺžka vypisovaného reťazca, ktorý bude až tretím parametrom.

Taktiež budú transformované funkcie, ktoré sa starajú o načítanie `_gfortran_transfer_integer()`, `_gfortran_transfer_logical()` na `scanf()` s použitím formátovacieho reťazca `"%i"` a `_gfortran_transfer_real()` a použitím reťazca `"%f"`. Funkcia `_gfortran_transfer_complex()` bude kvôli zložitosti transformácie vynechaná. Vynechaná bude taktiež funkcia `_gfortran_transfer_character()`. Dôvod je ten, že funkcia `scanf()` sa nedá nastaviť tak, aby načítala nebiele (angl. non-whitespace) znaky po prvý biely (angl. whitespace) znak do danej maximálnej veľkosti a zároveň aby nepridala ukončovací znak `'\0'`. V podstate ide o kombináciu funkcionality špecifikátorov `"c"` a `"s"` s určením maximálneho počtu načítaných znakov.

6.4 Jazyk D

Táto sekcia sa zaoberá problémami spojenými s migráciou jazyka D. Okrem problémov spomínaných v sekcii 6.1 to je problém s dekompilovanou funkciou `main()` a následnou

prácou s parametrami programu a problém vložených definícií funkcií knižníc.

6.4.1 Funkcia `main()` a práca s parametrami programu

Po migrácii jazyka D výstupný zdrojový kód obsahoval funkciu `main()` v neštandardnom tvare. Na obrázku 6.13 je vidieť, že funkcia prijíma jeden parameter a tým je štruktúra typu `struct2`. Prvý člen tejto štruktúry predstavuje dĺžku pola parametrov programu, druhý člen predstavuje samotné pole argumentov. Parameter je štruktúra typu `struct1` a obsahuje, podobne ako štruktúra `struct2`, dĺžku a pole znakov.

```
struct struct1 {
    int64_t e0;
    int8_t *e1;
};

struct struct2 {
    int64_t e0;
    struct struct1 *e1;
};

int32_t _Dmain(struct struct2 a) {
    return 0;
}
```

Obr. 6.13: Funkcia `main()` po migrácii jazyka D

Aby bol kód preložiteľný a bolo možné použiť parametre programu, je nutné dostať funkciu `main()` do správneho tvaru. To si vyžaduje jej premenovanie a zmenu typu. Po zmene typu bude potrebné ešte nahradiť prístup k argumentom cez pôvodné štruktúry za prístup cez premenné `argc` a `argv` z upravenej funkcie `main()`.

6.4.2 Definície funkcií knižníc

Ďalší problém, ktorý má zásadný vplyv na čitateľnosť kódu je existencia definícií funkcií z knižníc. Telá funkcií sú často vkladané pri preklade zdrojových kódov v jazyku D. Nie je žiadúce, aby sa tieto definície vo výslednom migrovanom kóde nachádzali. Definície sú vo výstupe zbytočné a ak uvážime, že prekladač takto vloží definície všetkých funkcií, ktoré sa navzájom volajú, výsledný zdrojový kód je značne nečitateľný a oproti pôvodnému zdrojovému kódu neúmerne dlhý. Riešením je opäť, tak ako v predošlých prípadoch, implementácia odstránenia definícií cez transformáciu prostrednej časti dekompilátoru. Po optimalizácii zostanú len deklarácie funkcií volaných priamo zo zdrojového kódu.

Kapitola 7

Implementácia

Táto kapitola sa venuje implementácii migračného nástroja, zavádzaní jednotlivých vstupných jazykov a implementácii transformácií zlepšujúcich kvalitu a čitateľnosť výstupu migrácie.

Kapitola je rozdelená do niekoľkých sekcií. Nasledujúca sekcia popisuje implementáciu migračného skriptu, sekcia 7.2 popisuje zavádzanie jednotlivých prekladačov, sekcia 7.3 popisuje najrozsiahlejšiu časť implementácie, a tou sú transformácie vylepšujúce výstup migrácie. V sekcii 7.4 je popísané riešenie problému s výnimkami a posledná sekcia 7.5 sa venuje transformácii `InstCombine`.

7.1 Migračný skript

Migračný nástroj predstavuje *shell* skript a transformácie aplikované prostrednou časťou dekompilátoru. Skript síce zastrešuje vybrané prekladače, prostrednú a zadnú časť dekompilátoru a riadi celú migráciu, ale nie je najvýznamnejšou časťou migračného nástroja. Tou sú implementované transformácie.

Tak ako ostatné skripty, ktoré sú s dekompilátorom k dispozícii, aj tento je vytvorený formou šablóny *migrate-hll.template*, do ktorej sú doplnené napríklad cesty k nástrojom, názvy použitých programov a pod. Doplnenie je vykonané pri inštalácii (*make install*), keď sa zo šablóny stane *shell* skript *migrate-hll.sh*.

Okrem spracovania parametrov pomocou *getopt* skript po spustení podľa prípony rozpozná, o aký vstupný jazyk ide a vyberie prekladač, ktorý bude použitý na preklad vstupného binárneho súboru do LLVM IR. V prípade, že je skript spustený v prostredí operačného systému Windows a cieľový jazyk je D, je do premennej `PATH` pridaná cesta k binárnym súborom *MinGW-w64* (Minimalist GNU for Windows verzia pre x86 aj x64), ktoré prekladač potrebuje pre svoj beh. Po preklade do LLVM IR je pôvodná premenná `PATH` obnovená. Následne sa výsledný *.ll* súbor dekompiluje existujúcim dekompilačným skriptom. Šablóna tohto skriptu má názov *decompile.template*. Skript dekompiluje *.ll* súbor do zvoleného vysoko úrovňového jazyka.

Migračný skript prijíma minimálne jeden parameter, ktorým je cesta k migrovanému súboru. Ďalej je možné použiť doplňujúce parametre a ručne špecifikovať:

- vstupný jazyk (nutné v prípade neznámej prípony),
- názov výstupného súboru (predvolený názov je názov vstupného súboru s príponou cieľového jazyka),

- zvoliť výpis nápovedy skriptu,
- špecifikovať parametre, ktoré budu predané dekompilačnému skriptu,
- cieľový jazyk (predvolený jazyk je C),
- zvoliť zmazanie dočasný súborov vytvorených počas migrácie (medzi mini je aj *.ll* súbor).

Posledné dve nastavenia sú tiež propagované dekompilačnému skriptu. Do dekompilačného skriptu bol implementovaný parameter (spolu s funkcionalitou), ktorý určuje, že sa namiesto transformácií používaných pri dekompilácii použijú transformácie určené pre migráciu. Tieto transformácie sú špecifikované v skripte *config.sh*, ktorý má šablónu *config.template*. V skripte *config.sh* sú okrem toho cesty k prekladačom vstupných jazykov spolu s niektorými parametrami (napr. špecifikácia, že výstup ma byť v LLVM IR), ošetrenie odlišností spojených s operačným systémom, na ktorom sú skripty spúšťané.

Migračný skript vracia nasledujúce návratové kódy (využíva sa najmä pri testovaní migrácie):

- `0` – skript prebehol bez chýb,
- `1` – obecná chyba (chyby nespadajúce do nasledujúcich dvoch kategórií, napr. chybné parametre, chyba pri rozpoznávaní vstupného jazyka atď.),
- `2` – chyba pri preklade vstupného súboru,
- `3` – chyba pri dekompilácii *.ll* súboru.

7.2 Vstupné jazyky a ich prekladače

Sekcia sa zaoberá zavádzaním vstupných jazykov do migračného nástroja. Zavedenie jazyka bolo realizované pridaním prekladača. Pri každom prekladači bolo okrem špecifických požiadavkov popísaných v podsekciách potrebné nastaviť, aby bol výstupom kód v LLVM IR.

7.2.1 D

V rámci projektu LDC je vyvíjaný prekladač jazyka D – *ldc2*. Tento prekladač je distribuovaný ako binárny balíček pre všetky platformy, ktoré sú požadované. Zavedenie prekladača bolo pomerne jednoduché. Ako už bolo spomenuté v popise migračného skriptu, pri platformách Windows prekladač potrebuje poznať cestu k *MinGW-w64*, ktorá mu je poskytnutá v migračnom skripte exportovaním premennej `PATH`.

7.2.2 C, C++, Objective-C a Objective-C++

Keďže prekladač *clang* je už integrovaný do dekompilátoru, jeho zavedenie do migračného nástroja bolo pomerne jednoduché. Pre preklad jazykov Objective-C a Objective-C++ bolo však nutné dodať potrebné knižnice. Tieto knižnice poskytuje projekt GNUstep[1].

[1] http://www.gnustep.org/

7.2.3 Fortran, jeho dialekty a ADA

O preklad týchto jazykov sa mal starať prekladač GCC s pluginom *Dragonegg*. Po množstve testov sa však nepodarilo spojazdniť preklad jazyka ADA. Prekladač *gnatgcc* (GCC s prednou časťou pre jazyk ADA) hlásil chybu pri zavádzaní pluginu *Dragonegg*. S jazykom Fortran tento problém nie je, a tak bol jazyk ADA vyradený z ďalšej integrácie.

Keďže plugin *Dragonegg* je vyvíjaný len pre linuxové platformy, ďalším problémom bolo spojazdniť preklad na platforme Windows. Existuje však projekt *pcxllvm*[2], ktorého cieľom je vytvoriť port pre platformu Windows. Zo stránok projektu je možné stiahnuť balíček MinGW s pluginom *Dragonegg* v binárnej podobe. Nejde však použiť kvôli závislostiam na rôznych knižniciach, ktoré nie sú s balíčkom distribuované. Ďalší projekt, ktorý poskytuje balíček MinGW s pluginom *Dragonegg* v binárnej podobe je *C::B advanced*[3]. V tomto balíčku plugin *Dragonegg* nejde použiť vôbec (pri preklade prestane reagovať). Balíček však obsahuje knižnice, ktoré v prvom spomínanom balíčku chýbajú. Pri dodaní knižníc k prvému balíčku je konečne možné plugin *Dragonegg* použiť na platforme Windows.

7.3 Transformácie

Táto sekcia popisuje implementované transformácie, ktoré sú aplikované pri dekompilácii kódu v LLVM IR prostrednou časťou dekompilátoru. Každá podsekcia popisuje jednu transformáciu.

7.3.1 Transformácia `remove-llvm-calls`

Táto transformácia sa stará o riešenie problému popísaného v podsekcii 6.1.1. Odstraňuje z výsledného migrovaného zdrojového kódu volania funkcie `llvm_lifetime_end()` a jej deklaráciu. Táto funkcia vzniká dekompiláciou LLVM IR funkcie `llvm.lifetime.end()`. V transformácii teda budeme odstraňovať túto funkciu z LLVM IR. Názov triedy, ktorá transformáciu implementuje je `LLVMCallsRemover`.

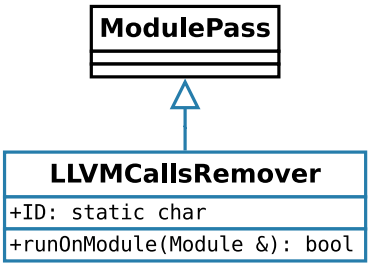
Pred tým, ako sa začne implementovať každá transformácia, je nutné stanoviť, od ktorej triedy bude dediť podľa toho, čo bude vykonávať. Na odstránenie volaní funkcie `llvm.lifetime.end()` by stačilo dediť od triedy `BasicBlockPass` a v implementácii funkcie `runOnBasicBlock()` iterovať inštrukciami (trieda `Instruction`) a hľadanú inštrukciu typu `CallInst` odstrániť. Keďže je ale potrebné odstrániť aj deklaráciu funkcie (tj. práca s modulom), je nutné, aby bola trieda `LLVMCallsRemover` potomkom triedy `ModulePass`.

Hľadanie volaní je teda implementované v metóde `runOnModule()` iterovaním cez funkcie pomocou iterátora `Module::iterator`, v ktorých sa iteruje cez základné bloky pomocou iterátora `Function::iterator` a v nich cez inštrukcie pomocou iterátora `BasicBlock::iterator`. Pomocou šablóny `dyn_cast<>` sa otestuje, či ide o inštrukciu typu `CallInst`. V prípade, že áno, metódou `getCalledFunction()` sa získa ukazovateľ typu `Function` na volanú funkciu a pomocou metódy `getName()` je možné porovnať jej názov s reťazcom `"llvm.lifetime.end"`. Ak sú zhodné, inštrukcia je odstránená metódou `eraseFromParent()`. Následne je nastavený príznak zmeny na hodnotu `true`.

Po odstránení všetkých volaní sa opäť iteruje cez funkcie. V prípade, že je funkcia deklarácia (`isDeclaration()`), nemá žiadne použitie (`use_empty()`) a jej názov je `"llvm.life-`

[2] https://code.google.com/p/pcxllvm/
[3] http://sourceforge.net/projects/cbadvanced/

`time.end"`, je taktiež odstránená metódou `eraseFromParent()`. Príznak zmeny je opäť nastavený na hodnotu `true` a iterácia je ukončená kvôli úspore času.

