VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ TESTOVACÍCH VSTUPŮ PODLE STOPY PROGRAMU

GENERATING TEST INPUTS BASED ON PROGRAM TRACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ SUŠOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Sušovský Tomáš, Bc.**
Program: Informační technologie Obor: Inteligentní systémy
Název: **Generování testovacích vstupů podle stopy programu**
Generating Test Inputs Based on Program Trace
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte instrukční sadu LLVM/IR. Nastudujte testování založené na modelech programu. Seznamte se s řešiči SMT vhodnými pro teorie nad celými čísly nebo ukazatelovou aritmetikou.
2. Analyzujte požadavky pro automatické generování testovacích případů v rámci testování založeného na modelech. Navrhněte transformaci stopy programu danou sekvencí instrukcí LLVM/IR do formule pro řešič SMT.
3. Implementujte program pro hledání testovacích vstupů založený na modelech programů a logik pro SMT.
4. Ověřte správnost implementace pomocí automatických testů základních částí převodu.

Literatura:

- Gyori, A.; Lahiri, S. K.; Partush, N. Refining interprocedural change-impact analysis using equivalence relations. 2017. In Proc. of ISSTA'17. doi: 10.1145/3092703.3092719.
- Kolektiv autorů Computer Science Laboratory SRI. Domovská stránka projektu llvm2smt. https://github.com/SRI-CSL/llvm2smt

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Tato práce se zabývá návrhem a implementací nástroje pro automatické generování testových vstupů na základě určené stopy programu. Cílem je zjednodušit a zefektivnit proces vytváření testových sad splňující pokročilá kritéria pokrytí (používaných v kritických aplikacích psaných v nízko úrovňových jazycích C/C++ splňující přísná omezení). Na základě modelu programu nástroj zkoumá, jaké přesné podmínky musí nastat pro průchod programu dle zadané stopy. Pro nalezení vhodných hodnot využívá existující pokročilý nástroj řešič SMT specializovaný na řešení problému splnitelnosti. Nástroj využívá knihovny překladačového rámce LLVM pro práci s modelem programu a knihovnu Z3 pro práci s řešičem SMT. Výsledkem této práce je návrh architektury nástroje pro generování testových vstupů, který dokáže vygenerovat vstupy pro vykonání zadané stopy programu díky analýzování modelu programu, a implementace jeho prototypu.

Abstract

This thesis focuses on design and implementation of a tool for automated generation of test inputs for a specified program trace. The aim of the thesis is to make development of testing suites (complying a given advanced coverage criteria) easier and more effective. These kinds of test suites are used in critical applications with code base written in low-level languages like C/C++ with strict restrictions applied. The tool investigates a program model and what conditions must be met to execute program in a way following provided trace. The tool uses advanced SMT-solver tool (software tool specialized for solving satisfiability problem) for generating fitting values. LLVM compiler framework libraries are used for modelling a program. Z3 library is used as a SMT-solver backend. This thesis brings results in architectural and implementation design of a tool capable of test inputs generation based on program analysis and provided program trace to cover.

Klíčová slova

Testování sfotwaru, LLVM, SMT-LIB, Generování testových vstupů

Keywords

Software testing, LLVM, SMT-LIB, Test inputs generation

Citace

SUŠOVSKÝ, Tomáš. *Generování testovacích vstupů podle stopy programu.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Generování testovacích vstupů podle stopy programu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Aleše Smrčky Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Sušovský
22.5. 2019

Poděkování

Děkuji vedoucímu této práce Ing. Aleši Smrčkovi Ph.D. za aktivní odbornou pomoc a podněty při řešení této práce.

Obsah

Kapitola 1

Úvod

Cílem této práce je vyvinout nástroj pro podporu automatického generovaní vstupních hodnot testových případů. Požadavkem pro vygenerované hodnoty je dodržení zadané cesty programu v testované jednotce při vykonání testu s těmito vstupními hodnotami (definováno v 2.2). Pro vhodně připravenou sadu cest programu nástroj vygeneruje odpovídající sadu hodnot testových vstupů, které mohou být použity pro vytvoření testovací sady splňující pokročilá kritéria pokrytí (jednotlivá kritéria pokrytí jsou vysvětlena v podkapitole 2.4).

Hodnoty jsou vždy generovány pro kód jedné testované jednotky a jednu jeho programovou cestu – volba správné kombinace cest je ponechána na uživateli. Nástroj je navržen tak, aby umožnil ovládání nejenom lidskému uživateli, ale i dalším automatizovaným nástrojům (které mohou obsahovat logiku pro určení správné kombinace cest).

Kapitola 2 obsahuje formální definice vlastností softwaru, které jsou v této práci zkoumány. Hlavní úlohou nástroje je ověření platnosti sémantické dosažitelnosti (definována v kapitole 2.3) zadané cesty. Ověření platnosti této vlastnosti je možné pomocí řešení problému dokazování správnosti programu (SAT problém – problém splnitelnosti 2.3.1). Pro sémanticky dostupné cesty nástroj generuje konkrétní hodnoty pro testové vstupy s využitím řešiče SMT (SMT rozšiřuje SAT) – programu specializovaného na řešení úloh SAT problémů. Řešiče SMT jsou optimalizovány pro nalezení řešení SAT problému ohodnocením proměnných ve formulích výrokové logiky, zapsaných ve formátu SMT-LIB (podrobněji vysvětleno v podkapitole 2.3.2). Nástroj bude pracovat na intraprocedurální úrovní -– v rámci kódu jedné funkce nebo procedury v kontextu své programové jednotky. Pro řešení SAT problému je třeba transformovat vstupní data (analyzovaný zdrojový kód programu a popis stopy) do reprezentace umožňující řešení tohoto problému – soustavy formulí výrokové logiky. Hledaným řešením je důkaz splnitelnosti zadané soustavy formulí pro zvolenou teorii. Pro zkoumání v doméně analýzy zdrojového kódu softwaru je možné zvolit jako cílovou teorii bitových vektorů, případně její rozšíření – teorii bitových vektorů a polí bitových vektorů (srovnání logik 2.1). Analyzované programy mohou být implementovány v různých programovacích jazycích s využitím různé úrovně abstrakce. Překladače, především ty, které podporují překlad více programovacích jazyků, využívají mezikód (vnitřní reprezentaci) pro reprezentaci modelu programu. Překladač při překladu převede program z kódu zdrojového programovacího jazyka do svého mezikódu (na kterém může provést optimalizace) a následně je z mezikódu vygenerován výstupní strojový kód programu pro cílovou architekturu. Použití modelu programu, generovaného ze souborů zdrojových kódů, by umožnilo zobecnit převod z původního kódu programu na formule výrokové logiky potřebné pro popis programu jako řešitelného SAT problému. Nástroj by tak nebyl omezen na analýzu programů implementovaných jen v jednom konkrétním programovacím jazyku. Vhodným formátem

pro popis modelů programů je LLVM IR (*LLVM Intermediate Representation* 2.5.3, využívaný v překladačích spadajících pod projekt LLVM 2.5.2).

Kapitola 3 obsahuje specifikaci formálních požadavků na řešení práce 3.1 a návrh nástroje `tindger`, jehož výsledná implementace by měla splňovat všechny uvedené požadavky. Výstupem této práce je prototyp nástroje, který v této fázi nepodporuje analýzu libovolného programu, ale pouze vymezenou podmnožinu programů neobsahujících určité komplexní vlastnosti uvedené v podkapitole 3.2.

Hledaným řešením je seznam ohodnocení proměnných (parametrů funkce, globálních proměnných, dat v paměti na symbolických adresách čtených z analyzovaného kódu), které ovlivňují tok řízení analyzované jednotky. U sémanticky nedosažitelných cest programu musí informovat uživatele o neúspěchu (pokud možno s lokalizací příčiny), aby uživateli umožnil zadat upravenou cestu a pokračovat v hledání nového řešení, potřebného pro dosažení cílového kritéria pokrytí. Návrh algoritmu transformace je blíže popsán v podkapitole 3.7.

Implementace nástroje `tindger` je detailně popsána v kapitole 4. Popsány jsou jednotlivé moduly aplikace a rozebrány konkrétní třídy dodávající funkcionalitu potřebnou ke splnění požadavků z kapitoly 3. U každého modulu je uveden popis hlavních algoritmů a významných datových struktur, které využívá ke splnění svého účelu.

Získání experimentálních výsledků a ověření jejich správnosti je zdokumentováno v kapitole 5.

Závěrečná kapitola 6 shrnuje, co patřilo mezi cíle této diplomové práce a obsahuje zhodnocení jejich dosažení. Dále poskytuje náhled do analýzy toho, které části mohou být dále rozšířeny. Závěr také přibližuje možné praktické využití vyvinutého nástroje v kontextu větší sady vzájemně spolupracujících nástrojů pro podporu testování softwaru (vyvíjené v rámci výzkumné skupiny Testos, která je blíže přiblížena v sekci 1.2).

1.1 Motivace

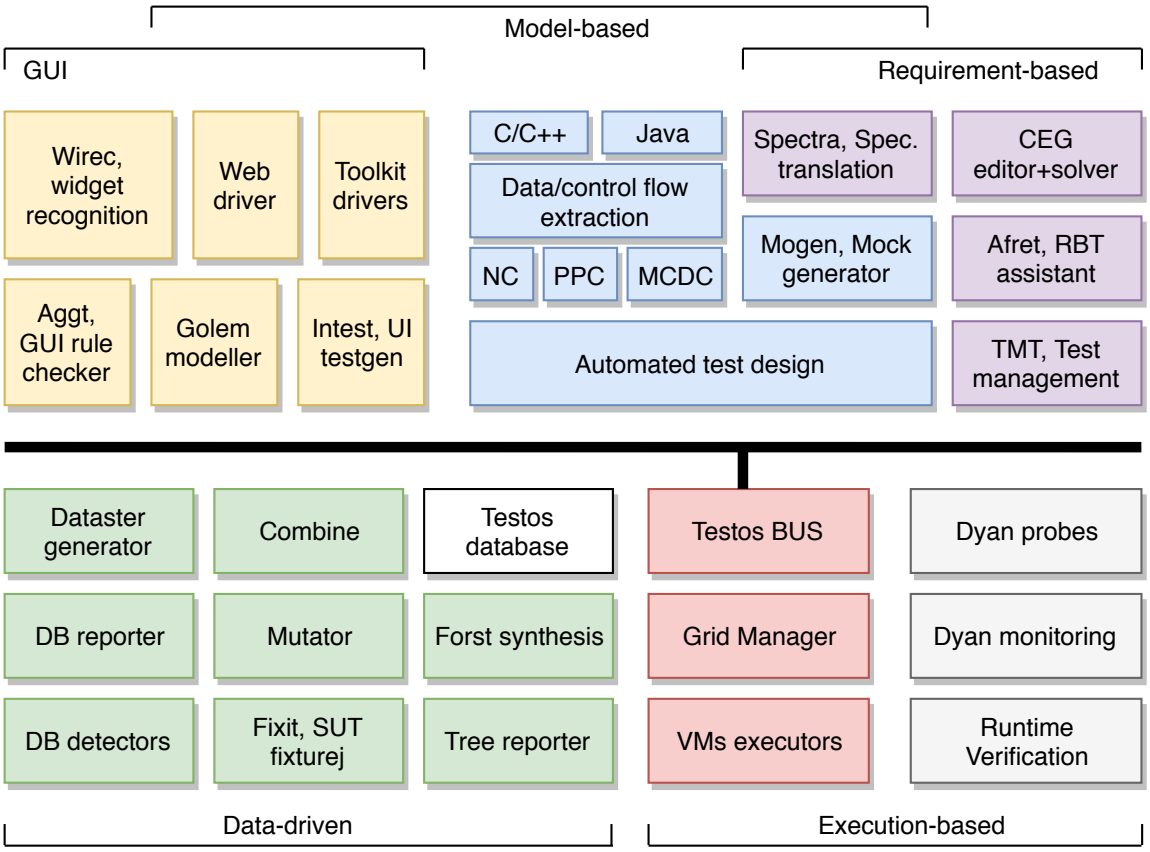
Analýza a testování softwaru tvoří důležitou součást procesu vývoje softwarových systému. Největší požadavky na testování jsou kladeny na systémy s vysokou úrovní spolehlivosti (SIL *angl. Safety Integrity Level*). Tyto *kritické aplikace*, mezi které se řadí systémy využívané v dopravě (automobilový průmysl, letectví), zdravotnictví, infrastruktuře, vesmírném programu a dalších odvětvích, musí splňovat přísné standardy a certifikace určené pro svou oblast. Vytvoření testové sady, která splňuje pokročilé požadavky pokrytí je časové náročná činnost, vyžadující práci kvalifikovaných testových inženýrů. Případné změny ve specifikaci požadavků nebo návrhu systému nevyžadují pouze úpravu v kódu implementace, ale také údržbu její testové sady. I pouze malé zvýšení efektivity procesu vytváření (a udržování) těchto testových sad může u velkých projektů ušetřit značnou část nákladů a času potřebného k dokončení projektu a jeho doručení do produkce (při zachování požadované kvality).

1.2 Platforma Testos

Nástroj pro automatizované generování testovacích vstupů je vyvíjen v rámci výzkumné skupiny Testos (Test Tool Set) [20]. Skupina Testos pracuje v rámci Fakulty informačních technologií VUT v Brně na vývoji platformy pro podporu automatizovaného testování softwaru. V současné fázi je většina nástrojů spadajících do platformy Testos vyvíjena v rámci studentských bakalářských a diplomových prací.

> Platforma Testos (Test Tool Set) umožňuje automatizaci testování softwaru. Nástroje této platformy kombinují různé úrovně testování (od jednotkového testování po akceptační testování) s různými kategoriemi přístupu k testování, jako je testování založené na modelech, testování založené na požadavcích, testování grafického uživatelského rozhraní, testování založené na datech a testování založené na běhu programu s dynamickou analýzou. Všechny části platformy jsou vyvíjeny s použitím špičkových technik a využívají poznatků současného výzkumu testování softwaru a dynamické analýzy. Platforma spojuje všechny své komponenty tak, aby si dokázala poradit se širokou řadou problémů, od analýzy použitelnosti frontendu testovaného systému, až po automatizované generování testů splňujících požadavky standardu IEC 64508 (např. pokrytí MCDC) pro systémy s nároky na vysokou úroveň spolehlivosti.
>
> (Manifest platformy Testos. Převzato z [20].)

Nástroj `tindger`, vyvinutý v rámci této diplomové práce, je navržen tak, aby umožňoval spolupráci s ostatními nástroji v rámci této platformy. Ostatní nástroje spadající do platformy Testos mohou automatizovaně generovat vstupní data pro tento nástroj dle požadavků uživatele (seznamy stop programu pro analýzu, splňující požadovaná kritéria pokrytí), nebo strojově zpracovat výstupy tohoto nástroje pro následné automatizované generování syntetických sad testových případů. V rámci platformy Testos tato práce náleží do části platformy zaměřené na testování založené na modelech. V této kategorii již byly vypracovány práce studentů (*Kondula Václav, 2017, Extrakce grafu toku řízení z formátu LLVM IR* [11], *Vít Radek, 2017, Control Flow Graph Query Engine* [22], *Sečkařová Petra, 2017, Extrakce grafu toku řízení z bajtkódu Java* [19]) a tato práce čerpá z dosažených výsledků a dále na ně volně navazuje. Souběžně s touto diplomovou prací byla vypracována práce na téma *Generování modelů pro testy ze zdrojových kódů* Danielem Krautem [14], se kterou tato práce sdílí část výzkumu. Očekává se, že další členové zapojení do skupiny Testos budou v následujících letech navazovat na výsledky této práce.



Obrázek 1.1: Schéma platformy Testos – platforma je rozdělena do částí dle zaměření (ve schématu barevně rozlišeno). Převzato z webových stránek platformy Testos [20].

Kapitola 2

Sémantická dosažitelnost cesty programu

Tato kapitola definuje pojmy, které se v následujících kapitolách využívají při popisu návrhu a implementačních detailů. Použitá Terminologie v této kapitole vychází z knihy *Introduction to Software Testing* (*P. Ammann a J. Offutt*, 2016 [1]).

2.1 Graf toku řízení

Graf toku řízení (CFG – *Control flow graph*) je abstraktní matematický model popisující program. Jedná se o orientovaný graf, kde uzly tvoří *základní bloky* a hrany jsou přechody mezi těmito uzly. Základní blok tvoří posloupnost instrukcí funkce, ukončená speciální instrukcí – *terminátorem* – která mění tok řízení (může se jednat o větvení (podmíněné či nepodmíněné) instrukcí skoku nebo předání řízení při návratu z funkce pomocí instrukce return). Základní bloky mohou být identifikovány pojmenovaným návěštím odkazujícím na první instrukci bloku anebo celočíselným indexem bloku v seznamu všech základních bloků funkce.

Definice 1 *Graf toku řízení CFG je čtveřice: CFG = (N,* N_0, N_F, E)
Kde platí:

- N – je konečná množina uzlů (základních bloků).
- N_0 – je konečná neprázdná množina počátečních uzlů, platí, že N_0 je podmnožina N.
- N_F – je konečná množina koncových uzlů, platí, že N_F je podmnožina N.
- E – je množina hran (orientovaných přechodů mezi dvěma uzly z N).

2.2 Cesta programu

Cesta programu mlžeme vyjádřit jako konečnou (neprázdnou) posloupnost uzlů z grafu toku řízení takových, že uzel cesty a jeho přímo následující uzel tvoří hranu grafu CFG. V kontextu této práce jsou zajímavé tzv. *testovací cesty*, pro které platí, že první uzel cesty patří mezi počáteční uzly analyzovaného CFG a poslední uzel cesty patří mezi koncové uzly analyzovaného CFG.

2.3 Sémantická dosažitelnost

Cestu programu považujeme za sémanticky dosažitelnou, v případě, že existuje ohodnocení vstupních proměnných, takové že stopa běhu program bude při jejich zadání obsahovat tuto cestu jako svou podcestu. Otázku, zda je cesta sémanticky dosažitelná, můžeme formulovat jako problém splnitelnosti.

2.3.1 SAT

Splnitelnost (*angl. Boolean Satisfiability Problem* nebo zkráceně *Satisfiability* – SAT) je problém řešící otázku zda existuje řešení, pro které zadaný výraz booleovské logiky bude vyhodnocen jako pravdivý – je splnitelný (`satisfiable`). V opačném případě musí být dokázáno, že daný výraz je kontradikcí (bude pro všechny možné vstupy vyhodnocen jako nepravdivý) a tedy nesplnitelný (`unsatisfiable`) – neexistuje žádné řešení, pro které by byl vyhodnocen jako pravdivý.

SAT je problém patřící do třídy složitosti NP-úplných (*NP-Complete*) problémů (byl prvním problémem vůbec, pro který bylo dokázáno, že se jedná o NP-úplný problém). NP (Nedeterministicky Polynomiální) problémy jsou problémy, které je možné teoreticky vyřešit v polynomiálním čase, pokud by byl použit k řešení počítač schopný v každém kroku provést neomezené větvení řešení a dále hledat řešení současně ve všech větvích (takový počítač existuje pouze jako teoretický model – nedeterministický Turingův stroj). Prakticky jde výsledné řešení takovýchto problému v polynomiálním čase ověřit, ale ne nalézt. NP-úplný problém je NP problém, pro který platí, že každý NP problém na něj může být redukován v polynomiálním čase.[1]

2.3.2 SMT-LIB

SAT modulo teorie (*angl. Satisfiability Modulo Theories* – SMT) rozšiřuje SAT problém splnitelnosti o splnitelnost výrazů predikátové logiky prvního řádu s rovností a prvky z různých teorií predikátové logiky určitého univerza (jako jsou lineární aritmetika, teorie polí, bitové vektory ad.). Ověřuje tedy, zda je zadaná formule splnitelná, ale ne libovolným ohodnocením jejich symbolů, ale takovým, které splňuje omezení kritérii definovanými určitou teorií.

Mezinárodní iniciativa SMT-LIB [2] se zaměřuje na výzkum a vývoj v oblasti řešení SMT problémů. Iniciativa SMT-LIB standardizuje jazyk SMT-LIB (aktuální verze standardu 2.6 [3]) pro zápis řešitelných SMT problémů. Jazyk SMT-LIB využívají programy specializované na řešení úloh z oblasti SMT (řešiče SMT). SMT-LIB dále vyvíjí řadu nástrojů a knihoven pro práci s SMT a sadu benchmarků (benchmarky jsou veřejně dostupné online viz [4]) pro testování účinnosti a efektivity programů řešících SMT (bylo provedeno přes 300 000 benchmarků založených na SMT-LIB). SMT-LIB verze 3 je ve vývoji, a měla by přinést lepší podporu pro kombinaci různých teorií a integraci vlastností formátu TIP, který rozšiřuje původní SMT-LIB verze 2 o podporu pro řešení problémů indukcí. Syntaxe jazyka SMT-LIB je definována v Backusova–Naurově formě a podobá se syntaxi jazyka LISP [3]. Používá se pro zápis termů a formulí (s typovým systémem) logiky prvního řádu, specifikaci nad nimi postavených teorií (mohou upravovat použití typů, funkcí a predikátových symbolů), a specifikaci nových logik v rámci zvolených teorií. A také slouží jako vstupní formát pro příkazy řešičů SMT.

[1] Podkapitola převzata z [21], zrevidováno.

Řešič SMT (*angl. SMT solver*) dostává na vstup vhodně zapsanou formuli, obsahující deklaraci proměnných (které jsou neměnné označuje je tedy za konstanty), deklaraci rozhraní funkcí, nebo i jejich přímou definici a sadu asercí, které určují omezení, co pro které proměnné musí platit. Výsledkem běhu programu řešiče je pak odpověď `SAT` (*satisfiable* – splnitelné), pokud existuje nějaké ohodnocení proměnných, takové že je možné splnit (vyhodnotit jako pravdivá) všechna tvrzení určená asercemi. V opačném případě vrací odpověď `UNSAT` (*unsatisfiable* – nesplnitelné). Pokud se mu podařilo najít aspoň jedno řešení, je možné získat výstup ohodnocení jednotlivých proměnných ve formě kódovaného výstupu. V praxi se setkáváme se dvěma případy, kdy je výhodné použít řešiče SMT. V prvním případě (který se týká i této práce) jde o nalezení konkrétních hodnot, které nám umožní docílit určitého stavu v systému, který jsme schopni formálně popsat (a je rozhodnutelný) — příkladem může být aplikace řešiče SMT pro řešení hry sudoku. V druhém případě, který se používá pro odhalení chyb v programech a verifikací jejich absence, se hledá proti-případ – nalezne-li řešič ohodnocení, pro které se může program dostat do chybového stavu, je zřejmé že program je chybový, ale opačné tvrzení, kdy se řešiči nepodaří najít ohodnocení takové, které by vedlo do chybového stavu, ještě nemusí zaručovat, že je program bezchybný.

