VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

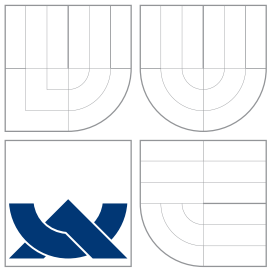
AUTOMATIZOVANÉ METODY HLEDÁNÍ CHYB V PŘEKLADAČÍCH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

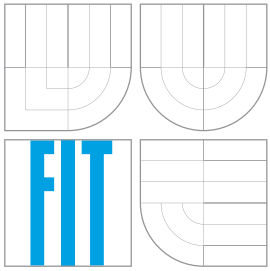
AUTOR PRÁCE
AUTHOR

PETR MÜLLER

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATIZOVANÉ METODY HLEDÁNÍ CHYB V PŘEKLADAČÍCH

AUTOMATED METHODS OF FINDING BUGS IN COMPILERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR MÜLLER

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Müller Petr**
Obor: Informační technologie
Téma: **Automatizované metody hledání chyb v překladačích**
Kategorie: Překladače

Pokyny:

1. Prostudujte metody automatizovaného testování překladačů na základě generování testovacích vstupů pro překladač a ověřování provedeného překladu srovnáním s překladem referenčním.
2. Seznamte se rovněž s metodami hledání chyb v programech pomocí statické analýzy.
3. Navrhněte a prototypově implementujte systém pro testování překladačů zvoleného jazyka, který bude využívat generátor vět daného jazyka a srovnávání výstupu testovaného překladače s referenčním překladem.
4. Aplikujte vytvořený systém na některý překladač zvoleného jazyka.
5. Zhodnoťte úspěšnost a efektivitu studované metody testování překladačů a její vámi vytvořené implementace. Porovnejte dosažené výsledky s přístupy založenými na statické analýze kódu.
6. Shrňte a zhodnoťte dosažené výsledky a navrhněte případná rozšíření vámi vytvořeného systému, příp. obecně metod vyhledávání chyb překladačích.

Literatura:

- Dle doporučení vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a alespoň část fáze návrhu z bodu třetího.

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

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

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

Vedoucí: **Vojnar Tomáš, doc. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

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

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Petr Müller**
Id studenta: 84232
Bytem: Svánovského 2341/9, 628 00 Brno
Narozen: 07. 07. 1985, Brno
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Automatizované metody hledání chyb v překladačích
Vedoucí/školitel VŠKP: Vojnar Tomáš, doc. Ing., Ph.D.
Ústav: Ústav inteligentních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

Tato práce se zabývá aplikací metody fuzz testing k testování překladačů a interpretů. V první části pojednává o překladačích, optimalizacích a chybách typických pro optimalizující překladač. Analyzuje vhodnost metod statické a dynamické analýzy pro hledání těchto chyb a jako vhodnou navrhuje dynamickou metodu fuzz testování. V rámci práce byl implementován nástroj pro testování překladačů používající tuto metodu, který byl aplikován na několik případů, přičemž se podařilo nalézt sérii chyb v rozšířených překladačích, a to včetně např. GCC.

Klíčová slova

překladače, testování, generátor náhodných vět, gramatiky, Python, jazyk C

Abstract

This thesis discusses an application of the fuzz testing method for testing compilers and interpreters. In the first part, it deals with compilers, optimizations, and bugs typical for an optimizing compiler. It analyzes applicability of static and dynamic analysis methods for searching such bugs and proposes dynamic fuzz testing as suitable for this task. A compiler testing tool suite using this method was implemented within this thesis and applied on several real compilers, including GCC, in which the tool revealed a series of bugs.

Keywords

compilers, testing, random sentence generator, grammars, Python, C language

Citace

Petr Müller: Automatizované metody hledání
chyb v překladačích, bakalářská práce, Brno, FIT VUT v Brně, 2008

Automatizované metody hledání chyb v překladačích

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením doc. Ing. Tomáše Vojnara, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Müller
14. května 2008

Poděkování

Děkuji doc. Tomáši Vojnarovi za vedení této práce a mnoho podnětných rad a připomínek. Rád bych také poděkoval panu Benovi Levensonovi za podporu a vedení mé práce v oboru testování překladačů.

Obsah

Kapitola 1

Úvod

Neodmyslitelnou součástí řetězce nástrojů pro vývoj moderních počítačových programů jsou překladače a interprety programovacích nebo jiných jazyků. Vývoj aplikací v assemblerech je v současnosti záležitostí vestavěných platforem a vysoce optimalizovaných operací. Naprostá většina programů je psána v kompilovaných nebo interpretovaných programovacích jazycích. Ty umožňují programátorovi formulovat program na vysokém stupni abstrakce a obstarávají mnoho automatizovaných úkolů, kterými se potom programátor nemusí zabývat. Překladače jsou velmi komplexní nástroje a poskytují svým uživatelů stále dokonalejší a složitější služby, např. automatickou paralelizaci pro víceprocesorové či vícejádrové systémy, masivní optimalizace apod. Většina programátorů je na těchto službách závislá, protože je mimo jejich možnosti tvořit ručně stejně výkonné programy, jako je tomu s použitím překladačů a dalších moderních nástrojů. Zároveň je tvorba programů bez těchto nástrojů neefektivní a nákladná.

Jako každý program, i překladače obsahují chyby. Moderní překladače jsou typicky velmi složité systémy, které během až stovek průběhů kódem mohou naprosto změnit jeho podobu. Vzhledem k výsadnímu postavení překladačů v řetezci vývoje programů je tedy velmi podstatná jejich správná funkčnost.

Tato práce si klade za cíl zhodnotit možnosti automatizace testování překladačů, na základě těchto znalostí implementovat systém pro hledání chyb v nich a otestovat jeho aplikaci v praxi. Tato práce se zaměřuje na testování překladačů jazyka C, ale obecné principy zde popsané je možné aplikovat i na testování nejen překladače kompilovaných jazyků, ale i pro hledání chyb v interpretech.

První část práce obsahuje teoretický základ pro testovaní překladačů a zahrnuje základní definice z oblasti. Dále pojednává o chybách, které se v překladačích vyskytují a metodách, které je možné při jejich hledání použít.

Druhá část obsahuje návrh a popis implementace systému, který tyto metody popsané v první části uvádí v rámci této práce do praxe přičemž využívá fakt, které byly zjištěny v první části, k úplnému a efektivnějšímu testovacího pokrytí překladače.

Ve třetí části byl implementovaný prototyp systému aplikován na některé rozšířené překladače jazyka C. Je zde popsána metodika testování, jeho průběh a analyzovány výsledky testování.

Kapitola 2

Překladače a optimalizace

Tato kapitola zpracovává teoretický základ pro oblast překladačů. Uvádí základní principy konstrukce překladače, jeho obvyklé součásti a jejich funkci. Zabývá se také optimalizacemi, které moderní překladače nad kódem provádějí pro vyšší efektivitu přeloženého programu.

2.1 Překladače a interprety

2.1.1 Překladače

Překladač [6] (též kompilátor, z angl. *compiler*) je program, schopný číst program v jednom jazyce – zvaném *zdrojovém* – a přeložit ho do ekvivalentního programu zapsaném v jiném jazyce – *cílovém*. Pojem *překladač* je typicky používán pro programy provádějící překlad programu zapsaného ve vyšším programovacím jazyce do jazyka nižšího, nejčastěji strojového kódu počítače. Příkladem jazyků, které jsou obvykle přímo překládány jsou C, C++ nebo rodina jazyků Fortran. Cílem je z abstraktního zápisu algoritmu získat spustitelný program. Programy provádějící opačný překlad se nazývají *dekompilátory*. Existují i programy, které provádějí překlad mezi jazyky zhruba stejné úrovně abstrakce.

2.1.2 Interprety

Interpret je naproti tomu program, který místo překladu programy napsané ve vyšším programovacím jazyce přímo spouští, tj. *interpretuje*. Interpret bývá obvykle postaven jako virtuální stroj přímo provádějící příkazy programovacího jazyka nebo instrukce vnitřního mezikódu. Výhodou tohoto přístupu je vyšší abstrakce od hardwaru a operačního systému, což s sebou nese větší pohodlí, robustnost, menší náchylnost k chybám a lepší možnosti diagnostiky. Naproti tomu bývá program přeložený do strojového kódu překladačem mnohem rychlejší než ekvivalentní program prováděný interpretem, a to z důvodu vyššího množství režijního kódu.

Typickými zástupci interpretovaných jazyků jsou moderní dynamické programovací jazyky, tedy např. Python, Ruby nebo Lua.

2.2 Principy konstrukce překladačů

Způsobů konstrukce překladačů je více, ale nejčastěji se uplatňuje dělení programu na dvě části. Prvním je front-end, který je orientovaný na zdrojový jazyk, a back-end, orientovaný na cílový jazyk. Některé překladačové systémy, jako například GCC, které mají rozsáhlou

část optimalizací nad mezikódem, s touto částí pracují jako se samostatnou [16] a nazývají ji middle-end.

2.2.1 Front-end

V části překladače nazvané front-end se obvykle provádějí tyto kroky: lexikální analýza, syntaktická analýza, sémantická analýza a překlad do mezikódu. Konstrukce front-endu je silně vázána na zdrojový jazyk a měla by být nezávislá na cílovém jazyce.

Lexikální analýza

Část překladače provádějící lexikální analýzu se nazývá lexikální analyzátor (angl. *scanner*).

Lexikální analýza je úvodní krok překladu, který v čistém textu rozeznává tzv. lexémy (angl. *tokeny*) – prvky zdrojového jazyka. Ty následně předává syntaktickému analyzátoru. Lexikální analyzátory bývají implementovány obvykle pomocí regulárních výrazů.

Pro tvorbu scannerů jsou běžně používané automatizované nástroje, např. Lex [13].

Syntaktická analýza

Syntaktická analýza rozeznává v sekvenci lexikálních symbolů syntaktickou strukturu podle pravidel gramatiky příslušného jazyka. Výstup této části překladače zachycuje strukturu programu ve tvaru vhodném k dalšímu zpracování, typicky ve tvaru derivačního stromu. Části překladače provádějící syntaktickou analýzu se říká syntaktický analyzátor (angl. *parser*). Metod sestrojení syntaktického analyzátoru je více. Základními dvěma přístupy (které se pak dále dělí) jsou přístup shora-dolů a zdola-nahoru. Různé přístupy [10] k sestrojení syntaktického analyzátoru produkují parsery schopné zpracovat různě složité gramatiky. Parsery se často generují automatizovanými nástroji, např. Yacc [12].

Sémantická analýza

Sémantický analyzátor v derivačním stromu rozeznává a zpracovává informace o významu příkazů. Uplatňuje sémantická omezení a oznamuje chyby v sémantice programu.

Překlad do mezikódu

Většina překladačů nepřekládá přímo z jednoho jazyka do druhého (i když i takové překladače existují), ale používají ještě třetí interní reprezentaci. Existují dva základní typy interní reprezentace: [6]

- Syntaktické a jiné stromy. V tomto případě je interní reprezentace generována jako výstup fází syntaktické a sémantické analýzy.
- Serializovaná reprezentace. Výstup sémantické analýzy se převede na posloupnost instrukcí jazyka interní reprezentace. Typickým příkladem je tříadresový kód.

Složité překladače mohou mít i několik interních reprezentací kódu [16]. V tom případě se část obsluhující mezikód obvykle neřadí do přední (*front-end*), ale do střední vrstvy (*middle-end*). Převody mezi interními reprezentacemi představují složité části překladačů, psané obvykle ručně.

2.2.2 Middle-end

Middle-end se nazývá ta část překladače, kde se na úrovni interní reprezentace provádějí optimalizace a jiné změny přímo v logice programu. Jen některé projekty nazývají tuto část překladače middle-end, jiné tento termín nepoužívají a operují jen s dvěma částmi. V této práci je termín *middle-end* používán.

Kromě operací nad interní reprezentací programu probíhají v této části optimalizace, které mají za úkol upravit program pro nějaký účel – obvykle k vyššímu výkonu nebo k minimalizaci velikosti výsledného programu. Optimalizacemi se blíže zabývá část 2.3.

2.2.3 Back-end

Back-end je část vázaná na cílový jazyk, obvykle strojový kód nebo kód v aassembleru Výsledná podoba interní reprezentace programu se zde převádí do cílového jazyka. Problémy řešené v této fázi jsou velmi různé, ale příkladem mohou být následující, které jsou nejobvyklejší:

- Efektivní využití instrukční sady cílového procesoru, včetně složitých a specializovaných instrukcí.
- Alokace registrů. Vzhledem k výkonu je vhodné maximum dat umisťovat do registrů místo do paměti.
- Paralelizace. Využití možností cílové architektury k paralelnímu běhu vhodných částí programu na multi/hyperskalárních procesorech, popř. systémech s více jádry nebo procesory.