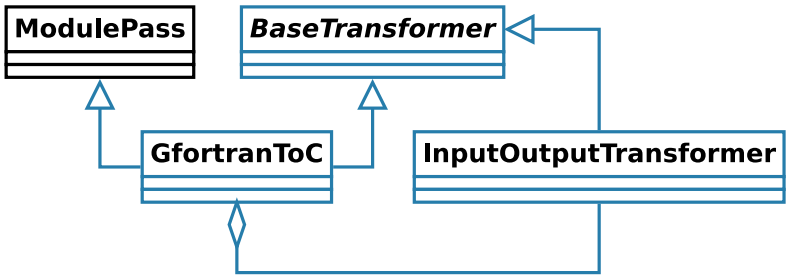


Obr. 7.1: Vzťah medzi triedamy transformácie `remove-llvm-calls`

Na obrázku 7.1 ja znázornený diagram tried. Modro zvýraznená časť bola implementovaná v rámci práce. Na obrázku je vidieť dedičnosť triedy `LLVMCallsRemover` od triedy `ModulePass`. Ako bolo spomenuté, trieda implementuje metódu `runOnModule()`. Trieda má verejnú statickú premennú `ID`, ktorá súvisí s identifikáciou transformácie. Hodnota tejto premennej nie je podstatná, pretože sa na identifikáciu transformácie používa jej adresa.

7.3.2 Transformácia `gfortran-to-c`

Táto transformácia zastrešuje riešenie všetkých problémov, ktoré sa vyskytujú vo výstupoch migrácie zdrojových kódov z jazyka Fortran za pomoci prekladača *gfortran*. Na obrázku 7.2 je znázornený diagram tried. Implementovaná časť je zvýraznená modrou farbou.



Obr. 7.2: Vzťah medzi triedamy transformácie `gfortran-to-c`

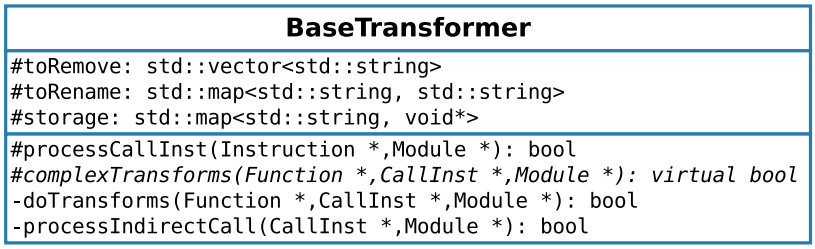
Základom celej transformácie je abstraktná trieda `BaseTransformer`. Od tejto triedy dedia triedy `InputOutputTransformer` a `GfortranToC`. Trieda `InputOutputTransformer` je súčasťou triedy `GfortranToC` ako jej privátna premenná. Trieda `GfortranToC` je potomkom dvoch tried. Okrem triedy `BaseTransformer` je to aj trieda `ModulePass`.

Trieda `BaseTransformer`

Trieda `BaseTransformer` je znázornená na obrázku 7.3. Obsahuje tri chránené (angl. protected) premenné. Sú nimi:

- vektor `toRemove`, ktorý ma slúžiť ako zoznam názvov volaní, ktoré môžu byť jednoducho odstránené,

- asociatívne pole `toRename`, ktoré slúži ako zoznam volaní určených na jednoduché premenovanie (pôvodný názov je kľúč, nový názov hodnota),
- asociatívne pole `storage`, ktoré slúži na odkladanie ukazovateľov na objekty pri vykonávaní transformácií (kľúčom je reťazec, hodnotou ukazovateľ na typ `void`).



Obr. 7.3: Trieda `BaseTransformer`

Trieda obsahuje taktiež chránené metódy `complexTransforms()` a `processCallInst()`. Metóda `complexTransforms()` je čisto virtuálna metóda (angl. pure virtual). Robí triedu abstraktnou. Metóda je určená na implementáciu kódu v potomkovi, ktorý bude volať jednotlivé zložitejšie transformácie implementované taktiež v potomkovi. Ako parameter prijíma ukazovateľ na volanú funkciu typu `Function`, ukazovateľ na inštrukciu volania typu `CallInst` a ukazovateľ na modul (typ `Module`), nad ktorým transformácia pracuje.

Metóda `processCallInst()` je metóda, ktorá zahajuje transformáciu volacích inštrukcií. Mala by byť volaná potomkom po tom, čo potomok určí, ktoré inštrukcie by mali byť spracované. Tie sú metóde predané cez ukazovateľ na inštrukciu (typ `Instruction`). Metóda prijíma taktiež ukazovateľ na modul (typ `Module`), nad ktorým transformácia pracuje.

Metóda skontroluje typ inštrukcie a v prípade, že ide o inštrukciu `CallInst`, zavolá sa jej metóda `getCalledFunction()`, ktorá vráti buď volanú funkciu, alebo hodnotu `NULL` v pripade, že ide o nepriame volanie (angl. indirect call). Príklad priameho a nepriameho volania je možné vidieť na obrázku 7.4. Súčasťou nepriameho volania v spodnej časti obrázku je inštrukcia pretypovania. Ak teda pôjde o priame volanie, metóda `processCallInst()` zavolá privátnu metódu `doTransforms()`. V prípade nepriameho volania zavolá privátnu metódu `processIndirectCall()`. Výsledok volanej privátnej metódy sa vráti ako výsledok. Výsledok je príznakom zmeny. Je to hodnota typu `bool`.

Priame volanie

```
call void @llvm.lifetime.end(i64 4, i8* %0)
```

Nepriame volanie

```
call void bitcast (void (...)* @_gfortran_exit_i4 to void
   ↪  (i32*)*)(i32* %b) #1
```

Obr. 7.4: Dva druhy volaní inštrukcie `CallInst`

Privátna metóda `processIndirectCall()` spracováva nepriame volania. Ako parameter prijíma ukazovateľ na inštrukciu `CallInst` a na modul. Účelom metódy je dostať sa k volanej funkcii. Keďže metóda `getCalledFunction()` vracia hodnotu `NULL` je nutné zavolať metódu `getCalledValue()`. Tá vracia ukazovateľ na typ `Value`. V prípade, že je možné tento ukazovateľ pretypovať na ukazovateľ na typ `Constant`, otestuje sa, či nejde o ukazovateľ na typ `GlobalValue`. V prípade, že nie a je možné tento ukazovateľ pretypovať na ukazovateľ na typ `ConstantExpr` môže byť prvý operand výrazu hľadaná funkcia. To otestujeme, a ak bude test úspešný, metóda zavolá privátnu metódu `doTransforms()` a predá jej ukazovateľ na funkciu, ukazovateľ na inštrukciu `CallInst` a ukazovateľ na modul. Výsledok volanej metódy je potom vrátený. V prípade, že by niektorý z popísaných testov neprešiel, je vrátená hodnota `false`.

Poslednou metódou je metóda `doTransforms()`. Tá sa stará o aplikovanie implementovaných konkrétnych transformácií. Najskôr iteruje vektorom `toRemove` a kontroluje, či sa názov volanej funkcie, ktorú obdrží ako parameter, nenachádza v tomto vektore. Ak áno, inštrukciu `CallInst`, ktorú metóda dostane ako druhý parameter, zmaže volaním `eraseFromParent()` a vráti hodnotu `true` ako indikátor zmeny. Ak sa názov funkcie vo vektore nenachádza, metóda pristúpi k druhému kroku, a tým je iterácia cez asociatívne pole `toRename`. Kontroluje názov funkcie voči kľúčom asociatívneho pola a ak narazí na zhodu, premenuje funkciu na hodnotu asociovanú ku kľúču metódou `setName()` a vráti hodnotu `true` ako indikátor zmeny. Ak sa názov funkcie zo spracovávanej inštrukcie nenachádza ani v tomto asociatívnom poli, metóda pristúpi k tretiemu kroku. V tom zavolá už spomenutú chránenú metódu `complexTransforms()` a vráti jej návratovú hodnotu.

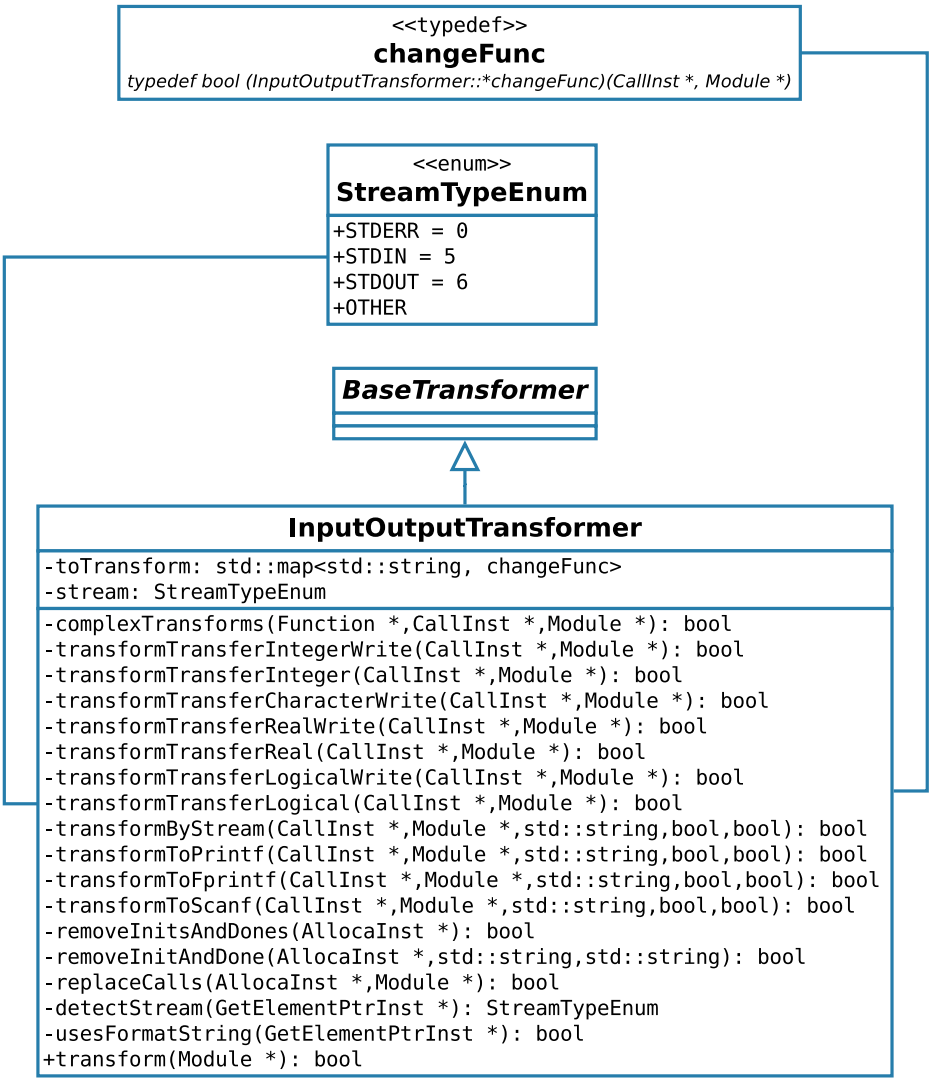
Trieda `InputOutputTransformer`

Trieda `InputOutputTransformer` zastrešuje transformácie volaní vstupno-výstupných funkcií na ich ekvivalenty zo štandardnej knižnice jazyka C. Ide o podstatnú časť transformácie `gfortran-to-c`. Trieda je znázornená na obrázku 7.5.

Súčasťou triedy je definícia vymenovania (angl. enum) `StreamTypeEnum`, ktorý trieda používa na označovanie prúdu (angl. stream), s ktorým pracuje volanie, ktoré je transformované a zároveň mapuje číslo prúdu použité v štruktúre `struct___st_parameter_common` (popísané v podsekcii 6.3.8) na názov. Trieda obsahuje privátnu premennú `stream`, ktorá je tohto typu a slúži na uchovanie typu prúdu aktuálne spracovávaných volaní. Môže to byť jeden z nasledujúcich:

- STDERR – štandardný chybový výstup mapovaný na číslo 0,
- STDIN – štandardný vstup mapovaný na číslo 5,
- STDOUT – štandardný výstup mapovaný na číslo 6,
- OTHER – ostatné vstupy a výstupy (súbory) mapované na ostatné čísla.

V triede je taktiež definovaný privátny typ `changeFunc`, ktorý definuje rozhranie transformačných funkcií volaných nad jednotlivými inštrukciami. Pomocou tohto typu je definované asociatívne pole `toTransform`, ktoré má ako kľúč názov volania, ktoré bude transformované funkciou, ktorá je priradená ako hodnota kľúča v asociatívnom poli. Pole je inicializované v konštruktore triedy. Jednotlivé kľúče, a teda transformované volania sú:



Obr. 7.5: Trieda `InputOutputTransformer`

- `"_gfortran_transfer_integer_write"`, hodnotou je ukazovateľ na metódu `transformTransferIntegerWrite()`,
- `"_gfortran_transfer_real_write"`, hodnotou je ukazovateľ na metódu `transformTransferRealWrite()`,
- `"_gfortran_transfer_logical_write"`, hodnotou je ukazovateľ na metódu `transformTransferLogicalWrite()`,
- `"_gfortran_transfer_character_write"`, hodnotou je ukazovateľ na metódu `transformTransferCharacterWrite()`,
- `"_gfortran_transfer_integer"`, hodnotou v tomto prípade je ukazovateľ na metódu `transformTransferInteger()`,

- `"_gfortran_transfer_real"`, hodnotou je ukazovateľ na metódu `transformTransferReal()`,
- `"_gfortran_transfer_logical"`, hodnotou v tomto prípade je ukazovateľ na metódu `transformTransferLogical()`.

