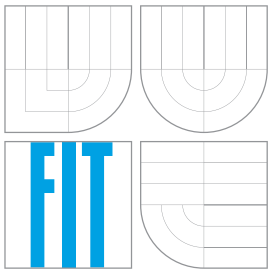
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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 [3, 8]).

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++ [6] a automatický nástroj na migrovanie zdrojových kódov z jazyka Fortran do jazyka C [5].

Dosiaľ vyvinuté nástroje často nie sú úplne automatické [6], 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 [5]. 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 [7]. 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

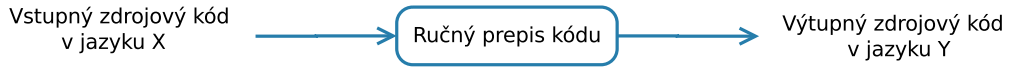
Dlhotrvajúce projekty sa musia potýkať s problémami spojenými s ich implementačným jazykom alebo prekladačom tohto jazyka. Mnoho aplikácií implementovaných pred niekoľkými desaťročiami v zastaralých jazykoch je stále dôležitou súčasťou existujúcich projektov. Udržovanie takýchto aplikácií je veľmi náročné 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 [11].

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 [3].

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

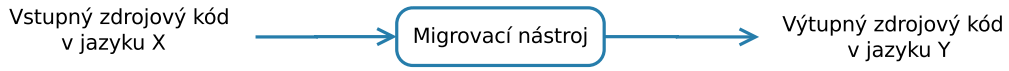
2.1 Možnosti migrácie

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 [3].



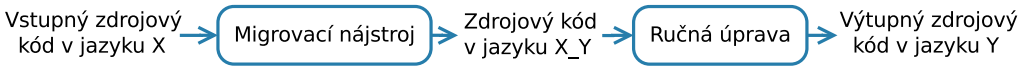
Obr. 2.1: Ručný prepis kódu

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* [5]. 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 [6]. 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.

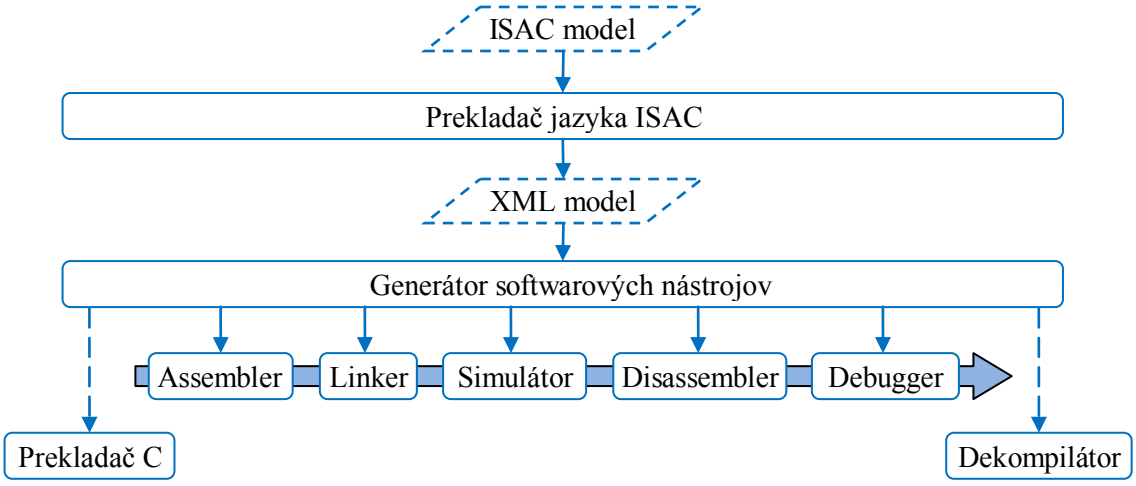
Kapitola 3

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 [7]

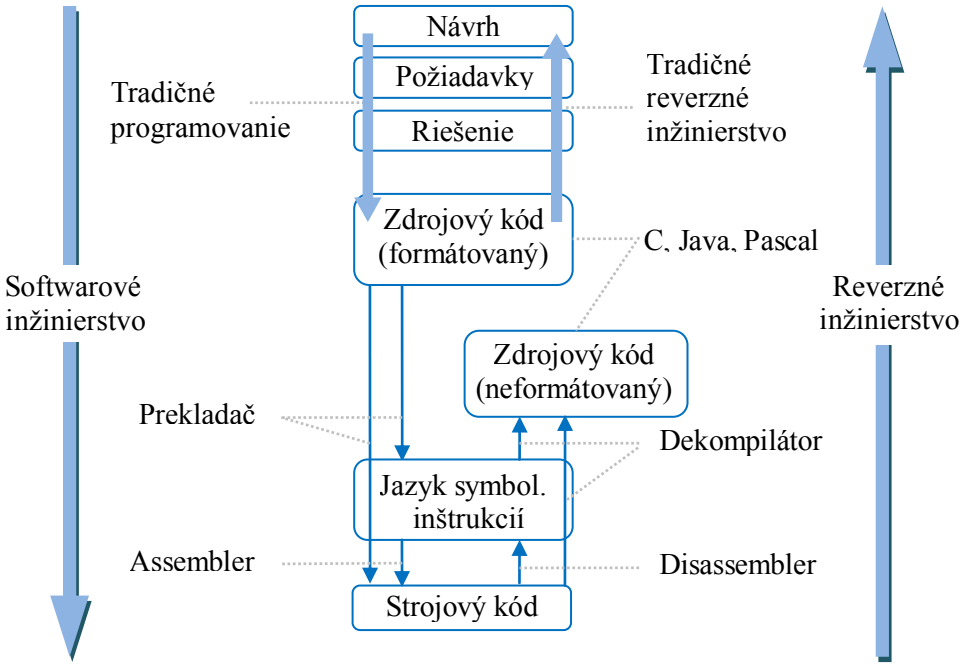
[1] http://www.fit.vutbr.cz/research/groups/lissom/project.html

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 [7]. Obrázok 3.1, prevzatý z [7], 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 [10].