2.3 Optimalizace

Rozšířené programovací jazyky a jejich sémantické konstrukce nejsou ze své podstaty vytvořené pro optimální využití zdrojů programy, které jsou v nich napsány. Jsou uzpůsobené primárně k snadnému vyjádření algoritmu člověkem a jeho následnému překladu do strojového kódu. Pokud bychom překládali program jen jako sekvenci na sobě nezávislých příkazů a každý příkaz doslovně přeložili do strojových instrukcí, vznikne množství neefektivního kódu. Pro optimální využití zdrojů (čas, paměť, místo na disku) je tedy nutné program zefektivnit. Toto může být provedeno buď programátorem, pokud přizpůsobí svuj program požadavkům na optimální využití zdrojů, nebo automaticky překladačem. Právě v překladačích se skryvá velký potenciál pro zefektivnění programu. Během překladu má překladač k dispozici velmi detailní informace o podobě překládaného programu. Tyto informace lze analyzovat a na základě této analýzy automaticky rozeznávat instrukce, které lze vypustit nebo nahradit jinými bez změny funkce programu. Moderní překladače obsahují rozsáhlou sadu často velmi složitých optimalizací.

Obecně se optimalizací nazývá jakýkoliv proces na kterékoliv úrovni tvorby programu, který zefektivňuje jeho využití kritických zdrojů. V této práci se z důvodu jejího zaměření na překladače používá termín “optimalizace” pro změny kódu, které automaticky vykonává kompilátor při překladu programu.

Konkrétním příkladem optimalizací [6], které může překladač nad kódem provádět může být eliminace mrtvého kódu, šíření konstant, nebo přímý výpočet výrazů, jejichž výsledek je znám už v době překladu.

Kapitola 3

Chyby v překladačích a metody jejich hledání

Jako všechny programy, i překladače obsahují chyby. Tato kapitola se zabývá chybami, které se vyskytují v překladačích, důsledky jejich výskytu, a metodami jejich hledání.

3.1 Chyby v překladačích

Překladač je důležitou součástí vývojového řetězce. Chyby v něm tedy mohou mít masivní důsledky. Takovým může být například zbrždění vývoje programu, kdy musí být kód, který překladač nedokáže přeložit, nahrazen jiným. Jiným příkladem může být zanášení chyb vzniklých špatným překladem do programu, popřípadě tichý překlad neplatného kódu s nedefinovaným chováním. Takové chyby se zvláště špatně diagnostikují, protože vyplynou na povrch v přeloženém programu, nikoliv překladači samém. Samotnému určení zdroje chyby v překladači tedy předchází fáze, kdy se chyba hledá na nesprávném místě, což zbytečně spotřebovává zdroje a při pokusu o opravu může způsobit zanesení další chyby.

3.1.1 Obecné chyby v programech

Překladače, zvláště moderní překladačové systémy pro větší množství jazyků a architektur, jsou typicky velmi komplexní programy. Jako takové jsou pochopitelně náchylné pro výskyt běžných programátorských chyb, stejných jako v jakémkoli jiném programu. Pro jazyk C jsou takovými typickými chybami například špatná práce s ukazateli, ignorování možných chybových stavů nebo nedostatečná opatření při práci s polemi konečné délky. Rozšířené překladače (GCC, Visual C++ Compiler) tyto běžné chyby, které by u obyčejného programu obvykle vedly ke zhroucení nebo nedefinovanému chování, zachytávají vnitřními mechanismy (zachytávání signálů, makra `assert` a jiné) a interpretují je jako tzv. ICE – Internal Compiler Error.

3.1.2 Chyby typické pro překladače

Kromě běžných chyb se v překladačích vyskytují chyby, které jsou pro ně specifické. Jedná se obvykle o sémantické chyby – překladač funguje tak, jak byl naprogramován, ale byl naprogramován špatně. Tyto chyby se dají roztřídit do několika kategorií a následně klasifikovat. Open-source komunita zabývající se překladači obvykle používá kategorie, které definuje projekt GCC [1], nebo alespoň kategorie velmi podobné.

Klasifikace chyb dle projektu GCC

Accepts-invalid Překladač tiše akceptuje a přeloží neplatný kód, i když by ho měl odmítnout. Chování neplatného kódu není definováno a výsledkem je nepředvídatelné chování preloženého programu. Jedná se o velmi vážnou chybu – chyba se projeví vadným chováním přeloženého programu a proces určování zdroje chyby je dlouhý a náročný.

Assemble-failure Překladač selže při generování výsledného kódu. Důvody mohou být různé, např. dvojitá definice jednoho symbolu. Protože se tento typ chyby projeví už při překladu, a to až po fázi přijetí kódu, jedná se o nepříliš závažný typ chyby.

Compile-time-hog Překlad trvá neadekvátně dlouho, nebo se nezastaví. Příčinou bývá neoptimalizovavý průběh kódem, kdy se vlivem podoby vnitřní reprezentace překladač zacyklí nebo zvolí velmi neoptimální cestu. Jedná se o málo závažný typ chyby, vyvstane už při použití překladače a pomocí profilovacích nástrojů nebo vnitřní introspekce překladače je možné relativně snadno neoptimální místo programu odhalit.

Diagnostic Selhání diagnostického hlášení. Překladač nezobrazí chybové hlášení nebo varování, když by měl, popř. je takové hlášení chybné nebo zavádějící.

ICE-on-invalid Překladač se zhroutí při zpracování neplatného kódu. Jedná se o nepříliš závažnou chybu. Neplatný kód by měl být stejně odmítnut, takže zhroucení překladače je pouze chyba robustnosti při ošetřování neplatného kódu. ICE může být způsobeno i běžnou programátorskou chybou.

ICE-on-valid Překladač se zhroutí při zpracovávání platného kódu. Jedná se o závažnou chybu. Platný kód by měl být přeložen, a selhání v tomto úkolu je tedy selhání základního účelu překladače. Diagnostika bývá obvykle nepříliš obtížná, s použitím moderních programátorských nástrojů může být poměrně snadné identifikovat místo kde ke zhroucení došlo a data nebo instrukce, které ho způsobily. ICE může být způsobeno i běžnou programátorskou chybou.

Link-failure Překladač vygeneruje objekt, který nejde sestavit (slinkovat) s jiným kódem. Jedná se o méně závažnou variantu chyby *wrong-code*, protože vyjde najevo už při linkování programu, což následuje obvykle hned po překladu. Důvody selhání sestavení jsou podobné jako u chyby *assemble-failure*.

Memory-hog Překladač spotřebovává při běhu neadekvátní množství paměti. Příčiny a ostatní znaky jsou podobné jako u chyby *compile-time-hog*.

Missed-optimization Překladač nezaregistruje část kódu, kterou by mohl optimalizovat. Velmi špatně odhalitelná chyba, zejména z toho důvodu, že se projevuje až v nižším výkonu přeloženého programu, což je zpozorovatelné jen při důkladném testování nebo opravdu výrazném snížení výkonu. Závažnost závisí na velikosti snížení výkonu aplikací.

Register-allocator Chyba typu *missed-opimization* způsobená vadnou funkcí alokátoru registrů. Překladač v tomto případě nevyužije možnosti uložit hodnotu do registru místo do paměti, i když by to v danou chvíli bylo možné.

Rejects-valid Překladač odmítne platný kód. Jedná se o středně závažnou chybu, která brání použití kompilátoru. Diagnostika není náročná, chyba se projevuje už při překladu.

Wrong-code Překladač generuje vadný kód – kód s jiným významem než měl původní platný zdrojový kód. Jedná se o nejzávažnější chybu překladače. Chyba se projevuje až vadným chováním přeloženého programu, přičemž není možné ve zdrojovém kódu vystopovat chybu – protože tam žádná není. Chybu je nutné izolovat a objevit na úrovni cílového jazyka přeloženého programu, což je zdlouhavé a náročné.

Klasifikace závažnosti chyb

Tabulka 3.1 uvádí klasifikaci závažnosti chyb v překladačích, vypracovanou v rámci této práce. V úvahu byla brána tři hlediska – dopad na uživatele, způsob, jakým se chyba projeví, a diagnostika.
Dopad na uživatele byl rozdělen do tří úrovní:

- *Nízký*: Dopad na uživatele je nízký, nebrání mu ve správném použití překladače.
- *Střední*: Omezuje možnosti uživatele a nutí jej používat nestandardní řešení. Překladá programy které fungují správně, ale s určitými omezeními.
- *Vysoký*: Zabraňuje uživateli v použití překladače, nebo generuje neplatné programy.

Způsob, kterým se chyba projeví, je také trojí:

- *Při překladu*: Chyba se objeví již při překladu.
- *Po překladu*: Chyba se projeví bezprostředně po použití výstupu překladače.
- *Neprojeví*: Chyba se neprojeví v souvislosti s překladačem.

Diagnostiku dělíme také do tří úrovní:

- *Zjevné*: Zjevně se jedná o chybu překladače.
- *Maskované*: Chyba se objevuje při použití překladače, ale zdá se, že překladač je v pořádku.
- *Skryté*: Chyba je skrytá a projevuje se při použití výsledného programu. Souvislost se špatnou funkcí překladače vyjde najevo až při důkladné analýze defektu.

3.2 Hledání a testovací pokrytí různých typů chyb

Obecné metody hledání chyb v překladačích se v zásadě neliší od metod používaných pro jiné programy. Obtíž spočívá ve faktu, že překladače jsou programy s nekonečným prostorem možných vstupů, a navíc není možné dopředu jednoznačně určit, jak má překlad vypadat – pro jeden vstupní program P obvykle existuje více možných ekvivalentních reprezentací v cílovém kódu. To výrazně snižuje možnosti testování správnosti výstupu.

Tabulka 3.1: Klasifikace typů chyb v překladačích

Typ chyby	**Dopad na uživatele**	**Způsob projevu**	**Diagnostika**
accepts-invalid	Vysoký	Neprojeví se	Skryté
assemble-failure	Vysoký	Po překladu	Maskované/Zjevné
compile-time-hog	Střední	Při překladu	Zjevné
diagnostic	Nízký	Při překladu	Maskované
ice-on-invalid	Nízký	Při překladu	Zjevné
ice-on-valid	Vysoký	Při překladu	Zjevné
link-failure	Vysoký	Po překladu	Maskované/Zjevné
memory-hog	Střední/Nízká	Při překladu	Maskované
missed-optimization	Střední	Neprojeví se	Skryté
register-allocator	Střední	Neprojeví se	Skryté
rejects-valid	Vysoký	Při překladu	Maskované/zjevné
wrong-code	Vysoký	Neprojeví se	Skryté

3.2.1 Statická analýza

Statická analýza je proces, kdy se zjišťují informace o programu bez jeho spuštění. Může se jednat o sběr dat analýzou zdrojového kódu, reprezentace v mezikódu během překladu nebo analýzu objektového a binárního tvaru přeloženého programu. Konkrétní metody jsou různé – nejběžnější jsou nástroje, které analyzují zdrojový kód a hledají vzorce indikující typické programátorské chyby, nebo nástroje typu `lint`, které hledají podezřelé sémantické konstrukce. Takové nástroje také dokáží například nalézt větve programu, které se nikdy neprovedou, nebo naopak takové, které se provedou vždy, přestože jsou zanořeny v podmíněném příkazu. Možností hledaných vzorů je mnoho a je možné staticky hledat i chyby, které nejsou jen obecné sémantické chyby, ale chyby v logice programu.

Možnosti metod statické analýzy pro testování překladačů jsou omezené na hledání chyb typu ICE, které jsou často způsobeny běžnými programátorskými chybami.

3.2.2 Dynamická analýza

Dynamickou analýzou se nazývá proces, kdy o programu získáváme informace jeho spouštěním. Spuštěním programu v prostředí virtuálního procesoru můžeme například detekovat chyby, které z kódu nemusí být zcela zjevné a tudíž je obtížné je odhalit statickými metodami. Typickým problémem, který je možné odhalit pomocí spuštění jsou problémy souběhu (angl. *race condition*), popřípadě problémy vzniklé nesprávným použitím paměti v jazyce C.

Příkladem nástroje pro dynamickou analýzu programů je soubor utilit *valgrind*. Obsahuje nástroje určené pro detekci paměťových chyb, chyb spojených s vlákny, detektor souběhů, a profilovací nástroje volání funkcí a využití hromady (*heapu*). Valgrind [9] používá právě přístup virtuálního procesoru, který provádí testované instukce, ke kterým jednotlivé nástroje přidávají svoje vlastní informace pro analýzu.

3.2.3 Metody testování překladačů cílené na jejich typické problémy

Všechny obecné nástroje pro zajištění kvality softwaru lze samozřejmě použít i k hledání chyb v překladačích – je to také software. Sofistikované nástroje dokáží najít mnoho chyb, ale nejsou cílené. Vzhledem k nejednoznačnosti správného výstupu překladače obecné metody selhávají v detekci sémantických chyb, zejména správně implementovaných špatných principů. Tyto chyby lze najít jen velmi špatně. Neexistuje mnoho metod, kterými je možné určit, zda překladač pro nějaký vstup vytvořil správný výstup.

Jednou z cest je pokrýt každou metodu nebo funkci uvnitř překladače, která nějak manipuluje s kódem, kvalitními jednotkovými testy. Pokud jsou jednotlivé části překladače navrženy dostatečně ortogonálně, a každá metoda nebo funkce má jasně definované vstupy, výstupy a činnost, pak je možné jednotkovými testy zjišťovat, zda každý takový úsek kódu dělá to, co je definováno. Zde platí, že správná činnost systému tvořeného mnoha menšími jednotkami (které musejí být co nejvíce ortogonální) se ověřuje snadněji [11], pokud ověřujeme činnost jednotlivých součástí, než pokud bychom ověřovali chování celého systému.