Pro deklaraci proměnné slouží výraz `(declare-const jméno_konstanty Sort)`, kde jméno musí být unikátní vůči ostatním proměnným a funkcím a Sort udává datový typ omezující rozsah hodnot proměnné. SMT-LIB podporuje aritmetické typy (nemají omezenou velikost) Int, Real, nebo například bitový vektor (zápis: `(_ BitVec n)`, kde n je šířka v bitech) -– nové typy mohou být definovány uživatelem výrazem `define-sort`. Nově definované sorty můžou obsahovat prvky existujících typů nebo typové šablony: př. `(define-sort Set (T) (Array T Bool))`. Novější verze umožňují definovat i tzv. algebraické datové typy (výrazem `declare-datatypes`) – záznamy (`records`) a výčtové typy (`enumeration types`). Záznam je struktura obsahující pevně daný počet pojmenovaných prvků a přesně jeden konstruktor. Výčtový typ může obsahovat konečný seznam hodnot své domény, jednotlivé jeho prvky jsou vyhodnocovány jako unikátní konstanty (nemůže nastat situace, kdy máme více od sebe vzájemně různých proměnných jednoho výčtového typu, než je počet jeho hodnot). Funkce mohou být pouze deklarovány výrazem `(declare-fun jméno_funkce (parametry) Sort)` a řešič pak může určit jejich implementaci libovolně (dle omezení asercemi) tak, aby jejich vstupy a použití jejich výstupů vedlo ke úspěšnému nalezení splnitelného modelu, případně mohou být přímo definovány výrazem `(define-fun jméno_funkce (parametry) Sort (tělo_funkce))` -– pak musí řešič použít zadanou implementaci. Pokud je deklarovaná funkce bez parametrů, je z pohledu řešiče považovaná za konstantu. Omezení pro řešení jsou zadány výrazem `(assert (body))`, kde tělo aserce je výraz typu boolean. Splnitelnost zadaných formulí se ověřuje příkazem `(check-stat)`, vypsání modelu ohodnocení proměnných příkazem `(get-model)` (relevantní jen pro splnitelné soustavy formulí). Některé řešiče, jako řešič z3, podporují práci se zásobníkem, kdy mohou výrazem `(push)` umístit dodatečnou aserci na zásobník, provést ověření splnitelnosti a výrazem `(pop)` jej ze zásobníku znovu odstranit. Toto umožňuje prozkoumat více podobných vlastností a sdílet část asercí a část mít specifickou pro konkrétní případ. Vlastnosti, které bude řešič při hledání řešení podporovat závisí na módu použité logiky. Na obrázku 2.1 je znázorněna hierarchie logik z SMT-LIB standardu.

2.3.3 Z3

Řešič SMT Z3 je vyvíjen verifikačním týmem firmy Microsoft Research (dceřiná společnost firmy Microsoft zaměřená na výzkum) jako otevřený software pod licencí MIT [6]. Nabízí

knihovny umožňující využití v široké řadě jazyků (C, C++, C#, Java, OCaml, Python, nově podporuje i WebAssembly). Pro implementaci v rámci jazyka C++ je možné využít jak z3 knihovnu pro jazyk C++, tak pro jazyk C – liší se úrovní abstrakce (u C++ knihovny vyšší, u varianty pro C se více blíží k zápisu holého SMT-LIB kódu) a podporou vlastností z SMT-LIB (pro některé prvky chybí vysokoúrovňové rozhraní používané pro C++ verzi).

2.4 Kritéria pokrytí

Tato podkapitola popisuje důležitá, v praxi využívaná, kritéria pokrytí, která vycházejí z grafu toků řízení a grafu toků dat. Kritéria pokrytí obecně představují pravidla pro vytvoření požadavků na testování systému. Pokrytí udává míru toho, jak moc byl testovaný systém skutečně otestován. Splnění vyžadovaného kritéria pokrytí může potvrdit, že byl systém dostatečně otestován. Základní kritéria pokrytí sledují, které části programu (například řádky zdrojového kódu, větve, nebo jednotlivé funkce) byly během (testového) běhu programu vykonány. Pokročilá kritéria, na rozdíl od těch základních, nesledují pouze že konkrétní část programu byla vykonána, ale zohledňují i kroky, které k jejímu vykonání vedly.

Pokrytí uzlů

Pokrytí uzlů (NC – *angl. node coverage*) vyžaduje, aby pro každý syntakticky dosažitelný uzel CFG, platilo že testovací sada obsahuje cestu zahrnující tento uzel.

Pokrytí hran

Pokrytí hran (EC – *angl. edge coverage*) vyžaduje, aby pro každou syntakticky dosažitelnou hranu CFG, platilo že testovací sada obsahuje cestu zahrnující tuto hranu.

Pokrytí párů hran

Pokrytí párů hran (EPC – *angl. edge-pair coverage*) vyžaduje, aby pro každou syntakticky dosažitelnou cestu v CFG, která prochází až dvě hrany, platilo že testovací sada obsahuje cestu zahrnující tuto cestu jako svou podcestu.

Jednoduchá cesta

Jednoduchá cesta (*angl. simple path*) v CFG neobsahuj žádný uzel (kromě počátečního a koncového uzlu – pokud se jedná o stejný uzel) více než jedenkrát.

Hlavní cesta

Hlavní cesta (*angl. prime path*) v CFG je jednoduchá cesta, která zároveň není podcestou jiné jednoduché cesty ve svém grafu.

Pokrytí hlavních cest

Pokrytí hlavních cest (PPC – *angl. prime paths coverage*) vyžaduje aby testovací sada obsahovala všechny hlavní cesty CFG.

Kritérium pokrytí změn podmínky a rozhodnutí

Pokrytí změn podmínky a rozhodnutí (MCDC – *angl. modified condition/decision coverage*) se považuje za nejvyšší používaný standard pokrytí. Vyžaduje platnost čtyř pravidel souvisejících s podmínkami a rozhodnutími. Podmínkou je zde myšlen booleovský výraz, který nemůže být dále rozložen na menší booleovské podvýrazy. Rozhodnutím je zde myšlen booleovský výraz, který je složen z podmínek a booleovských operátorů.

1. Každý vstupní a výstupní bod musí být navštíven.
2. Každé rozhodnutí musí být vyhodnoceno na všechny možné hodnoty.
3. Každá podmínka v rozhodnutí musí být vyhodnocena na všechny možné hodnoty.
4. Každá podmínka v rozhodnutí musí samostatně ovlivnit výsledné vyhodnocení celého rozhodnutí.

2.5 Analýza zdrojového kódu

Tato podkapitola popisuje techniky použité k analýze zdrojového kódu softwaru. Popsán je konkrétní příklad rámce LLVM, který využívá SSA formu a překlad do mezikódu (popsány v následujících sekcích jednotlivě).

2.5.1 SSA

Static Single Assignment Form (SSA) je forma transformovaného kódu využívána překladači při vytváření *mezi jazyku* (*angl.* intermediate representation). Aby kód splňoval SSA musí pro něj platit, že každé proměnné je přiřazena hodnota pouze jednou a každá proměnná je definovaná před svým použitím. Transformace spočívá v přidání *pseudo-přiřazení* a *pseudo-použití*, kdy je původní proměnná, které je v kódu vícekrát přiřazována hodnota, rozdělena na více proměnných (verzí původní proměnné) a použití původní proměnné je nahrazeno za použití její odpovídající verze. Problémem tohoto převodu je větvení kódu, které se při převodu řeší analýzou CFG (viz definice 2.1) a přidáním Φ-funkce na začátek základního bloku. Do nové verze proměnné se přiřadí výsledek Φ-funkce a dále se s ní pracuje. Převedení kódu na SSA umožňuje překladačům provádět mnoho optimalizací, mezi které patří propagace konstant, odstraňování mrtvého kódu a odstraňování zbytečné redundance.[2]

[2]Podkapitola převzata z [21], zrevidováno.

Příklad převodu kódu do SSA formy:

Původní kód:

```
1 int a = 5;
2 int b = 10 + a;
3 int c = a + b;
4 a = 2 * a + c;
5 c = 5 * a + c;
6
7 return a + b + c;
```

Kód po převodu do SSA formy:

```
1 int a = 5;
2 int b = 10 + a;
3 int c = a + b;
4 int a_2 = 2 * a + c;
5 int c_2 = 5 * a_2 + c;
6
7 return a_2 + b + c_2;
```

2.5.2 Rámec LLVM

LLVM (Low Level Virtual Machine) [17] projekt zahrnuje sadu modulárních a znovupoužitelných překladačů a nástrojů. Vznikl jako výzkumný projekt na University of Illionois [15] a brzy se rozšířil pro výzkumné i komerční použití. Cílem LLVM je přinést moderní přístup k překladu libovolných jazyků s využitím SSA (Static Single Assignment 2.5.1) formy vnitřního kódu a podporou jak pro statický, tak dynamický překlad. Překladače postavené nad LLVM (označované jako LLVM frontend – např. překladač Clang pro jazyky C, C++ a Objective C nebo Flang překladač jazyka Fortran) nejprve ze zdrojového kódu v podporovaném programovacím jazyce vygenerují mezikód ve formátu LLVM IR (LLVM Intermediate Representation), který je společný pro všechny podporované programovací jazyky. Nad IR kódem jsou prováděny analýzy a optimalizace. Po optimalizaci mezikódu je z něj generován strojový kód pro cílovou architekturu tzv. LLVM backendy (dnes je podporována řada cílových architektur jako x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU, MIPS, MSP430, SystemZ a XCore). LLVM backend také může generovat z IR mezikódu zdrojový kód v jazyce C (nebo jiném) pro následný překlad na platformě, která není přímo podporována.

2.5.3 LLVM IR

LLVM IR [16] je silně typovaná a využívá SSA (Single Static Assignment) formu. Pro zápis kódu v SSA formě, platí, že každá proměnná může mít právě jednou zapsánu hodnotu a čtení její hodnoty může být opakováno (pokud se hodnota proměnné změní, musí být uložena do nové proměnné). V LLVM IR jsou výsledky instrukcí, které vrací hodnotu zapisovány vždy do nového registru. To umožňuje použití neomezené sady registrů – o mapování na skutečné registry cílové architektury (s omezeným počtem registrů) se pak stará až LLVM backend dané cílové architektury při generování strojového kódu. V situaci, kdy má být použita hodnota registru, kde mohla být zapsána různá hodnota (podmíněné větvení kódu) a každý takový výsledek je uložen ve vlastním novém registru je pro čtení hodnoty použit tzv. PHI uzel, který na základě vyhodnocení logické podmínky zvolí, který z potenciálních registrů se má použít. Přístup do paměti v LLVM nepodléhá SSA, hodnota načtená z paměti pomocí instrukce load je uložena do lokálního registru, který dále podléhá pravidlům SSA. Typový systém IR obsahuje primitivní typy (integer – rozlišené bitovou šířkou, Floating-point – rozlišené dle přesnosti, void a label) a odvozené typy (pole, funkce, ukazatele, struktury, zabalené struktury, vektory a speciální opaque typy např. použité pro dopřednou deklaraci

struktur, které mohou být pozdější fázi překladu určeny konkrétním typem). O mapování interních typů IR na instrukce a registry konkrétní architektury se opět starají LLVM backendy.

Základní jednotkou překladu z pohledu LLVM je Modul, který odpovídá samostatně překládané jednotce zdrojového kódu. V kontextu jazyka C lze přeložit překladačem clang jeden zdrojový soubor na LLVM modul a jeho reprezentaci v LLVM IR zapsat do souboru. Modul obsahuje globální proměnné, funkce, informace o knihovnách a dalších modulech, na kterých sám závisí, tabulku symbolů a metadata. Obsahuje všechny IR objekty nižší úrovně (IR objekty funkce, instrukcí, proměnných).

Funkce jsou v IR reprezentovány jako kolekce základních bloků, které obsahují jednotlivé instrukce. Dále obsahuje objekt funkce informace o svých parametrech a vazbu na modul, ve kterém je definována (a odtud získává přístup ke globálním proměnným modulu a dalším funkcím) a metadata. Jméno funkce z pohledu LLVM je určeno pomocí dekorace jmen (*angl. name mangling*) a závisí na použitém překladači a jím preferované volací konvence (*angl. calling convention*), v případě použitého překladače Clang a jazyka C++ zpravidla bude vygenerované jméno funkce obsahovat zakódovanou informaci o návratovém typu funkce a typech jejích parametrů.

Základní blok (*angl. basic block*) -– je důležitou jednotkou z pohledu analýzy. Analyzovaná stopa programu je určena posloupností základních bloků v rámci jedné funkce. Skládá se z posloupnosti po sobě jdoucích instrukcí bez větvení, tak jak jsou vykonány programem. Instrukce provádějící větvení se označuje jako terminátor a základní blok jí musí být ukončen. Aktuálně LLVM IR využívá 8 různých terminálních instrukcí, které provádí řízení toku (instrukce podmíněného skoku, návratu z funkce a instrukcí pro zpracování výjimek a dalších). Pro kontrolu řízení toku programu se při větvení mezi bloky používají návěští identifikující daný základní blok. Pokud nejsou při překladu explicitně nastaveny, dokáže si je LLVM na úrovni IR kódu odvodit sám. Jeden základní blok může být během vykonávání kódu funkce vykonán vícenásobně (např. kód obsahující cyklus) – což je potřeba zohlednit dále při analýze.

Instrukce jsou nejnižší jednotkou na úrovni IR kódu. Instrukční sada LLVM obsahuje instrukce pro aritmetické a bitové operace, řízení toku, práci s pamětí, práci s vektory a instrukce pro konverze typů. Základní sada může být konkrétním překladačem nebo nástrojem rozšířena o další instrukce, ale pro překlad je pak nutné využít backend podporující tato rozšíření. Stejná instrukce může být použita pro různé datové typy (např. instrukce add pro 32-bitový integer i pro 64-bitový integer).

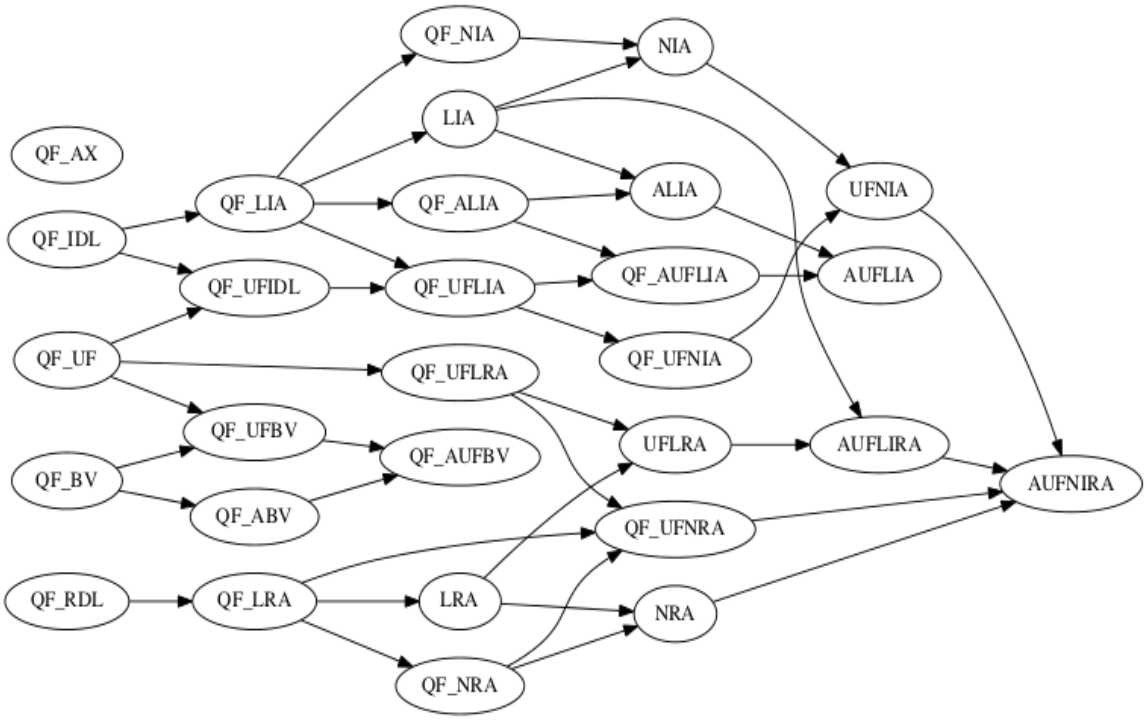
LLVM IR mezikód ze souboru s C++ zdrojovým kóde lze vygenerovat pomocí překladače Clang spuštěného se speciálním parametrem (`-emit-llvm`).

```
clang++ -emit-llvm -c main.cpp -o main.bc
```

Výpis 2.1: Příkaz pro vygenerování LLVM IR v bitovém kódu.

```
clang++ -emit-llvm -S main.cpp -o main.ll
```

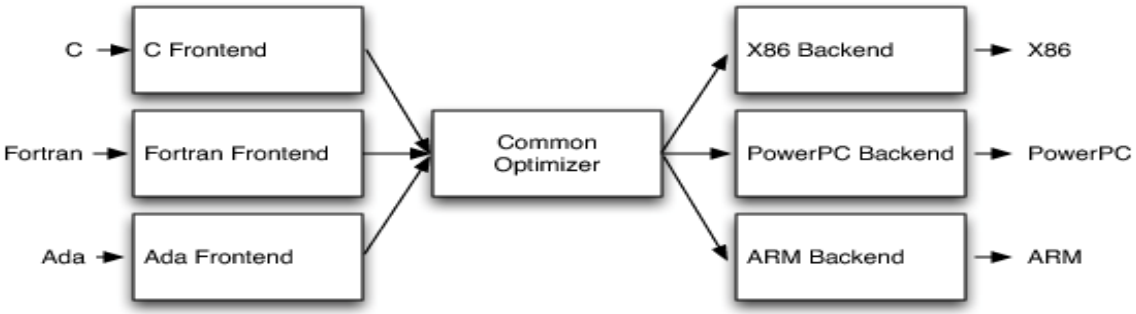
Výpis 2.2: Příkaz pro vygenerování LLVM IR v textové, lidsky čitelné podobě.



Obrázek 2.1: **Schéma hierarchie logik v SMT-LIB v2.**
Vysvětlivky použitých názvů logik:

- QF – omezení na formule bez kvantifikátoru (všeobecný kvantifikátor *forall*)
- A/AX – podpora polí
- BV – podpora bitových vektorů s pevnou délkou
- FP – podpora typů s plovoucí řádovou čárkou
- IA (Integer Arithmetic) – podpora aritmetických celých čísel (s neomezenou velikostí)
- RA (Real Arithmetic) – podpora reálných čísel (s neomezenou velikostí)
- IRA (mixed Integer Real Arithmetic) – podpora celých a reálných čísel (s neomezenou velikostí)
- LIA, LRA, LIRA – lineárně omezená verze IA, RA, IRA
- NIA, NRA, NIRA – nelineární omezená verze IA, RA, IRA
- IDL – diferenciální logika celých čísel
- RDL – diferenciální logika reálných čísel
- UF – rozšíření povolující volné typy a symboly funkcí

`Převzato z: http://smtlib.cs.uiowa.edu/logics.shtml`



Obrázek 2.2: **Schéma architektury LLVM.** Ilustrace zobrazuje rozvržení architektury sady nástrojů LLVM, kde různé frontend překladače (vlevo) generují stejný mezikód, na kterém jsou provedeny optimalizace (uprostřed) a různé backend překladače (vpravo) generují výsledný strojový kód pro cílové architektury. Převzato z: https://www.aosabook.org/en/llvm.html.

Kapitola 3

Návrh nástroje pro generování vstupních hodnot

Tato kapitola předkládá návrh nástroje `Tindger` (Test Input Data Generator) – generátoru testových vstupů pro specifikované stopy programu. Obsahuje seznam formálních funkcionálních a nefunkcionálních požadavků (viz 3.1) na řešení a jejich neformální detailní popis. Požadavky jsou upřesněny specifikací omezení (viz 3.2) vymezující možné případy, které mohou v analyzovaném kódu nastat, ale jejichž analýza nebude nástrojem podporována (a zdůvodnění).

3.1 Specifikace formálních požadavků

V následující tabulce jsou uvedeny požadavky na řešení této práce. Požadavky jsou rozděleny na funkcionální (F) a nefunkcionální (N). U jednotlivých požadavků je uveden krátký popis a seznam jejich prerekvizit (sloupec Pre).