Uvedené metódy sú jednoduché, všetky volajú metódu `transformByStream()` a vracajú jej výsledok, ktorým je príznak zmeny. Metóde predávajú svoje vlastné parametre, ktorými sú ukazovateľ na inštrukciu `CallInst` a ukazovateľ na modul. Ďalej volanej metóde predávajú formátovací reťazec (podľa popisu v podsekcii 6.3.8), príznak, či je potrebné získať hodnotu tlačenej premennej dereferencovaním (`false` v prípade metódy `transformTransferCharacterWrite()` a u metód spracúvajúcich načítanie) a posledným predávaným parametrom je, či má tlačená hodnota dĺžku (`true` iba v prípade metódy `transformTransferCharacterWrite()`).

Metóda `transformByStream()` podľa hodnoty privátnej triednej premennej `stream` volá funkciu, ktorá sa stará o samotnú transformáciu. Môže to byť `transformToPrintf()`, `transformToFprintf()` alebo `transformToScanf()`. Metóda všetky prijaté parametre predáva ďalej vymenovaným metódam. Vracia ich návratový kód ako príznak zmeny. Ak privátna premenná `stream` nemá hodnotu `STDOUT`, `STDERR` alebo `STDIN`, nie je volaná žiadna ďalšia metóda a je vrátená hodnota `false`.

Metódy `transformToPrintf()` a `transformToScanf()` sú podobné metódy. Obe nahradzujú inštrukciu `CallInst`. Prvá volaním `printf()`, druhá volaním `scanf()`. Obe sa pokúsia z modulu získať danú funkciu. Ak neuspejú, vytvoria vektor typu argumentov danej funkcie (jeden argument, ktorý bude použitý ako ukazovateľ na formátovací reťazec) a vytvoria typ funkcie `FunctionType` s premenným počtom parametrov. Následne vytvoria danú funkciu. Ďalej je vytvorený typ `ArrayType` určený pre formátovací reťazec, globálna premenná `GlobalVariable` s týmto typom, konštanta `Constant` obsahujúca formátovací reťazec, ktorou je inicializovaná globálna premenná. Potom je vytvorený ukazovateľ na danú globálnu premennú, ktorý bude predaný novovytvorenej funkcii. Ďalej je potrebné získať ostatné hodnoty predané novovytvorenej funkcii. Je to operand pôvodného volania (premenná, ktorá bude vytlačená alebo je cieľom načítania) a prípadne dĺžka, ak ide o reťazec. Parameter `loadValueToPrint` určuje, či je potrebné získanie hodnoty (dereferencia) inštrukciou `LoadInst`. Ak áno, táto inštrukcia je vytvorená a pridaná pred volania novej funkcie. Či je potrebné novej funkcii predať aj dĺžku prípadného reťazca určuje parameter `hasLength`. Ak áno, predá sa novovytvorenej funkcii aj dĺžka, ktorá je ako ďalší parameter pôvodného volania. Keď sú do nového volania pridané argumenty, môže ním byť nahradené pôvodné volanie.

Metóda `transformToFprintf()` je pripravená a z časti implementovaná. Funguje tak isto ako predchádzajúce dve popísané metódy. Nie je však odladená kvôli problému so `stderr` popísanom v podsekcii 6.3.8. Keď pokročí vývoj zadnej časti dekompilátoru, je možné, že bude možné metódu odladiť a začať používať. V súčasnej podobe vracia vždy hodnotu `false`, pretože nevykonáva žiadnu zmenu.

Jedinou verejnou metódou triedy je `transform()`. Táto metóda prijíma ukazovateľ na modul a vracia príznak zmeny. Metóda iteruje cez funkcie, následne cez bloky a v nich cez inštrukcie. Vyhľadáva inštrukciu typu `AllocaInst`, ktorej alokovaný typ je `Type::StructTyID` a má názov `"struct.__st_parameter_dt"`. Nájde tak štruktúru `struct___st_parameter_dt`, ktorá sa objavuje vo výsledkoch migrácie. Následne metóda iteruje cez použitie tejto štruktúry a vyhľadáva inštrukcie prístupu k jej členom (typ `GetElementPtrInst`). Ak nájde inštrukciu prístupu k prvému členu (štruktúra `struct___st_parameter_common`), zavolá me-

tódu `detectStream()`, ktorá vráti typ prúdu, s ktorým budú následne pracovať transformované vstupno-výstupné funkcie. Ak nájde inštrukciu prístupu k šiestej položke, zavolá metódu `usesFormatString()`, ktorá zistí, či bol použitý formátovací reťazec. V prípade, že formátovací reťazec použitý nebol a prúd nie je `OTHER`, je volaná metóda `replaceCalls()` a následne metóda `removeInitsAndDones()`. Potom je vrátený logický súčet návratových hodnôt týchto dvoch metód. V prípade, že k žiadnej zmene nedôjde, metóda `transform()` vráti hodnotu `false`.

`detectStream()` je metóda, ktorá prijíma ako parameter ukazovateľ na inštrukciu `GetElementPtrInst`, čo je inštrukcia prístupu ku štruktúre `struct___st_parameter_common`. Metóda iteráciou cez jej použitia hľadá inštrukciu prístupu k jej druhej položke, v ktorej sa nachádza číselný identifikátor prúdu. Identifikátor je do položky uložený inštrukciou typu `StoreInst`. Prvý operand tejto inštrukcie je hľadaný identifikátor. Metóda podľa tohto identifikátora vyberie odpovedajúcu položku vymenovania `StreamTypeEnum`, ktorú nastaví do privátnej premennej `stream` a zároveň ju vráti.

Ako už bolo spomenuté, metóda `usesFormatString()` zisťuje, či je použitý nejaký formátovací reťazec. Iteruje cez použitia inštrukcie prístupu k položke. Ak narazí na inštrukciu typu `StoreInst`, znamená to, že bol použitý formátovací reťazec. Metóda potom vráti hodnotu `true`. V opačnom prípade vráti hodnotu `false`.

Metóda `replaceCalls()` prijíma ako parameter ukazovateľ na inštrukciu typu `AllocaInst` štruktúry `struct___st_parameter_dt`. Iteruje cez jej použitia a na tieto inštrukcie volá metódu `processCallInst()` implementovanú v rodičovskej triede `BaseTransformer`. Metóda `replaceCalls()` potom vráti logický súčet všetkých volaní metódy `processCallInst()` ako príznak zmeny.

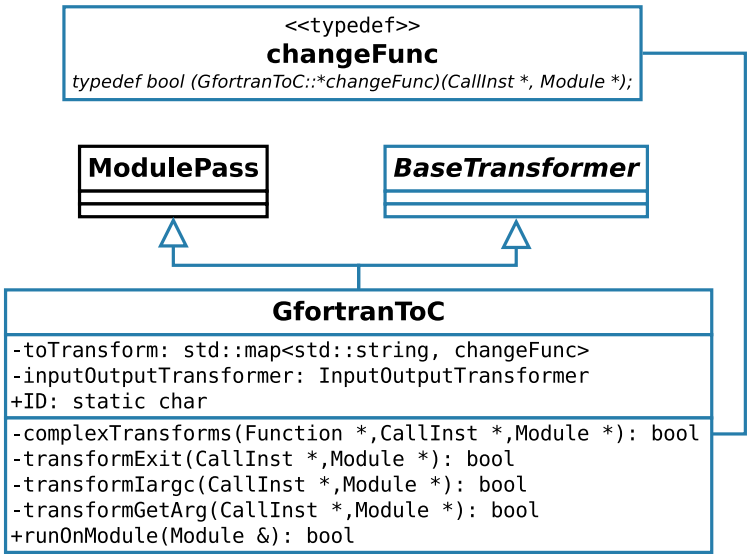
`removeInitsAndDones()` je jednoduchá metóda, ktorá volá dvakrát metódu `removeInitAndDone()` a vracia logický súčet týchto volaní ako príznak zmeny. Metóda prijíma ako parameter ukazovateľ na inštrukciu typu `AllocaInst` a tento aj posiela ďalej volaným metódam. Ďalej metódam predáva názov inicializačnej a ukončovacej funkcie. V prvom volaní ide o názvy funkcií spojené so zápisom `"_gfortran_st_write"` a `"_gfortran_st_write_done"`, v druhom volaní sú to názvy funkcií spojené s čítaním `"_gfortran_st_read"` a `"_gfortran_st_read_done"`.

Vyššie spomenutá metóda `removeInitAndDone()` iteruje cez použitia inštrukcie `AllocaInst` prijatej ako parameter a hľadá volania funkcií s názvom, ktoré dostala ako druhý a tretí parameter. Zároveň počíta počet použití. Ak nájde obe volania a počet použití je presne dva, znamená to, že všetky volania funkcií medzi inicializačným a ukončovacím volaním už boli transformované. Preto môžu byť obe odstránené. V takom prípade metóda vráti hodnotu `true`, v opačnom hodnotu `false`. Hodnota predstavuje príznak zmeny.

Trieda `InputOutputTransformer` je potomkom abstraktnej triedy `BaseTransformer`, a teda dedí aj jej funkcionalitu a implementuje čisto virtuálnu metódu `complexTransforms()`. Táto metóda prechádza pole `toTransform` a porovnáva kľúče s názvom volanej funkcie. Ak narazí na zhodu, zavolá metódu, na ktorú ukazuje ukazovateľ asociovaný ku kľúču v poli. Metóde predá ukazovateľ na inštrukciu `CallInst` a ukazovateľ na modul. Výsledok volania metódy je vrátený ako indikátor zmeny. Ak sa názov transformovanej funkcie nenachádza medzi kľúčmi v poli, metóda `complexTransforms()` vráti `false`, keďže žiadna zmena nebola vykonaná.

Trieda `GfortranToC`

Trieda `GfortranToC` je hlavnou triedou celej transformácie. Na obrázku 7.6 je vidieť, že je potomkom triedy `ModulePass`. Implementuje teda metódu `runOnModule()`. Tá volá metódu `transform()` triedy `InputOutputTransformer`, ktorú má ako svoju privátnu premennú. Potom iteruje cez funkcie, následne cez bloky a inštrukcie. Pre každú inštrukciu volá funkciu `processCallInst()`, ktorú dedí od abstraktnej triedy `BaseTransformer`. Metóda vracia logický súčet navratových hodnôt všetkých volaných metód ako príznak zmeny.



Obr. 7.6: Trieda `GfortranToC`

Trieda `GfortranToC`, podobne ako trieda `InputOutputTransformer`, definuje vlastný typ `changeFunc` a pomocou neho internú premennú `toTransform`. Taktiež úplne rovnako implementuje čisto virtuálnu metódu `complexTransforms()` rodiča `BaseTransformer`. Rozdiel je v type asociatívneho pola. Každá trieda má tento typ iný, aby mohla v poli uchovávať ukazovatele na svoje metódy.

V konštruktore triedy je inicializovaný vektor `toRemove` zdedený od triedy `BaseTransformer` hodnotami:

- `"_gfortran_set_options"`,
- `"_gfortran_set_args"`.

Asociatívne pole `toRename` nasledovne:

- `"_gfortran_pow_i4_i4"`, hodnotou je reťazec `"pow"`,
- `"llvm.powi.f32"`, hodnotou je reťazec `"pow"`,
- `"llvm.powi.f64"`, hodnotou je reťazec `"pow"`.

Asociatívne pole `toTransform` je inicializované v konštruktore triedy nasledovne:

- `"_gfortran_exit_i4"`, hodnotou je ukazovateľ na metódu `transformExit()`,

- "_gfortran_iargc", hodnotou je ukazovateľ na metódu `transformIargc()`,
- "_gfortran_getarg_i4", hodnotou je ukazovateľ na metódu `transformGetArg()`.

Metóda `transformExit()` transformuje volanie funkcie "_gfortran_exit_i4()" na volanie funkcie `exit()` zo štandardnej knižnice jazyka C. Keďže ide o nepriame volanie, metóda sa dostane, podobne ako metóda `processIndirectCall()` triedy `BaseTransformer`, k volanej funkcii. Z nej prevezme návratový typ, vytvorí nový typ parametra a celej funkcie. Následne vytvorí novú funkciu. Pred vložením jej volania namiesto pôvodného volania je ešte potrebné získať hodnotu (dereferencia) parametra, pretože pôvodné volanie pracuje s jeho adresou. Po nahradení volania metóda vráti hodnotu `true` ako príznak zmeny. Ak sa jej nepodarí dostať k volanej funkcii, vráti hodnotu `false`.

`transformIargc()` je metóda, ktorá sa stará o vytvorenie globálnej premennej na úschovu počtu parametrov programu a nahradenie volaní funkcie `_gfortran_iargc()` hodnotou tejto premennej. Keďže volaní funkcie `_gfortran_iargc()` môže byť v kóde viac, ale globálna premenná bude len jedna, je nutné si na ňu odložiť ukazovateľ. Na to poslúži asociatívne pole `storage` zdedené od triedy `BaseTransformer`. Predtým než metóda čokoľvek vykoná, obnoví z tohto pola ukazovateľ. Ak tam ešte nebol vložený, metóda nájde funkciu `main()`, vytvorí globálnu premennú a inicializuje ju prvým parametrom funkcie `main()` – počet parametrov programu. Následne uloží ukazovateľ na globálnu premennú do pola. Ak v poli už ukazovateľ bol, tento krok sa preskočí. Metóda nakoniec nahradí pôvodné volanie za globálnu premennú a vráti hodnotu `true` ako príznak zmeny.