Jinou metodou, jejímž použitím je možné se zaměřit právě na správný výstup překladače, je tzv. *fuzz testing*, na kterém byl založen systém dále navržený v této práci. Princip této medody a její praktická implementace pro toto použití je popsán v následující kapitole.

Kapitola 4

Návrh systému pro testování překladačů

V této kapitole představíme návrh metody pro testování správnosti výstupu překladače a implementaci systému, který tuto metodu využívá.

4.1 Principy metody fuzz testing

Základem celého systému je metoda zvaná *fuzz testing*. Jedná se o metodu testování typu *black-box*[1], kdy se pro program generuje velké množství testovacích vstupů, které jsou programem zpracovány. Chování programu se poté analyzuje. Je jasné, že zcela náhodně generovaný vstup bude pravděpodobně neplatný, a proto se za selhání považuje zhroucení programu. Pokud program dokáže náhodná data jakkoliv zpracovat, výsledkem testu je úspěch. Výhodou této metody je její až triviální implementace a automatizace. Je snadné generovat testovací případy, předkládat je programům a zjišťovat, zda se zhroutí. Velká rychlost generování testovacích případů znamená, že je možné otestovat mnoho případů v relativně krátkém čase. To zvyšuje pravděpodobnost, že některá náhodná data aktivují dosud skrytou chybu v programu, kterou spustí jen data s určitými charakteristikami. Zároveň tato metoda dokáže program vystavovat podmínkám, jaké by člověk-tester pravděpodobně nevytvořil.

Účelem testu samozřejmě není zjišťovat prostý fakt, zda program dokáže zvládnout libovolný neplatný vstup. Pravým důvodem je, že zhroucení programu na jakýchkoliv datech může ukazovat na neošetřenou speciální situaci, zpravidla typu přetečení zásobníku (*buffer overflow*).[2] Vzhledem k tomu, že chyby tohoto typu jsou hlavním zdrojem bezpečnostních problémů, je vhodné takto testovat zejména pro síťové služby a jiné programy, které dostávají data ke zpracování po síti nebo z jiného nedůvěryhodného zdroje. Schopnost zhroutit server síťové služby pouhým předložením speciálních dat je bezpečnostní chyba sama o sobě [15].

Fuzz testing je možné použít velmi cíleně, a to i na překladače. Příkladem může být práce Christiana Lindiga [14], který vytvořil program, testující dodržování volacích konvencí překladači jazyka C pro různé architektury. Při testování samotném dokázal velmi rychle najít chyby ve zpracování složitějších struktur v případě, že byly použity jako variadický

[1] *black-box* je testovací metoda zaměřená na zkoumání chování systému na základě jeho vnějších projevů, bez znalosti vnitřní implementace.

[2] *buffer overflow* je chyba, kdy se program pokouší zapsat data za hranice místa v paměti, které je jim vyhrazeno [8]

argument [18].

Zde navržený systém pro testování překladačů se zaměřuje na schopnost překladače dodat správnou reprezentaci programu v cílovém jazyce. Vzhledem k tomuto záměru je nutné jednoduchou koncepci fuzz testování rozšířit. V případě, že bychom generovali zcela náhodná data, byla by naprostá většina vstupních dat neplatným kódem, který by překladač měl odmítnout. Byla by tedy testována pouze ta část kódu překladače, která odmítá neplatné vstupy.

Je nutné sestrojit generátor náhodného platného kódu zdrojového jazyka, který překladač nebude mít problém přeložit. Tím se pokrývá celý proces překladu programu a je tak možné aktivovat chyby v každém kódu, který je zapojen do procesu překladu.

Dalším problémem, který je pak zapotřebí rozřešit, je zjišťování, zda je výstup překladače platný. Jak bylo řečeno v části 3.2, pro jeden program obvykle existuje více platných překladů, v závislosti např. na použitém překladači, optimalizačních procesech, nebo na úrovni, s jakou překladač dokáže využít specializovaných instrukcí cílového jazyka (instrukcí procesoru). Není tedy možné porovnávat samotný výstup, protože nemůžeme jednoznačně určit, zda je překlad správný nebo ne. Přístup našeho systému je přeložit program dvakrát nebo vícekrát v odlišných podmínkách, což může být přeložení jiným překladačem, nebo přeložení stejným překladačem s odlišným nastavením. Přeložené programy se spustí a zaznamená se jejich výstup. Program by měl produkovat stejný výstup bez ohledu na to, jakým překladačem byl preložen. Testem je tedy srovnání obou výstupů, popřípadě jiná pomocná kritéria (porovnání úspěšnosti překladu nebo způsobu ukončení programu). Pokud je úspěšnost překladu a výstup přeloženého programu stejný, pak je výsledkem testu úspěch. Pokud se liší, pak to může indikovat chybu v jednom z překladačů, protože je nepravděpodobné, že by stejná chyba byla přítomná v obou na sobě nezávislých překladačích.

Tomuto přístupu musí být přizpůsoben i generátor testovacího kódu, který by měl produkovat programy s deterministickým chováním. Programy by měly také poskytovat na výstup dostatek informací o svém vnitřním stavu, aby bylo možné zjistit anomálie.

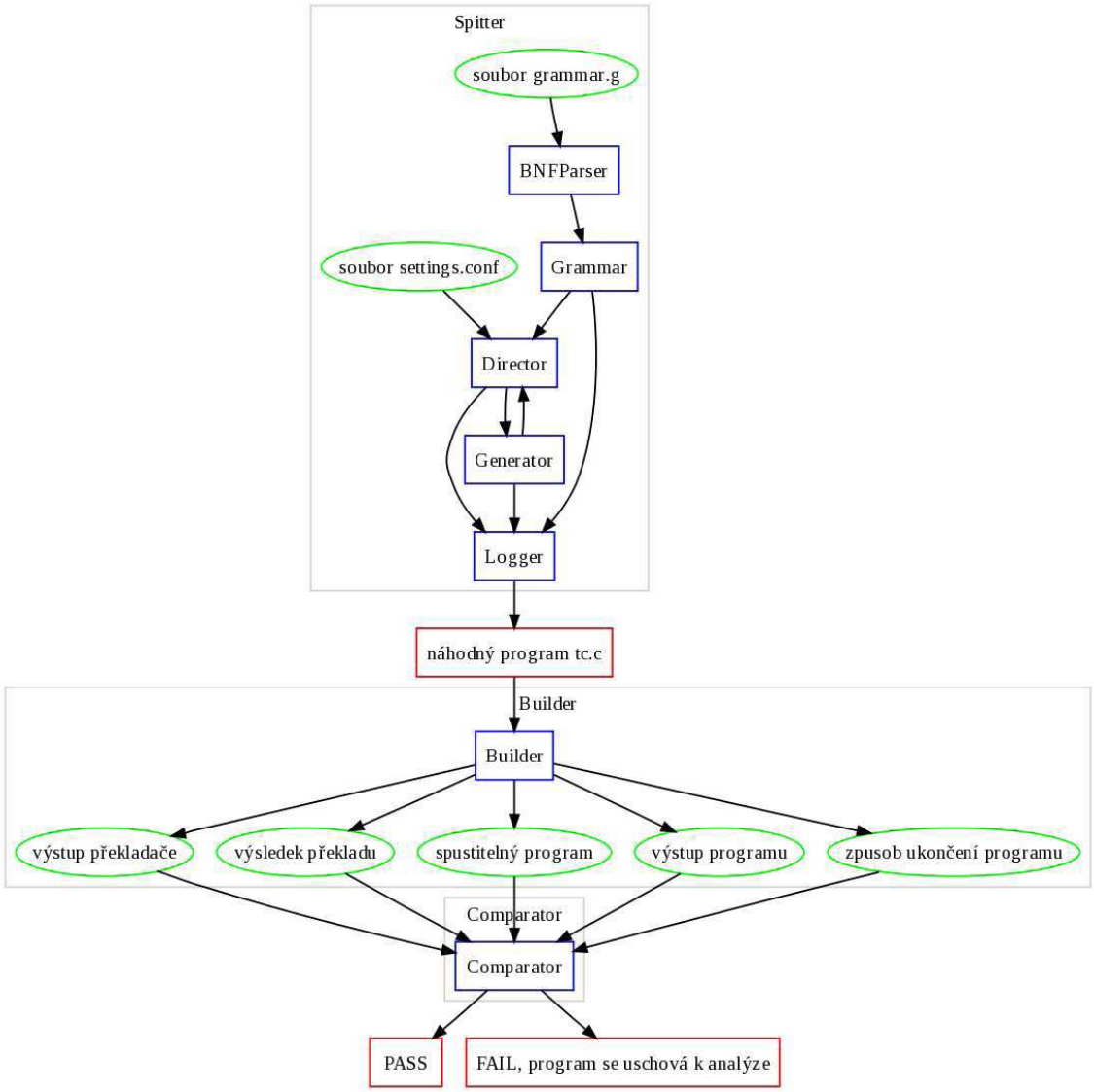
4.2 Celkový návrh a implementace

Navržený systém pro testování je rozdělen na tři části, spojené dohromady jednoduchým rámcem. První a nejdůležitější částí je generátor testovacího zdrojového kódu. Systém je navržen tak, aby se dal použít jakýkoliv program, který vyprodukuje použitelný vstup. Obsahuje ale svůj vlastní generátor kódu, jménem Spitter. Výstup generátoru je poté vstupem pro modul Builder, který vyprodukuje všechny výstupy k analýze. Tyto výstupy jsou poté analyzovány modulem Comparator, který rozhoduje, zda byla nalezena anomálie nebo ne. V případě že ano, pak se zdrojový kód a všechny výstupy uschovají k pozdější analýze. Jeden test proběhne velmi rychle, a proto je možné v krátkém čase otestovat velké množství testovacích případů. Tím je dosaženo většího pokrytí kódu testovaných překladačů.

Většina kódu v celém systému je napsána ve skriptovacím jazyce Python, část utilit je napsána v jazyce příkazového interpretu BASH. Oba interpretované jazyky byly zvoleny z důvodu snazší implementace, zejména v případě generátoru testovacího kódu jazyka C.

Celý systém je přehledně graficky znázorněn na obrázku 4.1.

Obrázek 4.1: Schéma systému



4.3 Spitter

Výstupem Spitteru je program v jazyce C, který je v pozdějších fázích překládán a analyzován. Jedná se o řízený generátor náhodných vět jazyka C. Skládá se ze dvou hlavních částí, pojmenovaných Director a Generator, a dalších tří menších. Podrobnější popis všech částí je uveden v této sekci.

4.3.1 Generátor

Generátor je modul, obsahující jednu třídu. Jediné, co generator provádí je tvorba náhodných vět podle gramatiky.

Princip generátoru je totožný [17] s konstrukcí syntaktického analyzátoru pomocí techniky rekurzivního sestupu, jen postup je obrácený. Parser užívající rekurzivní sestup je program, který se skládá ze vzájemně rekurzivních funkcí, které rozeznávají jednotlivé jazykové konstrukce. Schéma syntaktického analyzátoru pak kopíruje podobu gramatiky jazyka. Generátor je implementován podobně. Implementuje metodu `generate(symbol)`, která získá od Directoru seznam možných rozvojů. Z nich se náhodně vybere jeden rozvoj. Náhodný výběr je ovlivněn váhami, které mohou být jednotlivým rozvojům v seznamu přiřazeny. Rozvoj je reprezentován dalším seznamem, který obsahuje všechny symboly v rozvoji. Pro každý symbol v rozvoji se provede následující operace: pokud se jedná o terminál, pak se přímo zapíše do výstupního bufferu, který udržuje již vygenerované symboly. V případě, že se jedná o neterminál, pak se pro něj rekurzivně zavolá metoda `generate` a návratová hodnota se zapíše do bufferu. V případě že se jedná o volitelný terminál, pak se náhodně zjistí, zda se bude symbol generovat a v případě že ano, pak je postup stejný jako u neterminálu. Přehledně to ukazuje pseudokód 4.3.1. Jedná se o velmi zjednodušený princip, který znázorňuje jen samotné generování. Skutečná implementace je složitější, obsahuje další obslužný kód (logování a ošetřování speciálních situací), a navíc funkci návratu k předchozím verzím, pokud nějaký symbol z nějakého důvodu nemohl být vygenerován (tzv. *backtracking*). Tato funkce, přestože je v generátoru implementována, není systémem využívána, protože s ní nedokáže korektně pracovat vrstva Director. Pokud by toto omezení bylo odstraněno, umožnilo by to použít agresivnější potlačování rekurzivních problémů popsaných níže, protože by již nebylo nutné ponechávat každému symbolu alespoň jeden rozvoj i v případě, že i ten by byl za normálních okolností podle konfigurace vypuštěn.