#	Požadavek	Typ	Pre
1	Nástroj generuje SMT model cesty, pokud je cesta sémanticky splnitelná.	F	2, 20
2	Nástroj ověří sémantickou dostupnost cesty pomocí řešiče SMT.	N	
3	Nástroj vytváří seznam vygenerovaných hodnot, pokud je nalezeno řešení SMT modelu cesty.	F	1
4	Nástroj načítá vstupní zdrojový kód ze souboru, pokud uživatel zadá cestu souboru jako parametr `-input`.	F	
5	Nástroj načítá vstupní zdrojový kód ze standardního vstupu, pokud uživatel nezadá soubor.	F	
6	Nástroj vrací chybovou zprávu, pokud zadaný soubor neexistuje.	F	7
7	Nástroj načítá konfiguraci analýzy cesty ze souboru specifikovaného parametrem `-config`.	F	
8	Nástroj vrací chybovou zprávu, pokud zadaný konfigurační soubor neexistuje.	F	7
9	Nástroj vrací chybovou zprávu, pokud zadaný konfigurační soubor není ve formátu JSON.	F	7
10	Nástroj vrací chybovou zprávu, pokud zadaný konfigurační soubor neodpovídá schéma (viz C).	F	7, 9
11	Nástroj vrací chybovou zprávu, pokud je vstupní zdrojový kód není validní dle formátu kódu LLVM IR.	F	4, 5, 32
12	Nástroj vrací chybovou zprávu, pokud zadaná funkce není nalezena.	F	11, 7
13	Nástroj analyzuje cestu zadanou jako dvojici funkce a konečné posloupnosti indexů jejich základních bloků.	F	10
14	Nástroj provede intraprocedurální analýzu cesty v jedné funkci.	F	12, 13
15	Nástroj provádí analýzu cyklů do maximální hloubky omezené zadanou (konečnou) cestou.	F	14, 13
16	Nástroj vrací chybovou zprávu, není-li zadaná cesta syntakticky dosažitelná.	F	14, 15
17	Nástroj vrací chybovou zprávu, obsahuje-li cesta rekurzi.	F	14
18	Nástroj vrací chybovou zprávu, obsahuje-li cesta nepodporovanou instrukci.	F	14
19	Nástroj vrací chybovou zprávu, obsahuje-li cesta práci s nepodporovanými datovými typy.	F	14
20	Nástroj generuje výrazy SMT pro popis cesty a formulaci problému její sémantické dosažitelnosti.	F	16, 18,19
21	Nástroj ukládá vygenerované SMT výrazy do souboru specifikovaného parametrem `-smt`.	F	20
22	Nástroj vypíše vygenerované SMT výrazy na standardní výstup, není-li specifikován parametr `-smt`.	F	21
23	Nástroj využívá řešič Z3 verze 4.7 a její knihovny pro práci s SMT-LIB.	N	2
24	Nástroj generuje chybovou zprávu, při zjištění sémanticky nedostupné cesty.	F	2
25	Výstup nástroje je uložen ve formátu JSON. Obsahuje serializované hodnoty vstupů nebo chybu.	F	24, 3
26	Výstup je uložen do souboru, pokud je specifikovaný parametrem `-output`.	F	25
27	Výstup je vypsán na standardní výstup, není-li soubor specifikovaný parametrem.	F	25, 26
28	Chybový výstup je vypsán na standardní chybový výstup, není-li soubor specifikovaný parametrem `-error`	F	
29	Aplikační rozhraní nástroje je zdokumentováno.	N	
30	Nástroj vypíše nápovědu ke svému ovládání, je-li zadán parametr `-help`	F	
31	Nástroj podporuje integraci s ostatními programy v projektu Testos.	N	
32	Nástroj podporuje analýzu LLVM IR verze knihovny LLVM 7 a 8.	N	
33	Nástroj podporuje datový typ `Integer` o libovolné bitové velikosti.	F	
34	Nástroj podporuje datový typ `Bool`.	F	
35	Nástroj podporuje práci se strukturami a přístupem k jejich položkám.	F	
36	Nástroj podporuje větvení kódu pomocí řídicích struktur `if`, `else`, `else if`.	F	
37	Nástroj podporuje větvení kódu pomocí řídicí struktury `switch`.	F	5
38	Nástroj podporuje cykly `while`, `do while`, `for`.	F	5, 7
39	Nástroj obsahuje popis referenčního prostředí, ve kterém může být deterministicky sestaven.	N	
40	Nástroj obsahuje referenční testovou sadu ověřující validitu analýzy podporovaných vlastností.	N	

Tabulka 3.1:
Seznam formálních požadavků.
Typ: F - funkcionální, N - nefunkcionální.

3.2 Specifikace omezení

V rámci této práce je cílem dodat prototyp nástroje sloužící jako potvrzení konceptu (*angl. Proof of Concept* generování testových vstupů z modelů programů s využitím řešiče SMT. Konkrétní omezení vyplývají jednak z komplexity analyzované domény (rozsah programovacích jazyků C/C++, LLVM IR), a také technologických omezení možností současných implementací řešičů SMT. Složitost řešení SMT úloh v logice bitových vektorů s pevnou velikostí bez kvantifikátorů v praxi podrobně zkoumal tým Gergelyho Kovásznaie ve své práci [12] a navazující práci [13]. Prokázali vliv kódování délek bitových vektorů (srovnávané referenční unární kódování proti logaritmickému binárnímu kódování prakticky využívaného v SMT-LIB a Z3) na časovou složitost. Pro použití logaritmického binárního kódování se podařilo prokázat časovou složitost jako *NExpTime-těžkou*.

- Analýza kódu probíhá intraprocedurálně – v rámci jedné funkce – důvodem je řádově nižší komplexita analýzy (oproti interprocedurální variantě), přesto je dostačující pro ověření hlavní myšlenky prototypu generátoru.
- Zpracování výjimek není podporováno – zpracování výjimek v jazyce C++ vyžaduje vytvoření pokročilejšího modelu CFG, než se kterým počítá tato práce. U kritických aplikací je často zpracování výjimek vynuceně zakázáno na úrovni překladu a k ošetření chybových stavů v programu jsou použity jiné přístupy.
- Analýza rekurzivních volání funkce není podporována – analýza rekurzivních volání funkcí je nad rámec cílů stanovených prototypem.
- V analyzovaném kódu nedochází k volání polymorfních funkcí – analýza těchto volání souvisí s interprocedurální analýzou, která není podporována. Navíc je komplikována o nutnost analýzy vyhledávání ve VMT – tabulce virtuálních metod.
- Analýza je prováděna pro jednovláknové programy – analýza paralelních programů je samostatná kategorie analýzy software, vyžadující pokročilé speciální techniky nad rámec možností prototypu.
- Datové typy s plovoucí řádovou čárkou (*angl. Floating-point*) reprezentující reálná čísla nejsou podporovány – důvodem je kompilace s reprezentací floating-point typů na straně SMT-LIB (pro plnou podporu by bylo třeba vytvořit speciální model pro mapování floating-point hodnot o různých velikostech, na bitové vektory, tak aby je bylo možné využít při práci paměťovým modelem a konvertovat na celočíselné datové typy a naopak).
- Paměťový model předpokládá zarovnaný přístup do paměti – analýza nezarovnaného přístupu k paměti vyžaduje složitější postup, než se očekává od vyvíjeného prototypu.
- Instrukce pro vektorizaci – vektorizace dat při překladu je častým způsobem optimalizace programů, nicméně pro demonstraci prototypu nástroje vyvíjeného v rámci této postačí demonstrace analýzy neoptimalizovaného kódu a podpora vektorizace zůstává ponechána jako možné budoucí rozšíření této práce.
- Analýza variadických funkcí není podporována – variadické funkce, které mohou mít proměnný počet parametrů nejsou nezbytnou vlastností nutnou pro demonstraci korektní analýzy kódu vyvíjeným prototypem a jejich podpora zůstává ponechána jako možné budoucí rozšíření této práce.

Vylepšení nástroje přidáním podpory pro analýzu případů podléhajících těmto omezením bude možné jako rozšíření původní realizace nad rámec této diplomové práce. V rámci skupiny Testos (viz 1.2) se předpokládá další výzkum v této oblasti, včetně vývoje nových nástrojů a rozšiřování stávajících o novou funkcionalitu.

3.3 Návrh překladové tabulky

Překladová tabulka je jádrem řešení této práce. Popisuje způsob, jak je navržena transformace jednotlivých instrukcí mezikódu LLVM IR transformovány do podoby výrazů jazyka SMT-LIB. Jednotlivé SMT výrazy je třeba převést na výraz typu `boolean` tak, aby mohly být zadány řešiči jako aserce jedné formule. Řešič SMT má za úkol najít řešení (odpovídající model), které bude splňovat všechny aserce zároveň (pro výraz uvnitř aserce bude nalezeno ohodnocení výrazu na logickou hodnotu 1, a zároveň nebude vylučovat vyhodnocení platnosti ostatních výrazů). V kódu SMT-LIB pak bude aserce vypadat takto: `(assert výraz_instrukce)`.

Pro každou LLVM IR instrukci (číslovány dle operačních kódů LLVM IR ISA verze 7) je uveden stručný popis vysvětlující postup při transformaci na SMT výraz. Pod popisem jsou uvedeny příklady použití instrukce v LLVM IR (jedna instrukce může mít více způsobů použití s odlišnými druhy parametrů, pak je uvedeno více příkladů) a odpovídajícího převedeného SMT výrazu. Použité proměnné jsou v příkladech označeny stejnými názvy v LLVM ukázce i SMT. Aby mohly být proměnné použity jako parametry SMT výrazu musí být dříve definovány jako konstanty s odpovídajícím typem. U SMT výrazů s parametry typu bitových vektorů (s pevnou délkou) platí pravidlo, že operandy musí mít stejnou bitovou délku. Při použití regulérně vygenerovaného kódu LLVM IR je stejná bitová velikost operandů zpravidla zajištěna explicitním přetypováním operandů. V případě analýzy uměle vytvořeného kódu LLVM IR, kde by tato vlastnost LLVM nebyla dodržena se předpokládá selhání překladu a ukončení analýzy s chybovou hláškou. Speciálním případem jsou operandy typu `integer` o bitové délce 1 - ty se v LLVM IR používají pro reprezentaci hodnot typu `boolean`. Přeloženy do SMT mohou znamenat sémanticky dvě rozdílné hodnoty (boolean nebo bitový vektor o délce 1) a je nutné vložit explicitní konverzi podle konkrétního případu užití.

Nástroj v rámci rozsahu této práce nepodporuje překlad kompletní instrukční sady LLVM IR, ale pouze její podmnožiny. Omezení podpory překladu instrukcí a jejich zdůvodnění jsou uvedena v předcházející podkapitole 3.2. U nepodporovaných instrukcí je v popisu uvedeno krátké zdůvodnění, proč nejsou podporovány a je uveden pouze příklad pro podobu v LLVM IR. Příklad SMT výrazu pro nepodporované instrukce je vynechán.

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
1	`Ret`	Implicitní – není potřeba generovat speciální SMT výraz. Instrukce Ret předává řízení zpátky do volající funkce, intraprocedurální analýza zde končí.
Příklad:		
LLVM	`ret i32 5`	
SMT		
LLVM	`ret void`	
SMT		

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
2	`Br`	Instrukce Br (Branch) se nahradí aserci vyhodnocení booleovské podmínky dle požadované cesty. Implicitní u nepodmíněného skoku. Pokud tok řízení v analyzované cestě pokračuje do základního bloku označeného návěštím `label %IfTrue`, pak je přidána aserce podmínky `%cond` na hodnotu `True`. Pokud tok řízení v analyzované cestě pokračuje do základního bloku označeného návěštím `label %IfFalse`, pak je přidána aserce podmínky `%cond` na hodnotu `False`. V LLVM IR se můžeme setkat s tím, že je proměnná podmínky (LLVM typ `Bool`) reprezentována v IR datovým typem `i1` (1-bitový integer). Tento typ je možné do kódu SMT interpretovat jako SMT sort `Bool`, nebo `(_ BitVec 1)` (bitový vektor o délce 1 bit). Implicitní konverze mezi sortem `Bool` a `(_ BitVec 1)` v SMT výrazech není možná. Proměnná podmínky `%cond` se tedy generuje rovnou jako sort `Bool`, a pokud je potřeba s ní pracovat jako s bitovým vektorem, provede se explicitní konverze pomocí SMT výrazu `ite` (viz instrukce #53 PHI).
Příklad:		
LLVM	`br i1 %cond, label %IfTrue, label %IfFalse` ; podmíněný skok	
SMT	`(= %cond True)`	
LLVM	`br label %dest ; nepodmíněný skok`	
SMT		
3	`Switch`	Instrukce Switch podporuje větvení (viz instrukce #1 Br) s možností výběru cíle z více než dvou větví. Switch obsahuje seznam návěští (`label`) základních bloků, na které může skočit. Každý případ (`case`) obsahuje návěští a index, který porovnává s operandem instrukce switch. Výchozí návěští (v příkladu `label %otherwise`) je použito, pokud se operand nerovná žádnému z indexů. Do SMT transformuje na výraz pro aserci operandu na hodnotu požadovaného indexu (dle zadané cesty). Pokud cesta pokračuje přes výchozí návěští, pak se zajistí, že se hodnota operandu liší od všech indexů pomocí SMT výraz `distinct`.
Příklad:		

LLVM	`switch i32 %val, label %otherwise [ i32 0, label %onzero,`
	`  i32 1, label %onone,`
	`  i32 2, label %ontwo ]`	

SMT	`; příklad pro cestu pokračující přes case label:`
	`(= %val (_ bv1 32))`
	`; příklad pro cestu pokračující přes %otherwise label:`
	`(distinct %val (_ bv0 32) (_ bv1 32) (_ bv2 32))`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
4	`IndirectBr`	Instrukce IndirectBr provádí nepřímý skok (viz instrukce #1 Br), kdy cílový základní blok je určen adresou základního bloku v operandu instrukce. Seznam cílových destinací obsahuje adresy základních bloků (`blockaddress` konstanta bloku v LLVM IR) v rámci funkce (není možné skočit na základní blok z jiné funkce). Na vstupní základní blok funkce není možné skočit pomocí této instrukce.
Příklad:		
LLVM	`indirectbr i8* %Addr, [ label %bb1, label %bb2, label %bb3 ]`	
SMT	`(= %Addr %bb1)`	
5	`Invoke`	Instrukce Invoke slouží k volání funkce, která může vrátit řízení zpět do volající funkce normálním návratem pomocí instrukce `Ret`, nebo výjimkou. Instrukce obsahuje dvě návěští (v příkladu `label %Continue` a `label %TestCleanup`) - jedno pro normální návrat, druhé pro návrat pomocí výjimky (označuje se jako *landing pad* - "přistávací plocha" výjimky a musí vést na blok začínající instrukcí `landingpad` - viz #64 `landingpad`). Zpracování výjimek není nástrojem podporováno, viz 3.2. Při transformaci instrukce na SMT výraz se tedy nebere v potaz návěští pro návrat z výjimky a instrukce je zpracována jako volání funkce bez ošetření výjimky. Viz instrukce #54 Call.
Příklad:		
LLVM	`%retval = invoke i32 @Test(i32 15) to label %Continue unwind label %TestCleanup`	
SMT	`(= %retval (Test (_ bv15 32)))`	
6	`Resume`	Instrukce Resume slouží jako terminátor pro pokračování propagace výjimky v procesu zpracování výjimek. Zpracování výjimek není nástrojem podporováno, viz 3.2.
Příklad:		
LLVM	`resume i8*, i32 %exn`	
SMT		
7	`Unreachable`	Interní LLVM instrukce určená pro optimalizace (identifikace nedostupného kódu, který bude možno odstranit v rámci optimalizační transformace). Cesta obsahující instrukci Unreachable nebude nikdy sémanticky dostupná. Není potřeba vytvářet speciální SMT výraz.
Příklad:		
LLVM	`unreachable`	
SMT		
8	`CleanupRet`	Instrukce CleanupRet slouží pro předání řízení po dokončení obsluhy výjimky (zpátky směrem ke zdroji původního volání, nebo na další návěští obsluhy – `label %continue` v příkladu). Zpracování výjimek není nástrojem podporováno, viz 3.2.
Příklad:		

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
LLVM	`cleanupret from %cleanup unwind to caller`	
SMT		
LLVM	`cleanupret from %cleanup unwind label %continue`	
SMT		
9	`CatchRet`	Instrukce CatchRet slouží k ukončení obsluhy zachycené výjimky. Zpracování výjimek není nástrojem podporováno , viz 3.2.
Příklad:		
LLVM	`catchret from %catch label %continue`	
SMT		
10	`CatchSwitch`	Instrukce CatchSwitch se používá pro výběr následujícího bloku výjimky ze seznamu bloků obsluhy, nebo návrat z obsluhy k volající funkci. Zpracování výjimek není nástrojem podporováno, viz 3.2.
Příklad:		
LLVM	`%cs1 = catchswitch within none [label %handler0, label %handler1]`	
SMT		
LLVM	`%cs2 = catchswitch within %parenthandler [label %handler0] unwind label %cleanup`	
SMT		
11	`Add`	Instrukce pro celočíselného sčítání. Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost. Při přetečení operace je výsledek zarovnán pomocí modulo 2^n (n je bitová velikost proměnné pro uložení výsledku) – v LLVM i SMT.
Příklad:		
LLVM	`%result = add %a, %b`	
SMT	`(= %result (bvadd %a %b))`	
LLVM	`%result = add nuw nsw %a, i32 4`	
SMT	`(= %result (bvadd %a (_ bv4 32)))`	
12	`FAdd`	Instrukce pro sčítání reálných čísel. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%result = fadd %a, %b`	
SMT		
LLVM	`%result = fadd float 4.0, %a`	
SMT		
13	`Sub`	Instrukce pro celočíselné odčítání. Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost. Při přetečení operace je výsledek zarovnán pomocí modulo 2^n (n je bitová velikost proměnné pro uložení výsledku) – v LLVM i SMT.
Příklad:		
LLVM	`%result = sub %a, %b`	
SMT	`(= %result (bvsub %a %b))`	
LLVM	`%result = sub i32 4, nuw nsw %a`	
SMT	`(= %result (bvsub %a (_ bv4 32)))`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
14	`FSub`	Instrukce pro odčítání reálných čísel. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%result = fsub %a, %b`	
SMT		
LLVM	`%result = fsub float 4.0, %a`	
SMT		
15	`Mul`	Instrukce pro celočíselné násobení. Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost. Při přetečení operace je výsledek zarovnán pomocí modulo 2^n (n je bitová velikost proměnné pro uložení výsledku) – v LLVM i SMT.
Příklad:		
LLVM	`%result = mul %a, %b`	
SMT	`(= %result (bvmul %a %b))`	
LLVM	`%result = mul nuw nsw %a, i32 4`	
SMT	`(= %result (bvmul %a (_ bv4 32)))`	
16	`FMul`	Instrukce pro násobení reálných čísel. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%result = fmul %a, %b`	
SMT		
LLVM	`%result = fmul float 4.0, %a`	
SMT		
17	`UDiv`	Instrukce pro celočíselné dělení (čísel bez znaménka). Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost.
Příklad:		
LLVM	`%result = udiv %a, %b`	
SMT	`(= %result (bvudiv %a %b))`	
LLVM	`%result = udiv exact %a, i32 4`	
SMT	`(= %result (bvudiv %a (_ bv4 32)))`	
18	`SDiv`	Instrukce pro celočíselné dělení (čísel se znaménkem). Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost. Pokud dojde k přetečení (například -2147483648 děleno -1 pro proměnné typu 32-bitový integer), výsledek je z pohledu LLVM nedefinovaná hodnota.
Příklad:		
LLVM	`%result = sdiv %a, %b`	
SMT	`(= %result (bvsdiv %a %b))`	
LLVM	`%result = sdiv exact %a, i32 4`	
SMT	`(= %result (bvsdiv %a (_ bv4 32)))`	
19	`FDiv`	Instrukce pro dělení reálných čísel. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		

Transformace instrukcí LLVM IR na výrazy SMT-LIB		
#	instrukce	poznámka
LLVM	`%result = fdiv %a, %b`	
SMT		
LLVM	`%result = fdiv float 4.0, %a`	
SMT		
20	`URem`	Instrukce pro zbytek po dělení celých čísel bez znaménka. Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost.
Příklad:		
LLVM	`%result = urem %a, %b`	
SMT	`(= %result (bvurem %a %b))`	
LLVM	`%result = urem %a, i32 4`	
SMT	`(= %result (bvurem %a (_ bv4 32)))`	
21	`SRem`	Instrukce pro zbytek po dělení celých čísel se znaménkem. Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost. Pokud dojde k přetečení dělení (například -2147483648 děleno -1 pro proměnné typu 32-bitový integer), je výsledek dělení i zbytku po dělení z pohledu LLVM nedefinovaná hodnota.
Příklad:		
LLVM	`%result = srem %a, %b`	
SMT	`(= %result (bvsrem %a %b))`	
LLVM	`%result = srem %a, i32 4`	
SMT	`(= %result (bvsrem %a (_ bv4 32)))`	
22	`FRem`	Instrukce pro zbytek po dělení reálných čísel. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%result = frem %a, %b`	
SMT		
LLVM	`%result = frem float 4.0, %a`	
SMT		
23	`Shl`	Instrukce Shl slouží pro bitový posuv čísla (bitového vektoru) doleva. První operand slouží jako originální hodnota, druhý operand určuje o kolik bitů se hodnota posune.
Příklad:		
LLVM	`%shifted = shl i32 4, %var ; yields i32: 4 ≪ %var`	
SMT	`(= %shifted (bvshl (_ bv4 32) %var)`	
24	`LShr`	Instrukce LShr slouží pro logický bitový posuv čísla (bitového vektoru) doprava. Zprava se při posuvu doplňují nuly. První operand slouží jako originální hodnota, druhý operand určuje o kolik bitů se hodnota posune.
Příklad:		
LLVM	`%shifted = lshr i32 -4, i32 1 ; yields i32:shifted = 2147483646`	
SMT	`(= %shifted (bvlshr (_ bv4294967292 32) (_ bv1 32))`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
25	`AShr`	Instrukce AShr slouží pro aritmetický bitový posuv čísla (bitového vektoru) doprava. Zprava se při posuvu doplňují bity původního znaménka. První operand slouží jako originální hodnota, druhý operand určuje o kolik bitů se hodnota posune.
Příklad:		
LLVM	`%shifted = ashr i32 -4, 1 ; yields i32:shifted = -2`	
SMT	`(= %shifted (bvashr (_ bv4294967292 32) (_ bv1 32))`	
26	`And`	Instrukce And slouží pro získání logického bitového součinu. Při překladu z C++ reprezentuje bitový operátor AND (v C++ `&`). Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost.
Příklad:		
LLVM	`%result = and i32 4, %var`	
SMT	`(= %result (bvand (_ bv4 32) %var)`	
27	`Or`	Instrukce Or slouží pro získání logického bitového součtu. Při překladu z C++ reprezentuje bitový operátor OR (v C++ `|`). Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost.
Příklad:		
LLVM	`%result = or i32 4, %var`	
SMT	`(= %result (bvor (_ bv4 32) %var)`	
28	`Xor`	Instrukce Xor slouží pro získání exkluzivního logického bitového součtu. Při překladu z C++ reprezentuje bitový operátor XOR (v C++ `^`). Pro SMT validní výraz musí mít oba operandy bitové vektory stejnou velikost.
Příklad:		
LLVM	`%result = xor i32 4, %var`	
SMT	`(= %result (bvxor (_ bv4 32) %var)`	
29	`Alloca`	Alloca instrukce alokuje paměť na zásobníku pro lokální proměnné. Alokovaná paměť je opět uvolněna po opuštění funkce. V rámci intraprocedurální analýzy nástroj neanalyzuje alokování a uvolňování lokálních proměnných při přechodu mezi více funkcemi. Do proměnné je přiřazena hodnota dostupné adresy na zásobníku. Pomocná proměnná `SP` simuluje ukazatel na vrchol zásobníku (*Stack Pointer*) a při překladu každé Alloca instrukce se navýší pomocná proměnná `SP_offset` o počet alokovaných bytů. Nová adresa se tedy rovná součtu adresy `SP` (konstantní) a hodnoty `SP_offset` (přeloží se do SMT jako bit vektorová konstanta a následně se v překladači patřičně inkrementuje).