Poslednou privátnou metódou je metóda `transformGetArg()`. Podobne ako predchádzajúca metóda vytvára globálnu premennú pre ukazovateľ na pole parametrov programu. Preto rovnako testuje existenciu globálne premennej v poli `storage` a vytvára ju, len keď ešte neexistuje, s tým rozdielom, že globálna premenná je po vytvorení inicializovaná druhým parametrom funkcie `main()`. Ak je metóda v stave, keď má ukazovateľ na globálnu premennú, prejde k vytváraniu volania funkcie `memcpy()`. Najskôr vyskúša od modulu získať funkciu s názvom `llvm.memcpy.p0i8.p0i8.i32` (tá je zadnou časťou dekompilátoru dekompilovaná na `memcpy()`), ak neuspeje, vytvorí návratový typ funkcie, a potom aj samotnú funkciu. Získa prvý operand pôvodnej funkcie, ktorým je index do pola parametrov, druhý operand, ktorým je cieľová premenná a tretí operand, ktorým je veľkosť (pôvodná funkcia prijíma ako parameter veľkosť cieľového pola). Potom vytvorí inštrukciu pre získanie hodnoty z pola na danom indexe, ktorá sa použije ako zdroj kopírovania. Následne vytvorí pole argumentov a samotné volanie `memcpy()`, ktorým nahradí pôvodné volanie `_gfortran_getarg_i4()`.

Trieda má tak isto ako trieda `LLVMCallsRemover` verejnú premennú ID, ktorej adresa slúži na identifikáciu transformácie.

7.4 Spracovávanie výnimiek pri migrácii jazyka D a C++

Pôvodný návrh bol ošetriť tento problém cez vlastnú transformáciu. Po preštudovaní dokumentácie optimalizačného frameworku LLVM a existujúcich metód sa javila byť vstavaná optimalizácia `lowerinvoke` vhodná presne pre tento účel (nahradzuje inštrukciu `invoke` inštrukciou `call`). To sa potvrdilo po otestovaní na problémových zdrojových kódoch. Volanie tejto optimalizácie teda bolo zavedené pridaním parametra pre prostrednú časť dekompilátoru do zoznamu parametrov určujúceho, ktoré optimalizácie sa pri migrácii použijú. Tým sa problém vyriešil.

7.5 Transformácia `InstCombine`

Časť kódu, ktorá v transformácií `InstCombine` spôsobovala nežiadúce zmeny popísané v sekcii 6.1.3 bola identifikovaná a deaktivovaná pomocou direktív preprocesora `#if 0` a `#endif`. Bolo tak dosiahnuté požadovaného cieľa, kde sú žiadúce zmeny zachované a nežiadúce potlačené.

7.6 Jazyk D

Niektoré problémy pri migrácii jazyka D boli analyzované a bolo navrhnuté ich riešenie v sekciách 6.1 a 6.4. Riešenia zo sekcie 6.4 však neboli implementované kvôli rozsahu analýzy a implementácie pri predchádzajúcich jazykoch. Táto časť implementácie je ponechaná na budúce pokračovanie práce.

Kapitola 8

Testovanie

Táto kapitola popisuje spôsob testovania a obsahuje ukážky migrovaných zdrojových kódov. Spôsob testovania je popísaný v nasledujúcej sekcii, ukážky sa nachádzajú v sekcii 8.2.

8.1 Spôsob testovania

Celé testovanie sa dá rozdeliť na dve časti. Prvá časť je zameraná na testovanie behu migrácie. Cieľom bolo získať čo najviac rôznych existujúcich testovacích súborov. Na tento účel dobre poslúžili testovacie sady prekladačov. Pri jazyku Fortran to bola testovacia sada prekladača *gfortran*, pri jazyku C++ testovacia sada prekladača *g++* a pri jazyku D to bola testovacia sada prekladača *ldc2*. Počet testovacích súborov pre jednotlivé jazyky je zobrazený v tabuľke 8.1.

Tabuľka 8.1: Počet testovacích súborov na testovanie behu migrácie

Jazyk	**Počet súborov**
Fortran	1 611
C++	4 581
D	405
Spolu	6 597

Pri jazyku D testy okamžite odhalili problém so spracovaním výnimiek popísaný v podsekcii 6.1.2. Keďže bola testovacia sada pomerne rozsiahla a dekompilátor musel dekompilovať výstupy z troch rôznych prekladačov, bolo odhalené množstvo problémov a chýb v jeho zadnej časti. Chyby síce neboli opravené v rámci diplomovej práce, avšak boli vďaka nej odhalené. Práca teda má aj vedľajší prínos na vývoj dekompilátoru.

Druhá časť testovania sa zameriavala na konkrétne konštrukcie jazykov. Testovacie zdrojové kódy boli vytvárané za pomoci návodov. Pre Fortran to bol [21], pre C++ [3] a pre jazyk D [23]. Takto bolo možné postupne otestovať konštrukcie daných jazykov od jednoduchších po pokročilejšie. Testy začali postupne odhaľovať problémy popísané v kapitole 6, čo bolo ich hlavnou úlohou.

Počet testovacích súborov pre jednotlivé jazyky je zobrazený v tabuľke 8.2. Pri jazyku C++ boli testované len jeho pokročilé konštrukcie, ktoré sa nenachádzajú v jazyku C. Tie, ktoré sa v ňom nachádzajú, sú vyvíjané a ladené v rámci samotného dekompilátoru.

Tabuľka 8.2: Počet testovacích súborov na testovanie konštrukcií jazyka

Jazyk	**Počet súborov**
Fortran	80
C++	29
D	116

Práca sa primárne zaoberá jazykom Fortran. Kvôli rozsahu práce spojenej s týmto jazykom a následne s jazykom C++ neboli výstupy testov jazyka D detailnejšie analyzované. Bolo popísaných len niekoľko problémov a riešenie k väčšine z nich nebolo implementovaná. Hlavný problém, ktorý spôsoboval pád dekompilátoru, (popísaný v podsekcii 6.1.2) bol však odstránený.

Aby bolo možné efektívne vyvíjať transformácie a posudzovať, či došlo k zlepšeniu alebo zhoršeniu, bolo nutné testy automatizovať, a taktiež vytvoriť sadu referenčných výstupov. Sada referenčných výstupov bola vytvorená, a každou zmenou výstupov k lepšiemu aktualizovaná. Bol taktiež vytvorený testovací skript `run-hll-migration-tests.sh` (podobne ako migračný skript so šablónou `run-hll-migration-tests.template`).

Testovací skript spúšťa oba druhy testov. Predvolený je druhý typ popísaných testov. Ktoré testy sa ale budú vykonávať je možné zmeniť parametrami (je napríklad možné spustiť oba typy). Skript taktiež dokáže vygenerovať nové referenčné výstupy. Pri testovaní skript postupne prejde jednotlivé jazyky, vykoná zvolené testy a prípadné chyby vypisuje. Podľa návratového kódu z migračného skriptu, (viz sekcia 7.1) skript tiež zistí, v ktorom kroku migrácie nastala chyba, a túto informáciu zdelí na výstup. Je tak jednoduchšie identifikovať zdroj problému. V prípade druhého typu testov skript navyše porovnáva výstup s referenčným a v prípade, že sa líšia, vypíše výstup z programu `diff`. Je tak možné ihneď vidieť, či implementácia priniesla zlepšenie alebo zhoršenie výstupu.

8.2 Migrované zdrojové kódy

Táto sekcia obsahuje ukážky migrovaných zdrojových kódov a ich zhodnotenie. Niektoré príklady pochádzajú zo spomenutých návodov [3], [21] a [23]. Časť kódov bola z dôvodu veľkosti presunutá do príloh práce. Ostatné testovacie kódy a k nim prislúchajúce výstupy možno nájsť na CD priloženom k diplomovej práci.

Z dôvodu rozsahu sú v texte prezentované len ukážky výstupu v jazyku C. Ukážka výstupu v jazyku Python' sa nachádza len u jedného príkladu migrácie zdrojového kódu pre ilustráciu, ako výstup v jazyku Python' vyzerá. Ide o druhý príklad z prílohy B. Výstupy všetkých testovacích súborov na testovanie konštrukcií jazyka, ktorých počty zachytáva tabuľka 8.2, sa však nachádzajú na CD v oboch výstupných jazykoch.

Jednotlivé podsekcie, ktoré tvoria túto sekciu, odpovedajú sekciám kapitoly 6. Ide teda o Obecné problémy, Jazyk C++, Jazyk Fortran a Jazyk D.

8.2.1 Obecné problémy

Táto podsekcia ukazuje, ako sa zmenili príklady uvedené v sekcii 6.1 po implementovaní zmien popísaných v danej sekcii. Obrázok 8.1 odpovedá obrázku 6.1 po aplikovanej transformácií na odstránenie volaní `llvm_lifetime_end()`. Obrázok 8.2 odpovedná obrázku 6.2 po úpravách v transformácii `InstCombine`.

```
void func(void) {
    float32_t v1 = 1.6;
}
```

Obr. 8.1: Ukážka odstráneného volania `llvm_lifetime_end()`

```
int8_t g1[9] = "gt       ";
int8_t g2[9] = "lt or eq";

int32_t fun_(int8_t (*a1)[8], int32_t *a2, int32_t *a3) {
    int8_t v1[8];
    if (*a3 > 10) {
        memcpy(v1, g1, 8);
    } else {
        memcpy(v1, g2, 8);
    }
    memmove(a1, v1, 8);
/* ------------------------------------------------ */
    return *a2 > 10;
}
```

Obr. 8.2: Ukážka upravenej transformácie `InstCombine`

8.2.2 Jazyk C++

Jazyk C++ je blízky jazyku C, a tak s migráciou základných konštrukcii, ktoré majú tieto jazyky spoločné, nebol žiadny problém. Testy boli preto zamerané na konštrukcie, ktoré jazyky spoločné nemajú (napríklad triedy, dedičnosť, virtuálne metódy, šablóny, atď.).

```
class Class {
public:
    int attr;
    Class(){};
    Class(int a){attr=a;};
    ~Class(){};
};

int main(void) {
    Class c(4);
    return c.attr;
}
```

Obr. 8.3: Vstupný zdrojový kód

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

/* ------------- Structures ------------- */

struct class_Class {
    int32_t e0;
};

/* -------- Function Prototypes --------- */

void _ZN5ClassC1Ei(struct class_Class *a1, int32_t a2);
void _ZN5ClassD1Ev(struct class_Class *a1);
void _ZN5ClassD2Ev(struct class_Class *a1);
void _ZN5ClassC2Ei(struct class_Class *a1, int32_t a2);

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

int main() {
    struct class_Class v1;
    v1 = (struct class_Class){.e0 = 0};
    _ZN5ClassC1Ei(&v1, 4);
    _ZN5ClassD1Ev(NULL);
    return v1.e0;
}

void _ZN5ClassC1Ei(struct class_Class *a1, int32_t a2) {
    _ZN5ClassC2Ei(a1, a2);
}

void _ZN5ClassD1Ev(struct class_Class *a1) {
    _ZN5ClassD2Ev(NULL);
}

void _ZN5ClassD2Ev(struct class_Class *a1) {
    return;
}

void _ZN5ClassC2Ei(struct class_Class *a1, int32_t a2) {
    a1->e0 = a2;
}
```

Obr. 8.4: Výstupný zdrojový kód

Na obrázku 8.3 je kód jednoduchej triedy. Výsledný dekompilovaný kód je na obrázku 8.4. Vo výstupnom kóde je badateľná pôvodná funkcionalita. Kód je mierne nepre-

hľadný kvôli tzv. dekorácii alebo manglingu názvov[1]. S využitím takzvaného demangleru[2] je možné z týchto dekorovaných názvov dostať ich pôvodný význam, napríklad:

- `_ZN5ClassC1Ei` značí `Class::Class(int)`,
- `_ZN5ClassD1Ev` značí `Class::~Class()`.

Ďalšie príklady sa nachádzajú v prílohe A. Jednoduchá trieda a jej odpovedajúci výstup sa nachádza v A.1. Príklad A.2 ukazuje migráciu šablóny triedy a jej použitia a príklad A.3 ukazuje migráciu tried s dedičnosťou.

Ako je vidieť, migrované zdrojové kódy s niektorými konštrukciami jazyka C++ nie sú až tak zle čitateľné a je možné identifikovať pôvodnú funkcionalitu. Niektoré ďalšie však problém robili. Ide najmä o zdrojové kódy používajúce napríklad `std::map`, `std::vector` a štandardné prúdy (napr. `std::cout`). Príklady takýchto zdrojových kódov nie sú kvôli rozsahu zahrnuté do práce a nachádzajú sa na priloženom CD. Z týchto príkladov je možné vyvodiť záver, že migrácia jazyka C++ je zatiaľ len experimentálna a vyžaduje ďalší vývoj.

Značné zlepšenie by mohla priniesť napríklad integrácia demangleru do zadnej časti dekompilátoru tak, že by výstup demangleru bol dekorovaný iným spôsobom a to nahradením nepovolených znakov za povolené. Zároveň by to mohol byť krok smerom k pridaniu jazyka C++ ako výstupného jazyka zadnej časti dekompilátoru. Pomocou demangleru by z názvu `_ZN5ClassC1Ei` mohol vzniknúť napríklad názov `Class_Class_int_`, ktorý je sám o sebe čitateľnejší. Nahradenie všetkých názvov by značne zlepšilo čitateľnosť celého kódu, pretože ak by bol použitý jednotný spôsob ich vytvárania, nové názvy by zároveň napovedali, čo daný kód robí. Ak by bol čitateľ kódu oboznámený so spôsobom tvorby nových názvov, vedel by hneď, že `Class_Class_int_` predstavuje z pôvodného zdrojového kódu konštruktor triedy s názvom `Class`, ktorý prijíma jeden parameter typu `int` (čo sedí s typom migrovanej funkcie).

8.2.3 Jazyk Fortran

Táto podsekcia ukazuje, ako vyzerajú problematické výstupy zo sekcie 6.3 po aplikácii implementovaných transformácií, uvádza ďalšie príklady migrácie a kde je to možné, porovnáva výstup s pomerne úspešným existujúcim migrovacím nástrojom `f2c`.