Jednoduchý generátor sestrojený výše popsaným způsobem trpí výraznými problémy v případě, že generovaná gramatika obsahuje pravidla, která jsou rekurzivní. V programovacích jazycích jde zejména o pravidla popisující výrazy, kdy operandem většiny operátorů mohou být znovu další výrazy. V gramatice jazyka C [18] jsou výrazy popsány větším množstvím pravidel, jejichž hierarchie vyjadřuje prioritu operátorů. Všechna tato pravidla jsou rekurzivní, a pravidlo na nejnižší úrovni obsahuje rozvoj se symbolem výrazu opět na úrovni nejvyšší. V tomto konkrétním případě během generování docházelo k masivnímu zanořování generovaných výrazů a k dosažení maximální hloubky rekurzivních volání funkcí v interpretu jazyka Python. Tyto problémy byly vyřešeny ve vrstvě Director pomocí několika technik blíže popsaných v sekci 4.3.2.

4.3.2 Director

Director je nejrozsáhlejší modul z celého systému. Úkolem Directoru je řídit funkci generátoru takovým způsobem, aby výstupem byl validní a sémanticky správný kód. Funkce directoru jsou založeny na tzv. sémantických jednotkách.

Obrázek 4.2: Pseudokód principu metody `generate`

```
def generate(symbol):
    director.reportStart(symbol)
    productions =director. getSymbol(symbol)
    selected = weightedRandomSelection(productions)
    buffer = ""
    for symbol in selected:
        if symbol.type == TERMINAL:
            buffer += symbol.value
        elif symbol.type == OPTIONAL_NONTERMINAL
                        and rand(0,1) < symbol.probability:
            buffer += generate(symbol)
        elif symbol.type == NONTERMINAL:
            buffer += generate(symbol)
    buffer = director.reportFinish(symbol, buffer)
    return buffer
```

Sémantické jednotky

Sémantická jednotka je struktura, která udržuje informace o podobě jedné sémantické konstrukce jazyka C. Sémantickou konstrukcí rozumíme část programu, která nese nějakou informaci, která musí být dostupná i jiným sémantickým jednotkám (např. příkaz deklarace proměnné nese informaci, že v nadřazeném bloku je možné použít identifikátor proměnné ve výrazech, jejichž typ je s typem proměnné kompatibilní). Director buduje zásobník takovýchto sémantických jednotek a na základě informací, které jsou v něm obsaženy, řídí generování. Jednotlivé vlastnosti sémantických jednotek jsou určovány ihned, jak jsou generátorem vytvořeny a metodou `reportFinish` předány příslušné symboly. Přesný princip a informace o jednotlivých sémantických jednotkách budou popsány níže.

Sémantická jednotka udržuje následující informace:

Proměnné Udržuje informaci, zda daná sémantická jednotka vytváří rámec (*scope*) pro definici proměnných. Implementován je jako slovník, jehož klíčem je datový typ a hodnotou je seznam identifikátorů označujících proměnné tohoto typu v tomto rámci. Hodnota `None` znamená, že tato sémantická jednotka vlastní rámec nevytváří.

Identifikátor Udržuje informaci, zda daná sémantická jednotka má vlastní identifikátor. Implementován je jako řetězec, obsahující příslušný identifikátor. Pokud mu zatím identifikátor nebyl přiřazen (což je případ např. sémantické jednotky deklarace předtím, než je vygenerován symbol určující identifikátor), je řetězec prázdný. Hodnota None znamená, že daná sémantická jednotka nemůže mít přiřazen identifikátor.

Typ Datový typ sémantické jednotky, který zároveň omezuje datový typ jejích podsymbolů. Může nabývat těchto hodnot:

- True. Znamená "jakýkoliv typ". Tato sémantická jednotka nijak neomezuje typ svých podsymbolů.

- False. Tato sémantická jednotka omezuje typ svých podsymbolů, ale ještě jí nebyl přiřazen příslušný typ.
- None. Znamená "nezajímá se o typ". Tyto jednotky jsou z hlediska typu zcela ignorovány. Přejímají tak vlastně typ svých nadřazených sémantických jednotek.
- <datový typ>. Sémantická jednotka omezuje typy svých podsymbolů na datové typy kompatibilní se svým datovým typem.

Návratový typ Podobně jako typ, jen udržuje informaci o návratovém typu funkce a ničím neomezuje generování podsymbolu, kromě určení typu výrazu v příkazu `return`.

Funkce Informace a jejich reprezentace jsou podobné, jako je tomu u rámce pro proměnné. Rámec pro funkce vytváří pouze nejvyšší semantická jednotka (Program). Kromě identifikátoru a návratového typu se uchovává ještě informace o počtu a datových typech parametrů.

Parametry Opět sémantická informace specifická pro funkce. Uchovává se v podobě seznamu dvojic `(datový typ, identifikátor)`.

Funkce Directoru

Director slouží jako vrstva mezi abstraktní reprezentací gramatiky v programu a samotným generátorem, která modifikuje podobu gramatických pravidel předtím, než je generátoru předá. Obsahuje dvě základní vnitřní struktury, ve kterých udržuje aktuální podobu sémantiky programu:

Zásobník symbolů Udržuje podobu aktuální větve derivačního stromu generovaného programu. Tato data jsou použita k omezení problémů spojených s rekurzí některých gramatických symbolů.

Zásobník sémantických jednotek Udržuje podobu aktuální větve stromu sémantických jednotek. Princip sémantických jednotek je popsán níže. Tato data slouží k zaznamenávání sémantické podoby programu a řízení generování na jejím základě.

Funkce directoru spočívá v poskytování služby pro samotný generátor. Celý proces pro jeden symbol funguje následovně:

1. Director přijme od generátoru volání metody `reportStart(symbol)`. Director zařadí symbol na zásobník symbolů. V případě že se jedná o symbol, který vytváří sémantickou jednotku, pak ji vytvoří, přiřadí jí příslušné parametry a zařadí na zásobník sémantických jednotek.

2. Director přijme od generátoru volání metody `getSymbol(symbol)`, kterou generátor žádá o vážený seznam možných rozvojů pro symbol. Director zjistí tento seznam z gramatického pravidla, které získá od modulu 4.3.4. Poté analyzuje aktuální vrcholy obou vnitřních zásobníků, a na jejich základě může do seznamu přidat nové rozvoje nebo některé odebrat. Je také možné změnit podobu nebo váhu některých pravidel. Takto změněný seznam poté vrátí generátoru. Změny, které Director provádí, jsou pro každou sémantickou jednotku jiné, což je důvod, proč je Director jako modul závislý na jazyce, který se má generovat – algoritmus prováděných změn je popsán přímo programem, nikoliv vstupem programu nebo metadaty.

3. Director dále provádí tento proces pro všechny podsymboly aktuálně generovaného symbolu. Pro některé sémantické jednotky očekává konkrétní vygenerované podsymboly. Jejich hodnotami pak nastavuje vlastnosti takové sémantické jednotky.

4. Director přijme od generátoru volání metody `reportFinish(symbol, expanded)`, kterou generátor oznamuje ukončení generování jednoho symbolu, a také předává k posledním úpravám podobu vygenerovaného symbolu v parametru `expanded`. Director vyjme příslušný symbol ze zásobníku a v případě, že se jedná o symbol vytvářející sémantickou jednotku, i ze zásobníku sémantických jednotek. Případnou sémantickou jednotku zpracuje (může např. definovat nebo změnit vlastnost některé jiné, existující hlouběji v zásobníku). Výslednou podobu symbolu může poté ještě změnit a nakonec ji jako už zcela konečnou vrátí generátoru. I zde jsou zpracování a provedené změny specifické pro každou sémantickou jednotku. Jednotlivé prováděné akce jsou popsány výše u popisu samotných sémantických jednotek.

Implementované sémantické jednotky pro jazyk C

Implementace generátoru v této práci pokrývá poměrně omezenou množinu prostoru jazyka C. Pokryty byly základní konstrukce jazyka C, jako jsou přiřazovací příkaz, různá větvení a cykly, a výrazy zahrnující operace s proměnnými a funkcemi. Podporovány jsou aritmetické a relační výrazy. Nejsou podporovány všechny operátory a také operace se složenými (struct, union) nebo uživatelskými datovými typy. Také nejsou implementovány operace zahrnující pole a ukazatele. Náhodné generování takového kódu vnáší do chování programu velké množství entropie a složitost generátoru by v případě implementace všech těchto vlastností jazyka C vzrostla do míry, která je pro původní účel (prototypová implementace určená k experimentální aplikaci *fuzz testingu* na překladače) zbytečná.

Implementovány byly tedy sémantické jednotky uvedené v následujícím seznamu. V závorce za názvem jsou uvedeny symboly, které iniciují vznik této sémantické jednotky.

Program (`<program>`) Sémantická jednotka reprezentující celý program. Tvoří rámec pro definici proměnných i funkcí. Při ukončení generování se na začátek výsledku připojí seznam deklarací, které inicializují proměnné, které nebyly inicializovány v samotném programu.

Function (`<function_definition>`) Sémantická jednotka reprezentující definici funkce. Netvoří rámec pro definici proměnných, pouze parametrů. Pokud je na vrcholu zásobníku sémantických jednotek a nemá ještě definovanou příslušnou hodnotu, pak se první následující vygenerovaný datový typ přiřadí jako návratový typ, identifikátor jako název funkce, a uloží se vygenerované identifikátory parametrů s datovými typy. V okamžiku, kdyz se začne generovat blok funkce, jsou proměnné reprezentující parametry přidány do rámce proměnných v tomto bloku, takže je možné s nimi v těle funkce pracovat. Po skončení generování se funkce přidá do nejbližšího rámce funkcí v zásobníku sémantických jednotek, aby mohla být v tomto rámci také volána.

SimpleDeclaration (`<declaration>`) Reprezentuje jednoduchou deklaraci proměnné bez její inicializace. Director při jejím generování očekává vygenerované symboly typu a identifikátoru a přiřadí je do odpovídajících polí. Po ukončení se proměnná přidá do nejbližšího rámce pro proměnné a také do vnitřního seznamu neinicializovaných proměnných.

AssigningDeclaration (`<assgn_declaration>` : Reprezentuje deklaraci proměnné s její inicializací. Postup je stejný jako v případě sémantické jednotky SimpleDeclaration, jen se po vygenerování identifikátoru vygeneruje ještě operátor přiřazení a výraz, který má typ omezen na datové typy kompatibilní s datovým typem deklarované proměnné.

Block (`<block.` `<augmented_block>`, `<augmented_function_block>`) Reprezentuje blok, použitý jako tělo funkce, cyklu, nebo podmiňovacího příkazu. Tvoří rámec pro definice proměnných. Tři symboly definující bloky se od sebe liší použitím ukončovacích pojistek, což je zajištění výstupu a opuštění bloku. Zachytávají se následující symboly:

- `<block_epilogue>`, který zajistí vygenerování příkazů pro vypsání obsahu všech proměnných definovaných v rámci bloku na jeho konci. Tento symbol se vyskytuje pro všechny tři symboly tvořící blok.
- `<augmented_block_epilogue>` zajistí příkaz `break` pro opuštění bloku v případě, že tento už proběhl vícekrát, než je konfigurovatelná hodnota. To slouží k zabránění výskytu nekonečných smyček. Tento příkaz je vygenerován na konci bloku v cyklech.
- `<augmented_function_prologue>` zajistí vygenerování příkazu `return` pro opuštění bloku v případě, že tento už proběhl vícekrát, než je konfigurovatelná hodnota. To slouží k zabránění výskytu nekonečných smyček při rekurzivním volání funkce. Návratová hodnota v případě tohoto ukončení jen počet průběhů funkcí. Tento příkaz je vygenerován na začátku bloku funkce.

Main (`<main_function>`) Reprezentuje blok funkce `main()`. Vytváří rámec pro definici proměnných. V zásadě se jedná o stejnou sémantickou jednotku jako Block, ale na konci se kromě výpisu proměnných v rámci bloku vypíší také globální proměnné.

Assignment (`<assignment>`) Reprezentuje přiřazovací příkaz. Při generování identifikátoru Director zjistí seznam všech proměnných, viditelných v aktuálním rámci, a pro generátor vytvoří seznam rozvojů, které je obsahují jako jediný terminální symbol. V případě že neexistuje žádná proměnná, kterou by bylo možné na levé straně přiřazovacího příkazu použít, se přiřazovavací příkaz změní na deklaraci nové proměnné s inicializací. Výraz na pravé straně přiřazovacího příkazu má datový typ omezen na typ proměnné na levé straně.

Expression (`<expression>`) Nejsložitější sémantická jednotka, reprezentující výraz. Obsahuje omezení typu svých podsymbolů, někdy dané sémantikou předem (parametr funkce), někdy se určuje až po zjištění typu první vygenerované konstanty, funkce, nebo proměnné v něm (příkladem je operand v relačních výrazech). Při jeho zpracování Director provádí několik změn pro zajištění použití konstant, již vygenerovaných proměnných a funkcí, a také pro omezení rekurzivity, kdy se některé symboly mají sklony generovat donekonečna. Provádějí se následující akce:

- Generování konstant ve výrazu. V případě, že je typ nadřazené sémantické jednotky omezen, je vyloučené generování konstant, se kterými není typ výrazu kompatibilní. Rozsah číselných konstant není nijak ošetřován, protože návratovou hodnotu výrazu stejně nelze určit předem vzhledem k funkcím, u kterých není možné zajistit, že budou vracet hodnoty z platného rozsahu. Norma jazyka C pro celočíselné datové typy definuje pravidlo, že použitím hodnoty vyšší než rozsah datového typu se hodnota potichu určí jako hodnota modulo rozsah, takže

chování různých překladačů by mělo v těchto případech být na stejném stroji stejné [18]. Pro datové typy `float` a `double` jsou nelegální hodnoty interpretovány jako INF nebo NaN, a také by měly být stejné.