LLVM	`%2 = alloca i32, align 4`
	`%3 = alloca i32, align 4`
	`%4 = alloca i32, align 4`	

SMT	`(= %2 (bvadd SP (_ bv0 64)))` `(= %3 (bvadd SP (_ bv4 64)))`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
	`(= %4 (bvadd SP (_ bv8 64)))`	
30	`Load`	Instrukce Load slouží pro načtení dat z paměti. Jejími parametry jsou datový typ, určující velikost dat, která se budou načítat (v příkladu i32 – 32-bitový integer, což znamená že se budou načítat 4 byty dat) a ukazatel do paměti (představuje adresu, jejíž podoba se může implementačně lišit v závislosti na použitém adresném prostoru paměti, v rámci nástroje `tindger` předpokládáme, že se jedná o lineární adresný prostor indexovaný pomocí adresy vyjádřené jako N-bitový integer). Při překladu na výraz SMT-LIB je jedna instrukce rozgenerována na N výrazů pro čtení N bytů dat po 1 bytu. Každý byte je načten zvlášť pomocí výrazu `select` nad aktuálním obrazem modelu paměti `memory_X` (pole 8-bitových bitových vektorů, model se po aplikaci operace select nemění). Pro načtení N-tého bytu se používá zvýšení hodnoty adresy originálního ukazatele o N. Pro získaný byte dat z paměťového modelu je přidána aserce na shodu s bytem cílové proměnné, do které jsou načtená data uložena (SMT-LIB výraz `extract`, který umožňuje vybrat část bitového vektoru v rozsahu zadaných bitů). Paměťový model pracuje s předpokladem, že je pro pořadí bytů využita little-endian architektura.

LLVM	`%ptr = alloca i32 ; yields i32*:ptr`
	`store i32 3, i32* %ptr ; yields void`
	`%val = load i32, i32* %ptr ; yields i32:val = i32 3`	

SMT	`(= %ptr (bvadd SP (_ bv0 64)))`
	`(= memory_1_byte_write ((_ extract 7 0) (_ bv3 32))))`
	`(= memory_2_byte_write ((_ extract 15 8) (_ bv3 32))))`
	`(= memory_3_byte_write ((_ extract 23 16) (_ bv3 32))))`
	`(= memory_4_byte_write ((_ extract 31 24) (_ bv3 32))))`
	`(= memory_1 (store memory %ptr memory_1_byte_write)`
	`(= memory_2 (store memory_1 (bvadd %ptr (_ bv1 64) memory_2_byte_write)`
	`(= memory_3 (store memory_2 (bvadd %ptr (_ bv2 64) memory_3_byte_write)`
	`(= memory_4 (store memory_3 (bvadd %ptr (_ bv3 64) memory_4_byte_write)`
	`(= (select memory_4 %ptr) ((_ extract 7 0) %val))`
	`(= (select memory_4 (bvadd %ptr (_ bv1 64))) ((_ extract 15 8) %val))`
	`(= (select memory_4 (bvadd %ptr (_ bv2 64))) ((_ extract 23 16) %val))`
	`(= (select memory_4 (bvadd %ptr (_ bv3 64))) ((_ extract 31 24) %val))`
	`; nyni plati: (= %val (_ bv3 32))`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
31	`Store`	Instrukce Store slouží pro uložení dat do paměti. Jejími parametry jsou proměnná, jejíž hodnota je ukládaná do paměti (velikost dat se odvodí z typu proměnné. V příkladu i32 – 32-bitový integer, což znamená že se budou ukládat 4 byty dat) a ukazatel do paměti (představuje adresu, jejíž podoba se může implementačně lišit v závislosti na použitém adresném prostoru paměti, v rámci nástroje `tindger` předpokládáme, že se jedná o lineární adresný prostor indexovaný pomocí adresy vyjádřené jako N-bitový integer). Při překladu na výraz SMT-LIB je jedna instrukce rozgenerována na N výrazů pro zápis N bytů dat po 1 bytu. Každý byte je uložen zvlášť pomocí výrazu `store` nad aktuálním obrazem modelu paměti `memory_X` (pole 8-bitových bitových vektorů). Po aplikaci každé operace `store`, získáváme nový obraz modelu paměti (`memory_X+1`), tato verze musí být použita ve všech následujících operacích čtení a zápisu až do získání dalšího obrazu po příštím zápisu. Pro uložení N-tého bytu se používá zvýšení hodnoty adresy originálního ukazatele o N. Pro byte dat ukládaných do paměťového modelu je přidána aserce na shodu s bytem zdrojové proměnné, ze které jsou data čerpána (SMT-LIB výraz `extract`, který umožňuje vybrat část bitového vektoru v rozsahu zadaných bitů). Paměťový model pracuje s předpokladem, že je pro pořadí bytů využita little-endian architektura.

LLVM
```
%ptr = alloca i32 ; yields i32*:ptr
store i32 3, i32* %ptr ; yields void
%val = load i32, i32* %ptr ; yields i32:val = i32 3
```

SMT
```
(= %ptr (bvadd SP (_ bv0 64)))
(= memory_1_byte_write ((_ extract 7 0) (_ bv3 32))))
(= memory_2_byte_write ((_ extract 15 8) (_ bv3 32))))
(= memory_3_byte_write ((_ extract 23 16) (_ bv3 32))))
(= memory_4_byte_write ((_ extract 31 24) (_ bv3 32))))
(= memory_1 (store memory %ptr memory_1_byte_write)
(= memory_2 (store memory_1 (bvadd %ptr (_ bv1 64) memory_2_byte_write)
(= memory_3 (store memory_2 (bvadd %ptr (_ bv2 64) memory_3_byte_write)
(= memory_4 (store memory_3 (bvadd %ptr (_ bv3 64) memory_4_byte_write)
(= (select memory_4 %ptr) ((_ extract 7 0) %val))
(= (select memory_4 (bvadd %ptr (_ bv1 64))) ((_ extract 15 8) %val))
(= (select memory_4 (bvadd %ptr (_ bv2 64))) ((_ extract 23 16) %val))
(= (select memory_4 (bvadd %ptr (_ bv3 64))) ((_ extract 31 24) %val))
; nyni plati: (= %val (_ bv3 32))
```

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
32	`GetElementPtr`	Instrukce GetElementPtr slouží k výpočtu adresy pro přístup k prvkům struktur a polí. Tato instrukce neprovádí přístup do paměti, pouze výpočet efektivní adresy! Pro překlad je potřeba provést analýzu typů a počtu dimenzí, výsledný výraz pro získání nové adresy přičítá k původnímu bázovému ukazateli vypočítaný offset. V příkladu je použit výpočet pro offset třetího prvku pole obsahující prvky typu i8 (8-bitový integer).
Příklad:		
LLVM	`%eptr = getelementptr [12 x i8], [12 x i8]* %aptr, i64 0, i32 2`	
	`; yields i8*:eptr`	
SMT	`(= %eptr (bvadd%aptr (bvmul (_ bv1 64) (_ bv2 64))))`	
33	`Fence`	Instrukce Fence slouží k synchronizaci pořadí operací. Analýza více vláknových programů není nástrojem podporována, viz 3.2.
Příklad:		
LLVM	`fence acquire ; yields void`	
	`fence syncscope("singlethread") seq_cst ; yields void`	
	`fence syncscope("agent") seq_cst ; yields void`	
SMT		
34	`CmpXchg`	Instrukce CmpXchg slouží k atomické podmíněné modifikace paměti. Atomicky načte hodnotu z paměti, porovná ji a pokud se rovná, uloží do paměti novou hodnotu. Analýza více vláknových programů není nástrojem podporována, viz 3.2.
Příklad:		
LLVM	`%vals = cmpxchg i32* %ptr, i32 %cmp, i32 %squared acq_rel monotonic`	
	`; yields { i32, i1 }`	
SMT		
35	`AtomicRMW`	Instrukce AtomicRMW slouží k atomické nepodmíněné modifikace paměti. Analýza více vláknových programů není nástrojem podporována, viz 3.2.
Příklad:		
LLVM	`%old = atomicrmw add i32* %ptr, i32 1 acquire ; yields i32`	
SMT		
36	`Trunc`	Instrukce Trunc provádí typovou konverzi celočíselných datových typů na menší datový typ. Při překladu je využita operace SMT-LIB `extract`, která vytváří nový bitový vektor z bitů zadaného bitového vektoru v požadovaném rozsahu (v příkladu: 7 – index nejvyššího bitu, 0 – index nejnižšího bitu, nový bitový vektor bude mít velikost 8 bitů).
Příklad:		
LLVM	`%x = trunc i32 257 to i8 ; yields i8:1`	
SMT	`(= %x ((_ extract 7 0) (_ bv257 32)))`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
37	`ZExt`	Instrukce ZExt provádí typovou konverzi na větší datový typ rozšířením o 0 bity. Při překladu je využita dvojice operací `extract`. Dochází k aserci bitů v rozsahu datového typu původní proměnné na převáděnou hodnotu a horních bitů na nulový bitový vektor odpovídající velikosti.
Příklad:		
LLVM	`%x = zext i32 257 to i64 ; yields i64:257`	
SMT	`(= ((_ extract 31 0) %x) (_ bv257 32))` `(= ((_ extract 63 32) %x) (_ bv0 32))`	
38	`SExt`	Instrukce ZExt provádí typovou konverzi na větší datový typ rozšířením o znaménkové bity. Při překladu je využita trojice operací `extract` a operace `repeat`. Dochází k aserci bitů v rozsahu datového typu původní proměnné na převáděnou hodnotu. Nejvyšší bit původní proměnné se nejdříve vyextrahuje jako znaménko (1-bitový vektor), rozšíří se do požadované velikosti operací `repeat` (parametr repeat je rozdíl velikostí cílového a zdrojového typu) a nakonec je přidána aserce horních bitů na rozšířený znaménkový vektor.
Příklad:		
LLVM	`%x = sext i32 257 to i64 ; yields i64:257`	
SMT	`(= ((_ extract 31 0) %x) (_ bv257 32))` `(= ((_ extract 63 32) %x) (_ repeat ((_ extract 31 31) (_ bv257 32)) 32))`	
39	`FPToUI`	Instrukce FPToUI provádí typovou konverzi floating-point hodnoty na odpovídající celočíselnou hodnotu (bez znaménka). Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%x = fptoui double 123.0 to i32 ; yields i32:123`	
SMT		
40	`FPToSI`	Instrukce FPToSI provádí typovou konverzi floating-point hodnoty na odpovídající celočíselnou hodnotu (se znaménkem). Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%x = fptosi double -123.0 to i32 ; yields i32:-123`	
SMT		
41	`UIToFP`	Instrukce UIToFP provádí typovou konverzi celočíselné hodnoty (bez znaménka) na odpovídající floating-point hodnotu. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%x = uitofp i32 257 to float ; yields float:257.0`	
SMT		
42	`SIToFP`	Instrukce UIToFP provádí typovou konverzi celočíselné hodnoty (se znaménkem) na odpovídající floating-point hodnotu. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
LLVM	`%x = sitofp i8 -1 to double ; yields double:-1.0`	
SMT		
43	`FPTrunc`	Instrukce FPTrunc provádí typovou konverzi floating-point datových typů na menší datový typ. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%x = fptrunc double 16777217.0 to float ; yields float:16777216.0`	
SMT		
44	`FPExt`	Instrukce FPExt provádí typovou konverzi floating-point datových typů na větší datový. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM	`%x = fpext float 3.125 to double ; yields double:3.125000e+00`	
SMT		
45	`PtrToInt`	Instrukce PtrToInt provádí typovou konverzi ukazatelů na celočíselný datový typ. V LLVM IR mohou být ukazatele implementovány závisle na cílové architektuře, pro generovaných SMT-LIB v rámci nástroje `tindger` jsou pro potřeby analýzy kódovány jako N-bitové bitové vektory (32-bitové, 64-bitové). Pro konverzi je využit stejný postup jako pro instrukce ZExt (pokud má cílový typ větší velikost), Trunc (pokud má cílový typ menší velikost), nebo je vytvořen alias, stejně jako u instrukce Bitcast (pokud má cílový typ stejnou velikost, jako bitový vektor reprezentující ukazatele).
Příklad:		
LLVM	`%addr = ptrtoint i32* %ptr to i8` `; yields truncation on 64-bit architecture`	
SMT	`(= %addr ((_ extract 7 0) %ptr))`	
LLVM	`%addr = ptrtoint i32* %ptr to i64` `; yields no-op on 64-bit architecture`	
SMT	`(= %addr %ptr)`	
LLVM	`%addr = ptrtoint i32* %ptr to i128` `; yields zero extension on 64-bit architecture`	
SMT	`(= ((_ extract 63 0) %addr) %ptr)` `(= ((_ extract 123 64) %addr) (_ bv0 64))`	
46	`IntToPtr`	Instrukce IntToPtr provádí typovou konverzi celočíselných datových typů na ukazatele. V LLVM IR mohou být ukazatele implementovány závisle na cílové architektuře, pro generovaných SMT-LIB v rámci nástroje `tindger` jsou pro potřeby analýzy kódovány jako N-bitové bitové vektory (32-bitové, 64-bitové). Pro konverzi je využit stejný postup jako pro instrukce ZExt (pokud má zdrojový typ menší velikost), Trunc (pokud má zdrojový typ větší velikost), nebo je vytvořen alias, stejně jako u instrukce Bitcast (pokud má zdrojový typ stejnou velikost, jako bitový vektor reprezentující ukazatele).

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
Příklad:		
LLVM	`%ptr = inttoptr i128 255 to i32*`	
	`; yields truncation on 64-bit architecture`	
SMT	`(= %ptr ((_ extract 63 0) (_ bv255 128)))`	
LLVM	`%ptr = inttoptr to i64 255 to i32*`	
SMT	`(= %ptr (_ bv255 64))`	
	`; yields no-op on 64-bit architecture`	
LLVM	`%ptr = inttoptr i8 255 to i32*`	
	`; yields zero extension on 64-bit architecture`	
SMT	`(= ((_ extract 7 0) %ptr) (_ bv255 8))`	
	`(= ((_ extract 64 8) %ptr) (_ bv0 64))`	
47	`BitCast`	Instrukce BitCast slouží ke konverzi typu, u které nedochází ke změně bitů (zdrojový i cílový typ má stejnou velikost). Jelikož jsou všechny hodnoty na úrovni výrazu SMT-LIB reprezentovaný bitovými vektory, je konverze implicitní. Pro výsledek konverze je přidán alias na původní hodnotu.
Příklad:		
LLVM	`%x = i8 255 to i8`	
	`%y = bitcast %x to i8 ; yields i8 :-1`	
SMT	`(= %x %y)`	
48	`AddrSpaceCast`	Instrukce AddrSpaceCast provádí konverzi ukazatele pro jeden adresný prostor na ukazatel pro jiný adresný prostor. Nástroj podporuje pouze analýzu lineárního paměťového prostoru, viz 3.2.
Příklad:		
LLVM	`%y = addrspacecast i32* %x to i32 addrspace(1)*`	
	; yields i32 addrspace(1)*:%x	
SMT		
49	`CleanupPad`	Instrukce CleanupPad slouží pro označení základního bloku jako *cleanup* bloku pro ošetření výjimky. Zpracování výjimek není nástrojem podporováno, viz 3.2.
Příklad:		
LLVM	`%tok = cleanuppad within %cs []`	
SMT		
50	`CatchPad`	Instrukce CatchPad slouží pro označení základního bloku jako *catch handler* bloku pro ošetření výjimky. Zpracování výjimek není nástrojem podporováno, viz 3.2.
Příklad:		
LLVM	`dispatch:`	
	`%cs = catchswitch within none [label %handler0] unwind to caller`	
	`; A catch block which can catch an integer.`	
	`handler0:`	
	`%tok = catchpad within %cs [i8** @_ZTIi]`	
SMT		

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
51	`ICmp`	Instrukce ICmp provádí porovnání dvou (celočíselných) hodnot dle zadaného predikátu. Podporované predikáty: 1. eq: je roven. 2. ne: není roven. 3. ugt: je větší než (bez znaménka). 4. uge: je větší nebo roven než (bez znaménka). 5. ult: je menší než (bez znaménka). 6. ule: je menší nebo roven než (bez znaménka). 7. sgt: je větší než (se znaménkem). 8. sge: je větší nebo roven než (se znaménkem). 9. slt: je menší než (se znaménkem). 10. sle: je menší nebo roven než (se znaménkem).
Příklad:		
LLVM	`%result = icmp eq %x, %y`	
SMT	`(= %result (= %x %y))`	
LLVM	`%result = icmp neq %x, %y`	
SMT	`(= %result (distinct %x %y))`	
LLVM	`%result = icmp ugt %x, %y`	
SMT	`(= %result (bvugt %x %y))`	
LLVM	`%result = icmp uge %x, %y`	
SMT	`(= %result (bvuge %x %y))`	
LLVM	`icmp ult i32 4, 5 ; yields: result=true`	
SMT	`(bvult (_ bv4 32) (_ bv5 32))`	
LLVM	`%result = icmp ule %x, %y`	
SMT	`(= %result (bvule %x %y))`	
LLVM	`%result = icmp sgt %x, %y`	
SMT	`(= %result (bvsgt %x %y))`	
LLVM	`%result = icmp sge %x, %y`	
SMT	`(= %result (bvsge %x %y))`	
LLVM	`%result = icmp slt %x, %y`	
SMT	`(= %result (bvslt %x %y))`	
LLVM	`%result = icmp sle %x, %y`	
SMT	`(= %result (bvsle %x %y))`	
52	`FCmp`	Instrukce FCmp provádí porovnání dvou (floating-point) hodnot dle zadaného predikátu. Floating-point typy nejsou nástrojem podporovány, viz 3.2.
Příklad:		

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
LLVM	`%result = fcmp oeq float 4.0, 5.0 ; yields: result=false`	
SMT		
53	`PHI`	Instrukce PHI implementuje Φ-uzávěr proměnných v SSA formě. Například, pokud je jedné proměnné přiřazená v různých větvích programu jiná hodnota musí se pro každou větev použít jiná proměnná (pro dodržení SSA) a pro použití proměnné po opuštění větve je třeba provést podmíněné přiřazení do nové proměnné reprezentující výběr korektní hodnoty. PHI instrukce musí být uvedeny vždy na začátku základního bloku (nepředchází jim žádné jiné instrukce, co nejsou také PHI instrukce).
Příklad:		
LLVM	`%11 = phi i1 [ true, %0 ], [ %9, %7 ]`	
SMT	`(= %11 (ite (= %0 True) %9 %7))`	
54	`Call`	Instrukce Call slouží k volání funkce. Jelikož nástroj nepodporuje interprocedurální analýzu, je zpracování této instrukce omezené. Pokud je funkce zadefinována v SMT-LIB (a jedná-li se o čistou funkci), může být její volání přeloženo jako aplikace výrazu. Příklad: `(jmeno_funkce arg1 arg2 ...)`. V případě že není funkce explicitně definována, ale pouze deklarována řešič SMT se pokusí v rámci sestavení modelu dosadit funkci vlastní vygenerovaný výraz, pro její implementaci, která se bude lišit od té skutečné.
Příklad:		
LLVM	`%retval = call i32 @foo(i32 %argc)`	
SMT	`(= %retval (foo %argc))`	
55	`Select`	Instrukce Select souží k podmíněnému přiřazení hodnoty (bez větvení základních bloků). Instrukci je možno překládat pomocí výrazu `ite`, jehož první parametr (typu bool) slouží jako podmínka.
Příklad:		
LLVM	`%cond = i1 true` `%x = select %cond, i8 17, i8 42 ; yields i8:17`	
SMT	`(= %x (ite %cond (_ bv17 8) (_ bv42 8)))`	
56	`UserOp1`	UserOp1 není skutečnou instrukcí, ale speciálním kódem operací použitým interně, v rámci průchodů překladu. Na výraz SMT-LIB se nepřekládá.
Příklad:		
LLVM		
SMT		
57	`UserOp2`	UserOp2 není skutečnou instrukcí, ale speciálním kódem operací použitým interně, v rámci průchodů překladu. Na výraz SMT-LIB se nepřekládá.
Příklad:		
LLVM		
SMT		

Transformace instrukcí LLVM IR na výrazy SMT-LIB

#	instrukce	poznámka
58	`VAArg`	Instrukce VAArg slouží pro přístup k parametrům variadických funkcí. Variadické funkce nejsou nástrojem podporovány, viz 3.2.
Příklad:		
LLVM		
SMT		
59	`ExtractElement`	Instrukce ExtractElement vrací prvek vektoru na zadaném indexu. Vektorizace není nástrojem podporována, viz 3.2.
Příklad:		
LLVM	`%element = extractelement <4 x i32> %vec, i32 0 ; yields i32`	
SMT		
60	`InsertElement`	Instrukce InsertElement vloží prvek do vektoru na zadaný index. Vektorizace není nástrojem podporována, viz 3.2.
Příklad:		
LLVM	`%newVec = insertelement <4 x i32> %vec, i32 1, i32 0 ; yields <4 x i32>`	
SMT		
61	`ShuffleVector`	Instrukce ShuffleVector vytváří permutaci z prvků dvou vstupních vektorů. Vektorizace není nástrojem podporována, viz 3.2.
Příklad:		
LLVM	`%result = shufflevector <4 x i32> %v1, <4 x i32> %v2,` `<4 x i32> <i32 0, i32 4, i32 1, i32 5> ; yields <4 x i32>`	
SMT		
62	`ExtractValue`	Instrukce ExtractValue vrací hodnotu prvku struktury nebo pole. Pro výběr prvku využívá konstantní index.
Příklad:		
LLVM	`%result = extractvalue {i32, float} %agg, 0 ; yields i32`	
SMT		
63	`InsertValue`	Instrukce InsertValue ukládá hodnotu prvku struktury nebo pole. Pro výběr prvku využívá konstantní index.
Příklad:		
LLVM	`%agg2 = insertvalue {i32, float} %agg1, float %val, 1` `; yields {i32 1, float %val}`	
SMT		
64	`LandingPad`	Instrukce LandingPad slouží k označení základního bloku jako *landingpad* – vstupního bodu pro zpracování ošetření zachycené výjimky. Zpracování výjimek není nástrojem podporováno, viz 3.2.
Příklad:		
LLVM	`; A landing pad which can catch an integer.` `%res = landingpad { i8*, i32 }` `catch i8** @_ZTIi` `; A landing pad that is a cleanup.` `%res = landingpad { i8*, i32 }` `cleanup` `; A landing pad which can catch an integer and can only throw a double.` `%res = landingpad { i8*, i32 }`	

Transformace instrukcí LLVM IR na výrazy SMT-LIB		
#	instrukce	poznámka
	`catch i8** @_ZTIi` `filter [1 x i8**] [@_ZTId]`	
SMT		

Tabulka 3.2: Instrukční sada LLVM IR ISA se vztahuje k referenční verze LLVM 7. Ukázky převzaty z referenčního manuálu jazyka LLVM verze 7.0.0 [16]. LLVM 8 rozšiřuje instrukční sadu o instrukci FNeg a vývojová verze LLVM 9 představuje novou instrukci CallBr, tyto instrukce nejsou podporovány.