```
int main(int a1, char **a2) {
    return 0;
}
```

Obr. 8.5: Ukážka odstráneného volania `_gfortran_set_options()`

Obrázok 8.5 odpovedá obrázku 6.4 po aplikovaní transformácií a odstránení volania `_gfortran_set_options()`. Obrázok 8.6 odpovedá obrázku 6.5 po odstránení volania `_gfortran_iargc()`, obrázok 8.7 odpovedá obrázku 6.6 po transformácii volania `_gfortran_getarg_i4()`, obrázok 8.8 odpovedá obrázku 6.7 po odstránení volania `_gfortran_set_args()`, obrázok 8.9 odpovedá obrázku 6.8 po odstránení volania `_gfortran_exit_i4()`,

[1] https://en.wikipedia.org/wiki/Name_mangling
[2] Online demangler je dostupný na http://demangler.com/

obrázok 8.10 odpovedá obrázku 6.9 po odstránení volania `_gfortran_pow_i4_i4()` a obrázok 8.11 odpovedá obrázkom 6.11 a 6.12. Je vidieť, že v tomto prípade došlo k podstatnému skráteniu zdrojového kódu a výraznému zlepšeniu jeho čitateľnosti.

```
int32_t g1 = 0;

int32_t fun_(int32_t *a) {
    return *a + g1;
}

int main(int a1, char **a2) {
    g1 = a1;
    int32_t x = 1;
    int32_t y = fun_(&x);
    return y;
}
```

Obr. 8.6: Ukážka odstráneného volania `_gfortran_iargc()`

```
int8_t **g1 = NULL;

int main(int a1, char **a2) {
    g1 = a2;
    int8_t v1[32];
    int32_t v2 = 1;
    memcpy(v1, g1[(int64_t)v2], 32);
}
```

Obr. 8.7: Ukážka odstráneného volania `_gfortran_getarg_i4()`

```
int main(int a1, char **a2) {
    return 0;
}
```

Obr. 8.8: Ukážka odstráneného volania `_gfortran_set_args()`

```
int main(int a1, char **a2) {
    exit(0);
}
```

Obr. 8.9: Ukážka odstráneného volania `_gfortran_exit_i4()`

```
int main(int a1, char **a2) {
    int32_t v1 = 256;
    return pow(v1, v1 + 1);
}
```

Obr. 8.10: Ukážka odstráneného volania `_gfortran_pow_i4_i4()`

```
int8_t g1[5] = "Hello";
int8_t g2[5] = "World";

int main(int a1, char **a2) {
    printf("%.*s", 5, g1);
    printf("%d", 47);
    printf("%.*s", 5, g2);
    printf("%f", 3.1400001);
}
```

Obr. 8.11: Problém pri migrácii IO funkcií po aplikácii transformácií

Nasledujúce popísané príklady demonštrujú migráciu na reálnych zdrojových kódoch. Nachádzajú sa v prílohe B. Na obrázku B.1 je vstupný zdrojový kód, ktorý vypočíta faktoriál počtu parametrov programu. V hornej časti kódu sa nachádza rekurzívna funkcia na výpočet faktoriálu, v spodnej časti telo programu. Aby bolo vidieť prínos transformácií, na obrázkoch B.2, B.3 a B.4 je zobrazený výstup bez transformácií. Kód zaberá tri strany aj napriek tomu, že bol ručne skrátený naformátovaním definícií štruktúr do troch stĺpcov. Definícia by inak zabrala tri-krát toľko miesta, teda jeden a pol strany. Na obrázku B.5 je konečný výstupný kód po transformáciách. Pri tomto príklade sa prejavila nevýhoda migračného nástroja `f2c`. Keďže podporuje iba dialekt FORTRAN77 a vstupný zdrojový kód je v dialekte Fortran90, migrácia pomocou nástroja `f2c` končí s chybou.

Druhý príklad je prevzatý z [7]. Počíta najväčší spoločný deliteľ dvoch zadaných čísel pomocou Euklidovho algoritmu. Vstupný zdrojový kód je na obrázku B.6. Výstupný zdrojový kód bez transformácií sa tentokrát rozlieha cez sedem obrázkov B.7, B.8, B.9, B.10, B.11, B.12 a B.13. Pri tomto príklade je obzvlášť viditeľný prínos transformácií. Výstup s ich zapojením je na obrázkoch B.14 a B.15. Výstup sa podstatne skrátil a je čitateľnejší.

Keďže vstupný zdrojový kód je v dialekte FORTRAN77, mohol byť migrovaný taktiež pomocou nástroja `f2c`. Výstup z tohto nástroja je na obrázkoch B.16, B.17 a B.18. Pri porovnaní výstupu z implementovaného migračného nástroja s výstupom z nástroja `f2c` je na prvý pohľad vidieť, že výstup z nástroja `f2c` je dlhší. Ďalej je vidieť, že tento výstup

obsahuje volania interných funkcií, na ktoré boli funkcie jazyka Fortran prevedené. To robí výstup menej čitateľným a závislým na knižnici `libf2c`. Oproti tomu výstup implementovaného migračného nástroja je pomerne dobre čitateľný a obsahuje len volania funkcií zo štandardných knižníc.

Pri tomto príklade je pre ilustráciu výstupného jazyka Python' uvedený aj výstup v tomto jazyku. Nachádza sa na obrázkoch B.19 a B.20. Ako už bolo v práci spomenuté, jazyk Python' predstavuje jazyk Python obohatený o konštrukcie jazyka C. Vo výstupe sa nachádzajú volania knižníc jazyka C.

Z predchádzajúcich príkladov je vidieť, že implementovaný migračný nástroj pri jazyku Fortran produkuje pomerne kvalitné výsledky. V uvedenom prípade dokonca lepšie než existujúci migračný nástroj `f2c`. Ďalšou veľkou výhodou implementovaného migračného nástroja oproti nástroju `f2c` je, že nie je obmedzený na konkrétny dialekt jazyka Fortran. Podpora nasledujúcich nových dialektov sa odvíja od vývoja prekladača GCC a nebude si vyžadovať žiadne veľké úpravy, ako by tomu bolo v prípade zavedenia nového dialektu do nástroja `f2c`.

8.2.4 Jazyk D

Príloha C obsahuje príklad migrácie jednoduchého zdrojového kódu v jazyku D. Na obrázku C.1 je vstupný zdrojový kód, ktorý obsahuje funkciu s ternárnym operátorom. Odpovedajúci výstup je na obrázkoch C.2 a C.3. Na výstupnom kóde je vidieť problém popísaný v podsekcii 6.4.1. Ďalšie zdrojové kódy je možné nájsť na priloženom CD. Pri väčšine z nich je vidieť, že výstupný zdrojový kód je rozsiahly, často kvôli problému popísanom v podsekcii 6.4.2. Migrácia jazyka D zatiaľ nie je reálne použiteľná. Výstupy testov neboli podrobne analyzované tak, ako tomu bolo pri predchádzajúcich jazykoch. Tu je priestor pre pokračovanie práce a ďalší vývoj.

Kapitola 9

Záver

V práci bol rozobratý problém migrácie zdrojových kódov, motivácia a možnosti migrácie. Bolo spomenuté množstvo existujúcich migračných nástrojov, ktoré však majú určité obmedzenia. Hlavným obmedzením je ich podpora jedného vstupného jazyka a jedného výstupného jazyka. Po teoretickom úvode k projektu Lissom, LLVM a dekompilátoru projektu Lissom bol predstavený návrh riešenia postavený na tomto dekompilátore. Ďalej boli predstavené prekladače, ktoré dokážu generovať LLVM IR na výstupe, a jazyky, ktoré tieto prekladače prekladajú. Z uvedených prekladačov boli vybratí vhodní kandidáti na implementáciu. Následne boli popísané návrhy a implementácia migračného nástroja a integrácia prekladačov vstupných jazykov.

Výstupom práce je migračný nástroj postavený na prostrednej a zadnej časti dekompilátoru projektu Lissom a vybraných prekladačoch. Prekladače zo vstupného jazyka generujú kód v LLVM IR, ktorý je ďalej optimalizovaný a dekompilovaný prostrednou a zadnou časťou dekompilátoru. Z popísaných prekladačov boli vybrané *ldc2* (prekladač jazyka D), *Clang* (prekladač jazykov C, C++, Objective-C a Objective-C++) a GCC s pluginom *Dragonegg* (prekladač jazyka Fortran a jeho dialektov). Výstupom dekompilácie je zdrojový kód v jazyku C alebo v jazyku Python rozšírenom o niektoré konštrukcie jazyka C. Migračný nástroj teda umožňuje migráciu jazykov D, C, C++, Objective-C a Objective-C++, Fortran a jeho dialektov do jazykov C a rozšíreného jazyka Python. Migračný nástroj je jednoducho rozšíriteľný o ďalšie vstupné jazyky pridaním príslušných prekladačov generujúcich LLVM IR.

Výstupy migrácie po implementácii migračného nástroja neboli veľmi uspokojivé. Preto bolo nutné vytvoriť a analyzovať testovacie zdrojové kódy zamerané na rôzne konštrukcie migrovaného jazyka. Tie boli vytvorené pre jazyky C++ a Fortran a D, detailne analyzované však boli len pre jazyky C++ a Fortran. Na základe analýz bolo navrhnutých a implementovaných niekoľko metód. Väčšinou ide o transformácie vykonané prostrednou časťou dekompilátoru. Tieto metódy podstatne zlepšujú migrované výstupy a tvoria jadro celej práce.

V práci boli taktiež prezentované ukážky migrovaných zdrojových kódov a ich zhodnotenie. Pri jazyku C++ vyplynulo, že migrácia v súčasnom stave nie je prakticky využiteľná kvôli problémom pri migrácii zdrojových kódov, v ktorých sú použité triedy ako `std::map` alebo `std::cout`. Aby sa výstupy zlepšili, je potrebný ďalší vývoj. V práci bol prezentovaný návrh na integráciu demangleru do zadnej časti dekompilátoru.

Výstupy migrácie jazyka Fortran boli veľmi uspokojivé. V uvedenom príklade dokonca predčili výstup existujúceho migračného nástroja `f2c`. Implementovaný migračný nástroj má navyše oproti nástroju `f2c` výhodu podpory viacerých dialektov jazyka Fortran (podľa

podpory prekladača). Výslednej kvality výstupov migrácie bolo dosiahnuté pomocou implementovaných transformácií. Na príkladoch je vidieť úžitok, aký transformácie prinášajú. Ide predovšetkým o skrátenie zdrojových kódov (väčšinou niekoľko násobne) a podstatné zlepšenie ich čitateľnosti. Aj pri jazyku Fortran je však priestor pre ďalší vývoj. Ten sa môže zamerať napríklad na doteraz nepodporované formátovacie reťazce pri vstupno-výstupných príkazov, podporu práce so súbormi. Taktiež je možné, že testami neboli postihnuté úplne všetky konštrukcie a príkazy jazyka. Nové testy by mohli odhaliť priestor pre ďalší vývoj.

Pri jazyku D okrem riešení obecných problémov neboli implementované žiadne metódy vylepšujúce výstup. Migrácia zdrojových kódov z jazyka D v súčasnom stave nie je prakticky použiteľná a je potrebný ďalší vývoj. Niekoľko návrhov bolo v práci prezentovaných. Išlo o úpravu funkcie `main()` a práce s parametrami programu a o odstránenie prebytočného kódu funkcií z knižníc jazyka.

Výstupy migrácie jazyka C nebolo nutné nijak vylepšovať. Práca sa nezaoberala týmto jazykom, pretože sa ním zaoberá vývoj samotného dekompilátoru. Z dôvodu rozsahu práce bola taktiež vynechaná analýza a testovanie jazykov Objective-C a Objective-C++.

Práca mala taktiež vedľajší pozitívny prínos. Vďaka nej bolo odhalených množstvo chýb v zadnej časti dekompilátoru a taktiež poukázala na problematické konštrukcie, ktorých spôsob dekompilácie bol následne vylepšený (napríklad dekompilácia globálnych premenných obsahujúcich reťazce). K lepším výsledkom dekompilátoru taktiež prispieva úprava vstavanej transformácie `InstCombine`.

V práci bol teda implementovaný migračný nástroj podporujúci viacero vstupných a viacero výstupných jazykov, metódy vylepšujúce jeho výstup, bola zhodnotená ich kvalita a prezentované návrhy na možný budúci vývoj.

Literatúra

[1] *Ada (programming language)* [online]. last modified on 7 December 2013 at 05:37 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/Ada_%28programming_language%29>.

[2] *C++* [online]. last modified on 21 December 2013 at 20:49 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/?title=C%2B%2B>.

[3] *C++ Language* [online]. [cit. 9. júna 2014]. Dostupné na: <http://www.cplusplus.com/doc/tutorial/>.

[4] *C (programming language)* [online]. last modified on 28 December 2013 at 08:39 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/C_%28programming_language%29>.

[5] *D (programming language)* [online]. last modified on 22 December 2013 at 17:09 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/D_%28programming_language%29>.

[6] *Fortran* [online]. last modified on 21 December 2013 at 22:27 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/Fortran>.

[7] *Fortran 77 examples: Greatest common divisor* [online]. last modified on 26 November 2013, at 19:18 [cit. 13. júna 2014]. Dostupné na: <https://en.wikibooks.org/wiki/Fortran/Fortran_examples#Fortran_77_examples>.

[8] *The GNU Fortran Compiler* [online]. [cit. 11. mája 2014]. Dostupné na: <https://gcc.gnu.org/onlinedocs/gfortran/>.

[9] *LLVM Language Reference Manual* [online]. Last updated on 2014-01-01 [cit. 3. januára 2014]. Dostupné na: <http://llvm.org/docs/LangRef.html>.

[10] *[LLVMdev] LLVM IR is a compiler IR* [online]. Tue Oct 4 15:23:20 CDT 2011 [cit. 8. januára 2014]. Dostupné na: <http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-October/043720.html>.

[11] *Lua (programming language)* [online]. last modified on 13 December 2013 at 18:15 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/Lua_%28programming_language%29>.

[12] *Objective-C* [online]. last modified on 29 December 2013 at 11:38 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/Objective-C>.

[13] *Pascal (programming language)* [online]. last modified on 27 December 2013 at 02:29 [cit. 29. decembra 2013]. Dostupné na: <https://en.wikipedia.org/wiki/Pascal_%28programming_language%29>.

[14] Brown, A. a Wilson, G. *The Architecture Of Open Source Applications* [Paperback]. [b.m.]: lulu, jun 2011. Dostupné na: <http://www.aosabook.org/en/>. ISBN 978-1257638017.

[15] Dickens, T. Migrating Legacy Engineering Applications to Java. In *OOPSLA 2002 Practitioners Reports*. New York, NY, USA: ACM, 2002. Dostupné na: <http://doi.acm.org/10.1145/604251.604260>. ISBN 1-58113-471-1.

[16] Eilam, E. *Reversing: Secrets of Reverse Engineering*. Canada: Wiley Publishing, Inc., 2005. 589 s. ISBN 0-7645-7481-7.

[17] Feldman, S. I. A Fortran to C Converter. *SIGPLAN Fortran Forum*. Oct 1990, roč. 9, č. 2. Dostupné na: <http://doi.acm.org/10.1145/101363.101366>. ISSN 1061-7264.

[18] Kontogiannis, K., Martin, J., Wong, K. et al. Code Migration Through Transformations: An Experience Report. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*. [b.m.]: IBM Press, 1998. CASCON'98. Dostupné na: <http://dl.acm.org.ezproxy.lib.vutbr.cz/citation.cfm?id=783160.783173>.

[19] Křoustek, J. *Analýza a převod kódů do vyššího programovacího jazyka*. Brno: Fakulta Informačních Technologií, Vysoké Učení Technické v Brne, 2009. 83 s. Diplomová práca.

[20] Lerew, E. Migration of legacy test programs to a modern computer platform [for avionics testing]. In *AUTOTESTCON '99. IEEE Systems Readiness Technology Conference, 1999. IEEE*. 1999. S. 293–298. ISSN 1080-7725.

[21] Shene, C.-K. *Fortran 90 Tutorial* [online]. Last update: April 15, 2009 [cit. 12. mája 2014]. Dostupné na: <http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html>.

[22] Van Emmerik, M. J. *Static Single Assignment for Decompilation*. The University of Queensland, 2007. PhD Thesis. Dostupné na: <http://espace.library.uq.edu.au/view/UQ:158682>.

[23] Çehreli, A. *D Programming Language Tutorial: Learn to program in the D programming language* [online]. [cit. 9. júna 2014]. Dostupné na: <http://ddili.org/ders/d.en/>.

[24] Ďurfina, L., Křoustek, J. a Zemek, P. Psyb0t Malware: A Step-by-Step Decompilation Case Study. In *20th Working Conference on Reverse Engineering (WCRE'13)*. Koblenz, DE: IEEE, 2013. S. 449–456.

[25] Ďurfina, L., Křoustek, J. a Zemek, P. Generic Source Code Migration Using Decompilation. In *10th Annual Industrial Simulation Conference (ISC'2012)*. [b.m.]: EUROSIS, 2012. S. 38–42. ISBN 978-90-77381-71-7.

Príloha A

Príklady migrácie jazyka C++

A.1 Príklad 1