- Generování proměnných ve výrazu. Při tvorbě seznamu možných rozvojů jsou zjištěny všechny viditelné proměnné, které mají typ kompatibilní s typem výrazu. Pro každou proměnnou je poté vytvořen rozvoj s pravděpodobností generování rovnou jedné, který obsahuje indetifikátor proměnné jako terminální symbol.
- Generování volání funkcí ve výrazu. Při tvorbě seznamu možných rozvojů pro operandy jsou zjištěny všechny funkce s návratovým typem kompatibilním s typem výrazu. Pro každou funkci je poté vygenerován rozvoj obsahující identifikátor funkce jako terminál, a poté v závorkách uzavřený seznam umělých neterminálů. Každý takový neterminál má tvar `<param/function/index>`, např pro volání funkce `double D(int,double,char)` bude mít příslušný rozvoj podobu `D(<param/D/0>, <param/D/1>, <param/D/2>)`. Generátor neví, že tyto neterminály jsou použity pouze ve vnitřním mechanismu Directoru a normálně požádá o jejich rozvoje. Když o ně požádá, Director generátoru vrátí jediný rozvoj obsahující symbol `<expression>`. Typ sémantické jednotky, vytvořené pro tento výraz, bude nastaven na datový typ `index`-tého parametru funkce `function`. Tyto informace Director zjistí v sémantické jednotce, v jejímž rámci je `function` definována. Vzhledem k tomu, že je zbytečné volat jednu funkci vícekrát, je použití volání funkce omezeno. Jakmile je její volání použito vícekrát než je hodnota v konfiguraci, pak se už její další volání generovat nebudou.
- Omezení rekurzivity. Výraz je ze své podstaty rekurzivní – každým operandem může být zase další výraz. To může při špatné konfiguraci vést k neukončení procesu generování, nebo jeho selhání. Pro ošetření této situace si Director udržuje hloubku hierarchie aktuálně generovaných výrazů a od určité konfigurovatelné hloubky začne při generování operandů pravděpodobnostně preferovat hodnoty před dalšími výrazy. Se vzrůstající hloubkou výrazu také zvyšuje pravděpodobnost vygenerování literální konstanty před generováním proměnných nebo funkcí.

4.3.3 Příklad celého procesu

Director má na vrcholu zásobníku sémantickou jednotku *Block*, která obsahuje již definované proměnné `int A, float B`. V sémantické jednotce Program na dně zásobníku je také definována funkce `int C(int)`. Director přijme volání `reportStart('assgn_declaration')`. Tento symbol vytváří sémantickou jednotku *AssigningDeclaration*, Director ji vytvoří a vloží na zásobník. Na následující volání `getSymbol('assgn_declaration')` vrátí rozvoj `<type> <identifier> = <expression>`. Následuje generování typu, který nevytváří sémantickou jednotku a jako rozvoje jsou pro něj vráceny jednotlivé datové typy. Director registruje volání `reportFinish('type', 'int')`, zjistí že SJ na vrcholu zásobníku je AssigningDeclaration, a přiřadí jí vygenerovaný datový typ, tedy 'int'. Dále je vygenerován identifikátor, a Director opět registruje až volání `reportFinish('identifier', 'IDEN')`. Stejným způsobem jako v případě typu přiřadí SJ na vrcholu zásobníku identifikátor IDEN. Při volání metody `reportStart('expression')` zjistí, že aktuální sémantická jednotka je AssigningDeclaration s již určeným typem, kterým je 'int'. Vytvoří tedy SJ *Expression*, rovnou jí přiřadí typ 'int' a zařadí ji na zásobník. Při generování podsymbolů výrazu je vždy zjištěno, že nadřazená SJ má typ int a v nadřazených SJ, které vytvářejí rámce pro proměnné jsou

definovány proměnné A,B a funkce C. Director zjistí základní pravidlo pro rozvoj operandu, které vypadá takto:

```
<operand> ::= <integer_constant> | <floating_constant> | <character_constant>.
```

Z tohoto rozvoje vyřadí rozvoj se symbolem `floating_constant`, protože není kompatibilní s datovým typem int. K rozvoji dále přiřadí rozvoj s identifikátorem A, ale ne identifikátorem B, opět z důvodu nekompatibilního typu. Nakonec přiřadí ještě rozvoj obsahující volání funkce C. Výsledný seznam rozvojů vypadá takto:

```
<operand> ::= <integer_constant> | <floating_constant> | <character_constant>
| A | C(<param/A/0>.
```

Na případné volání metody `reportStart('param/A/0')` vytvoří director SJ *Expression* s datovým typem nultého parametru funkce A. Tyto informace jsou přítomné v sémantické jednotce Program na dně zásobníku. Dále se v tomto případě postupuje zase jako při generování výrazu.

Nakonec celého procesu je volána metoda `reportFinish('assgn_declaration', 'int IDEN = 168 + C(0/A)')`. Při volání této metody Director vyhledá v zásobníku nejbližší SJ, která vytváří rámec pro proměnné, a vloží do ní informaci o přítomnosti proměnné IDEN s datovým typem 'int'. Znamená to, že následující SJ, které budou v rámci stejného bloku generovány, mohou s proměnnou IDEN pracovat.

4.3.4 Pomocné moduly

Součástí programu Spitter jsou také tři pomocné moduly. Tyto moduly obsluhují specifické části programu, které přímo nesouvisejí s generováním vět. Modul Grammar tvoří abstraktní vrstvu, ve které jsou uchovávány informace o gramatice. Modul Logger se stará o výstup. Modul BNF Parser rozeznává v textovém souboru pravidla gramatiky definovaná v Backus-Naurově Formě.

Grammar

Modul Grammar představuje abstraktní vrstvu uchovávající podobu gramatiky generovaného jazyka. Obsahuje tyto třídy reprezentující různé elementy, použité pro popis gramatiky:

- **Element** je bázová třída pro všechny prvky gramatiky. Obsahuje pouze metody pro logování, které mohou využívat modul Logger.
- **Symbol** reprezentuje jeden symbol gramatiky. Jeho prvky jsou jméno a typ. Typ může být terminál nebo neterminál. Pokud je typ symbolu terminál, pak se nevyužívá vlastnost jméno.
- **OptSymbol** je potomkem třídy Symbol. Reprezentuje symbol, který má navíc volitelnou možnost generování, což je číslo s plovoucí desetinnou čárkou v intervalu $< 0, 1 >$.
- **Production** reprezentuje jeden možný rozvoj neterminálu. Obsahuje pouze seznam symbolů v pořadí, v jakém se mají vygenerovat.

- **Rule** reprezentuje pravidlo. Obsahuje jméno neterminálu, pro který je určeno, a také seznam možných rozvojů. Rozvoje mohou mít váhu, která určuje poměr pravděpodobností s jakou se vygenerují.
- **Grammar** reprezentuje celou gramatiku. Obsahuje seznam neterminálů, pro které existují pravidla. Obsahuje také seznam pravidel. Implementuje navíc metodu pro ověření korektnosti gramatiky – zda pro každý neterminál použitý na pravé straně pravidla existuje pravidlo pro jeho generování.

Logger

Modul Logger obsluhuje výstup a logování programu. Podstatnou vlastností je konfigurovatelnost směrování obou výstupů. Vzhledem k povaze logování činnosti generátoru umožňuje odsazování výstupu podle aktuální hloubky v derivačním stromu a také volbu úrovně logování.

BNF Parser

Modul BNF Parser obsluhuje čtení a rozeznávání pravidel gramatiky z textového vstupu a předává je modulu Grammar. Jedná se o velmi jednoduchý modul, využívající lexikální a syntaktický Ply [7].

4.4 Builder

Builder je část systému, která se stará o vytvoření všech výstupů, které se budou později Comparatorem analyzovat. Jedná se o velmi jednoduchý skript, který jako parametr bere cestu k souboru a seznam značek konfigurací překladačů, pro které se testovací kód zkompiluje. Ze souboru s nastavením potom načítá konfigurace pro spouštění jednotlivých překladačů. Každá konfigurace musí mít unikátní značku, podle které je možné ji identifikovat. Konfigurace dále obsahuje značky, podle kterých se určují pozice zdrojového souboru a výstupu v příkazu, který volá překladač. Přeložený program je Builderem spuštěn a jeho výstup zaznamenán. Pokud se program sám neukončí do nastaveného počtu vteřin, je mu odeslán signál SIGTERM. Výstupem builderu jsou soubory:

- `TAG.build` obsahuje zprávy, které překladač během zpracovávání zdrojového kódu vypsal na standardní nebo chybový výstup.
- `TAG.buildresult` obsahuje výsledek překladu, tj. informaci, zda byl překlad úspěšný.
- `TAG` je vlastní spustitelný soubor.
- `TAG.termination` obsahuje informaci, zda se přeložený program ukončil standardně, nebo zda musel být ukončen signálem.
- `TAG.output` obsahuje výstup přeloženého programu po spuštění.

4.5 Comparator

Comparator je jednoduchý modul, jehož úkolem je porovnat výstupy, které vytvořil modul Builder, a na jejich základě rozhodnout o tom, zda bude výsledek testu to že testované

překladače uspěly, nebo některý selhal. Selhání dělí na několik druhů, které se později uchovávají a reportují odděleně. Tato selhání jsou označena jako anomálie. Existují tři druhy anomálií: anomálie překladu (rozdílná úspěšnost překladu), anomálie ukončení (rozdílný způsob ukončení) a konečně anomálie výstupu (spuštěné programy mají odlišný výstup).

Comparator nejprve porovná výsledky překladů. Pokud jsou odlišné, pak se výsledek testu náhlásí jako anomálie překladu. Případ kdy oba překlady selžou je možné nastavit jako anomálii i jako platný výsledek. Pokud používáme generátor, o kterém víme, že produkuje vždy jen naprosto validní kód, pak je jakékoliv selhání anomálií a všechny výsledky se zachovají k analýze člověkem. Pro generátor produkující neplatný kód je odmítnutí překladu správně fungujícím překladačem očekávané a proto se za anomálii nepovažuje.

V případě že nebyla objevena anomálie ve výsledku překladu, porovná se způsob ukončení programu. Pokud se liší, pak jsou prohlášeny za anomálii ukončení. Stejně jako u výsledků překladu, i zde je možné nastavit, jakým způsobem se bude nakládat se stejnými, ale negativními výsledky (pro tento případ je to násilné ukončení). I v tomto případě, pokud máme jistotu, že generátor tvoří programy, které by se měly ukončit samy, je násilné ukončení přeloženého programu anomálií.

Pokud není objevena anomálie ve dvou předchozích krocích, pak se porovnávají výstupy spuštěných programů. Zde se za anomálii považují už jen rozdílné výstupy. Oba výstupy jsou volitelně prověřeny na přítomnost znaků falešného selhání, kdy systém nahlásí anomálii, která ale může být způsobena buď již známou chybou, nebo jiným známým nedeterministickým chováním generovaných programů. Takovým typickým znakem je ukončení programu signálem SIGFPE[3], který nastává v případě dělení nulou [15]. Toto dělení nulou se ale může nacházet ve výrazu, který překladač odoptimalizuje a potom se může stát, že jeden překlad selže se signálem SIGFPE, ale druhý doběhne v pořádku. Ověřování na falešné selhání je možno konfigurovat.

Výstupem Comparatoru je návratový kód, který určuje, zda byla nalezena nějaká anomálie, a její typ v případě že ano.

4.6 Pokrytí pro překladače specifických chyb navrženým systémem

V případě optimální implementace celého systému je možné předpokládat, že dosáhneme takového pokrytí typů chyb v překladačích, jaké uvádí tabulka 4.1. Z tabulky je vidět, že kvalitní pokrytí přináší už první fáze porovnávání, ihned při překladu. Pro dobré pokrytí není tedy potřeba generátor deterministického kódu, stačí platný kód. Deterministický výstup je ale třeba pro detekci chyb typu *wrong-code*, na něž je tato práce zaměřena. Pro pokrytí některých chyb, které jsou označeny jako “Možné” by byly třeba modifikace jednotlivých částí.

[3]signál Floating Point Exception, indikující obecnou chybu při operaci s číslem s plovoucí desetinnou čárkou

Tabulka 4.1: Pokrytí typů chyb v překladačích navrženým systémem

Typ chyby	**Pokrytí**	**Typ anomálie**
accepts-invalid	Možné (s generátorem neplatného kódu)	Anomálie překladu
assemble-failure	Ano	Anomálie překladu
compile-time-hog	Možné	Anomálie překladu
diagnostic	Ne	Žádná
ice-on-invalid	Možné (s generátorem neplatného kódu)	Anomálie překladu
ice-on-valid	Ano	Anomálie překladu
link-failure	Ano	Anomálie překladu
memory-hog	Ne	Žádná
missed-optimization	Ne	Žádná
register-allocator	Ne	Žádná
rejects-valid	Ano	Anomálie překladu
wrong-code	Ano	Anomálie výstupu/ukončení

Kapitola 5

Aplikace na reálné případy

Celý systém byl podroben čtyřem experimentům, porovnávajícím různé konfigurace různých překladačů. Tato kapitola popisuje jejich nastavení, průběh a výsledky.