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 [4]. Reverzné inžinierstvo v informatike si môžeme predstaviť ako proces inverzný k softvérovému inžinierstvu [7]. Obrázok 3.2 prevzatý z [7] tento vzťah ilustruje.



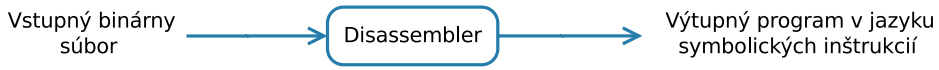
Obr. 3.2: Vzťah softwarového a reverzného inžinierstva [7]

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 [4]. 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 [4].

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 [4].

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) [4].

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 [4].

3.4 Systém LLVM

Táto sekcia z veľkej časti vychádza z [2] a [1]. 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.

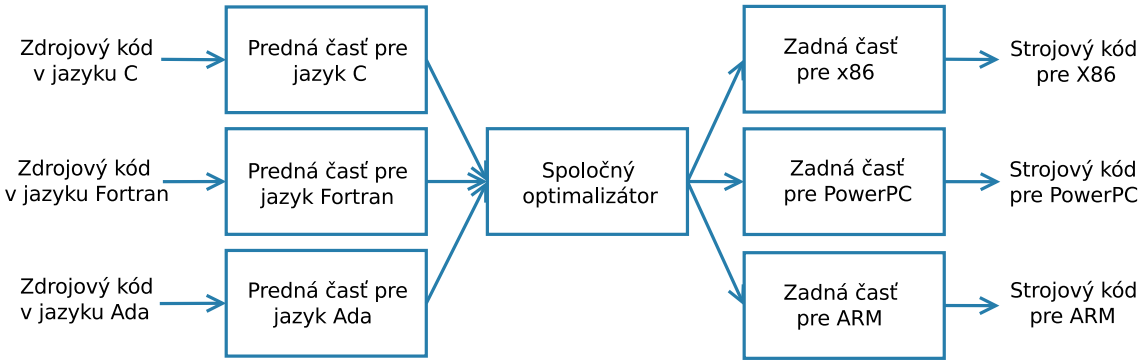
Obr. 3.5: Typická architektúra prekladača, prevzaté z [2]

[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 [2]

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 [9]. 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 [4].

```
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 [2]

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 [10]. 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.

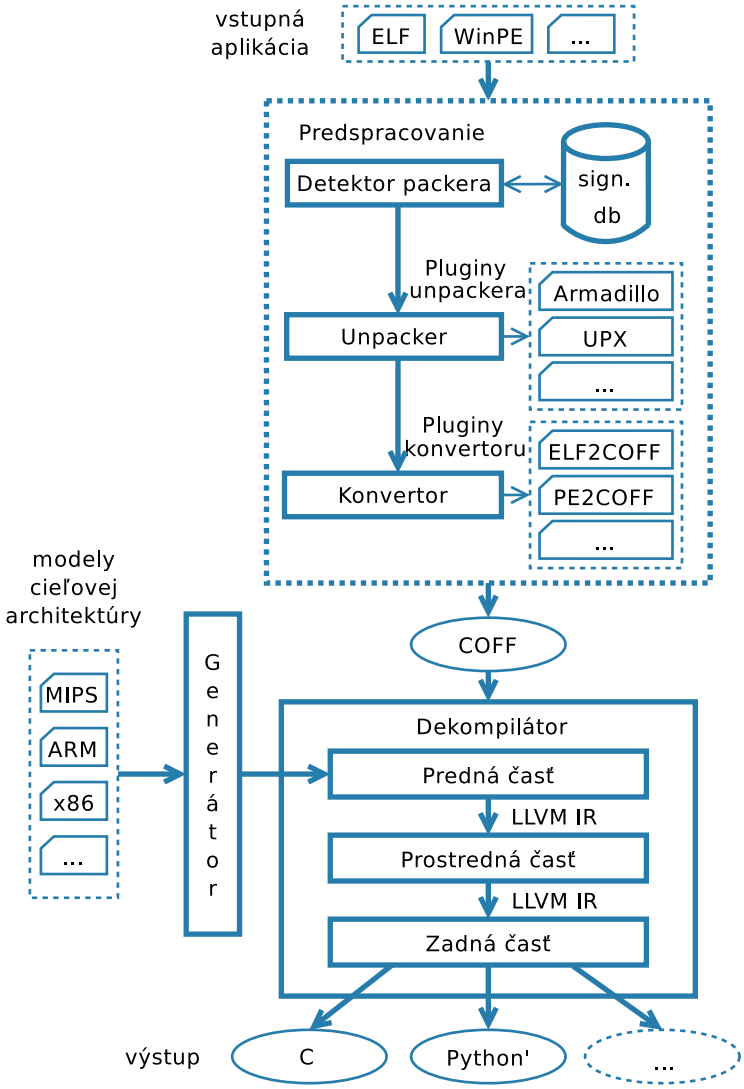
3.5.1 Predspracovanie

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 [7].

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 [10]

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

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

Kapitola 5

Výber vstupných jazykov a prekladačov

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

Kapitola 6

Návrh metód vylepšujúcich výstup migrácie

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

Kapitola 7

Implementácia

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

Kapitola 8

Testovanie

Obsah tejto kapitoly je klasifikovaný ako utajený, viz licenčné ujednanie.

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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>.

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

[8] 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.

[9] 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>.

[10] 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.

[11] 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