3.4 Vysokoúrovňový návrh

Z pohledu vysokoúrovňového návrhu můžeme aplikaci rozdělit na logicky oddělené vrstvy a její datový model.

UI

Nejvyšší vrstva UI (*user interface*) uživatelského rozhraní slouží pro ovládání aplikace. Jejími zodpovědnostmi jsou zpracování vstupních dat (ze vstupního formátu na interní uložení do strukturovaných datových struktur), zpracování výstupních dat (prezentace) a zpracování ovládacích příkazů od uživatele. Nástroj `tindger` je navržen jako aplikace příkazové řádky bez grafického uživatelského rozhraní. Vstupy od uživatele a vstupní data může přijímat ze standardního vstupu nebo ze souboru. Textový výstup aplikace může ukládat na standardní výstup nebo do specifikovaného souboru. Pro snadné použití jinými automatizovanými nástroji je vhodné nespoléhat na interaktivní vstup od uživatele. Dodatečné příkazy je třeba uložit jako součást vstupních dat.

Aplikační vrstva

Aplikační vrstva (*bussiness logic*) má na starost řízení zpracování dat. Probíhá zde transformace modelu programu a analýza výsledku transformace. Dle výsledku analýzy se generují výrazy SMT-LIB a předávají řešiči SMT. Po zadání všech potřebných dat se spouští řešič SMT a ověřuje se jeho výsledek.

Perzistentní vrstva

Perzistentní vrstva pro ukládaní dat (do souboru, databáze) není v rámci nástroje v této fázi potřeba. Všechna vstupní data jsou zkonzumována programem v rámci jednoho běhu programu a všechna potřebná výstupní data jsou přímo vrácena uživateli. Pro budoucí rozšíření nástroje o pokročilé techniky analýzy podcest bude vhodné přidání perzistence částečných mezivýsledků pro optimalizaci výkonu.

Datový model

Datový model nástroje `tindger` obsahuje struktury potřebné pro uložení vstupních dat, dat interně zpracovávaných a výstupních dat.

Vstupní data tvoří kód LLVM IR analyzovaného programu, strukturovaný popis analyzované cesty a metadata obsahující příkazy pro optimalizaci analýzy. Kód LLVM IR je načítán ze samostatného souboru a jeho formát je dán standardem LLVM verze 7. Data popisující cestu a metadata jsou společně obsaženy v konfiguračním JSON souboru, popis je uvedeno v příloze: schéma formátu vstupních dat C.

Interní data zahrnují paměťová reprezentaci LLVM IR modelu programu (objektový model vytvořený LLVM knihovnou), transformovaný model (pomocné struktury pro uložení informací o analyzovaných proměnných, parametrech funkce, globálních proměnných). Patří zde také objekty reprezentující SMT-LIB výrazy (použity datové typy z knihovny Z3) a struktury pro jejich ukládání a organizaci.

Výstupní data tvoří JSON soubor obsahující seznam nalezených ohodnocení proměnných. Jeho popis je uveden v příloze: schéma formátu výstupních dat D.

3.5 Návrh architektury

Nástroj se skládá ze tří základních modulů -– LLVMExtractor, Model a SMT, které definují základní rozhraní používaná v aplikaci a jejich výchozí implementaci. Hlavní řídicí třída App, která se stará o zpracování vstupů a vygenerování příslušných výstupů, závisí pouze na rozhraních. Konfigurace, která implementace rozhraní se použije je provedena mimo třídu App pomocí injektování závislostí (*angl. Dependency Injection*) – návrhového principu inverze řízení (angl. Inversion of Control) s využitím knihovny [Boost].DI (viz [9]). O zpracování parametrů programu, které se pak zpracované na objekt AppData předají objektu App to spuštění analýzy slouží třída ArgParser, která využívá knihovnu getopt. Program může být spuštěn s parametrem `--input` (nebo zkráceně `-i`) a načíst IR kód pro analýzu ze souboru podle zadané cesty, pokud není tento parametr zadán, předpokládá načtení IR kódu ze standardního vstupu. Soubor se řídicími daty (seznam stop k analýze) je předán ve formátu JSON, cesta k němu je zadána parametrem `--config` (zkráceně `-c`).

Modul LLVMExtractor

Modul LLVMExtractor obsahuje třídy použité k rekonstrukci LLVM IR paměťového modelu ze zadaného kódu na vstupu, manipulaci s ním a extrakci požadovaných modelových dat pro další zpracování. Třídy v tomto modulu úzce závisí na třídách LLVM, se kterými pracují přes jejich C++ API. Práce s modelem je prováděna nad zrekonstruovaným modelem, tak jak byl vytvořen LLVM při překladu. Modul Model obsahuje třídy, které umožňují modelování a manipulaci s daty získanými z LLVM modelu, ale přímo nezávisí na LLVM třídách. Umožňují přípravu dat potřebných pro následné generování kódu SMT-LIB avytvoření paměťového modelu pro potřeby generování. Pomocné třídy pomáhají vytvářet a průběžně aktualizovat kontext generování výstupního kódu, například při opakovaném průchodu stejným základním blokem využít místo původní proměnné jejich očíslovanou verzi tak, aby zůstala zachována SSA forma 2.5.1 a ve vygenerovaných formulích se použily správně označené proměnné.

Modul Model

Modul Model obsahuje třídy pro podporu analýzu transformovaného modelu cesty a připravuje data pro přípravu dat pro generování výrazů SMT-LIB (které probíhá v modulu SMT). Centrální komponentou tohoto modulu je třída `LLVMInstructionVisitor`, která

vychází z návrhového vzoru visitor a používá se k inspekci jednotlivých instrukcí a rozhodování jestli se pro danou instrukci bude generovat výraz SMT-LIB (a která data se SMT vrstvě předají), či ne.

Modul SMT

Modul SMT obsahuje třídy, které se starají o vygenerování cílového kódu SMT, který je následně vyhodnocen řešičem SMT a výsledek je zpracován pro další zpracování nástrojem. Poskytuje základní obecné rozhraní, na kterém závisí třída `App`, nezávislé na konkrétním řešiči SMT, a také výchozí implementaci pro konkrétní SMT řešič – Z3 (viz 2.3.3).

3.6 Návrh zpracování LLVM modelu

V této podkapitole přiblížím varianty návrhu modelu, které jsem v návrhové části prostudoval a srovnal.

LLVM neměnný model a vlastní modelová mezivrstva:

První zvažovaný způsob byl vytvořit LLVM paměťový model (angl. in-memory model) na základě načteného IR kódu pomocí LLVM knihovny IRReader. K načtenému LLVM modelu by nástroj přistupoval (po potřebném přidání jmen proměnných a labelů základních bloků) jako k neměnnému (angl. imutable) objektu, a pro potřeby modifikace modelu a samotné generování kódu SMT-LIB by si vytvářel vlastní model. Tento přístup by měl výhodu v možné budoucí podpoře jiných forem mezikódu (např. mezikódu překladače GCC) a odizolování části nástroje generujícího SMT od závislosti na LLVM implementaci. Zpočátku se tento přístup jevil jako nejvhodnější, ale po prostudování všech prvků LLVM, které by musel podporovat (například rozsáhlý typový systém) a znovu implementovat se ukázal jako nevhodný.

LLVM model transformovaný v LLVM průchodu:

Druhý přístup počítal s prováděním analýzy přímo nad LLVM modelem, včetně potřebných transformací (eliminace základních bloků mimo stopu programu, rozbalení smyček dle stopy, odstranění větvení a PHI uzlů) přímým editováním LLVM objektů jejich přidáváním a odstraňováním. Tento postup je vhodný, pokud by se nástroj implementoval jako plugin LLVM překladače (např. clang) a modifikace nad IR modelem by prováděl v LLVM průchodu (LLVM Pass), který je připraven pro rozšíření optimalizací a analýzy překládaných programů, a je možné cílit ho na průchod modelu na určité úrovni (funkce, modulu, smyčky...). Tento přístup by byl nejblíže přístupu, který doporučuje tým vyvíjející LLVM, nicméně byl by na něj příliš vázaný a cílem projektu je vytvořit nástroj, který bude od LLVM vyžadovat jen validní výstupy a nebude jeho přímým rozšířením.

LLVM model a tenká mezivrstva pro ukládání kontextu generování:

Třetí varianta využívá LLVM model jako jediný neměnný model, ale pro potřeby generování k němu přidává nadstavbu pomocného kontextu, který se v průběhu generování mění. Takto je model procházen pomocí návrhového vzoru visitor (který je LLVM objekty podporován) a mezi procházení modelových objektů aktualizuje záznamy v pomocných třídách. Pomocné třídy jsou spolu s referencí na konstantní objekt modelu zapouzdřeny v jedné struktuře

nazvané `ModelAnalyseData`. Obsahuje mimo jiné hašovací tabulku pro sledování aktuálních verzí proměnných (řeší problém, jak mapovat vícenásobně zapsanou proměnnou na úrovní jazyka SMT-LIB, například při vícenásobném průchodu stejným základním blokem v cyklu), registr typů, které byly skutečně použity a bude je třeba deklarovat při generování kódu SMT-LIB, a hlavně paměťový model, který se stará o správné zpracování instrukcí s přístupem do paměti. Modul, který se stará o generování kódu SMT-LIB je tedy částečně odizolován od modelu a pracuje s výsledky visitor objektů (prochází dle zadané stopy programu jednotlivé instrukce ve všech základních blocích cesty stopy) a kontextem generování. Tento přístup odstraňuje problém vytváření druhého modelu duplikujícího logiku a složitost původního LLVM modelu a zároveň je efektivní díky podpoře využití návrhového vzoru visitor v objektech LLVM modelu (i pro načtené kompletní modely). Velikost ukládaných dat pro udržení kontextu může růst lineárně s počtem analyzovaných instrukcí, ale bude zabírat minimální potřebné množství paměti, na rozdíl od nově vytvořené modelové mezivrstvy, která s sebou nese nároky na režii (komplexní hierarchie objektů a jejich propojení). Nevýhodou je, že optimalizace (např. nepoužité proměnné, použité v jiné větvi, než navštíví zkoumaná stopa programu) vyžaduje druhý průchod a pokud jsou data pro řešič SMT generována již v prvním průchodu, musí se filtrovat při dalších průchodech (z pohledu výkonu a zvýšení šance na nalezení vhodného řešení je potřeba minimalizovat datový model předávaný do SMT řešiče). Jednoduché smyčky mohou být optimalizovány rozbalením, anebo z pohledu řešiče SMT deklarováním pomocného uzávěru nad tělem cyklu, který bude v SMT kódu deklarován jako rekurzivní funkce a její výsledek bude aplikován na finální verzi proměnné – tento způsob optimalizace eliminuje hlavně u jednoduchých cyklů s vysokým počtem iterací do velké míry nároky na řešič SMT, který má problémy s velkým počtem proměnných (které by jinak musel udržovat pro každou rozbalenou iteraci), ale vyžaduje analýzu datových závislostí. Také pokud je v cyklu zapisováno do více (datově nezávislých) proměnných, je třeba vygenerovat více pomocných rekurzivních funkcí. V případě že mezi zapisovanými proměnnými v těle cyklu existují datové závislosti nebo dochází k přístupu do paměti není vytvoření této pomocné rekurzivní funkce v této fázi návrhu možné.

Tuto variantu jsem zvolil jako nejvhodnější možnost pro implementaci zpracování modelu programu a použil ji při implementaci nástroje.

Úprava LLVM IR v textové podobě:

Tuto alternativní variantu jsem krátce prozkoumal během objevení implementačních problémů s transformací LLVM objektového modelu přidáváním a odebíráním IR objektů.

Pro vyhnutí se přímé manipulaci s objektovým modelem by se prováděly požadované úpravy (rozgenerování smyček, eliminace nevyužitých instrukcí) na úrovní textové reprezentace analyzovaného LLVM IR kódu. Nejprve by se tedy kód zanalyzoval, podle výsledků by se provedla úprava textové reprezentace, která by se uložila do dočasného souboru a z něj se znovu načetla pro vytvoření nového, již transformovaného, objektového modelu programu. Přestože by se jednalo o zajímavý přístup k řešení, stále by zde zůstala potřeba složité analýzy celého modelu před vytvořením sady úprav (které by mohlo být problematické v některých případech namapovat z objektového modelu zpět na správné místo v textové reprezentaci) a proto se tato varianta hlouběji nezkoumala.

3.7 Generování kódu SMT-LIB:

U generování kódu SMT-LIB bylo třeba zvážit, zda generovat zápis ve formátu SMT-LIB pro řešič do textové reprezentace a předat ho samostatně spuštěnému řešiči, anebo využít dostupné knihovny pro práci s konkrétními řešiči a generovat SMT kód pomocí jimi nabízených rozhraní. Zhodnotil jsem dva nejdůležitější požadavky – zda a jak bude nástroj podporovat různé SMT řešiče a jak bude probíhat generování komplexních struktur z IR kódu. Podpora různých SMT řešičů je problematická kvůli různé úrovni podpory standardu SMT-LIB u jednotlivých SMT řešičů, což nemusí být problém při ručním zápisu formulí pro konkrétní cílový SMT řešič, ale při automatickém generování se jedná o komplikaci. Pod komplexními strukturami IR kódu (pro generování) můžeme zahrnou podporovaný typový systém, rekurzivní volání funkcí, struktury s dynamickými prvky a podobně. Hlavním kritériem pro výběr podporovaného SMT řešiče ve výsledku byla podpora knihoven a dokumentace – a zde zcela převládl řešič z3 (viz kapitola 2.3.3) nad ostatními. Z3 nabízí knihovny pro práci s řešičem pro jazyk C, tak pro jazyk C++. Do jisté míry lze obě knihovny kombinovat (při použití v C++ projektu).

V rámci rozsahu této práce bude implementována podpora pouze pro řešič Z3, ale pro možnost přidání podpory jiného řešiče bude implementována vrstva abstraktního rozhraní pro obecné použití libovolného SMT řešiče. Pro přidání podpory nového druhu řešiče tedy bude stačit přidat speciální třídu implementující toto rozhraní -– řadič pro konkrétní řešič (a změnit výchozí implementace SMT rozhraní v konfiguraci DI kontejneru).

Výrazy budou generovány dle navržené překladové tabulky 3.3.

3.8 Návrh vstupního a výstupního formátu

Vstupní formát

Vstupní data jsou nástroji předána ve formátu JSON dodržujícího schéma JSON input schema viz příloha C. Na nejvyšší úrovni obsahuje objekt `function`, který v sobě ukládá seznam stop k analýze pro danou funkci. Objekt function má dvě položky – `name` (řetězec, obsahuje jméno analyzované funkce, ve stejném formátu jako je uloženo v textové podobě LLVM IR kódu kde se nachází jméno funkce po provedení dekorace jmen C++ funkcí (*angl. name mangling*), např. `_Z3fooii` u funkce `foo` typu integer se dvěma parametry typu integer) a `paths` (pole objektů stop). Objekt stopy obsahuje tři prvky: `tag` (řetězec, sloužící jako unikátní identifikátor stopy, využívá se pro spárování výsledků s konkrétní stopu), `metadat` (pole řetězcových příkazů pro zefektivnění analýzy) a `path`, který obsahuje seznam celočíselných indexů základních bloků analyzované stopy. První index v seznamu musí být index 0, jakožto vstupní bod do funkce (analýza podcest v rámci funkce zatím není plně podporována). Pro zadané indexy musí platit, že funkce obsahuje základní blok na daném indexu a že její CFG 2.1 diagram umožňuje pro každý indexovaný základní blok přechod na základní blok s následujícím indexem v kolekci. Pokud vstupní datová struktura nesplňuje tyto podmínky, nemůže nástroj najít správné řešení.

Výstupní formát

Vstupní data jsou nástroji předána ve formátu JSON dodržujícího schéma viz příloha JSON output schema D. Na nejvyšší úrovni obsahuje kolekci záznamů objektů výsledku analýzy stop. Záznam výsledku stop obsahuje buďto záznam `error` (řetězec) obsahující popis zprávy

o chybě analýzy nebo kolekci obsahující jednotlivé záznamy zakódovaných vstupních hodnot. Záznamy vstupních hodnot obsahují zakódovaný název: pokud se jedná o argument funkce, má název ve tvaru arg_X, kde X je index pořadí argumentu v seznamu argumentů funkce, pokud se jedná o globální proměnou, má název odpovídající svému zakódovanému názvu v LLVM IR kódu, pokud se jedná o hodnotu v paměti, je ji přiřazena symbolická hodnota ve tvaru `memory_X_@Y_offset_Z`, kde X je index obrazu modelu paměti(*memory snapshot*) a `@Y_offset_Z` je symbolická adresa odpovídající LLVM IR proměnné použité pro přístupu do paměti – Y je jméno proměnné (ukazatele) použitého pro přístup do paměti (LLVM IR jméno proměnné) a Z je index bytu (v rozsahu 0 až $N - 1$ pro proměnnou o velikosti N bytů). Záznamy vstupních hodnot dále obsahují informaci o typu (odpovídá LLVM IR typu) a do textově zobrazitelné podoby serializovanou hodnotu získanou z SMT řešiče.

Kapitola 4

Implementační detaily nástroje Tindger

V této kapitole detailně přiblížím realizaci návrhu nástroje `tindger` z předešlé kapitoly 3. Nástroj `tindger` je realizován jako spustitelný program ovládaný přes rozhraní příkazové řádky. Aplikace byla implementována v programovacím jazyce C++ a extenzivně využívá knihovny LLVM a Z3. Je rozdělená do tří modulů `LLVMExtractor` (obsahuje třídy zodpovědné za přípravu vstupních dat s LLVM IR 2.5.3 kódem programu do vhodného objektového modelu, použitelného pro další zpracování, podrobně popsán v podkapitole 4.2), `Model` (obsahuje třídy starající se o analýzu modelu programu, podrobně popsán v podkapitole 4.3), `SMT` (obsahuje třídy pro vytváření SMT výrazů pro popis programu a jejich zadání řešiči SMT, podrobně popsán v podkapitole 4.4) a modulu `App` (podrobně popsán v podkapitole 4.1), do kterého spadají třídy pro zpracování příkazů od uživatele a řízení běhu výpočtu (předávání získaných zpracovaných dat mezi moduly).

4.1 Kostra aplikace

`App` modul tvoří nejvyšší vrstvu aplikace. Obsahuje stejnojmennou třídu `App` (centrální třída aplikace), která řídí zadávání zpracování dat a průběh výpočtu (v rámci jednoho běhu programu je možno analyzovat více zadaných cest programu - v rámci jedné analyzované jednotky - funkce - tato třída řídí pořadí jejich analýzy a sloučení výsledků do jednoho výstupního souboru).

Přehled tříd modulu:

- `App` – třída koordinující spolupráci specializovaných modulů (závisí na rozhraních modulu, konkrétní implementace lze překonfigurovat) na získání hledaného výsledku – seznamu vygenerovaných hodnot.
- `ArgParser` – třída pro zpracování argumentů příkazové řádky.
- `ConfigParser` – třída pro parsování vstupních konfiguračních dat v JSON formátu.
- `AppStreams` – pomocná třída spravující vstupně-výstupní proudy aplikace.
- `Diagnostic` – pomocná třída pro měření délky výpočtu aplikace.
- `AppParams` – struktura pro uložení zpracovaných parametrů příkazové řádky.

- `ExtractionPath` – struktura pro uložení zadaní cesty pro analýzu.

Mimo modul `App` stojí soubor main.cpp, který obsahuje jediný vstupní bod programu (funkci `main`). Dochází zde ke zpracování řídicích argumentů příkazové řádky spuštění programu a konfiguraci dependency injection kontejneru (Boost.DI [9]). DI kontejner spravuje závislosti aplikace, tak, že váže třídy rozhraní na konkrétní implementující třídy (případně na konkrétní instance). Stará se o vytváření objektů (vytváří objekt `App` s nakonfigurovanými závislostmi).

4.2 Zpracování LLVM IR dat

Tato část projektu je již v této fázi navržená a implementována. Modul `LLVMExtractor` obsahuje definici rozhraní `LLVMExtractorInterface` (které by se již neměla v budoucnu příliš měnit, pouze je možné přidat novou implementaci tohoto rozhraní) a jeho výchozí implementaci ve třídě `LLVMExtractor`. Hlavní funkcí rozhraní `LLVMExtractor` (a celého modulu) je zpracování vstupního souboru se zdrojovým kódem LLVM IR na jeho objektový model překladové jednotky (LLVM IR modulu). Model je náležitě upraven a poté vrácen zabalen ve *wrapper* objektu třídy `ModuleSelector`. Zbytek aplikace pracuje s LLVM IR modelem skrze aplikační rozhraní tohoto wrapperu a pro získávání informací o konkrétní funkci `FunctionSelector`. Třídy ModuleSelector a FunctionSelector nabízí jednoduché a minimalistické rozhraní pro výběr objektů z LLVM IR modelu pro analýzu.

Přehled tříd modulu:

- `LLVMExtractorInterface` – veřejné rozhraní modulu pro zprostředkování převodu textové reprezentace mezikódu LLVM IR na objektový model.
- `LLVMExtractor` – výchozí implementace rozhraní `LLVMExtractorInterface`.
- `LLVMModuleSelectorInterface` – veřejné rozhraní modulu pro *wrapper* poskytující k objektům objektového modelu zpracovaného kódu LLVM IR.
- `ModuleSelector` –výchozí implementace rozhraní `LLVMModuleSelectorInterface`.
- `FunctionSelector` – *wrapper* zapouzdřující přístup k objektům argumentů funkce a základních bloků funkce.
- `BasicBlockLabeler` – třída, která se stará o přiřazení názvu proměnným v modelu LLVM IR (a jmen návěští základních bloků).
- `TypeConverter` – třída pro převod původních LLVM IR typů na zjednodušenou sadu typů podporovaných nástrojem `tindger`.
- `PathView` – struktura pro reprezentaci analyzované cesty programu obsahující seznam základních bloků cesty a rozhraní pro přístup k nim.

Získání objektového modelu LLVM IR

Získání objektového modelu LLVM IR je provedeno třídou `LLVMExtractor`. `LLVMExtractor` obsahuje metodu `extractIRModule`, která přijímá jako parametr řetězec obsahující cestu k souboru s kódem LLVM IR. Obsah souboru je načten do dočasného řetězce. Načtený řetězec je zpracován pomocí knihovní funkce z knihovny LLVM `llvm::parseIR`. Objektům v modelu je následně potřeba přiřadit názvy a poté může být zabalen do wrapperu `ModuleSelector` (ten jediný opouští tento modul).