```
class Class {
private:
    int attr;
    int addOne(int par) {
        return par + 1;
    };
public:
    Class(){};
    ~Class(){};
    void setAttr(int attr) {
        this->attr = this->addOne(attr);
    };
    int getAttr() {
        return this->attr;
    };
};


int main(void) {
    Class c;
    c.setAttr(5);
    return c.getAttr();
}
```

Obr. A.1: Príklad 1, vstupný zdrojový kód

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

/* ------------- Structures ------------- */

struct class_Class {
    int32_t e0;
};

/* -------- Function Prototypes --------- */

void _ZN5ClassC1Ev(struct class_Class *a1);
void _ZN5Class7setAttrEi(struct class_Class *a1, int32_t
    ↪ a2);
int32_t _ZN5Class7getAttrEv(struct class_Class *a1);
void _ZN5ClassD1Ev(struct class_Class *a1);
void _ZN5ClassD2Ev(struct class_Class *a1);
int32_t _ZN5Class6addOneEi(struct class_Class *a1,
    ↪ int32_t a2);
void _ZN5ClassC2Ev(struct class_Class *a1);

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

int main() {
    struct class_Class v1;
    _ZN5ClassC1Ev(NULL);
    v1 = (struct class_Class){.e0 = 0};
    _ZN5Class7setAttrEi(&v1, 5);
    int32_t v2 = _ZN5Class7getAttrEv(&v1);
    _ZN5ClassD1Ev(NULL);
    return v2;
}

void _ZN5ClassC1Ev(struct class_Class *a1) {
    _ZN5ClassC2Ev(NULL);
}

void _ZN5Class7setAttrEi(struct class_Class *a1, int32_t
    ↪ a2) {
    a1->e0 = _ZN5Class6addOneEi(NULL, a2);
}

int32_t _ZN5Class7getAttrEv(struct class_Class *a1) {
    return a1->e0;
}
```

Obr. A.2: Príklad 1, výstupný zdrojový kód, prvá časť

```
void _ZN5ClassD1Ev(struct class_Class *a1) {
    _ZN5ClassD2Ev(NULL);
}

void _ZN5ClassD2Ev(struct class_Class *a1) {
    return;
}

int32_t _ZN5Class6addOneEi(struct class_Class *a1,
    ↪ int32_t a2) {
    return a2 + 1;
}

void _ZN5ClassC2Ev(struct class_Class *a1) {
    return;
}
```

Obr. A.3: Príklad 1, výstupný zdrojový kód, druhá časť

A.2 Príklad 2

```
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  return myint.increase() + mychar.uppercase();
}
```

Obr. A.4: Príklad 2, vstupný zdrojový kód

```c
#include <stdint.h>

/* ------------- Structures ------------- */

struct class_mycontainer {
    int32_t e0;
};

struct class_mycontainer_0 {
    int8_t e0;
};

/* -------- Function Prototypes --------- */

void _ZN11mycontainerIiEC1Ei(struct class_mycontainer *a1
   ↪ , int32_t a2);
void _ZN11mycontainerIcEC1Ec(struct class_mycontainer_0 *
   ↪ a1, int8_t a2);
int32_t _ZN11mycontainerIiE8increaseEv(struct
   ↪ class_mycontainer *a1);
int8_t _ZN11mycontainerIcE9uppercaseEv(struct
   ↪ class_mycontainer_0 *a1);
void _ZN11mycontainerIiEC2Ei(struct class_mycontainer *a1
   ↪ , int32_t a2);
void _ZN11mycontainerIcEC2Ec(struct class_mycontainer_0 *
   ↪ a1, int8_t a2);

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

int main() {
    struct class_mycontainer_0 v1;
    struct class_mycontainer v2;
    v2 = (struct class_mycontainer){.e0 = 0};
    _ZN11mycontainerIiEC1Ei(&v2, 7);
    v1 = (struct class_mycontainer_0){.e0 = 0};
    _ZN11mycontainerIcEC1Ec(&v1, 106);
    int32_t v3 = _ZN11mycontainerIiE8increaseEv(&v2);
    return (int32_t)_ZN11mycontainerIcE9uppercaseEv(&v1)
       ↪ + v3;
}

void _ZN11mycontainerIiEC1Ei(struct class_mycontainer *a1
   ↪ , int32_t a2) {
    _ZN11mycontainerIiEC2Ei(a1, a2);
}
```

Obr. A.5: Príklad 2, výstupný zdrojový kód, prvá časť

```
void _ZN11mycontainerIcEC1Ec(struct class_mycontainer_0 *
    ↪ a1, int8_t a2) {
    _ZN11mycontainerIcEC2Ec(a1, a2);
}

int32_t _ZN11mycontainerIiE8increaseEv(struct
    ↪ class_mycontainer *a1) {
    int32_t v1 = a1->e0 + 1;
    a1->e0 = v1;
    return v1;
}

int8_t _ZN11mycontainerIcE9uppercaseEv(struct
    ↪ class_mycontainer_0 *a1) {
    int8_t v1 = a1->e0;
    int8_t v2 = v1;
    if (v1 < 123) {
        int8_t v3 = v1 - 32;
        a1->e0 = v3;
        v2 = v3;
    }
    return v2;
}

void _ZN11mycontainerIiEC2Ei(struct class_mycontainer *a1
    ↪ , int32_t a2) {
    a1->e0 = a2;
}

void _ZN11mycontainerIcEC2Ec(struct class_mycontainer_0 *
    ↪ a1, int8_t a2) {
    a1->e0 = a2;
}
```

Obr. A.6: Príklad 2, výstupný zdrojový kód, druhá časť

A.3 Príklad 3

```
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  return rect.area() + trgl.area();
}
```

Obr. A.7: Príklad 3, vstupný zdrojový kód

```c
#include <stdint.h>

/* ------------- Structures ------------- */

struct class_Polygon {
    int32_t e0;
    int32_t e1;
};

struct class_Rectangle {
    struct class_Polygon e0;
};

struct class_Triangle {
    struct class_Polygon e0;
};

/* -------- Function Prototypes --------- */

void _ZN7Polygon10set_valuesEii(struct class_Polygon *a1,
   ↪  int32_t a2, int32_t a3);
int32_t _ZN9Rectangle4areaEv(struct class_Rectangle *a1);
int32_t _ZN8Triangle4areaEv(struct class_Triangle *a1);

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

int main() {
    struct class_Rectangle v1;
    struct class_Triangle v2;
    v1 = (struct class_Rectangle){.e0 = (struct
       ↪ class_Polygon){.e0 = 0, .e1 = 0}};
    _ZN7Polygon10set_valuesEii(&v1.e0, 4, 5);
    v2 = (struct class_Triangle){.e0 = (struct
       ↪ class_Polygon){.e0 = 0, .e1 = 0}};
    _ZN7Polygon10set_valuesEii(&v2.e0, 4, 5);
    int32_t v3 = _ZN9Rectangle4areaEv(&v1);
    return _ZN8Triangle4areaEv(&v2) + v3;
}
```

Obr. A.8: Príklad 3, výstupný zdrojový kód, prvá časť

```
void _ZN7Polygon10set_valuesEii(struct class_Polygon *a1,
    ↪  int32_t a2, int32_t a3) {
    a1->e0 = a2;
    a1->e1 = a3;
}

int32_t _ZN9Rectangle4areaEv(struct class_Rectangle *a1)
    ↪ {
    return a1->e0.e1 * a1->e0.e0;
}

int32_t _ZN8Triangle4areaEv(struct class_Triangle *a1) {
    return a1->e0.e1 * a1->e0.e0 / 2;
}
```

Obr. A.9: Príklad 3, výstupný zdrojový kód, druhá časť

Príloha B

Príklady migrácie jazyka Fortran

B.1 Príklad 1

```
        recursive integer function factorial(a) &
         result(res)
                implicit none
                integer, intent(in) :: a
                if (a <= 0) then
                        res = 1
                else
                        res = a * factorial(a - 1)
                end if
        end function factorial

        program test
                implicit none
                integer :: a, b
                integer :: factorial
                a = iargc()
                b = factorial(a)
                print *, " | Result: ", b
                call EXIT(b)
        end program test
```