5.1 Metodika

Systém byl testován na čtyřech různých případech, které pokrývají spektrum možných použití systému. Těmito případy jsou:

- produkční překladač v různých konfiguracích,
- produkční překladač proti překladači, který ještě nedosáhl verze 1,
- open-source produkční překladače proti produkčnímu proprietárnímu,
- produkční překladač ve stabilní verzi proti stejnému překladači ve vývojové verzi.

Původně bylo úmyslu testovat více open-source neprodukčních překladačů proti produkčnímu, ale nepodařilo se zprovoznit jiné open-source překladače než GCC a TCC.

Pro každou variantu byl systém testován se třemi různými generátory, použitými pro tvorbu testovacího vstupu. Tento přístup byl zvolen kvůli objektivitě. Pokud by mnou implementovaný generátor (Spitter popsaný v sekci 4.3) neměl vlastnosti potřebné pro úspěšné vyhledávání chyb, pak je třeba zjistit chování systému s jiným generátorem. Pokud by ani tento generátor neměl požadované výsledky, pak by bylo možné tuto metodu prohlásit za neefektivní a vyvodit z toho závěry.

Použité generátory jsou tyto:

- **Spitter** je mnou implementovaný generátor. Dokáže generovat omezenou podmnožinu jazyka C. Programy jsou validní a ukončují se. Implementačním jazykem je Python.
- **Randprog** je generátor náhodných programů jazyka C, implementovaný Bryanem Turnerem [19]. Dokáže generovat omezenou podmnožinu jazyka C. Programy jsou validní, a ačkoliv program obsahuje kód který má zabránit nekonečným cyklům, programy musejí být občas ukončeny signálem. Implementačním jazykem je C++.
- **fuzz** představuje čistě náhodný vstup o délce 1000 byte, realizovaný příkazem `dd if=/dev/urandom`. Tento generátor byl zvolen z důvodu záměru porovnání čistého *fuzz testingu* s variantou navrženou v této práci.

Vzhledem k relativní rychlosti jednoho testu byl pro jednu variantu zvolen počet 5000 testů. V případě, že během testování jedné varianty byly testovány různé konfigurace překladačů (např. optimalizací), pak byl pro každou konfiguraci proveden poměrný počet testů (např. pro dvě konfigurace bylo s každou provedeno 2500 testů). Pro odlišné konfigurace byly generovány odlišné testovací případy. Po uběhnutí všech testů byly analyzovány výsledky, a objevené anomálie byly klasifikovány jako falešné nebo pravé.

5.2 Testované překladače

Testovány byly překladače uvedené v následujícím seznamu. Další podmínky v testovacím prostředí se v jednotlivých variantách testování lišily, a jsou uvedeny u každého případu zvlášť.

- GCC (GNU Compiler Collection)[2] je překladačový systém vyvíjený jako součást projektu GNU. Jedná se o nejrozšířenější překladač aplikací v open-source komunitě používající systém GNU/Linux[4]. Obsahuje front-endy pro jazyky C, C++, ObjC, Fortran, Java a Ada. Je dostupný pod licencí GPLv3.

- TCC (Tiny C Compiler)[5] je překladač jazyka C, zaměřený na kompaktnost a rychlost. Obsahuje vlastní assembler a linker, a implementuje dvě zajímavé vlastnosti: schopnost interpretovat zdrojový kód jazyka C jako skript, a zabudované testy na některé programátoské sémantické chyby (*bound checker*). TCC zatím nedosáhl stabilní verze. Je dostupný pod licencí BSD.

- ICC (Intel C Compiler)[3] je překladač jazyků C, C++ a Fortran pro platformy firmy Intel, kterou je také vyvíjen. Výhodou jsou vynikající optimalizace na nativních platformách. Jedná se o komerční, proprietární překladač dostupný zdarma pro nekomerční využití. Akademické využití není považováno za nekomerční, bylo tedy využito instalace tohoto překladače na školních serverech s patřičnou licencí.

5.3 Klasifikace výsledků

- Anomálie překladu znamená, že výsledek překladu byl u obou překladačů odlišný.

- Anomálie ukončení znamená, že jeden program byl ukončen signálem, zatímco druhý korektně doběhl.

- Anomálie výstupu označuje případ, kdy se výstupy spuštěných programů liší.

- Selhání překladu označuje případ, kdy při překladu selžou oba překladače.

- Ignorované výsledky byly takové, kdy alespoň jeden program byl ukončen signálem SIGFPE.

Anomálie byly následně ručně analyzovány a klasifikovány jako pravé chyby, nebo falešná hlášení. Častý byl případ, kdy byla jedna chyba nahlášena několikrát. V takovém případě je započítána pouze jednou, stejně jako několik falešných hlášení, které byly způsobeny stejnou chybou v testovacím systému.

5.4 Případ 1: gcc s různými optimalizacemi

V této části byl systém aplikován na open-source produkční překladač s vypnutými optimalizacemi jako referenční a stejný překladač s různými úrovněmi optimalizací.

5.4.1 Prostředí a podmínky

Testovaný překladač

Testovaný překladač byl GNU Compiler Collection ve verzi 4.1.2-33, dodávaný v RPM balíčku distribuce Fedora gcc-4.1.2-33.fc8.rpm. Bylo provedeno 1500 experimentů pro tři různé úrovně optimalizací – optimalizace výkonu `-O2`, silnější optimalizace výkonu `-O3` a optimalizace velikosti výsledného kódu `-Os`.

Referenční překladač

Referenčním překladačem byl v tomto experimentu rovněž GNU Compiler Collection ve verzi 4.1.2-33, dodávaný v RPM balíčku distribuce Fedora gcc-4.1.2-33.fc8.rpm, s explicitně deaktivovanými optimalizacemi (volba `-O0`).

Prostředí

Testování probíhalo v následujícím prostředí.

- **Procesor**: AMD Athlon(tm) 64 Processor 3800+
- **Operační systém**: Fedora 8
- **Jádro**: Linux 2.6.24.4-64.fc8 #1 SMP i686 athlon i386 GNU/Linux
- **Linker**: GNU ld verze 2.17.50.0.18-1 20070731
- **Knihovna C**: glibc-2.7-2

5.4.2 Spitter

Výsledky jsou uvedeny v tabulce 5.1. Během tohoto testování nebyly nalezeny žádné chyby v testovaném překladači. Selhání překladu byla způsobena nedokonalým generováním Spitteru, kdy je jeden identifikátor, nejčastěji jednoznakový, použit v programu vícekrát.

5.4.3 Randprog

Výsledky jsou uvedeny v tabulce 5.2. Vysoký počet anomálií ukončení je dán podobou programů generovaných programem Randprog. Programy jím generované mají občas poměrně dlouhou dobu trvání, a proto byla často anomálie vyvolaná tím, že po uplynutí limitu 5 sekund byl takový program uměle zabit. Byla nalezena jedna chyba (demonstrovaná v dodatcích A.1 a A.2) , kdy byl v optimalizovaném kódu špatně vypočítaný výsledek bitové operace.

5.4.4 fuzz

Výsledky uvádí tabulka 5.3. Test dopadl zcela dle očekávání, všechny náhodné vstupy byly překladačem korektně odmítnuty. Nebylo zjištěno ani jedno zhroucení překladače.

Tabulka 5.1: GCC -O0 vs. GCC -O2, -O3, -Os, Spitter

Testy	`-O2`	`-O3`	`-Os`
Celkem	1500	1500	1500
OK	1456	1457	1460
Anomálie překladu	0	0	0
Anomálie ukončení	0	0	0
Anomálie výstupu	0	0	0
Selhání překladu	15	3	11
Ignorované výsledky	29	40	29
Počet skutečných chyb	0	0	0
Počet falešných chyb	0	0	0

Tabulka 5.2: GCC -O0 vs. GCC -O2, -O3, -Os, Randprog

Testy	`-O2`	`-O3`	`-Os`
Celkem	1500	1500	1500
OK	1345	1335	1349
Anomálie překladu	0	0	0
Anomálie ukončení	155	162	151
Anomálie výstupu	0	3	0
Selhání překladu	0	0	0
Ignorované výsledky	0	0	0
Počet skutečných chyb	0	2	0
Počet falešných chyb	0	0	0

Tabulka 5.3: GCC -O0 vs. GCC -O2, -O3, -Os, fuzz

Testy	`-O2`	`-O3`	`-Os`
Celkem	1500	1500	1500
OK	0	0	0
Selhání překladu	1500	1500	1500
Počet skutečných chyb	0	0	0
Počet falešných chyb	0	0	0

5.4.5 Zhodnocení experimentu

Naproti očekávání nebyly nalezeny téměř žádné chyby. Důvodem může být fakt, že překladač GCC verze 4.1.2 je užíván již poměrně dlouhou dobu a zjevné chyby byly již objeveny. Je také možné, že při větším počtu testovacích případů by k nalezení chyby mohlo dojít. Zajímavý bude tento experiment aplikovaný na vývojovou verzi překladače GCC.

5.5 Případ 2: GCC, TCC

V této části byl systém aplikován na open-source produkční překladač jako referenční a open-source nedospělý překladač jako testovaný.

5.5.1 Prostředí a podmínky

Testovaný překladač

Testovaný překladač byl Tiny C Compiler ve verzi 0.9.24, přeložený a sestavený ze zdrojových kódů dostupných na webových stránkách projektu, přeložený standardním GNU vývojovým řetězcem (překladač GCC 4.1.2-33, linker ld 2.17.50.0.18-1, knihovna C glibc glibc-2.7-2).

Referenční překladač

Referenčním překladačem byl v tomto experimentu GNU Compiler Collection ve verzi 4.1.2-33, dodávaný v RPM balíčku distribuce Fedora gcc-4.1.2-33.fc8.rpm.

Prostředí

- **Procesor**: AMD Athlon(tm) 64 Processor 3800+
- **Operační systém**: Fedora 8
- **Jádro**: Linux 2.6.24.4-64.fc8 #1 SMP i686 athlon i386 GNU/Linux
- **Linker**: GNU ld verze 2.17.50.0.18-1 20070731
- **Knihovna C**: glibc-2.7-2

5.5.2 Spitter

V tomto případě bylo nalezeno velké množství anomálií. Výsledky jsou uvedeny v tabulce 5.4. Všechny anomálie překladu jsou způsobeny různými variantami chyby uvedené v příloze A.5, kdy překladač TCC neodmítne kód, kde jsou předefinovány symboly na jiný typ, popřípadě jdou definovány jako jiný typ symbolu. Velké množství anomálií výstupu je způsobeno variantami chyby A.9 a dalšími. Objevily se také některé falešné chyby, zejména ve spojitosti s přesností operací velmi vysokých čísel s plovoucí řádovou čárkou, kdy se ztrácí přesnost tohoto datového typu a překladače se s tím vyrovnávají různým způsobem.

5.5.3 Randprog

Výsledky jsou uvedeny v tabulce 5.4. Příčiny anomálií nebyly do detailu vyšetřovány, vzhledem k velikosti případů generovaných programem Randprog.

Tabulka 5.4: GCC vs. TCC

Testy	Spitter	Randprog	fuzz
Celkem	5000	5000	5000
OK	4083	4427	0
Anomálie překladu	14	0	0
Anomálie ukončení	0	514	0
Anomálie výstupu	680	59	0
Selhání překladu	7	0	5000
Ignorované výsledky	216	0	0
Počet skutečných chyb	7	0	0
Počet falešných chyb	2	0	0

5.5.4 fuzz

Výsledky jsou uvedeny v tabulce 5.4. Čisté fuzz testování nenalezlo v obou překladačích žádnou chybu.

5.6 Zhodnocení experimentu

V této kombinaci bylo testování nejúspěšnější. V poměru k relativně lehkému testování bylo nalezeno sedm různých chyb, což potvrzuje vhodnost metody pro testování nedospělého překladače. Zároveň se ukázala výrazná nevhodnost programů generovaných programem Randprog pro testování. I v případě, že byla nalezena anomálie, vyšetřování příčiny v generovaném kódu by bylo velmi obtížné a namáhavé, protože Randprog generuje velmi obsáhlé programy s extrémně složitým průběhem kódem.

5.7 Případ 3: icc, gcc

Tato část se zabývá experimentem, kdy byl systém testován na kombinaci produkčního open-source překladače a produkčního proprietárního překladače. Jako referenční překladač byl zvolen open-source překladač GCC, jako testovaný překladač proprietární. Testy proběhly ve dvou variantách nastavení optimalizací testovaného překladače.

5.7.1 Prostředí a podmínky

Testovaný překladač

Testovaný překladač byl Intel C++ Compiler ve verzi 10.0, nainstalovaný na počítači `merlin.fit.vutbr.cz`. Bylo provedeno 2500 experimentů pro dvě různé úrovně optimalizací – optimalizace výkonu `-O2` a explicitně deaktivované optimalizace `-O0`.

Referenční překladač

Referenčním překladačem byl v tomto experimentu GNU Compiler Collection ve verzi 4.2.3, nainstalovaný na počítači `merlin.fit.vutbr.cz`. Optimalizace byly explicitně deaktivovány (volba `-O0`).