Přiřazení názvů proměnných

Zparsovaný LLVM IR model neobsahuje názvy proměnných, návěští základních bloků a dalších objektů nacházejících se v modelu. Pro potřeby serializace do textové reprezentace LLVM IR jsou *ad-hoc* generovány, ale při deserializaci se nezachovají. Pokud chceme dále v programu pracovat s názvy proměnných, je třeba je vygenerovat znovu a přiřadit objektům modelu se kterým pracujeme.

Třída `BasicBlockLabeler` je použita pro vygenerování jmen objektů LLVM modelu. Původní implementace vycházela z třídy LLVM knihovny `llvm::CFGPrinter` tak, aby generovala stejné názvy jako při exportu IR kódu z paměťové reprezentace do textové reprezentace. Knihovna interně využívá čítač, který ovlivňuje hodnotu vygenerovaného názvu v závislosti na pořadí zpracování objektů. Základní objekt `llvm::Value`, od kterého jsou pojmenovávané objekty odvozeny obsahuje metodu `printAsOperand`, která je použita právě při převodu do textová podoby třídou `llvm::CFGPrinter`. Pro převod je potřeba použít pomocný stream `llvm::raw_string_ostream`, přes který je možné poslat vygenerovaný název do standardního řetězce `std::string`.

Na rozdíl od názvu v `.ll` souboru s textovou reprezentací obsahují takto vygenerované názvy nulu na začátku názvu navíc. Například pro proměnnou, která má v textové podobě název `"%3"` se tímto postupem vygeneruje název `"%03"`. Pro zachování konzistentních názvů je tedy potřeba provést jeden dodatečný krok, ve kterém je pro každý nově vygenerovaný název nahrazen prefix názvu `"%0"` za `"%"` bez nadbytečné nuly.

Ověření syntaktické dostupnosti cesty

Na začátku analýzy máme dostupnou dvojici jména funkce a seznamu indexů základních bloků cesty. Před zahájením zkoumání sémantické dostupnosti je třeba ověřit, že je cesta dostupná syntakticky.

První krok je ověření, že model skutečně obsahuje zadanou funkci (je možné použít metodu `LLVMModuleSelectorInterface::function`, která vrátí platný `FuctionSelector`, pokud funkci v modelu najde).

Druhým krokem je ověření přechodů mezi základními bloky. Platí, že první index cesty musí být roven 0 a vést tedy na základní blok obsahující vstupní bod funkce. Pro každý další index pak musí platit, že daná funkce obsahuje základní blok na daném indexu (ověříme metodou `FunctionSelector::basicBlock`) a že tento základní blok je v seznamu následníku předcházejícího bloku cesty. Z terminátoru (ukončující instrukce) základního bloku je možné získat seznam následníku (aplikace metod `llvm::BranchInst::getNumSuccessors` a `llvm::BranchInst::getSuccessor(i)`). Následnost bloků ověřím pro všechny bloky v cestě až na poslední. Pro poslední základní blok cesty ověříme že nemá následníka, ale končí terminátorem `Ret` (viz instrukce Ret).

Vytvoření náhledu cesty

Nezbytnou součástí vytvoření náhledu cesty (*angl. path view*), reprezentovaného objektem třídy `PathView`, je výběr základních bloků analyzované cesty. Ze seznamů indexů bloků, který je zadán uživatelem, se přes `FunctionSelector` vyberou ukazatele na odpovídající bloky ležící v cestě a uloží se v pořadí cesty do seznamu (v implementaci použit standardní kontejner `std::vector<llvm::BasiBlock*>`). Při analýze nad náhledem cesty se pak již prochází tento nový seznam obsahující pouze základní bloky cesty. Tím dojde ke dvěma vedlejším efektům – jednak jsou vyfiltrovány základní bloky funkce nepatřící do analyzované

cesty (a jejich instrukce se nebudou dále analyzovat a generovat na SMT-LIB výrazy), a také dojde k rozbalení smyček. V případě rozbalení smyček je třeba ošetřit zachování SSA formy. Pro bloky, které jsou navštíveny v průchodu cesty vícenásobně by mohlo dojít k vícenásobnému zápisu do stejné proměnné, což by SSA formu porušilo. Náhled cesty je předán do modulu Model, kde se provede odstranění duplikátů.

4.3 Vrstva pro analýzu LLVM modelu

`Model` modul se zaměřuje na analýzu modelu cesty a přípravu dat pro generování výrazů SMT-LIB, které se provádí v modulu SMT. Hlavní třídou tohoto modulu je třída `LLVMInstructionVisitor`, která provádí analýzu LLVM instrukcí rozbalené cesty. Třída `LLVMInstructionVisitor` využívá objekt třídy `Context` pro udržování kontextu analýzy, také zprostředkovává visitorovi přístup k řadiči řešiče SMT (viz popis modulu SMT 4.4) a přístup k náhledu cesty.

Přehled tříd modulu:

- `LLVMInstructionVisitor` – centrální třída modulu, analyzující jednotlivé instrukce cesty. Využívá návrhový vzor visitor (dědí od LLVM třídy `llvm::InstVisitor`).
- `Context` – třída držící kontext analýzy a zpřístupňující SMT API visitoru.
- `VariablesRegistry` – třída zaznamenávající analýzu proměnných a určující, která verze proměnných bude použita pro generování výrazu SMT-LIB.
- `AbstractModelFactory` – rozhraní pro továrnu vytvářející Model.
- `ModelFactory` – výchozí implementace rozhraní pro továrnu vytvářející Model.
- `Model` – třída, modelující analyzovaný program, pro vygenerování výrazů SMT-LIB. V současné verzi se však využívá pouze pro registraci argumentů analyzované funkce a globální proměnné (lokální proměnné a instrukce registruje přímo visitor).
- `Instruction` – struktura pro uložení zjednodušených dat instrukce.
- `Value` – struktura pro uložení dat potřebných pro registraci proměnné z modelu do SMT.
- `LLVMValuePrinter` – pomocná třída pro serializaci LLVM objektů (pro možnost jejich výpisu jako řetězce).

Třídy zjednodušeného typového systému podporovaného analýzou:

- `Type` – bázová třída pro tindger typy. `Tindger` používá pro analýzu zjednodušené typy (oproti LLVM, některé nejsou podporovány).
- `IntegerType` – základní datový typ reprezentující celá čísla o zvolené bitové šířce.
- `PointerType` – Tindger datový typ pro analýzu paměťových ukazatelů.
- `BooleanType` – Tindger datový typ pro reprezentaci logických (booleových) hodnot (true or false).
- `VoidType` – Tindger datový typ pro reprezentaci prázdného návratového typu.
- `UnsupportedType` – Tindger datový typ pro reprezentaci nepodporovaných datových typů (např. floating-point typ).

Odstranění duplikátů

Při transformaci do SMT jsou proměnné původního kódu vyjádřeny jako SMT výrazy – konstanty. SMT konstanty jsou pevně vázány na jednu hodnotu (ať již explicitně zadanou, nebo nalezenou řešičem v průběhu vytváření SMT modelu). Použití LLVM IR splňující SSA formu umožňuje přímý převod se zachováním názvů proměnných (LLVM IR proměnná %3 bude v SMT výrazu reprezentována konstantou s názvem %3). Během vytváření náhledu cesty však může dojít k rozbalení smyček a porušení SSA formy. Pro vyřešení tohoto problému je zavedena pomocná struktura `VariablesRegistry`, sloužící jako centrální registr analyzovaných proměnných v kontextu překladu celé cesty. Registr je implementován pomocí mapy obsahující původní název proměnné jako klíč a počet zápisů této proměnné jako hodnotu. Při analýze LLVM instrukce je tak možné z LLVM objektu proměnné získat původní název proměnné a pomocí registru získat nový název, který bude využit v SMT výrazu. Generované verze proměnných označujeme jako *primes*. Pro čtení hodnoty se použije poslední vygenerovaný název. Pro zápis hodnoty se vždy generuje nový název. Je potřeba zachovat pořadí že nejdříve se při analýze instrukce generují všechna čtení hodnot proměnných (typicky operandů) a až následně se vygeneruje název pro zápis (pro proměnnou pro uložení výsledku instrukce). Pokud byla proměnné přiřazena hodnota pouze jednou, použije se originální název. Byla-li proměnné přiřazena hodnota dvakrát čí více, je jí vygenerován název ve tvaru `%originalinnazev_X`, kde X představuje počet registrovaných zápisů do proměnné. Po každém generování názvu pro zápis se navýší čítač proměnné v registru. Použití tohoto postupu zaručí prevenci konfliktů proměnných (dodržením SSA formy) ve vygenerovaném SMT zápisu, přidává však další krok nutný pro správnou interpretaci finálních výsledků nástroje.

Inspekce instrukcí třídou LLVMInstructionVisitor

Třída `LLVMInstructionVisitor` provádí analýzu instrukcí náhledu cesty, jednotlivě, v pořadí tak jak jsou v náhledu cesty uvedeny. Nejprve dochází k rozpoznání, jedná-li se o podporovanou instrukci, nebo je třeba vrátit chybu, protože analyzovaný kód obsahuje nepodporovanou instrukci. `LLVMInstructionVisitor` vychází z návrhového vzoru vizitor (dědí od třídy z knihovny LLVM `llvm::InstVisitor`) a podle toho přistupuje ke zpracování instrukce, která je ji v rámci metody `visit` předána. Nejprve se snaží rozpoznat instrukci podle obecné kategorie a delegovat ji na specializovanou metodu obsluhy této kategorie. Mezi kategorizovaná zpracování patří visit metody: `visitBinaryOperator`, `visitBranchInst`, `visitCmpInst`, `visitMemoryInst`, `visitCastInst`, na úrovni kategorie jsou instrukce zpracovány již podle konkrétního operačního kódu (případně obecné části zpracováni jsou provedeny společně, viz binární operátory). Ne všechny instrukce, které se podaří přiřadit do kategorie jsou podporovány a tak i zde probíhá filtrování nepodporovaných instrukcí. Pokud je instrukce podporována jsou analyzovány její operandy a proměnná pro zápis výsledku operace. Pro uvedené proměnné jsou získány nové názvy s využitím `VariablesRegistry`. Pak je vybráno konkrétní API služby `SMTDriver` a je mu zadáno přidání výrazu instrukce. Pro některé instrukce, jako jsou například instrukce skoku, je třeba mít přístup k informacím nejen o aktuální instrukci (a jejím základním bloku), ale i ostatním základním bloku cesty, pro tyto účely využívá visitor svého modelového kontextu v objektu třídy Context (který zprostředkovává přístup k objektům `SMTDriver`, `VariablesRegistry` nebo `PathView`).

4.4 Generování kódu SMT-LIB

Generování kódu SMT-LIB je primární úloha modulu `SMT`. Jako veřejné rozhraní modulu slouží `SMTDriver` (abstraktní třída). Konkrétní použitou implementací rozhraní SMTDriver je třída `Z3Driver` implementující rozhraní řadiče SMT pro konkrétní zvolený řešič SMT Z3 (popsán v kapitole 2.3.3). Z3Driver zahrnuje rozhraní pro práci s paměťovým modelem `MemoryMangementModel` a jeho referenční implementaci `AlignedMemoryModel` postavou na modelování paměti se zarovnaným přístupem. Paměť je modelována jako jednotný adresný prostor pomocí SMT pole 8-bitových bitových vektorů. Toto pole je indexováno bitvektorem o zvolené velikosti (podpora pro 32-bitové nebo 64-bitové ukazatele, dle cílové architektury).

Ve vrstvě pracující s SMT řešičem je v modulu SMT nachystáno rozhraní pro přístup k řadičům řešičů SMT – SMTDriver – a jeho implementace pro řešič z3 (třída Z3Driver). Aktuálně pracuje s C++ verzí z3 knihovny a umožňuje přidat výraz v objektu Expr do kontextu řešiče, ověřit splnitelnost jeho modelu a v případě získat hodnoty hledaných proměnných v textové formě (zatím hledá pouze hodnoty argumentů funkce, pokud se jedná o proměnné typu integer, který převádí na bitový vektor). Pro generování výrazu bude využívat třídy odvozené od llvm::InstVisitor (implementované zvláště pro jednotlivé typy instrukce LLVM IR ISA), které umožňuje LLVM objektový model přijmout jako visitory pro objekty svých IR instrukcí. U těchto visitor objektů musí být zajištěna vazba na kontext překladu, který musí být průběžně aktualizován. Jedna instrukce z LLVM modelu se pak přeloží na jeden Expr objekt výrazu pro řešič SMT, dodatečně musí být před spuštěním deklarovány použité datové typy.

Třídy modulu:

- `SMTDriver` – veřejné rozhraní modulu pro práci s obecným API řešiče SMT (není vázané na konkrétní řešič).
- `Z3Driver` – výchozí implementace rozhraní `SMTDriver` implementovaná pro řešič Z3.
- `Z3Context` – třída pro zapouzdření kontextu SMT výrazů. Je úzce vázaná na Z3Driver, který zde registruje výrazy SMT-LIB.
- `Sort` – struktura pro uložení dat typu proměnné pro SMT (bitové vektory).
- `ResultModel` – struktura pro uložení výsledků řešiče (seznamu proměnných a jejich ohodnocení), případně chybové zprávy. Také obsahuje řetězec s kódem SMT-LIB, který byl použit pro řešič SMT.

Paměťový submodul:

- `MemoryManagementEngine` – obecné rozhraní paměťového modelu obsahují API pro přístup do paměti (operace čtení a zápisu dat). Vztahuje se k implementaci pro řešič Z3 (používá ve svých metodách parametry typu `z3::expr`).
- `AlignedMemoryModel` – výchozí implementace rozhraní `MemoryManagementEngine`, modeluje paměť pomocí pole 8-bitových bitových vektorů, které je indexováno bitovým vektorem (o zvolené velikosti 32/64 bitů).

Pomocné datové struktury třídy Z3Driver

- `primedVariables` – vektor SMT výrazů (typu `z3::expr`), slouží k organizovanému ukládání výrazů konstant reprezentující proměnné (index na kterém jsou uloženy je

spravován v mapě `variablesIndexes`), pokud je proměnná použita (pro čtení), je její výraz získán z tohoto vektoru a nemusí se znovu vytvářet.

- `constExpressions` – vektor SMT výrazů (typu `z3::expr`), slouží pro ukládání analyzovaných konstant, které se mohou použít například při vynucení návratové hodnoty volané funkce na konkrétní hodnotu.
- `variablesIndexes` – mapa pro ukládání indexů výrazů proměnných (klíč `std::string` – název proměnné, hodnota `uint32_t` – index proměnné ve vektoru `primedVariables`).
- `functionDeclarationIndexes` – mapa indexů výrazů deklarací funkcí.
- `functionDefinitionsIndexes` – mapa indexů výrazů definic funkcí.
- `declaredFunctions` – vektor SMT výrazů (typu `z3::func_decl_vector`), slouží k ukládání výrazů deklarace funkcí.

Generování SMT-LIB výrazů

SMT-LIB výrazy jsou generovány přes C++ API knihovny Z3. Jsou reprezentovány objekty typu `z3::expr`. Všechny objekty `z3::expr` jsou vázany na svůj kontext (objekt `z3::context`). Z3 API umožňuje kombinovat jednodušší výrazy pomocí svých funkcí do složených výrazů (také typu `z3::expr`). Přetěžování operátorů v C++ umoňuje přímočarý zápis pro skládání složitějších výrazů z jednodušších (například proměnných obsahujících výrazy konstant bitových vektorů spojit pomocí operátoru `'+'` do výrazu `bvadd`).

```
1 z3::context context = z3::context();
2 z3::expr a = context.bv_val(42, 8); // a - bitovy vektor o velikosti 8 bitu s
      hodnotou 42
3 z3::expr b = context.bv_val(1, 8); // b - bitovy vektor o velikosti 8 bitu s
      hodnotou 1
4
5 z3::expr c = a + b;
6 // c == (bvadd (_ bv42 8) (_ bv1 8))
7 // c bude vyhodnoceno jako (_ bv43 8)
```

Vytvořené objekty `z3::expr` jsou ukládány do vektoru v objektu třídy `Z3Context` a po dokončení analýzy jsou odsud nahrány do objektu `z3::solver`.

Seznam metod rozhraní `SMTDriver` pro vytváření SMT výrazů:

- `addArg` – vytvoří výraz pro argument analyzované funkce.
- `addAlias` – vytvoří výraz pro aserci stejné hodnoty ve dvou proměnných (SMT konstantách) – alias. Příklad: `(= %original %aliased)`.
- `addExtensio` – slouží pro vytváření výrazů odpovídacích přetypování a změny velikosti bitových vektorů (zmenšení, zvětšení rozšířením se znaménkem, zvětšení rozšířením bez znaménka).
- `addBinaryOperator` – vytváří výrazy pro binární operátory (aritmetické a logické instrukce), zároveň registruje výrazy pro použité proměnné.
- `addPredicate` – vytváří výrazy pro práci s predikáty.

- `assertCondition` – vytváří výrazy pro aserci vyhodnocení podmínky na požadovanou hodnotu.
- `addVariableAllocation` – vytváří výrazy pro alokaci proměnných (na zásobníku).
- `addStore` – vytváří výraz pro zápis dat do paměti (delegace zpracování operace na `MemoryMangementModel`).
- `addLoad` – vytváří výraz pro čtení dat z paměti (delegace zpracování operace na `MemoryMangementModel`).
- `addGetElementPtr` – vytváří výraz výpočet adresy prvku struktury nebo pole.
- `declareFunction` – vytváří výraz pro deklaraci funkce (`z3::func_decl`).
- `defineFunction` – vytváří výraz pro definici funkce (svázání deklarace s interpretací těla funkce).
- `addFunctionCall` – vytváří výraz volání funkce a uložení návratové hodnoty.

Zadání výrazů do řešiče a ověření splnitelnosti

Pro práci s řešičem používá `Z3Driver` objekt třídy `z3::solver` vázaný na jeden určitý objekt třídy `z3::context`. Jednotlivé výrazy, které chceme řešičem asertovat tak, abychom získali řešení splňující všechny podmínky (pro výsledný model musí platit všechny aserce zároveň) je možné přidat řešiči voláním jeho metody `z3::solver::add`. Metoda pro přidání aserce `z3::solver::add` přijímá jeden parametr výrazu `z3::expr` – musí se však jednat o výraz booleovského typu. To znamená, že pokud bychom chtěli vyjádřit například výraz operace nad bitovými vektory (která se vyhodnotí jako bitový vektor), je třeba ji zaregistrovat jako aserci rovnosti tohoto výrazu na nějakou konstantu (kterou si můžeme vygenerovat pro tento účel) – získáme tím potřebný booleovský výraz. Při zadávání výrazů řešiči postupujeme tak, že první zaregistrujeme všechny výrazy pro operace nad paměťovým model a pak výrazy získané transformací ostatních instrukcí. Deklarace funkcí se neregistrují objektu `z3::solver`, jsou již v objektu `z3::context`, nad kterým `z3::solver` pracuje.

Výsledek řešiče je třeba ověřit metodou `metodou z3::solver::check`. Výsledek může nabývat hodnot `z3::check_result::sat` (zadaný problém je splnitelný můžeme získat model jeho řešení), `z3::check_result::unsat` (řešič nenalez žádné řešení pro zadaný program, do výstupu se předá chybová zpráva), nebo `z3::check_result::unknown` (do výstupu se zapíše chybová zpráva). Objekt nalezeného řešení lze získat pomocí metody `z3::solver::get_model`.

4.5 Zpracování výsledků řešiče

Hledané ohodnocení proměnných je možné získat z objektu třídy `z3::model`. Pro přenos hodnot jsou uložený do pomocné struktury, `ResultModel`, kde se ukládá do mapy název proměnné a její nalezená hodnota, převedena do textové serializace. Booleovské proměnné jsou reprezentovány řetězcem "true"nebo "false", pro bitové vektory je hodnota získána jako jejich hexadecimální reprezentace v řetězci (obsahuje prefix "0x"a decimální zápis se rozšířen o nuly do délky bitového vektoru).

Pro výstup programu je obsah získaného `ResultModel` převeden do formátu JSON (formát viz příloha schéma formátu vstupních dat D). Pro každou z analyzovaných cest je ve výstupním souboru přidán jeden objekt, konkrétní cesta je zde identifikována klíčem `tag`. Položka `data` obsahuje seznam proměnných a jejich serializovaných hodnot (klíč je název proměnné). Pokud nebyla analyzovaná cesta sémanticky splnitelná, nebo došlo k jiné chybě během její analýzy, je chybová hláška v objektu cesty uložena do položky `error`.

4.6 Závislosti

Nástroj je implementován v jazyce C++ (standardu C++17). Sestavování programu je automatizováno s využitím multiplatformního open source nástroje pro správu a konfiguraci sestavování projektů CMake [10], nezávislého na použitém operačním systému a překladači. Referenční prostředí pro použití programu je systém s linuxovým operačním systém distribuce Fedora 29 (64-bitová verze), obsahující verzi LLVM knihovny 7.0.1 a překladač clang, také ve verzi 7.0.1.

Referenční verze knihovny pro práci s řešičem SMT `Z3` pro C++ je 4.7.1 (odpovídá nainstalované verzi řešiče Z3). Pro injektování závislostí (*angl. dependency injection*) je v aplikaci použita knihovna `[Boost].DI` [9] ve verzi 1.1.0 (distribuovaná jako jediný C++ hlavičkový soubor, nezávislý na ostatních boost knihovnách), veřejně dostupná pod licencí Boost Software License. Pro práci s datovým formátem JSON, který je použit, jak pro vstupní data, tak pro formátování výstupu aplikace, je v aplikaci využita knihovna `JSON for Modern C++` vyvíjená Nielsem Lohmannem [18] (použita verze 3.6.1) distribuovaná jako jediný C++ hlavičkový soubor, bez dalších závislostí, veřejně dostupná pod licencí MIT. Pro implementaci testové sady nebyla použita specializovaná knihovna, ale sada bash skriptů (pro práci s formátem JSON používají skripty utilitu jq [7] dostupnou pod licencí Creative Commons CC BY 3.0).

Kapitola 5

Ověření výsledků

V této kapitole je popsáno navržení a implementace testové sady 5.2, která ověří, že nástroj tindger splňuje požadavky 3.1 a je schopný vygenerovat validní výsledky pro sadu zdrojový souborů pokrývajících rozsah LLVM instrukční sady mimo zadaná omezení 3.2 (validita výsledků je demonstrována demonstračními skripty 5.3 sledujícími cestu toku řízení programu základními bloky po zadání vygenerovaných hodnot).