Obr. B.1: Príklad 1, vstupný zdrojový kód

```
struct struct___st_parameter_common {
    int32_t e0;   int32_t e1;   int8_t *e2;
    int32_t e3;   int32_t e4;   int8_t *e5;
    int32_t *e6;
};

struct struct___st_parameter_dt {
    struct struct___st_parameter_common e0;
    int64_t e1;       int64_t *e2;   int64_t *e3;
    int8_t *e4;       int8_t *e5;    int32_t e6;
    int32_t e7;       int8_t *e8;    int8_t *e9;
    int32_t e10;      int32_t e11;   int8_t *e12;
    int8_t e13[256]; int32_t *e14;  int64_t e15;
    int8_t *e16;      int32_t e17;   int32_t e18;
    int8_t *e19;      int8_t *e20;   int32_t e21;
    int32_t e22;      int8_t *e23;   int8_t *e24;
    int32_t e25;      int32_t e26;   int8_t *e27;
    int8_t *e28;      int32_t e29;   int8_t e30[4];
};

/* -------- Function Prototypes --------- */

int32_t factorial_(int32_t *a1);
void MAIN__(void);

// The following external functions do not have any
   ↪ associated header file:
void llvm_lifetime_end(int64_t var1, int8_t *var2);
int32_t _gfortran_iargc(void);
void _gfortran_st_write(struct struct___st_parameter_dt *
   ↪ var3);
void _gfortran_transfer_character_write(struct
   ↪ struct___st_parameter_dt *var4, int8_t *var5,
   ↪ int32_t var6);
void _gfortran_transfer_integer_write(struct
   ↪ struct___st_parameter_dt *var7, int8_t *var8,
   ↪ int32_t var9);
void _gfortran_st_write_done(struct
   ↪ struct___st_parameter_dt *var10);
void _gfortran_exit_i4(...void);
void _gfortran_set_args(int32_t var11, int8_t **var12);
void _gfortran_set_options(int32_t var13, int32_t *var14)
   ↪ ;
```

Obr. B.2: Príklad 1, výstupný zdrojový kód bez transformácií, prvá časť

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

int8_t g1[108] = "tests/factorial_with_print.f90\x00";
int8_t g2[11] = " | Result: ";
int32_t g3[8] = {68, 1023, 0, 0, 1, 1, 0, 1};

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

int32_t factorial_(int32_t *a1) {
    uint32_t v1 = *a1;
    int32_t v2;
    if (v1 >= 1) {
        int32_t v3 = v1 - 1;
        v2 = factorial_(&v3) * v1;
        llvm_lifetime_end(4, (int8_t *)&v3);
    } else {
        v2 = 1;
    }
    return v2;
}

void MAIN__(void) {
    struct struct___st_parameter_dt v1;
    int32_t v2 = _gfortran_iargc();
    int32_t v3 = factorial_(&v2);
    v1 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v1.e0.e2 = &g1[0];
    v1.e0.e3 = 18;
    v1.e0.e0 = 128;
    v1.e0.e1 = 6;
    _gfortran_st_write(&v1);
    _gfortran_transfer_character_write(&v1, g2, 11);
    _gfortran_transfer_integer_write(&v1, (int8_t *)&v3,
       ↪ 4);
```

Obr. B.3: Príklad 1, výstupný zdrojový kód bez transformácií, druhá časť

```
    _gfortran_st_write_done(&v1);
    llvm_lifetime_end(480, (int8_t *)&v1);
    ((void (*)(int32_t *))_gfortran_exit_i4)(&v3);
}

int main(int a1, char **a2) {
    _gfortran_set_args(a1, a2);
    _gfortran_set_options(8, g3);
    MAIN__();
}

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

// void _gfortran_exit_i4(...void);
// int32_t _gfortran_iargc(void);
// void _gfortran_set_args(int32_t var11, int8_t **var12)
   ↪ ;
// void _gfortran_set_options(int32_t var13, int32_t *
   ↪ var14);
// void _gfortran_st_write(struct
   ↪ struct___st_parameter_dt *var3);
// void _gfortran_st_write_done(struct
   ↪ struct___st_parameter_dt *var10);
// void _gfortran_transfer_character_write(struct
   ↪ struct___st_parameter_dt *var4, int8_t *var5,
   ↪ int32_t var6);
// void _gfortran_transfer_integer_write(struct
   ↪ struct___st_parameter_dt *var7, int8_t *var8,
   ↪ int32_t var9);
// void llvm_lifetime_end(int64_t var1, int8_t *var2);
```

Obr. B.4: Príklad 1, výstupný zdrojový kód bez transformácií, tretia časť

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

/* -------- Function Prototypes --------- */

int32_t factorial_(int32_t *a1);
void MAIN__(void);

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

int8_t g1[11] = " | Result: ";
int32_t g2 = 0;

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

int32_t factorial_(int32_t *a1) {
    uint32_t v1 = *a1;
    int32_t v2;
    if (v1 >= 1) {
        int32_t v3 = v1 - 1;
        v2 = factorial_(&v3) * v1;
    } else {
        v2 = 1;
    }
    return v2;
}

void MAIN__(void) {
    int32_t v1 = g2;
    int32_t v2 = factorial_(&v1);
    printf("%.*s", 11, g1);
    printf("%d", v2);
    exit(v2);
}

int main(int a1, char **a2) {
    g2 = a1;
    MAIN__();
}

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

// void exit(int32_t var2);
// int32_t printf(int8_t *var1, ...);
```

Obr. B.5: Príklad 1, výstupný zdrojový kód po transformáciách

B.2 Príklad 2

```
      PROGRAM EUCLID
      PRINT *, "A?"
      READ *, NA
      IF (NA .LE. 0) THEN
        PRINT *, "A must be a positive integer."
        CALL EXIT(1)
      END IF
      PRINT *, "B?"
      READ *, NB
      IF (NB .LE. 0) THEN
        PRINT *, "B must be a positive integer."
        CALL EXIT(1)
      END IF
      PRINT *, "The GCD of", NA, " and", NB, " is",
     ↪ NGCD(NA, NB), "."
      CALL EXIT(0)
    END

    FUNCTION NGCD(NA, NB)
      IA = NA
      IB = NB
1     IF (IB .NE. 0) THEN
        ITEMP = IA
        IA = IB
        IB = MOD(ITEMP, IB)
        GOTO 1
      END IF
      NGCD = IA
      RETURN
    END
```

Obr. B.6: Príklad 2, vstupný zdrojový kód

```
struct struct___st_parameter_common {
    int32_t e0;   int32_t e1;   int8_t *e2;
    int32_t e3;   int32_t e4;   int8_t *e5;
    int32_t *e6;
};

struct struct___st_parameter_dt {
    struct struct___st_parameter_common e0;
    int64_t e1;       int64_t *e2;   int64_t *e3;
    int8_t *e4;       int8_t *e5;    int32_t e6;
    int32_t e7;       int8_t *e8;    int8_t *e9;
    int32_t e10;      int32_t e11;   int8_t *e12;
    int8_t e13[256]; int32_t *e14;  int64_t e15;
    int8_t *e16;      int32_t e17;   int32_t e18;
    int8_t *e19;      int8_t *e20;   int32_t e21;
    int32_t e22;      int8_t *e23;   int8_t *e24;
    int32_t e25;      int32_t e26;   int8_t *e27;
    int8_t *e28;      int32_t e29;   int8_t e30[4];
};

/* -------- Function Prototypes --------- */

int32_t ngcd_(int32_t *a1, int32_t *a2);
void MAIN__(void);
// The following external functions do not have any
   ↪ associated header file:
void llvm_lifetime_end(int64_t var1, int8_t *var2);
void _gfortran_st_write(struct struct___st_parameter_dt *
   ↪ var3);
void _gfortran_transfer_character_write(struct
   ↪ struct___st_parameter_dt *var4, int8_t *var5,
   ↪ int32_t var6);
void _gfortran_st_write_done(struct
   ↪ struct___st_parameter_dt *var7);
void _gfortran_st_read(struct struct___st_parameter_dt *
   ↪ var8);
void _gfortran_transfer_integer(struct
   ↪ struct___st_parameter_dt *var9, int8_t *var10,
   ↪ int32_t var11);
void _gfortran_st_read_done(struct
   ↪ struct___st_parameter_dt *var12);
void _gfortran_stop_string(int8_t *var13, int32_t var14);
void _gfortran_transfer_integer_write(struct
   ↪ struct___st_parameter_dt *var15, int8_t *var16,
   ↪ int32_t var17)
```

Obr. B.7: Príklad 2, výstupný zdrojový kód bez transformácií, prvá časť

```
void _gfortran_set_args(int32_t var18, int8_t **var19);
void _gfortran_set_options(int32_t var20, int32_t *var21)
    ↪ ;

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

int8_t g1[65] = "../../testsuite/hll_migration/fortran/
    ↪ output_quality_tests/gcd.f\x00";
int32_t g10[8] = {68, 1023, 0, 0, 1, 1, 0, 1};
int8_t g2[2] = "A?";
int8_t g3[29] = "A must be a positive integer.";
int8_t g4[2] = "B?";
int8_t g5[29] = "B must be a positive integer.";
int8_t g6[10] = "The GCD of";
int8_t g7[4] = " and";
int8_t g8[3] = " is";
int8_t g9[1] = ".";

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

int32_t ngcd_(int32_t *a1, int32_t *a2) {
    int32_t v1 = *a1;
    int32_t v2 = *a2;
    if (v2 == 0) {
        return v1;
    }
    int32_t v3 = v1 % v2;
    while (v3 != 0) {
        v1 = v2;
        v2 = v3;
        v3 = v1 % v2;
    }

    return v2;
}

void MAIN__(void) {
    struct struct___st_parameter_dt v1;
    struct struct___st_parameter_dt v2;
    struct struct___st_parameter_dt v3;
    struct struct___st_parameter_dt v4;
    struct struct___st_parameter_dt v5;
    struct struct___st_parameter_dt v6;
    struct struct___st_parameter_dt v7;
```

Obr. B.8: Príklad 2, výstupný zdrojový kód bez transformácií, druhá časť

```
    v1 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v1.e0.e2 = &g1[0];
    v1.e0.e3 = 6;
    v1.e0.e0 = 128;
    v1.e0.e1 = 6;
    _gfortran_st_write(&v1);
    _gfortran_transfer_character_write(&v1, g2, 2);
    _gfortran_st_write_done(&v1);
    llvm_lifetime_end(480, (int8_t *)&v1);
    v2 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v2.e0.e2 = &g1[0];
    v2.e0.e3 = 7;
    v2.e0.e0 = 128;
    v2.e0.e1 = 5;
    _gfortran_st_read(&v2);
    int32_t v8 = 0;
    _gfortran_transfer_integer(&v2, (int8_t *)&v8, 4);
    _gfortran_st_read_done(&v2);
    llvm_lifetime_end(480, (int8_t *)&v2);
```

Obr. B.9: Príklad 2, výstupný zdrojový kód bez transformácií, tretia časť

```
    if (v8 < 1) {
        v3 = (struct struct___st_parameter_dt){.e0 = (
           ↪ struct struct___st_parameter_common){.e0 =
           ↪ 0, .e1 = 0, .e2 = NULL, .e3 = 0, .e4 = 0, .
           ↪ e5 = NULL, .e6 = NULL}, .e1 = 0, .e2 = NULL,
           ↪  .e3 = NULL, .e4 = NULL, .e5 = NULL, .e6 =
           ↪ 0, .e7 = 0, .e8 = NULL, .e9 = NULL, .e10 =
           ↪ 0, .e11 = 0, .e12 = NULL, .e14 = NULL, .e15
           ↪ = 0, .e16 = NULL, .e17 = 0, .e18 = 0, .e19 =
           ↪  NULL, .e20 = NULL, .e21 = 0, .e22 = 0, .e23
           ↪  = NULL, .e24 = NULL, .e25 = 0, .e26 = 0, .
           ↪ e27 = NULL, .e28 = NULL, .e29 = 0};
        v3.e0.e2 = &g1[0];
        v3.e0.e3 = 9;
        v3.e0.e0 = 128;
        v3.e0.e1 = 6;
        _gfortran_st_write(&v3);
        _gfortran_transfer_character_write(&v3, g3, 29);
        _gfortran_st_write_done(&v3);
        llvm_lifetime_end(480, (int8_t *)&v3);
        _gfortran_stop_string(NULL, 0);
    }
    v4 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v4.e0.e2 = &g1[0];
    v4.e0.e3 = 12;
    v4.e0.e0 = 128;
    v4.e0.e1 = 6;
    _gfortran_st_write(&v4);
    _gfortran_transfer_character_write(&v4, g4, 2);
    _gfortran_st_write_done(&v4);
    llvm_lifetime_end(480, (int8_t *)&v4);
```

Obr. B.10: Príklad 2, výstupný zdrojový kód bez transformácií, štvrtá časť

```
    v5 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v5.e0.e2 = &g1[0];
    v5.e0.e3 = 13;
    v5.e0.e0 = 128;
    v5.e0.e1 = 5;
    _gfortran_st_read(&v5);
    int32_t v9 = 0;
    _gfortran_transfer_integer(&v5, (int8_t *)&v9, 4);
    _gfortran_st_read_done(&v5);
    llvm_lifetime_end(480, (int8_t *)&v5);
    if (v9 < 1) {
        v6 = (struct struct___st_parameter_dt){.e0 = (
           ↪ struct struct___st_parameter_common){.e0 =
           ↪ 0, .e1 = 0, .e2 = NULL, .e3 = 0, .e4 = 0, .
           ↪ e5 = NULL, .e6 = NULL}, .e1 = 0, .e2 = NULL,
           ↪  .e3 = NULL, .e4 = NULL, .e5 = NULL, .e6 =
           ↪ 0, .e7 = 0, .e8 = NULL, .e9 = NULL, .e10 =
           ↪ 0, .e11 = 0, .e12 = NULL, .e14 = NULL, .e15
           ↪ = 0, .e16 = NULL, .e17 = 0, .e18 = 0, .e19 =
           ↪  NULL, .e20 = NULL, .e21 = 0, .e22 = 0, .e23
           ↪  = NULL, .e24 = NULL, .e25 = 0, .e26 = 0, .
           ↪ e27 = NULL, .e28 = NULL, .e29 = 0};
        v6.e0.e2 = &g1[0];
        v6.e0.e3 = 15;
        v6.e0.e0 = 128;
        v6.e0.e1 = 6;
        _gfortran_st_write(&v6);
        _gfortran_transfer_character_write(&v6, g5, 29);
        _gfortran_st_write_done(&v6);
        llvm_lifetime_end(480, (int8_t *)&v6);
        _gfortran_stop_string(NULL, 0);
    }
```