Prostředí

- **Procesor**: Dual-Core AMD Opteron(tm) Processor 2216
- **Operační systém**: CentOS verze 5
- **Jádro**: Linux 2.6.16.60 #2 SMP x86_64 x86_64 x86_64 GNU/Linux
- **Linker**: GNU ld verze 2.18
- **Knihovna C**: glibc-2.5-18

5.7.2 Spitter

Výsledky uvádí tabulka 5.5. Tento experiment ukázal mnoho chyb a nedostatečných zajištění implementovaného generátoru, což vyústilo ve velký počet různých falešných hlášení. Důvodem byl fakt, že ICC je překladač, jehož funkcionalita je výrazněji odlišná od GCC, i když ještě stále v rámci normy. ICC dokázal na rozdíl od GCC odhalit příliš velkou konstantu už v době překladu (jediná anomálie překladu). Byly odhaleny dva druhy falešných chyb. Jedním důvodem byla schopnost překladače ICC odhalit nedostatečnou přesnost proměnné s hodnotou s plovoucí řádovou čárkou, pokud byly zapnuté optimalizace. V tomto případě překladač změnil datový typ `float` na datový typ `double`, což se odrazilo na výstupu odlišném od GCC, který tuto optimalizaci nezvládá. Za chybu to ale nelze považovat v ani jednom případě.

Druhým falešným hlášením byl odlišný obsah proměnné typu `long long` (nebo jiného celočíselného typu nižší velikosti) v případě, že do ní byla přiřazena hodnota typu s plovoucí řádovou čárkou, která byla větší než maximální hodnota typu `long long`. Příklad viz. pseudokód 5.7.2. Program přeložený GCC do takové proměnné přiřadil hodnotu `LLONG_MAX`. Program přeložený ICC do ní přiřadil hodnotu o jedna vyšší, což způsobí přetečení na nejnižší hodnotu v rozsahu příslušného datového typu. Autorovi se nepodařilo v normě jazyka C nalézt definici, které by toto chování odporovalo, takže nemohla být prohlášena za chybu.

5.7.3 Randprog

Výsledky uvádí tabulka 5.6. V tomto experimentu se prokázala relativní nevhodnost generátoru Randprog pro účely fuzz testování různých překladačů. Vysoké číslo anomálií ukončení znamená fakt, že mnoho testů nebylo ukončeno do stanoveného času. Testovací případy, které probíhají dlouhou dobu, nejsou pro fuzz testování vhodné, protože se tím ztrácí výrazná výhoda – možnost zpracovat velké množství testů během krátké doby. Zároveň v tomto případě nebyly analyzovány chyby ve výstupu. Randprog občas generuje i skutečně velmi rozsáhlé programy, v řádu desetitisíců řádků. Zároveň jediný jeho výstup (a jeho výpočet) je proveden až zcela na konci programu, a je tak velmi obtížné nalézt místo, ve kterém se funkcionalita liší od referenčního překladu. Vzhledem k náchylnosti kombinace GCC/ICC na rozdílnost výstupu popsaného v sekci 5.7.2 je také velmi pravděpodobné, že tyto anomálie jsou způsobeny právě takovým rozdílným výsledkem přiřazení.

5.7.4 fuzz

Výsledky uvádí tabulka 5.7. Ani v této kombinaci nebyla čistým fuzz testováním nalezena žádná chyba. Všechny neplatné vstupy byly korektně odmítnuty oběma překladači.

Tabulka 5.5: GCC vs. ICC, Spitter

Testy	`-O0`	`-O2`
Celkem	2500	2500
OK	2455	2411
Anomálie překladu	1	0
Anomálie ukončení	0	0
Anomálie výstupu	13	28
Selhání překladu	18	13
Ignorované výsledky	13	48
Počet skutečných chyb	0	0
Počet falešných chyb	1	1

Tabulka 5.6: GCC vs. ICC, Randprog

Testy	`-O0`	`-O2`
Celkem	2500	2500
OK	2203	2249
Anomálie překladu	0	0
Anomálie ukončení	277	232
Anomálie výstupu	20	19
Selhání překladu	0	0
Ignorované výsledky	0	0
Počet skutečných chyb	0	0
Počet falešných chyb	0	0

Tabulka 5.7: GCC vs. ICC, fuzz

Testy	`-O0`	`-O2`
Celkem	2500	2500
OK	0	0
Selhání překladu	2500	2500
Ignorované výsledky	0	0
Počet skutečných chyb	0	0
Počet falešných chyb	0	0

```
$ cat testcase.c
#include <stdio.h>
int main(){
  unsigned long long a = 0x1.P+150000;  //> ULLONG_MAX
  signed long long b = 0x1.P+150000;    //> LLONG_MAX
  printf("Unsigned: %llu\n", a);
  printf("Signed:   %lli\n", b);
}

$ gcc testcase.c; ./a.out
Unsigned: 18446744073709551615          //ULLONG_MAX
Signed:   9223372036854775807           //LLONG_MAX

$ icc testcase.c; ./a.out
Unsigned: 0                             //ULLONG_MIN
Signed:   -9223372036854775808          //LLONG_MIN
```

Obrázek 5.1: Příklad chování konverze typu pro vysoké hodnoty

5.7.5 Zhodnocení experimentu

Výstupy tohoto experimentu byly velmi složité na vyhodnocení. Objevilo se mnoho anomálií, které byly způsobeny rozdílným chováním popsaným výše, a mezi jejich velkým množstvím bylo obtížné a pracné hledat jakékoliv jiné vadné chování, což vyústilo ve fakt, že nebyla nalezena jediná opravdová chyba. Zároveň také teto experiment ukázal na nepohodlnost práce s anomáliemi v případě, že jich je větší množství, způsobené jednou příčinou.

5.8 Případ 4: gcc-stable, gcc-bleeding edge

Poslední experiment se zabýval testováním poslední vývojové verze překladače GCC, jehož zdrojový kód byl stažen přímo ze systému správy verzí. Tato kombinace byla zvolena pro otestování možnosti použít systém při regresním testování překladače.

5.8.1 Prostředí a podmínky

Testovaný překladač

Testovaný překladač byl GCC ve verzi 4.4.0, jehož zdrojový kód byl získán z větve HEAD systému správy verzí dne 28.4.2008. Přeložen byl provedením plného 3-fázového bootstrapu [2], což dokazuje, že překladač je alespoň tak kvalitní, aby přeložil sám sebe.

Referenční překladač

Referenčním překladačem byl v tomto experimentu GNU Compiler Collection ve verzi 4.1.2-33, dodávaný v RPM balíčku distribuce Fedora gcc-4.1.2-33.fc8.rpm.

Prostředí

- **Procesor**: AMD Athlon(tm) 64 Processor 3800+
- **Operační systém**: Fedora 8
- **Jádro**: Linux 2.6.24.4-64.fc8 #1 SMP i686 athlon i386 GNU/Linux
- **Linker**: GNU ld verze 2.17.50.0.18-1 20070731
- **Knihovna C**: glibc-2.7-2

5.8.2 Spitter

Výsledky uvádí tabulka 5.8. Toto testování objevilo jednu chybu typu *ICE-on-valid*, která byla přítomná v obou překladačích a projevovala se jen s vypnutými optimalizacemi. Podrobnosti jsou uvedeny v dodatku A.3. Tato chyba byla nalezena při zběžném procházení logu selhaných překladů a přinesla nápad na možné rozšíření systému pro zachytávání a hlášení těchto případů jako anomálie. Falešné hlášení má svuj původ ve vlastnosti jazyka C, kdy není určeno v jakém pořadí jsou vyhodnocovány jednotlivé části výrazu. Obě anomálie obsahovaly výraz, ve kterém je jedna funkce volána dvakrát s různými argumenty a výsledkem je rozdílný výstup v případě rozdílného pořadí volání funkce.

5.8.3 Randprog

Výsledky uvádí tabulka 5.9. Objevena byla jedna chyba typu *ICE-on-invalid* (popsána v dodatku A.4) v případě, že byly vypnuté optimalizace. Anomálie výstupu nebyly analyzovány vzhledem k velikosti případů, které selhaly. Ve všech případech se jednalo o náhodné programy s více než 4000 řádky a více než dvaceti funkcemi.

5.8.4 fuzz

Nebyla nalezena žádná chyba. Výsledky uvádí tabulka 5.10.

5.8.5 Zhodnocení experimentu

Tento experiment ukázal na nedokonalost systému v případě, že překlad nestandardně (ICE) selže u obou překladačů. Takový případ je zařazen jako Selhání překladu a může tak být přehlédnutý, pokud jsou analyzovány jen anomálie. Toto by mělo být v budoucnosti ošetřeno lépe. Nalezeny byly dvě chyby, což je nižší výsledek, než se očekávalo u HEAD větve projektu.

Tabulka 5.8: GCC 4.1.2 vs. GCC 4.4.0, Spitter

Testy	`-O0`	`-O2`
Celkem	2500	2500
OK	2476	2370
Anomálie překladu	0	0
Anomálie ukončení	0	0
Anomálie výstupu	2	0
Selhání překladu	13	13
Ignorované výsledky	9	117
Počet skutečných chyb	1	0
Počet falešných chyb	1	0

Tabulka 5.9: GCC 4.1.2 vs. GCC 4.4.0, Randprog

Testy	`-O0`	`-O2`
Celkem	2500	2500
OK	2249	2234
Anomálie překladu	1	0
Anomálie ukončení	250	261
Anomálie výstupu	0	5
Selhání překladu	0	0
Ignorované výsledky	0	0
Počet skutečných chyb	1	0
Počet falešných chyb	0	0

Tabulka 5.10: GCC 4.1.2 vs. GCC 4.4.0, fuzz

Testy	`-O0`	`-O2`
Celkem	2500	2500
OK	0	0
Selhání překladu	2500	2500
Ignorované výsledky	0	0
Počet skutečných chyb	0	0
Počet falešných chyb	0	0

Kapitola 6

Závěr

V závěru jsou zhodnoceny dosažené výsledky a také nastíněn další možný vývoj projektu.

6.1 Zhodnocení výsledků

V této práci byla popsána problematika chyb v překladačich, se zaměřením na typické problemy s nimi spojené. Byl navržen přístup k testování překladačů pomocí automatizace metody *fuzz testing*. Podle tohoto přístupu byl implementován prototyp systému pro testování překladačů, se kterým byly provedeny experimenty na třech rozšířených překladačích v různých verzích. Ve dvou ze tří takto testovaných překladačů byly během experimentů nalezeny chyby. Následuje zhodnocení jednotlivých částí projektu.

6.1.1 Generátory náhodného kódu

Byly prostudovány možnosti sestrojení generátoru náhodných vět platného kódu v jazyce C. Jeden generátor byl implementován autorem této práce, a pro lepší zhodnocení byly provedeny experimenty s generátorem kódu jiného autora.

Spitter

Generátor implementovaný jako součást této práce. Podařilo se dosáhnout relativně slušné úrovně použitelnosti generovaných vět pro testování. Generované věty byly rychle přeložitelné a také s nízkým trváním běhu. Avšak byly také zjištěny chyby a problémy, které s sebou nese zvolené řešení. V průběhu experimentů se vyskytovaly dva větší problémy, které byly zdrojem falešných chybových hlášení:

- Nedefinované pořadí vyhodnocování podvýrazů v jednom výrazu. V případě, že byla např. jedna funkce volána v jednom výrazu vícekrát s různými parametry, pak se výstup programu lišil v případě, že překladače zvolily odlišné pořadí vyhodnocení výrazů.
- Příliš vysoké hodnoty konstant a výsledků výrazů. Zejména při operacích s velmi vysokými čísly typu `float` či `double`, které ztrácejí přesnost, se vyskytovaly problémy s odlišným výstupem.

Generátor také dokázal produkovat jen velmi omezenou podmnožinu jazyka C.

Randprog

Tento generátor volí jiný přístup k výpisu výstupu generovaných programů. Výstup je pouze jeden, a jsou do něj promítnuty všechny operace provedené při běhu programu. Tento přístup s sebou nese značné obtíže při zjišťování místa, kde došlo k chybě. Výrazné jsou zvláště v kombinaci s druhou vlastností generátoru, kterou je často velmi dlouhý vygenerovaný program – často v řádu tisíců až desetitisíců řádků, obsahujících mnoho dlouhých cyklů a rekurzivních volání. Programy mají velmi dlouhou dobu běhu, což neguje zásadní výhodu testovací metody – možnost testovat velké množství případů za krátký čas. Výhodou Randprogu bylo, že neprodukoval neplatný kód. Fakt, že jím vytvořené programy se velmi špatně analyzují, ale zabránil nalezení možných chyb v programech, což velmi snižuje možnost jeho použití pro tuto metodu testování.

6.1.2 Systém pro testování

Celý systém fungoval poměrně spolehlivě. S jeho pomocí bylo nalezeno několik chyb v překladačích. Výhodou bylo snadné použití různých generátorů testovacích případů. Poněkud nepohodlná je práce s nalezenými anomáliemi, kdy je třeba ručně analyzovat vytvořený zdrojový kód a hledat místo, kde se chyba vytváří. Je to také pracné a zdlouhavé v případě většího množství anomálií, které jsou následkem jediné chyby. Systém by také mohl lépe zvládat situace, kdy dojde k ukončení programu násilně z důvodu časového limitu, nebo doručením některého ze signálů způsobujících ukončení programu.