5.1 Průběžná integrace a referenční prostředí

Projekt je verzován systémem pro správu verzí git, repozitář projektu je hostován v systému gitlab skupiny Testos. Součástí gitlabového repozitáře je zprovoznění systému průběžné integrace (GitLab CI/CD pipelines [8]). GitLab CI umožňuje provést automatické sestavení aplikace po nahrání kódových změn a spuštění testů. Průběžná integrace je konfigurovatelná pomocí konfiguračních souborů ve formátu yaml.

V rámci projektu jsem nakonfiguroval dvoustupňovou CI linku. V prvním stupni se provádí čisté sestavení. V rámci druhého stupně se provádí dvě úlohy – sestavení se spuštěním testové sady a sestavení s měřením pokrytí nad během testové sady. Obě úlohy produkují artefakty sestavení, které jsou po doběhnutí úlohy dostupné ke stažení a analýze. Gitlab runner, program vykonávající úlohy z gitlab CI pipeline, může být nakonfigurován pro běh úloh nad různými prostředími – pro lokální systém, Docker kontejner, nebo přes vzdálený přístup na specifickém serveru. Využil jsem dva obrazy Docker kontejnerů – referenční obraz `fedora:29`, se stabilní verzí LLVM 7 a testovací větev nastavenou pro obraz `fedora:30` obsahující vývojovou verzi knihovny LLVM 8.

5.2 Testovací sada

Testové příklady se nacházejí ve složce projektu test/demo, obsahují tyto kategorie:

- basic – základní testy podpory řídicích struktur pro větvení (`IF-THEN-ELSE`).
- operators – testy podpory jednotlivých operátorů (aritmetických a logických).
- ptr – základní testy podpory ukazatelů (práce s hodnotou předanou funkci přes ukazatel).
- aliasing – pokročilé testy podpory ukazatelů zaměřující se na detekci překryvu ukazatelů (*pointer aliasing*, kdy dva různé ukazatele ukazují na stejné místo v paměti.

- while – základní testy podpory řídicích struktur pro cykly (`while`).
- struct – základní testy podpory práce se strukturami (přístup k atributům struktury přes ukazatel na strukturu).
- function – experimentální testy pro analýzu volání funkcí (omezené vzhledem k intra-procedurálnímu zaměření nástroje).
- boolean – testy podpory práce s datovým typem `boolean`.
- casting – testy podpory přetypování.
- global – testy podpory práce s globálními proměnnými (omezen na globální proměnné definované v rámci analyzované překladové jednotky).
- notfound – testy ošetření nevalidního zadání (např. zadání analýzy funkce, která se nevyskytuje v zadaném zdrojovém kódu).
- cli – testy ověřující správné zpracování parametrů příkazové řádky nástroje.
- unsupported – testy ošetření chyb analýzy při detekci nepodporovaných vlastností (např. `floating-point` instrukcí).

Spuštění testů a ověření jejich výsledků je prováděno skriptem run_test.sh, který se nachází v adresáři test.

5.3 Demonstrační skripty

Projekt obsahuje v rámci svých testů demonstrační ukázku, toho že vygenerované hodnoty zaručí provedení zadané cesty programu. Demonstrační program volající analyzovanou funkci obsahuje textový výstup, reportuje skutečně navštívené základní bloky. Parametry funkce jsou nastaveny na získané hodnoty pomocí konstant definovaných překladačem. Při testu je testový skript nejprve přeloží analyzovaný zdrojový soubor do LLVM IR kódu, který použije pro analýzu nástrojem `tindger` pomocí nachystaného konfiguračního souboru obsahujícího zadanou cestu. Z výstupního JSON souboru ověří, že cesta byla sémanticky dostupná (kontrola atributu error). Po úspěšné kontrole vyextrahuje nalezené hodnoty vstupů. Hodnoty vstupů předá jako parametry překladači, který je nadefinuje při překladu demonstračního programu. Testový skript provede spuštění demonstračního programu a uloží jeho výstup, ze který vyfiltruje, aby zde zůstal pouze výpis reportu pokrytí (řádky začínající prefixem `"BB#"`). Porovnáním reportu vůči cestě ze zadání bylo ověřeno, že hodnoty testovacích vstupů byly korektně vygenerovány.

5.4 Testové pokrytí

Tato podkapitola přibližuje přípravu projektu na měření pokrytí zdrojových souborů projektu. Pro měření pokrytí byl použit nástroj `gcov` a jeho rozšíření `lcov`, příprava nástrojů je popsána v následující sekci 5.4. Výsledky získané z měření pokrytí jsou uvedeny v sekci 5.4.

Postup měření pokrytí

Pro měření pokrytí byl použit linuxový nástroj `gcov`, vyvíjeny jako součást překladače GCC pro testování pokrytí. Nástroj `lcov` rozšiřuje možnosti `gcov` o generování vizualizace pokrytí ze získaných analytických dat do dobře prezentovatelné podoby v HTML formátu. Pro použití spolu s překladačem `clang`, který jsem v projektu využil, je třeba použít nástroj `llvm-cov`, který slouží jako adaptér pro práci s `gcov` pro překladače postavené nad LLVM. Kvůli měnícímu se rozhraní `llvm-cov` je potřeba s novějšími verzemi LLVM použít jednoduchý skript, který vyřeší problém s jinými parametry `llvm-cov` a `lcov`. Příklad jednoduchého skriptu řešící tento problém `llvm-gcov.sh`:[1]

```
1 #!/bin/bash
2 exec llvm-cov gcov "$@"
```

Pro podporu generování analytických dat je třeba program přeložit se speciálním nastavením pro přidání debugovacích informací. Použil jsem tyto přepínače překladače `-g -O0 -coverage -fprofile-instr-generate -fcoverage-mapping -fno-inline-functions`. Knihovnu lcov jsem přilinkoval přidáním parametrů linkeru `-lgcov -coverage`. Vhodně přeložený program je pak třeba spustit (použil jsem běh testovací sady 5.2). Vygenerovaná data je třeba shromáždit a zanalyzovat. Pro vytvoření souboru `coverage.info` jsem použil příkaz: `lcov -directory ./../CMakeFiles/tindger.dir/src -base-directory . -rc lcov_branch_coverage=1 -gcov-tool ./llvm-gcov.sh -capture -o coverage.info`. Pro přesnější výsledky jsem z výsledků vyfiltroval knihovny třetích stran pomocí příkazu: `lcov -remove coverage.info '*include/*' -o coverage.info`. Finální report je vygenerován příkazem `genhtml`, pro který jsem použil parametr `-demangle-cpp` pro rekonstrukci originálních názvů funkcí v reportu. Celý použitý příkaz: `genhtml coverage.info -o coverage -demangle-cpp`. Nově vygenerovaný adresář obsahuje soubor `index.html` (hlavní strana reportu), ze kterého je přístupný náhled na pokrytí jednotlivých modulů a původních zdrojových souborů.

[1] Skript převzat z [5].

Výsledky měření pokrytí

Následující tabulky obsahují výsledky měření testu pokrytí nástroje `tindger` vůči jeho testové sadě.

	Průchodů	Celkem	Pokrytí
Řádky kódu:	1490	1593	93.5 %
Funkce:	267	285	93.7 %

Tabulka 5.1: Pokrytí napříč celým projektem.

Adresář	Pokrytí řádků %	Pokrytí řádků	Pokrytí funkcí %	Pokrytí funkcí
src	97.7 %	301/308	97.1 %	34/35
src/llvmextractor	96.7 %	203/210	90.0 %	45/50
src/model	91.8 %	380/414	92.2 %	71/77
src/model/types	96.9 %	31/32	100.0 %	34/34
src/smt	90.9 %	507/558	93.4 %	71/76
src/smt/memory	95.8 %	68/71	92.3 %	12/13

Tabulka 5.2: Pokrytí jednotlivých modulů a submodulů (adresář src odpovídá modulu `App`).

Kapitola 6

Závěr

V diplomové práci jsem zkoumal způsob zefektivnění procesu testování software. Zaměřil jsem se na oblast automatizovaného generování testových sad, konkrétně na možnost automaticky generovat testovací vstupy, pro které bude testovaný kód při vykonání testu dodržovat určenou cestu grafem toku řízení. Prostudoval jsem existující řešení zabývající se automatizovaným generováním testových sad. Pro návrh svého nástroje pro generování testovacích vstupů jsem se rozhodl využít technologií LLVM 2.5.2 a řešičů SMT-LIB 2.3.2 (konkrétně řešiče Z3 2.3.3).

Při implementaci nástroje `tindger` jsem zvolil přístup zpracování vstupního kódu LLVM IR na objektový model vytvořený knihovnou LLVM. Tento model je dále transformován (s využitím pomocných tříd) tak, aby bylo možno analyzovat jednotlivé cesty. Pro analyzovanou cestu je vytvořen seznam základních bloků, je ověřena jejich syntaktická dostupnost a jsou odstraněny duplicity (které mohou vzniknout při rozvinutí smyček). Pomocné třídy analyzují instrukce cesty (využil jsem návrhový vzor `visitor`) a určují, na jaký typ výrazu SMT-LIB se bude daná instrukce transformovat. Výrazy SMT-LIB jsou vytvořeny třídou `Z3Driver`, která implementuje obecné rozhraní `SMTDriver` pro konkrétní řešič SMT – Z3. Pro přidání podpory jiného řešiče stačí doplnit implementaci tohoto rozhraní a změnit konfiguraci jeho výchozí implementace (zbytek aplikace závisí pouze na obecném rozhraní). Pro práci s pamětí jsem využil pole 8-bitový bitových vektorů, které je indexováno (dle cílové architektury) 32-bitovým nebo 64-bitovým bitovým vektorem. Po každém zápisu do paměti je vytvořen alias pro novou verzi paměťového modelu, který je použit pro následující operace čtení z paměti až do dalšího zápisu. Po transformaci celé cesty na výrazy SMT-LIB jsou všechny výrazy (a definice všech použitých proměnných a funkcí) zadány řešiči SMT Z3. Výstup řešiče je zpracován a serializován do formátu JSON. Výstup obsahuje seznam ohodnocení proměnných pro každou z analyzovaných cest. U proměnných je uveden jejich název z LLVM IR a serializovaná hodnota.

Výsledkem této práce je prototyp generátoru hodnot testovacích vstupů podle zadané stopy programu – nástroj `tindger`. Nástroj `tindger` se zaměřuje na intraprocedurální analýzu (interprocedurální zatím není dostatečně podporována) a pracuje nad modely programů popsanými mezikódem LLVM IR. Ne všechny LLVM IR instrukce jsou podporovány (omezení prototypu), ale pro všechny podporované instrukce byl ověřen korektní překlad v rámci testování implementace. Pro testování byl vygenerován z C++ zdrojového souboru kód LLVM IR, spolu s připraveným seznamem cest byl použit jako vstup nástroje `tindger`. Z vygenerovaného seznamu ohodnocení proměnných byly hledané hodnoty vyextrahovány a vloženy překladačem do demonstračního programu (jako preprocesorové definice). Demonstrační program volající analyzovanou funkci obsahuje textový výstup, reportuje sku-

tečně navštívené základní bloky. Porovnáním reportu vůči cestě ze zadání bylo ověřeno, že hodnoty testovacích vstupů byly korektně vygenerovány. Výsledky byly ověřeny pro verze knihovny LLVM 7 a 8. V současné době nástroj podporuje analýzu základních (celočíselných) aritmetických a logických operací, změnu toku řízení řídicími strukturami (podmíněné větvení, cykly), analýzu práce s pamětí (pro zarovnaný přístup do paměti) a ukazateli (včetně překryvu ukazatelů) a práce s datovými strukturami.

Literatura

[1] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2016, ISBN 9781107172012, doi:DOI:10.1017/9781316771273.

[2] Barrett, C.; Fontaine, P.; Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technická zpráva, Department of Computer Science, The University of Iowa, 2017, [Online; navštíveno 16.01.2019].
URL `http://smtlib.cs.uiowa.edu/`

[3] Barrett, C.; Fontaine, P.; Tinelli, C.: The SMT-LIB StandardVersion 2.6. http://smtlib.cs.uiowa.edu, 2017, [Online; navštíveno 22.05.2019].
URL
`http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf`

[4] Barrett, C.; Fontaine, P.; Tinelli, C.: SMT-LIB Benchmarks. http://smtlib.cs.uiowa.edu, 2018, [Online; navštíveno 22.05.2019].
URL `http://smtlib.cs.uiowa.edu/benchmarks.shtml`

[5] hsiang Chien, T.: Check Code Coverage with Clang and LCOV. Logan's Note, 2015, [Online; navštíveno 22.05.2019].
URL `https://logan.tw/posts/2015/04/28/check-code-coverage-with-clang-and-lcov/`

[6] De Moura, L.; Bjørner, N.: Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 3-540-78799-2, 978-3-540-78799-0, s. 337–340.
URL `https://dl.acm.org/citation.cfm?id=1792734.1792766`

[7] Dolan, S.: *jq*. [Online; navštíveno 22.05.2019].
URL `https://stedolan.github.io/jq/`

[8] GitLab: *GitLab CI/CD*. [Online; navštíveno 22.05.2019].
URL `https://about.gitlab.com/product/continuous-integration/`

[9] Jusiak, K.: *[Boost].DI*. [Online; navštíveno 22.05.2019].
URL `https://boost-experimental.github.io/di/`

[10] Kitware: *CMake*. [Online; navštíveno 22.05.2019].
URL `https://cmake.org/`

[11] Kondula, V.: *Extrakce grafu toku řízení z formátu LLVM IR*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017.
URL `http://www.fit.vutbr.cz/study/DP/BP.php?id=19239`

[12] Kovásznai, G.; Fröhlich, A.; Biere, A.: On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width. In *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories*, *EPiC Series in Computing*, ročník 20, editace P. Fontaine; A. Goel, EasyChair, 2013, ISSN 2398-7340, s. 44–56, doi:10.29007/cvnz.
URL `https://easychair.org/publications/paper/SF7`

[13] Kovásznai, G.; Fröhlich, A.; Biere, A.: Complexity of Fixed-Size Bit-Vector Logics. *Theory of Computing Systems*, ročník 59, č. 2, Aug 2016: s. 323–376, ISSN 1433-0490, doi:10.1007/s00224-015-9653-1.
URL `https://doi.org/10.1007/s00224-015-9653-1`

[14] Kraut, D.: *Generování modelů pro testy ze zdrojových kódů*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2019, k nalezení na http://www.fit.vutbr.cz/study/DP/DP.php.

[15] Lattner, C.: *LLVM: An Infrastructure for Multi-Stage Optimization*. Diplomová práce, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[16] LLVM-Foundation: *LLVM Language Reference Manual*. [Online; navštíveno 22.05.2019].
URL `https://releases.llvm.org/7.0.0/docs/LangRef.html`

[17] LLVM-Foundation: *The LLVM Compiler Infrastructure*. [Online; navštíveno 22.05.2019].
URL `https://llvm.org/`

[18] Lohmann, N.: *JSON for Modern C++*. [Online; navštíveno 22.05.2019].
URL `https://nlohmann.github.io/json/`

[19] Sečkařová, P.: *Extrakce grafu toku řízení z bajtkódu Java*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017.
URL `http://www.fit.vutbr.cz/study/DP/BP.php?id=19241`

[20] Smrčka, A.; aj.: *Testos Platform*. [Online; navštíveno 22.05.2019].
URL `https://testos.org/`

[21] Sušovský, T.: *Překladač grafu toků dat do logiky bitových vektorů*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.
URL `http://www.fit.vutbr.cz/study/DP/BP.php?id=18543`

[22] Vít, R.: Control Flow Graph Query Engine. Testos, 2017, [Online; navštíveno 22.05.2019].
URL `https://pajda.fit.vutbr.cz/testos/cfgqe`

Seznam příloh

Příloha A

Obsah CD

- `/`
  - `bin` ..... Adresář pro přeložené spustitelné artefakty projektu.
    - `tindger` . Spustitelný binární soubor programu tindger (přeložený pro referenční prostředí).
  - `doc` ..... Adresář s dokumentací projektu.
    - `src` ..... Adresář se zdrojovými soubory technické zprávy.
    - `xsusov01.pfd` ..... PDF verze této technické zprávy.
    - `config.json` ..... Referenční JSON schema formátu vstupních dat.
    - `result.json` ..... Referenční JSON schema formátu výstupních dat.
  - `include` ..... Adresář pro knihovny použité projektu.
    - `boost`
      - `di.hpp` ..... Single header knihovna [Boost].DI.
    - `nlohmann .4 json.hpp` ..... Single header knihovna Nlohmann/JSON.
  - `src` ..... Adresář se zdrojovými soubory
  - `test` ..... Adresář s testy.
    - `run-test.sh` ..... Skript pro spuštění testovací sady.
    - `cli` ..... Testové soubory testovací sady zaměřené na UI programu.
    - `coverage` ..... Adresář pro ukládání výstupu analýzy pokrytí.
    - `demo` ..... Testové soubory demonstrační testovací sady.
      - `out` ..... Adresář pro ukládání výstupu testů.
  - `CMakeLists.txt` ..... CMake skript pro překlad programu.
  - `README.md` ..... Stručný popis popis programu a jeho ovládání.
  - `LICENSE` ..... MIT licence zdrojových souborů projektu.
  - `.clang-format` ..... Kódový styl projektu.
  - `.gitlab-ci.yml` ..... Konfigurační soubor pro nastavení CI pipeline.

Příloha B

Manuál ovládání programu tindger

```
Pouziti: tindger -c FILEPATH_CONFIG.JSON [SEZNAM VOLITELNYCH ARGUMENTU]

Povinne argumenty:
-c, --config FILEPATH cesta k JSON souboru se zadanim cest k analyze

Volitelne argumenty:
-i, --input FILEPATH cesta zdrojoveho souboru s kodem LLVM IR
                     STDIN pokud neni zadan
-o, --output FILEPATH cesta pro vystupni JSON soubor pro zapis vysledku
                     STDOUT pokud neni zadan
-s, --smt FILEPATH cesta pro vystupni soubor pro vygenerovany kod SMT-LIB
                     STDOUT pokud neni zadan
-e, --error FILEPATH cesta error logu
                     STDERR pokud neni zadan
-d, --debug FILEPATH cesta debug logu
                     STDERR pokud neni zadan
-v, --version vypise cislo verze
-h, --help vypise napovedu
```

Výpis B.1: Manuál použití nástroje tindger

Příloha C

Schéma formátu vstupních dat

V této příloze je uveden popis použitého JSON schéma pro formátování vstupních dat programu. Schéma je uvedeno dle standardu JSON schema (verze 7).

JSON SCHEMA

```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://pajda.fit.vutbr.cz/testos/tindger/blob/master/doc/config.json
      ",
  "type": "object",
  "title": "Tindger path config",
  "required": [
    "function"
  ],
  "properties": {
    "function": {
      "$id": "/properties/function",
      "type": "object",
      "title": "Analyzed function description",
      "required": [
        "name",
        "paths"
      ],
      "properties": {
        "name": {
          "$id": "/properties/function/properties/name",
          "type": "string",
          "title": "Analyzed function name (LLVM IR generated name)",
          "default": "",
          "examples": [
            "_Z3fooii",
            "_Z1xi",
            "_Z11constantFoom"
          ],
          "pattern": "^_Z[0-9]+[a-zA-Z0-9_]+$"
        },
        "paths": {
```

```
33          "$id": "/properties/function/properties/paths",
34          "type": "array",
35          "title": "List of paths description for analysis",
36          "items": {
37            "$id": "/properties/function/properties/paths/items",
38            "type": "object",
39            "title": "Path record",
40            "required": [
41              "path",
42              "tag",
43              "metadata"
44            ],
45            "properties": {
46              "path": {
47                "$id": "/properties/function/properties/paths/properties/path",
48                "type": "array",
49                "title": "List of paths description for analysis",
50                "items": {
51                  "$id": "/properties/function/properties/paths/properties/path/
                        items",
52                  "type": "integer",
53                  "title": "Basic Block index in function's basic blocks list",
54                  "default": 0,
55                  "examples": [
56                    0,
57                    1,
58                    5
59                  ]
60                }
61              },
62              "tag": {
63                "$id": "/properties/function/properties/paths/properties/tag",
64                "type": "string",
65                "title": "Single path unique identificator",
66                "default": "",
67                "examples": [
68                  "ifTrue",
69                  "ifFalse",
70                  "foo-allArgsAreZero"
71                ],
72                "pattern": "^(.*)$"
73              },
74              "metadata": {
75                "$id": "/properties/function/properties/paths/properties/
                      metadata",
76                "type": "array",
77                "title": "List of additional for analysis optimization",
78                "items": {
79                  "$id": "/properties/function/properties/paths/properties/
                        metadata/items",
80                  "type": "string",
81                  "title": "Meta tag in format $key:$vale",
82                  "examples": [
83                    "puts:ignore",
```

```
                    "_Z11constantFoom:define",
                    "%15:assert==42"
                  ],
                  "pattern": "^(.+):(.+)$"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Výpis C.1: JSON schéma formátu vstupních dat dle stadardu JSON schema v7

Příloha D

Schéma formátu výstupních dat

V této příloze je uveden popis použitého JSON schéma pro formátování výstupních dat programu. Schéma je uvedeno dle standardu JSON schema (verze 7).

JSON SCHEMA

```
{
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://pajda.fit.vutbr.cz/testos/tindger/blob/master/doc/result.json",
  "type": "array",
  "title": "Tindger generator result data format",
  "items": {
    "$id": "/items",
    "type": "object",
    "title": "Generated values object",
    "required": [
      "function",
      "tag",
      "data",
      "error"
    ],
    "properties": {
      "function": {
        "$id": "/items/properties/function",
        "type": "string",
        "title": "Analyzed function name (LLVM IR generated name)",
        "default": "",
        "examples": [
          "_Z3fooii",
          "_Z1xi",
          "_Z11constantFoom"
        ],
        "pattern": "^_Z[0-9]+[a-zA-Z0-9_]+$"
      },
      "tag": {
        "$id": "/items/properties/tag",
        "type": "string",
        "title": "Single path unique identificator",
```

```
34            "default": "",
35            "examples": [
36              "ifTrue",
37              "ifFalse",
38              "foo-allArgsAreZero"
39            ],
40            "pattern": "^(.*)$"
41          },
42          "data": {
43            "$id": "/items/properties/data",
44            "type": "object",
45            "title": "Single path unique identificator"
46          },
47          "error": {
48            "$id": "/items/properties/error",
49            "type": "string",
50            "title": "Error message.",
51            "default": "",
52            "examples": [
53              "Invalid type float detected in analysis.",
54              "Unsupported insteuction found FMul."
55            ],
56            "pattern": "^(.*)$"
57          }
58        }
59    }
60 }
```

Příloha E

Ukázka LLVM IR kódu a seznamu vygenerovaných hodnot

Zdrojový kód programu v jazyce C++

`foo.cpp`:

```
1  int foo(int a, int b)
2  {
3      int retVal;
4      if ((a + b) > (a * b)) {
5          retVal = 1;
6      } else {
7          retVal = 0;
8      }
9
10     return retVal;
11 }
```

Přeložený kód LLVM IR

```
1  ; ModuleID = 'foo.cpp'
2  source_filename = "foo.cpp"
3  target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
4  target triple = "x86_64-unknown-linux-gnu"
5
6  ; Function Attrs: noinline nounwind optnone uwtable
7  define dso_local i32 @_Z3fooii(i32, i32) #0 {
8    %3 = alloca i32, align 4
9    %4 = alloca i32, align 4
10   %5 = alloca i32, align 4
11   store i32 %, i32* %3, align 4
12   store i32 %1, i32* %4, align 4
13   %6 = load i32, i32* %3, align 4
14   %7 = load i32, i32* %4, align 4
15   %8 = add nsw i32 %6, %7
16   %9 = load i32, i32* %3, align 4
```

```
17   %10 = load i32, i32* %4, align 4
18   %11 = mul nsw i32 %9, %10
19   %12 = icmp sgt i32 %8, %11
20   br i1 %12, label %13, label %14
21
22 ; <label>:13: ; preds = %2
23   store i32 1, i32* %5, align 4
24   br label %15
25
26 ; <label>:14: ; preds = %2
27   store i32 0, i32* %5, align 4
28   br label %15
29
30 ; <label>:15: ; preds = %14, %13
31   %16 = load i32, i32* %5, align 4
32   ret i32 %16
33 }
34
35 attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-
       math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-
       pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "
       no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="
       false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="
       x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "
       use-soft-float"="false" }
36
37 !llvm.module.flags = !{!0}
38 !llvm.ident = !{!1}
39
40 !0 = !{i32 1, !"wchar_size", i32 4}
41 !1 = !{!"clang version 7.0.1 (Fedora 7.0.1-6.fc29)"}
```

JSON s konfigurací pro analýzu cest

```
1  {
2    "function": {
3      "name": "_Z3fooii",
4      "paths": [
5        {
6          "path": [
7            0,
8            1,
9            3
10         ],
11         "tag": "condIsTruePath",
12         "metadata": []
13       },
14       {
15         "path": [
16           0,
17           2,
18           3
19         ],
20         "tag": "condIsFalsePath",
```

```
        "metadata": []
      },
      {
        "path": [
          0,
          1,
          2,
          3
        ],
        "tag": "invalidPath",
        "metadata": []
      }
    ]
  }
}
```


Vygenerované hodnoty


```
{
"condIsTruePath": {
  "data": {
    "#arg_0": "0x00000000",
    "#arg_1": "0x00000001",
    "%10": "0x00000001",
    "%11": "0x00000000",
    "%12": true,
    "%16": "0x00000001",
    "%3": "0x0000000000000801",
    "%4": "0x0000000000000805",
    "%5": "0x0000000000000809",
    "%6": "0x00000000",
    "%7": "0x00000001",
    "%8": "0x00000001",
    "%9": "0x00000000",
    "SP": "0x0000000000000801",
    "arg_0": "0x00000000",
    "arg_1": "0x00000001",
    "memory_10_byte_write": "0x00",
    "memory_11_byte_write": "0x00",
    "memory_12_byte_write": "0x00",
    "memory_1_byte_write": "0x00",
    "memory_2_byte_write": "0x00",
    "memory_3_byte_write": "0x00",
    "memory_4_byte_write": "0x00",
    "memory_5_byte_write": "0x01",
    "memory_6_byte_write": "0x00",
    "memory_7_byte_write": "0x00",
    "memory_8_byte_write": "0x00",
    "memory_9_byte_write": "0x01"
  },
  "error": ""
},
"condIsFalsePath": {
  "data": {
    "#arg_0": "0x00000000",
    "#arg_1": "0x80000001",
    "%10": "0x80000001",
    "%11": "0x00000000",
    "%12": false,
    "%16": "0x00000000",
    "%3": "0x0000000000000801",
    "%4": "0x0000000000000805",
    "%5": "0x0000000000000809",
    "%6": "0x00000000",
```

```
      "%7": "0x80000001",
      "%8": "0x80000001",
      "%9": "0x00000000",
      "SP": "0x0000000000000801",
      "arg_0": "0x00000000",
      "arg_1": "0x80000001",
      "memory_10_byte_write": "0x00",
      "memory_11_byte_write": "0x00",
      "memory_12_byte_write": "0x00",
      "memory_1_byte_write": "0x00",
      "memory_2_byte_write": "0x00",
      "memory_3_byte_write": "0x00",
      "memory_4_byte_write": "0x00",
      "memory_5_byte_write": "0x01",
      "memory_6_byte_write": "0x00",
      "memory_7_byte_write": "0x00",
      "memory_8_byte_write": "0x80",
      "memory_9_byte_write": "0x00"
    },
    "error": ""
  },
  "invalidPath": {
    "data": {
    },
    "error": "Blocks in analyzed path not reachable from each other. Aborting path analysis"
  }
}
```


Vygenerovaný kód SMT-LIB

```
; condIsTruePath SMT:
(set-info :status unknown)
(declare-fun arg_0 () (_ BitVec 32))
(declare-fun memory_1_byte_write () (_ BitVec 8))
(declare-fun %3 () (_ BitVec 64))
(declare-fun memory () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_1 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_2_byte_write () (_ BitVec 8))
(declare-fun memory_2 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_3_byte_write () (_ BitVec 8))
(declare-fun memory_3 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_4_byte_write () (_ BitVec 8))
(declare-fun memory_4 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun arg_1 () (_ BitVec 32))
(declare-fun memory_5_byte_write () (_ BitVec 8))
(declare-fun %4 () (_ BitVec 64))
(declare-fun memory_5 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_6_byte_write () (_ BitVec 8))
(declare-fun memory_6 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_7_byte_write () (_ BitVec 8))
(declare-fun memory_7 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_8_byte_write () (_ BitVec 8))
(declare-fun memory_8 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun %6 () (_ BitVec 32))
(declare-fun %7 () (_ BitVec 32))
(declare-fun %9 () (_ BitVec 32))
(declare-fun %10 () (_ BitVec 32))
(declare-fun memory_9_byte_write () (_ BitVec 8))
(declare-fun %5 () (_ BitVec 64))
(declare-fun memory_9 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_10_byte_write () (_ BitVec 8))
(declare-fun memory_10 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_11_byte_write () (_ BitVec 8))
(declare-fun memory_11 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun memory_12_byte_write () (_ BitVec 8))
(declare-fun memory_12 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun %16 () (_ BitVec 32))
```

```
(declare-fun |#arg_0| () (_ BitVec 32))
(declare-fun |#arg_1| () (_ BitVec 32))
(declare-fun SP () (_ BitVec 64))
(declare-fun %8 () (_ BitVec 32))
(declare-fun %11 () (_ BitVec 32))
(declare-fun %12 () Bool)
(assert
 (= memory_1_byte_write ((_ extract 7 0) arg_0)))
(assert
 (= memory_1 (store memory %3 memory_1_byte_write)))
(assert
 (= memory_2_byte_write ((_ extract 15 8) arg_0)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (= memory_2 ?x33)))
(assert
 (= memory_3_byte_write ((_ extract 23 16) arg_0)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (= memory_3 (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write))))
(assert
 (= memory_4_byte_write ((_ extract 31 24) arg_0)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
      memory_4_byte_write)))
 (= memory_4 ?x49))))
(assert
 (= memory_5_byte_write ((_ extract 7 0) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
      memory_4_byte_write)))
 (= memory_5 (store ?x49 %4 memory_5_byte_write)))))
(assert
 (= memory_6_byte_write ((_ extract 15 8) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
      memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (= memory_6 ?x81)))))
(assert
 (= memory_7_byte_write ((_ extract 23 16) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
      memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (= memory_7 (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write))))))
(assert
 (= memory_8_byte_write ((_ extract 31 24) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
      memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
      memory_8_byte_write)))
 (= memory_8 ?x95))))))
(assert
```

```
93  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
94  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
95  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
96  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
97  (= (select ?x95 %3) ((_ extract 7 0) %6)))))))
98 (assert
99  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
100 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
101 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
102 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
103 (= (select ?x95 (bvadd %3 (_ bv1 64))) ((_ extract 15 8) %6)))))))
104 (assert
105 (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
106 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
107 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
108 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
109 (= (select ?x95 (bvadd %3 (_ bv2 64))) ((_ extract 23 16) %6)))))))
110 (assert
111 (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
112 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
113 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
114 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
115 (= (select ?x95 (bvadd %3 (_ bv3 64))) ((_ extract 31 24) %6)))))))
116 (assert
117 (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
118 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
119 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
120 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
121 (= (select ?x95 %4) ((_ extract 7 0) %7)))))))
122 (assert
123 (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
124 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
125 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
126 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
127 (= (select ?x95 (bvadd %4 (_ bv1 64))) ((_ extract 15 8) %7)))))))
128 (assert
129 (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
130 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
131 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
132 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
        memory__8_byte_write)))
133 (= (select ?x95 (bvadd %4 (_ bv2 64))) ((_ extract 23 16) %7)))))))
134 (assert
135 (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
        )))
136 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
        memory__4_byte_write)))
137 (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
```

```
138  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
139  (= (select ?x95 (bvadd %4 (_ bv3 64))) ((_ extract 31 24) %7)))))))
140 (assert
141  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
142  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
143  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
144  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
145  (= (select ?x95 %3) ((_ extract 7 0) %9)))))))
146 (assert
147  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
148  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
149  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
150  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
151  (= (select ?x95 (bvadd %3 (_ bv1 64))) ((_ extract 15 8) %9)))))))
152 (assert
153  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
154  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
155  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
156  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
157  (= (select ?x95 (bvadd %3 (_ bv2 64))) ((_ extract 23 16) %9)))))))
158 (assert
159  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
160  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
161  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
162  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
163  (= (select ?x95 (bvadd %3 (_ bv3 64))) ((_ extract 31 24) %9)))))))
164 (assert
165  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
166  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
167  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
168  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
169  (= (select ?x95 %4) ((_ extract 7 0) %10)))))))
170 (assert
171  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
172  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
173  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
174  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
175  (= (select ?x95 (bvadd %4 (_ bv1 64))) ((_ extract 15 8) %10)))))))
176 (assert
177  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
178  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
179  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
180  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
181  (= (select ?x95 (bvadd %4 (_ bv2 64))) ((_ extract 23 16) %10)))))))
182 (assert
```

```
183  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
184  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
185  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
186  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
187  (= (select ?x95 (bvadd %4 (_ bv3 64))) ((_ extract 31 24) %10)))))))
188 (assert
189  (= memory_9_byte_write ((_ extract 7 0) (_ bv1 32))))
190 (assert
191  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
192  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
193  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
194  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
195  (= memory_9 (store ?x95 %5 memory_9_byte_write)))))))
196 (assert
197  (= memory_10_byte_write ((_ extract 15 8) (_ bv1 32))))
198 (assert
199  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
200  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
201  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
202  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
203  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
204  (= memory_10 ?x163)))))))
205 (assert
206  (= memory_11_byte_write ((_ extract 23 16) (_ bv1 32))))
207 (assert
208  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
209  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
210  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
211  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
212  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
213  (= memory_11 (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write)))))))))
214 (assert
215  (= memory_12_byte_write ((_ extract 31 24) (_ bv1 32))))
216 (assert
217  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
218  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
219  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
220  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
221  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
222  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
223  (= memory_12 ?x177)))))))))
224 (assert
225  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
226  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
227  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
228  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
```

```
229  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
230  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
231  (= (select ?x177 %5) ((_ extract 7 0) %16)))))))))
232 (assert
233  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
234  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
235  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
236  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
237  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
238  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
239  (= (select ?x177 (bvadd %5 (_ bv1 64))) ((_ extract 15 8) %16)))))))))
240 (assert
241  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
242  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
243  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
244  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
245  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
246  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
247  (= (select ?x177 (bvadd %5 (_ bv2 64))) ((_ extract 23 16) %16)))))))))
248 (assert
249  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
250  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
251  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
252  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
253  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
254  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
255  (= (select ?x177 (bvadd %5 (_ bv3 64))) ((_ extract 31 24) %16)))))))))
256 (assert
257  (= |#arg_0| arg_0))
258 (assert
259  (= |#arg_1| arg_1))
260 (assert
261  (bvsgt SP (_ bv2048 64)))
262 (assert
263  (= %3 (bvadd SP (_ bv0 64))))
264 (assert
265  (= %4 (bvadd SP (_ bv4 64))))
266 (assert
267  (= %5 (bvadd SP (_ bv8 64))))
268 (assert
269  (= memory_1_byte_write ((_ extract 7 0) arg_0)))
270 (assert
271  (= memory_1 (store memory %3 memory_1_byte_write)))
272 (assert
273  (= memory_2_byte_write ((_ extract 15 8) arg_0)))
274 (assert
275  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
276  (= memory_2 ?x33)))
277 (assert
278  (= memory_3_byte_write ((_ extract 23 16) arg_0)))
```

```
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (= memory_3 (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write))))
(assert
 (= memory_4_byte_write ((_ extract 31 24) arg_0)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (= memory_4 ?x49))))
(assert
 (= memory_5_byte_write ((_ extract 7 0) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (= memory_5 (store ?x49 %4 memory_5_byte_write)))))
(assert
 (= memory_6_byte_write ((_ extract 15 8) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (= memory_6 ?x81)))))
(assert
 (= memory_7_byte_write ((_ extract 23 16) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (= memory_7 (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write))))))
(assert
 (= memory_8_byte_write ((_ extract 31 24) arg_1)))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
       memory_8_byte_write)))
 (= memory_8 ?x95))))))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
       memory_8_byte_write)))
 (= (select ?x95 %3) ((_ extract 7 0) %6)))))))
(assert
 (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
       memory_4_byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
       memory_8_byte_write)))
 (= (select ?x95 (bvadd %3 (_ bv1 64))) ((_ extract 15 8) %6)))))))
(assert
```

```
329  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
330  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
331  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
332  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
333  (= (select ?x95 (bvadd %3 (_ bv2 64))) ((_ extract 23 16) %6)))))))
334 (assert
335  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
336  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
337  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
338  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
339  (= (select ?x95 (bvadd %3 (_ bv3 64))) ((_ extract 31 24) %6)))))))
340 (assert
341  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
342  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
343  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
344  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
345  (= (select ?x95 %4) ((_ extract 7 0) %7)))))))
346 (assert
347  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
348  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
349  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
350  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
351  (= (select ?x95 (bvadd %4 (_ bv1 64))) ((_ extract 15 8) %7)))))))
352 (assert
353  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
354  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
355  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
356  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
357  (= (select ?x95 (bvadd %4 (_ bv2 64))) ((_ extract 23 16) %7)))))))
358 (assert
359  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
360  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
361  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
362  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
363  (= (select ?x95 (bvadd %4 (_ bv3 64))) ((_ extract 31 24) %7)))))))
364 (assert
365  (= %8 (bvadd %6 %7)))
366 (assert
367  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
368  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3_byte_write) (bvadd %3 (_ bv3 64))
         memory__4_byte_write)))
369  (let ((?x81 (store (store ?x49 %4 memory__5_byte_write) (bvadd %4 (_ bv1 64)) memory__6_byte_write)))
370  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7_byte_write) (bvadd %4 (_ bv3 64))
         memory__8_byte_write)))
371  (= (select ?x95 %3) ((_ extract 7 0) %9)))))))
372 (assert
373  (let ((?x33 (store (store memory %3 memory__1_byte_write) (bvadd %3 (_ bv1 64)) memory__2_byte_write
         )))
```

```
374  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
375  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
376  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
377  (= (select ?x95 (bvadd %3 (_ bv1 64))) ((_ extract 15 8) %9)))))))
378 (assert
379  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
380  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
381  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
382  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
383  (= (select ?x95 (bvadd %3 (_ bv2 64))) ((_ extract 23 16) %9)))))))
384 (assert
385  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
386  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
387  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
388  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
389  (= (select ?x95 (bvadd %3 (_ bv3 64))) ((_ extract 31 24) %9)))))))
390 (assert
391  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
392  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
393  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
394  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
395  (= (select ?x95 %4) ((_ extract 7 0) %10)))))))
396 (assert
397  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
398  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
399  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
400  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
401  (= (select ?x95 (bvadd %4 (_ bv1 64))) ((_ extract 15 8) %10)))))))
402 (assert
403  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
404  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
405  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
406  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
407  (= (select ?x95 (bvadd %4 (_ bv2 64))) ((_ extract 23 16) %10)))))))
408 (assert
409  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
410  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
411  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
412  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
413  (= (select ?x95 (bvadd %4 (_ bv3 64))) ((_ extract 31 24) %10)))))))
414 (assert
415  (= %11 (bvmul %9 %10)))
416 (assert
417  (= %12 (bvsgt %8 %11)))
418 (assert
419  (= %12 true))
420 (assert
421  (= memory_9_byte_write ((_ extract 7 0) (_ bv1 32))))
```

```
(assert
 (let ((?x33 (store (store memory %3 memory__1__byte_write) (bvadd %3 (_ bv1 64)) memory__2__byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3__byte_write) (bvadd %3 (_ bv3 64))
      memory__4__byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory__5__byte_write) (bvadd %4 (_ bv1 64)) memory__6__byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7__byte_write) (bvadd %4 (_ bv3 64))
      memory__8__byte_write)))
 (= memory__9 (store ?x95 %5 memory__9__byte_write)))))))
(assert
 (= memory__10__byte_write ((_ extract 15 8) (_ bv1 32))))
(assert
 (let ((?x33 (store (store memory %3 memory__1__byte_write) (bvadd %3 (_ bv1 64)) memory__2__byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3__byte_write) (bvadd %3 (_ bv3 64))
      memory__4__byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory__5__byte_write) (bvadd %4 (_ bv1 64)) memory__6__byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7__byte_write) (bvadd %4 (_ bv3 64))
      memory__8__byte_write)))
 (let ((?x163 (store (store ?x95 %5 memory__9__byte_write) (bvadd %5 (_ bv1 64)) memory__10__byte_write))
      )
 (= memory__10 ?x163)))))))
(assert
 (= memory__11__byte_write ((_ extract 23 16) (_ bv1 32))))
(assert
 (let ((?x33 (store (store memory %3 memory__1__byte_write) (bvadd %3 (_ bv1 64)) memory__2__byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3__byte_write) (bvadd %3 (_ bv3 64))
      memory__4__byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory__5__byte_write) (bvadd %4 (_ bv1 64)) memory__6__byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7__byte_write) (bvadd %4 (_ bv3 64))
      memory__8__byte_write)))
 (let ((?x163 (store (store ?x95 %5 memory__9__byte_write) (bvadd %5 (_ bv1 64)) memory__10__byte_write))
      )
 (= memory__11 (store ?x163 (bvadd %5 (_ bv2 64)) memory__11__byte_write)))))))
(assert
 (= memory__12__byte_write ((_ extract 31 24) (_ bv1 32))))
(assert
 (let ((?x33 (store (store memory %3 memory__1__byte_write) (bvadd %3 (_ bv1 64)) memory__2__byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3__byte_write) (bvadd %3 (_ bv3 64))
      memory__4__byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory__5__byte_write) (bvadd %4 (_ bv1 64)) memory__6__byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7__byte_write) (bvadd %4 (_ bv3 64))
      memory__8__byte_write)))
 (let ((?x163 (store (store ?x95 %5 memory__9__byte_write) (bvadd %5 (_ bv1 64)) memory__10__byte_write))
      )
 (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory__11__byte_write) (bvadd %5 (_ bv3 64))
      memory__12__byte_write)))
 (= memory__12 ?x177)))))))
(assert
 (let ((?x33 (store (store memory %3 memory__1__byte_write) (bvadd %3 (_ bv1 64)) memory__2__byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3__byte_write) (bvadd %3 (_ bv3 64))
      memory__4__byte_write)))
 (let ((?x81 (store (store ?x49 %4 memory__5__byte_write) (bvadd %4 (_ bv1 64)) memory__6__byte_write)))
 (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory__7__byte_write) (bvadd %4 (_ bv3 64))
      memory__8__byte_write)))
 (let ((?x163 (store (store ?x95 %5 memory__9__byte_write) (bvadd %5 (_ bv1 64)) memory__10__byte_write))
      )
 (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory__11__byte_write) (bvadd %5 (_ bv3 64))
      memory__12__byte_write)))
 (= (select ?x177 %5) ((_ extract 7 0) %16)))))))))
(assert
 (let ((?x33 (store (store memory %3 memory__1__byte_write) (bvadd %3 (_ bv1 64)) memory__2__byte_write
      )))
 (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory__3__byte_write) (bvadd %3 (_ bv3 64))
      memory__4__byte_write)))
```

```
467  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
468  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
469  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
470  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
471  (= (select ?x177 (bvadd %5 (_ bv1 64))) ((_ extract 15 8) %16)))))))))
472 (assert
473  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
474  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
475  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
476  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
477  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
478  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
479  (= (select ?x177 (bvadd %5 (_ bv2 64))) ((_ extract 23 16) %16)))))))))
480 (assert
481  (let ((?x33 (store (store memory %3 memory_1_byte_write) (bvadd %3 (_ bv1 64)) memory_2_byte_write
         )))
482  (let ((?x49 (store (store ?x33 (bvadd %3 (_ bv2 64)) memory_3_byte_write) (bvadd %3 (_ bv3 64))
         memory_4_byte_write)))
483  (let ((?x81 (store (store ?x49 %4 memory_5_byte_write) (bvadd %4 (_ bv1 64)) memory_6_byte_write)))
484  (let ((?x95 (store (store ?x81 (bvadd %4 (_ bv2 64)) memory_7_byte_write) (bvadd %4 (_ bv3 64))
         memory_8_byte_write)))
485  (let ((?x163 (store (store ?x95 %5 memory_9_byte_write) (bvadd %5 (_ bv1 64)) memory_10_byte_write))
         )
486  (let ((?x177 (store (store ?x163 (bvadd %5 (_ bv2 64)) memory_11_byte_write) (bvadd %5 (_ bv3 64))
         memory_12_byte_write)))
487  (= (select ?x177 (bvadd %5 (_ bv3 64))) ((_ extract 31 24) %16)))))))))
488 (check-sat)
```