Obr. B.11: Príklad 2, výstupný zdrojový kód bez transformácií, piata časť

```
    v7 = (struct struct___st_parameter_dt){.e0 = (struct
       ↪ struct___st_parameter_common){.e0 = 0, .e1 = 0,
       ↪ .e2 = NULL, .e3 = 0, .e4 = 0, .e5 = NULL, .e6 =
       ↪ NULL}, .e1 = 0, .e2 = NULL, .e3 = NULL, .e4 =
       ↪ NULL, .e5 = NULL, .e6 = 0, .e7 = 0, .e8 = NULL,
       ↪ .e9 = NULL, .e10 = 0, .e11 = 0, .e12 = NULL, .
       ↪ e14 = NULL, .e15 = 0, .e16 = NULL, .e17 = 0, .
       ↪ e18 = 0, .e19 = NULL, .e20 = NULL, .e21 = 0, .
       ↪ e22 = 0, .e23 = NULL, .e24 = NULL, .e25 = 0, .
       ↪ e26 = 0, .e27 = NULL, .e28 = NULL, .e29 = 0};
    v7.e0.e2 = &g1[0];
    v7.e0.e3 = 18;
    v7.e0.e0 = 128;
    v7.e0.e1 = 6;
    _gfortran_st_write(&v7);
    _gfortran_transfer_character_write(&v7, g6, 10);
    _gfortran_transfer_integer_write(&v7, (int8_t *)&v8,
       ↪ 4);
    _gfortran_transfer_character_write(&v7, g7, 4);
    _gfortran_transfer_integer_write(&v7, (int8_t *)&v9,
       ↪ 4);
    _gfortran_transfer_character_write(&v7, g8, 3);
    int32_t v10 = ngcd_(&v8, &v9);
    _gfortran_transfer_integer_write(&v7, (int8_t *)&v10,
       ↪  4);
    llvm_lifetime_end(4, (int8_t *)&v10);
    _gfortran_transfer_character_write(&v7, g9, 1);
    _gfortran_st_write_done(&v7);
    llvm_lifetime_end(480, (int8_t *)&v7);
    _gfortran_stop_string(NULL, 0);
}

int main(int a1, char **a2) {
    _gfortran_set_args(a1, a2);
    _gfortran_set_options(8, g10);
    MAIN__();
}

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

// void _gfortran_set_args(int32_t var18, int8_t **var19)
   ↪ ;
// void _gfortran_set_options(int32_t var20, int32_t *
   ↪ var21);
```

Obr. B.12: Príklad 2, výstupný zdrojový kód bez transformácií, šiesta časť

```
// void _gfortran_st_read(struct struct___st_parameter_dt
   ↪  *var8);
// void _gfortran_st_read_done(struct
   ↪ struct___st_parameter_dt *var12);
// void _gfortran_st_write(struct
   ↪ struct___st_parameter_dt *var3);
// void _gfortran_st_write_done(struct
   ↪ struct___st_parameter_dt *var7);
// void _gfortran_stop_string(int8_t *var13, int32_t
   ↪ var14);
// void _gfortran_transfer_character_write(struct
   ↪ struct___st_parameter_dt *var4, int8_t *var5,
   ↪ int32_t var6);
// void _gfortran_transfer_integer(struct
   ↪ struct___st_parameter_dt *var9, int8_t *var10,
   ↪ int32_t var11);
// void _gfortran_transfer_integer_write(struct
   ↪ struct___st_parameter_dt *var15, int8_t *var16,
   ↪ int32_t var17);
// void llvm_lifetime_end(int64_t var1, int8_t *var2);
```

Obr. B.13: Príklad 2, výstupný zdrojový kód bez transformácií, siedma časť

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

/* -------- Function Prototypes --------- */

int32_t ngcd_(int32_t *a1, int32_t *a2);
void MAIN__(void);

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

int8_t g1[2] = "A?";
int8_t g2[29] = "A must be a positive integer.";
int8_t g3[2] = "B?";
int8_t g4[29] = "B must be a positive integer.";
int8_t g5[10] = "The GCD of";
int8_t g6[4] = " and";
int8_t g7[3] = " is";
int8_t g8[1] = ".";

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

int32_t ngcd_(int32_t *a1, int32_t *a2) {
    int32_t v1 = *a1;
    int32_t v2 = *a2;
    if (v2 == 0) {
        return v1;
    }
    int32_t v3 = v1 % v2;
    while (v3 != 0) {
        v1 = v2;
        v2 = v3;
        v3 = v1 % v2;
    }
    return v2;
}

void MAIN__(void) {
    printf("%.*s", 2, g1);
    int32_t v1 = 0;
    scanf("%i", &v1);
    if (v1 < 1) {
        printf("%.*s", 29, g2);
        exit(1);
    }
```

Obr. B.14: Príklad 2, výstupný zdrojový kód po transformáciách, prvá časť

```
    printf("%.*s", 2, g3);
    int32_t v2 = 0;
    scanf("%i", &v2);
    if (v2 < 1) {
        printf("%.*s", 29, g4);
        exit(1);
    }

    printf("%.*s", 10, g5);
    printf("%d", v1);
    printf("%.*s", 4, g6);
    printf("%d", v2);
    printf("%.*s", 3, g7);
    int32_t v3 = ngcd_(&v1, &v2);
    printf("%d", v3);
    printf("%.*s", 1, g8);
    exit(0);
}

int main(int a1, char **a2) {
    MAIN__();
}
```

Obr. B.15: Príklad 2, výstupný zdrojový kód po transformáciách, druhá časť

```
#include "f2c.h"

/* Table of constant values */

static integer c__9 = 9;
static integer c__1 = 1;
static integer c__3 = 3;
static integer c__0 = 0;

/* Main program */ int MAIN__(void)
{
    /* System generated locals */
    integer i__1;

    /* Builtin functions */
    integer s_wsle(cilist *), do_lio(integer *, integer
       ↪ *, char *, ftnlen),
            e_wsle(void), s_rsle(cilist *), e_rsle(void);

    /* Local variables */
    static integer na, nb;
    extern integer ngcd_(integer *, integer *);
    extern /* Subroutine */ int exit_(integer *);

    /* Fortran I/O blocks */
    static cilist io___1 = { 0, 6, 0, 0, 0 };
    static cilist io___2 = { 0, 5, 0, 0, 0 };
    static cilist io___4 = { 0, 6, 0, 0, 0 };
    static cilist io___5 = { 0, 6, 0, 0, 0 };
    static cilist io___6 = { 0, 5, 0, 0, 0 };
    static cilist io___8 = { 0, 6, 0, 0, 0 };
    static cilist io___9 = { 0, 6, 0, 0, 0 };


    s_wsle(&io___1);
    do_lio(&c__9, &c__1, "A?", (ftnlen)2);
    e_wsle();
    s_rsle(&io___2);
    do_lio(&c__3, &c__1, (char *)&na, (ftnlen)sizeof(
       ↪ integer));
    e_rsle();
```

Obr. B.16: Príklad 2, zdrojový kód migrovaný pomocou nástroja `f2c`, prvá časť

```
    if (na <= 0) {
        s_wsle(&io___4);
        do_lio(&c__9, &c__1, "A must be a positive
            ↪ integer.", (ftnlen)29);
        e_wsle();
        exit_(&c__1);
    }
    s_wsle(&io___5);
    do_lio(&c__9, &c__1, "B?", (ftnlen)2);
    e_wsle();
    s_rsle(&io___6);
    do_lio(&c__3, &c__1, (char *)&nb, (ftnlen)sizeof(
        ↪ integer));
    e_rsle();
    if (nb <= 0) {
        s_wsle(&io___8);
        do_lio(&c__9, &c__1, "B must be a positive
            ↪ integer.", (ftnlen)29);
        e_wsle();
        exit_(&c__1);
    }
    s_wsle(&io___9);
    do_lio(&c__9, &c__1, "The GCD of", (ftnlen)10);
    do_lio(&c__3, &c__1, (char *)&na, (ftnlen)sizeof(
        ↪ integer));
    do_lio(&c__9, &c__1, " and", (ftnlen)4);
    do_lio(&c__3, &c__1, (char *)&nb, (ftnlen)sizeof(
        ↪ integer));
    do_lio(&c__9, &c__1, " is", (ftnlen)3);
    i__1 = ngcd_(&na, &nb);
    do_lio(&c__3, &c__1, (char *)&i__1, (ftnlen)sizeof(
        ↪ integer));
    do_lio(&c__9, &c__1, ".", (ftnlen)1);
    e_wsle();
    exit_(&c__0);
    return 0;
} /* MAIN__ */

integer ngcd_(integer *na, integer *nb)
{
    /* System generated locals */
    integer ret_val;

    /* Local variables */
    static integer ia, ib, itemp;
```

Obr. B.17: Príklad 2, zdrojový kód migrovaný pomocou nástroja `f2c`, druhá časť

```
    ia = *na;
    ib = *nb;
L1:
    if (ib != 0) {
        itemp = ia;
        ia = ib;
        ib = itemp % ib;
        goto L1;
    }
    ret_val = ia;
    return ret_val;
} /* ngcd_ */

/* Main program alias */ int euclid_ () { MAIN__ ();
   ↪ return 0; }
```

Obr. B.18: Príklad 2, zdrojový kód migrovaný pomocou nástroja `f2c`, tretia časť

```
# ---------- Global Variables ----------

g1 = "A?"
g2 = "A must be a positive integer."
g3 = "B?"
g4 = "B must be a positive integer."
g5 = "The GCD of"
g6 = " and"
g7 = " is"
g8 = "."

# ------------ Functions ---------------

def ngcd_(a1, a2):
    v1 = *a1
    v2 = *a2
    if v2 == 0:
        return v1

    v3 = v1 % v2
    while v3 != 0:
        v1 = v2
        v2 = v3
        v3 = v1 % v2

    return v2

def MAIN__():
    printf("%.*s", 2, &g1[0])
    v1 = 0
    scanf("%i", &v1)
    if v1 < 1:
        printf("%.*s", 29, &g2[0])
        exit(1)

    printf("%.*s", 2, &g3[0])
    v2 = 0
    scanf("%i", &v2)
    if v2 < 1:
        printf("%.*s", 29, &g4[0])
        exit(1)
```

Obr. B.19: Príklad 2, výstupný zdrojový kód v jazyku Python', prvá časť

```
    printf("%.*s", 10, &g5[0])
    printf("%d", v1)
    printf("%.*s", 4, &g6[0])
    printf("%d", v2)
    printf("%.*s", 3, &g7[0])
    v3 = ngcd_(&v1, &v2)
    printf("%d", v3)
    printf("%.*s", 1, &g8[0])
    exit(0)

def main(a1, a2):
    MAIN__()

# --------- External Functions ---------

# exit()
# printf()
# scanf()

# ------------ Entry Point -------------

if __name__ == '__main__':
    import sys
    sys.exit(main(len(sys.argv), sys.argv))
```

Obr. B.20: Príklad 2, výstupný zdrojový kód v jazyku Python', druhá časť

Príloha C

Príklady migrácie jazyka D

C.1 Príklad 1

```
int f(bool isLeapYear) {
    int days = isLeapYear ? 366 : 365;
    return days;
}

int main()
{
    bool isLeapYear = true;

    return f(isLeapYear);
}
```

Obr. C.1: Príklad 1, vstupný zdrojový kód

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

/* ------------- Structures ------------- */

struct struct1 {
    int64_t e0;
    int8_t *e1;
};

struct struct2 {
    int64_t e0;
    struct struct1 *e1;
};

struct ModuleReference {
    struct ModuleReference *e0;
    struct object_ModuleInfo *e1;
};

struct object_ModuleInfo_New {
    int32_t e0;
    int32_t e1;
};

struct object_ModuleInfo {
    struct object_ModuleInfo_New e0;
    int64_t e1;
    int64_t e2;
    int64_t e3;
    int64_t e4;
    int64_t e5;
    int64_t e6;
    int64_t e7;
    int64_t e8;
    int64_t e9;
    int64_t e10;
    int64_t e11;
    int64_t e12;
    int64_t e13;
    int64_t e14;
    int64_t e15;
};
```

Obr. C.2: Príklad 1, výstupný zdrojový kód, prvá časť

```
/* -------- Function Prototypes --------- */

int32_t _D17ternary_operator11fFbZi(bool a1);
int32_t _Dmain(struct struct2 a1);
void _D17ternary_operator116__moduleinfoCtorZ(void);

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

struct {int32_t e0; int32_t e1; int8_t e2[18];} g2 = {.e0
   ↪  = -0x7ffffffc, .e1 = 0, .e2 = "ternary_operator1"};
struct ModuleReference *g3;
struct ModuleReference g1 = {.e0 = NULL, .e1 = (struct
   ↪ object_ModuleInfo *)&g2};

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

int32_t _D17ternary_operator11fFbZi(bool a1) {
    return a1 ? 366 : 365;
}

int32_t _Dmain(struct struct2 a1) {
    return _D17ternary_operator11fFbZi(true);
}

void _D17ternary_operator116__moduleinfoCtorZ(void) {
    g1.e0 = g3;
    g3 = &g1;
}
```

Obr. C.3: Príklad 1, výstupný zdrojový kód, druhá časť