6.2 Zhodnocení použitelnosti metody

V této práci byla zkoumána použitelnost metody *fuzz testing* k testování překladačů. Bylo dosaženo pozitivních výsledků, ale negativem zůstává obtížná analýza samotného selhávajícího testovacího případu. Ve srovnání se statickými metodami také tato dynamická metoda produkuje velké množství falešných upozornění. Nepříjemný je také fakt, že jedna chyba může způsobit velké množství selhání, a v takovém množství se obtížně hledají jiné chyby. V případě, že by tyto problémy byly odstraněny, pak je dle názoru autora tuto metodu možné použít k úspěšnému a snazšímu testování překladačů.

6.3 Další vývoj

Pro systém byl založen projekt, umístěný na http://code.googlecode.com/p/fucc/, kde bude pokračovat jeho vývoj. Z této práce vyplývá poměrně jednoznačný postup pro další vývoj projektu. Je potřeba se zaměřit zejména na generátor, kde je nutné opravit chyby, které způsobují generování neplatného nebo nedeterministického kódu. Bylo by také dobré zvětšit podmnožinu jazyka C, kterou Spitter dokáže generovat. Dále je nutné refaktorovat nebo rozdělit třídu Director generátoru, je příliš rozsáhlá a nepřehledná. V neposlední řadě je také třeba poskytnout programu možnost konfigurací a větší robustnost. Velkým přínosem by také bylo zhotovení jednoduchého uživatelského rozhraní, které by zjednodušilo použití systému. Pro analýzu by byl velmi přínosný nástroj, který by dokázal z předloženého zdrojového kódu odstranit všechno, co neovlivní výstup programu.

V dlouhodobém horizontu by mohlo být zajímavé zaměření na Director, který byl méně vázán na konkrétní jazyk.

Dodatek A

Nalezené chyby

V této části jsou uvedeny chyby v překladačích, nalezené ve fázi testování vytvořeného systému. Jedná se o ručně vyextrahované testovací případy, které jsou upraveny tak, aby obsahovaly co nejmenší množství kódu potřebného pro demonstraci defektu.

A.1 Chyba 1

Umístění chyby GCC 4.1.2-33 se zapnutými optimalizacemi

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
 $ cat tc1052.c
#include <limits.h>
int main(){
  long var = 0x19DF1618;
  printf("var:              %i\n", var);
  printf("var*var:          %i\n", var*var);
  printf("(var*var) << var: %i\n", (var*var) << var);
  printf("LONG_MAX:         %i\n", LONG_MAX);
  return 0;
}
$ gcc -O3 tc1052.c ; ./a.out
var:              434050584
var*var:          800596544
(var*var) << var: 0
LONG_MAX:         2147483647
```

Předpokládané chování

```
$ gcc -O3 tc1052.c ; ./a.out
var:              434050584
var*var:          800596544
(var*var) << var: 1073741824
LONG_MAX:         2147483647
```

A.2 Chyba 2

Umístění chyby GCC 4.1.2-33 se zapnutými optimalizacemi

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
$ cat tc661.c
int main (int argc, void *argv[])
{
      unsigned long l_40452115 = 0x027F1A57;
      printf ("l_40452115:                %i\n", l_40452115);
      printf ("l_40452115 >> l_40452115:  %i\n", l_40452115 >> l_40452115);
  return 0;
}

$ gcc -O3 tc661.c ; ./a.out
l_40452115:                41884247
l_40452115 >> l_40452115:  0
```

Předpokládané chování

```
$ gcc -O3 tc661.c ; ./a.out
l_40452115:                41884247
l_40452115 >> l_40452115:  4
```

A.3 Chyba 3

Umístění chyby GCC 4.1.2-33, a GCC 4.4.0, s vypnutými optimalizacemi

Typ chyby ICE-on-valid

Zdrojový kód a reprodukce

```
$ cat tc250.c
int main(){
  float c2 = 1.0;
  if ( ! ! c2 * 07ll == 0 ){
    printf("BOGUS!\n");
  }
  else{
    printf("PASS\n");
  }
}

$ gcc tc250.c && ./a.out
tc250.c: In function ‘main’:
tc250.c:11: internal compiler error: in simplify_subreg, at simplify-rtx.c:3818
```

Předpokládané chování

```
$ gcc tc250.c && ./a.out
PASS
```

A.4 Chyba 4

Umístění chyby GCC 4.4.0, s vypnutými optimalizacemi

Typ chyby ICE-on-invalid

Zdrojový kód a reprodukce

```
$ cat tc1047.c
func_86234633 (unsigned long p_27126309)
{
          unsigned char l_20121738 = 0xF9;
          if ((l_20121738 >> 0x97B05850))
            {
            }
}
$ gcc tc1047.c -O0 -c
tc1047.c: In function ‘func_86234633’:
tc1047.c:4: warning: right shift count >= width of type
tc1047.c:7: error: unrecognizable insn:
(insn 6 5 7 3 tc1047.c:4 (set (reg:QI 60)
        (const_int -1750050736 [0x97b05850])) -1 (nil))
tc1047.c:7: internal compiler error: in extract_insn, at recog.c:1983
```

Předpokládané chování

```
$ gcc tc1047.c -O0 -c
tc1047.c: In function ‘func_86234633’:
tc1047.c:4: warning: right shift count >= width of type
```

A.5 Chyba 5

Umístění chyby TCC 0.9.24

Typ chyby accepts-invalid

Zdrojový kód a reprodukce

```
$ cat aaa.c
int aaa(int P, long P, short P, double P){
  int a = P;
}

$ tcc aaa.c -c && echo PASS || echo FAIL
PASS
```

Předpokládané chování

```
$ tcc aaa.c -c && echo PASS || echo FAIL
FAIL
```

A.6 Chyba 6

Umístění chyby TCC 0.9.24

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
$ cat 1536.c
int main (int argc, char *argv[]) {
  int  m1D = 'K';
  double FT = 61.11;
  long ct = 0xB466;
  if ( (3795ll) % (long long)(FT) - FT > (long long)(L'\xaD' + ct ) % (int)m1D){
    printf ("NONE");
    return 1;
  }
  return 0;
}

$ tcc 1536.c
$ ./a.out && echo PASS
Neoprávněný přístup do paměti (SIGSEGV)
```

Předpokládané chování

```
$ tcc 1536.c
$ ./a.out && echo PASS
PASS
```

A.7 Chyba 7

Umístění chyby TCC 0.9.24

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
$ cat 16.c
long long P4 = 1;
int main (int argc, char *argv[])
{
  unsigned long long aaa = ((((long long)(0007Lu)) % ((long long)(0x81)))+P4);
  unsigned long long bbb = (17l);
  printf("aaa: %i\n", aaa);
  printf("bbb: %i\n", bbb);
  if (aaa >= bbb)
      printf ("--- This is under aaa >= bbb block from variables \n");
}

$ tcc 16.c; ./a.out
aaa: 8
bbb: 17
--- This is under (aaa >= bbb) block from variables
```

Předpokládané chování

```
$ tcc 16.c; ./a.out
aaa: 8
bbb: 17
```

A.8 Chyba 8

Umístění chyby TCC 0.9.24

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
$ cat 16.c
unsigned int B (short K)
{
  K = (2 + 'U');
  printf("K397L5: %i\n", K);
  if ((K - K) <= K)
    printf ("THIS IS IN ((%i-%i) <= %i BLOCK\n", K, K, K);
  else
    printf ("THIS IS IN ((%i-%i) <= %i ELSE BLOCK\n", K, K, K);
  return L'g' + ((((long long) (L'~')) % ((long long) (L'\?' + ('g')))));
};

int
main (int argc, char *argv[])
{
  B (L'\v'-2979u);
}

$ tcc 232.c; ./a.out
K397L5: 87
THIS IS IN ((87-87) <= 87 ELSE BLOCK
```

Předpokládané chování

```
$ tcc 232.c; ./a.out
K397L5: 87
THIS IS IN ((87-87) <= 87 BLOCK
```

A.9 Chyba 9

Umístění chyby TCC 0.9.24

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
$ cat try.c
double a =1;
int main(){
  printf("%f\n",a);
  return 0;
}

$ cat try2.c
long long Q = 1;
int main (int argc, char *argv[])
{
  printf("Global long long Q: %llu\n", Q);
}

$ tcc try.c; ./a.out
0.000000
$ tcc try2.c; ./aout
Global long long Q: 629445874048565249
```

Předpokládané chování

```
$ tcc try.c; ./a.out
1.000000
$ tcc try2.c; ./aout
Global long long Q: 1
```

A.10 Chyba 10

Umístění chyby TCC 0.9.24

Typ chyby wrong-code

Zdrojový kód a reprodukce

```
$ cat tc551.c
long long D = 1;
int main(int argc, char *argv[]){
  printf("LEFT : %u\n", (D));
  printf("RIGHT: %u\n", ('+' - 0l) + ';' - D + 0234620 );

  if ( (D) <= ('+' - 0l) + ';' - D + 0234620 ){
    printf("PASS - WE ARE IN (LEFT <= RIGHT) IF BLOCK\n");
  }
}
$ tcc tc551.c; ./a.out
LEFT : 1
RIGHT: 80373
```

Předpokládané chování

```
LEFT : 1
RIGHT: 80373
PASS - WE ARE IN (LEFT <= RIGHT) IF BLOCK
```

Literatura

[1] *GCC bugzilla keyword descriptions* [online]. 2008 [cit. 2008-04-15]. Dostupný z WWW: <http://gcc.gnu.org/bugzilla/describekeywords.cgi>.

[2] *GCC online documentation* [online]. 2008 [cit. 2008-05-13]. Dostupný z WWW: <http://gcc.gnu.org/onlinedocs/>.

[3] Intel, Inc. *Intel® C++ compiler documentation* [online]. c2008 [cit. 2008-05-13]. Dostupný z WWW: <http://www.intel.com/software/products/compilers/docs/clin/main_cls/index.htm>.

[4] Ohloh. *Ohloh.net : Open Source, revealed* [online]. 2008 [cit. 2008-05-13]. Dostupný z WWW: <http://www.ohloh.net>.

[5] *Tiny C Compiler Reference Documentation* [online]. 2008 [cit. 2008-05-13]. Dostupný z WWW: <http://fabrice.bellard.free.fr/tcc/tcc-doc.html>.

[6] AHO, Alfred V., et al. *Compilers : Principles, Techniques, and Tools*. 2nd edition. [s.l.] : Addison-Wesley, 2007. 964 s.

[7] BEAZLEY, David M. *PLY : Python Lex-Yacc* [online]. [2007] [cit. 2008-04-12]. Dostupný z WWW: <http://www.dabeaz.com/ply/ply.html¿'.

[8] COWAN, Crispin, et al. *Buffer overflows: Attacks and defenses for the vulnerability of the decade*. In Foundations of Intrusion Tolerant Systems. [s.l.] : [s.n.], 2003. s. 14. Dostupný z WWW: <http://crispincowan.com/discex00.pdf>.

[9] *Valgrind User Manual* [online]. 2008 [cit. 2008-04-18]. Dostupný z WWW: <http://valgrind.org/docs/manual/manual.html>.

[10] GRUNE, Dick, JACOBS, Ceriel J.H. *Parsing techniques: A practical guide*. Chichester, Anglie : Ellis Horwood Limited, 1990.

[11] THOMAS, David, HUNT, Andrew. *Programátor Pragmatik*. Brno : Computer Press, 2007. 219 s.

[12] JOHNSON, Stephen C. *Yacc : Yet Another Compiler-Compiler* [online]. [2007] [cit. 2008-04-12]. Dostupný z WWW: <http://dinosaur.compilertools.net/yacc/index.html¿'.

[13] LESK, Michael E., SCHMIDT, Eric *Lex - A Lexical Analyzer Generator* [online]. [2007] [cit. 2008-04-12]. Dostupný z WWW: <http://dinosaur.compilertools.net/lex/index.html¿'.

[14] LINDIG, Cristian. *Random testing if C calling conventions*. Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG) [s.l.] : [s.n.], 2003. Dostupný z WWW: <www.st.cs.uni-sb.de/ lindig/papers/lindig-aadebug-2005.pdf>.

[15] MITCHELL, Mark, OLDHAM, Jeffrey, SAMUEL, Alex. *Pokročilé programování v operačním systému Linux*. Praha : Softpress, c2002. 265 s. ISBN 80-86497-29-1.

[16] NOVILLO, Diego. From source to binary. *Red Hat Magazine* [online]. 2004, no. 2 [cit. 2008-03-23]. Dostupný z WWW: <http://www.redhat.com/magazine/002dec04/features/gcc/>.

[17] SCHWARTZ, R. Writing nonsense with perl. *Linux Magazine* [online]. 1999, no. 9 [cit. 2008-03-23]. Dostupný z WWW: <http://www.linux-mag.com/id/309>.

[18] ISO 9899 TC3. *Programming Languages - C*. ISO, Geneva, Švýcarsko, 1999.

[19] TURNER, Bryan. *Random C Program Generator* [online]. [2007] [cit. 2008-04-24]. Dostupný z WWW: <http://brturn.googlepages.com/randomcprogramgenerator>.