Kapitola 1

Úvod

1.1 Motivace

Exploze stavového prostoru při model checkingu, Partial Order Redukce to řeší, LLVM model checking je super a chceme tam POR.

To, co chceme, je vygenerovaný NTS, který není paralelní. Tedy výstupem redukce není explicitní stavový prostor.

1.2 Přístup

Cílem této práce tedy je vytvořit ze vstupního programu v jazyce LLVM IR, jež využívá posixová vlákna, odpovídající sekvenční přechodový systém v jazyce NTS. Tuto úlohu je možné přirozeně rozdělit na dvě části, a to na

1. přirozený překlad programu z LLVM IR do NTS se zachováním paralelismu (část dále označovaná jako překlad), a
2. převod paralelního programu v jazyce NTS na sekvenční program (část dále označovaná jako sekvencializace).

Tyto dvě části jsou poměrně nezávislé a mohou být implementovány jako rozdílné nástroje, pracující nad společnou knihovnou. Rozdělením navíc získáváme možnost ověřovat vlastnosti sekvenčních programů v jazyce LLVM IR nástrojem, který rozumí jazyku NTS .

Až na uvedená rozšíření. Nechtěli jsme mít možnost použít unbounded integery?

1.2.1 Funkční volání

Chceme tedy z paralelního NTS, který vznikl překladem z LLVM IR, učinit sekvenční. Pokud o vstupním paralelním NTS nebudeme nic předpokládat, nástroj pro sekvencializaci bude muset umět korektně zacházet s funkčními voláními, protože ta mohou být součástí NTS. To zejména znamená, že nástroj si bude muset pro každý stav výstupního systému udržovat přehled o tom, jaké funkce jsou v jakém vlákně aktivní. Implementace takového nástroje nemusí být snadná, proto jsme učinili následující rozhodnutí:

Rozhodnutí 1. *Nástroj pro sekvencializaci předpokládá, že vstupní paralelní systém neobsahuje funkční volání, tedy (v termínech) NTS není hierarchický. Takové NTS budeme dále označovat jako "ploché".*

Většina skutečných programů v jazyce LLVM IR ovšem obsahují funkční volání, a rádi bychom takové programy podporovali. Nabízejí se dvě možnosti, jak takovou podporu zajistit. První spočívá v tom, že paralelní program v jazyce LLVM IR přeložíme na paralelní a (potenciálně) hierarchický přechodový systém v jazyce NTS, který následně převedeme na ekvivalentní paralelní plochý přechodový systém. Alternativní možností je nejprve odstranit funkční volání z paralelního LLVM IR programu a následně tento plochý program převést na plochý paralelní přechodový systém. Oba způsoby však znamenají, že programy s rekurzivním voláním nebudou podporovány.

Zplošťování LLVM Zploštění programu v LLVM IR nevyžaduje téměř žádnou další práci, protože již na tuto úlohu existuje llvm průchod. Jmenuje se inline a je součástí projektu LLVM. Nicméně nástroj pro překlad LLVM IR na NTS musí umět korektně přeložit volání funkce pthread_create(3), které v tomto případě nemůže být přeloženo jako volání BasicNts na úrovni NTS. Volání této funkce by tedy muselo být vhodně přeloženo již na úrovni LLVM kódu.

Zplošťování NTS Přestože je pro zplošťování NTS potřeba udělat o něco více práce, napsání odpovídajícího nástroje nám umožní sekvencializaci libovolného NTS - ne jen toho, které vzniklo překladem z LLVM, ale libovolného hierarchického NTS.

Rozhodnutí 2. *Protože kvůli rozhodnutí 1 nástroj na sekvencializaci NTS podporuje pouze ploché NTS, vytvoříme nástroj, který z hierarchického NTS udělá nehierarchické (ploché) NTS.*

1.2.2 Architektura

Celý problém se tedy rozpadá na několik částí. Nejprve potřebujeme přeložit program z jazyka LLVM IR do formalizmu NTS, poté z přeloženého programu odstranit volání subsystémů, plochý program sekvencializovat s využitím partial order redukce a nakonec ho uložit do souboru. Pro spolupráci těchto částí je třeba mít vybudovanou vhodnou paměťovou reprezentaci programu v jazyce NTS. Existence parseru by byla užitečná (zejména pro testování), ale není nutná.

1.3 Popis následujících kapitol

Kapitola 2

Použité technologie

2.1 LLVM

Projekt LLVM [1] je sada knihoven a nástrojů, tvořící infrastrukturu pro tvorbu překladačů. Ke své činnosti využívá mezijazyk známý jako LLVM intermediate representation [2] (dále LLVM IR nebo jen IR) a sadu dobře definovaných softwarových rozhraní pro manipulaci s ním. Práci překladače vybudovaného nad LLVM lze rozdělit do několika fází: nejprve je zpracován kód ve vstupním jazyce (proběhne lexikální, syntaktická a sémantická analýza), poté je vygenerován mezikód v jazyce LLVM IR, nad tímto pak proběhnou zvolené optimalizace, z výsledného IR kódu je vygenerován platformově specifický kód a proběhne linkování. Tato architektura umožňuje využít při tvorbě překladače již jednou napsaných částí [3].

2.1.1 LLVM IR

LLVM IR je typovaný jazyk podobný jazyku symbolických adres, od kterého se však liší v několika zásadních vlastnostech.

- LLVM IR je platformově nezávislý a není určený pro žádný fyzicky existující procesor.
- V jazyce LLVM IR je možné používat neomezené množství registrů.
- Do každého registru je možné přiřadit hodnotu pouze jednou. Tato vlastnost je nazývána *static single assignment form* (dále *SSA*).

Kód je uchováván v modulech a rozčleněn do funkcí, které se skládají ze základních bloků. Základní blok (basic block, dále blok) je taková posloupnost instrukcí, která má jeden vstupní bod a jeden výstupní. Zejména tedy není možné provádět skoky dovnitř bloku nebo vyskočit z bloku jinde, než na konci. Sémantika tohoto jazyka je dobře zdokumentována, navíc probíhá práce na její formalizaci [4].

Instrukční sada

Instrukční sada LLVM IR je svou jednoduchostí podobná RISCovým instrukčním sadám. Instrukce jsou rozděleny na ukončovací, binární, bitové, paměťové a os-

tatní, přičemž pouze ukončovací instrukce mohou měnit tok řízení. Tyto instrukce vždy ukončují Basic Block a rozhodují, který blok bude vykonán po dokončení aktuálního.

Příklad

Obrázek 2.1 zobrazuje funkci, vygenerovanou nástrojem clang [5] z kódu v jazyce C. Funkce je globálním symbolem (globální symboly mají prefix `@`), má návratovou hodnotu `i32` (dvaatřicetibitové celé číslo) a jako parametr přijímá proměnnou téhož typu. V těle funkce se vyskytuje pouze jedna ukončovací instrukce, totiž `ret`, a celá funkce je tak tvořena jedním basic blockem. Symbol `%1` označuje výsledek instrukce `alloca`, což je lokální proměnná typu ukazatel na `i32` (lokální symboly mají prefix `%`). Následuje instrukce `store`, která zkopíruje hodnotu parametru `%x` na právě alokované paměťové místo a nevrací žádnou hodnotu. Po načtení hodnoty z paměti a přičtení jedničky je vrácen výsledek poslední operace.

```
define i32 @add(i32 %x) #0 {
%1 = alloca i32, align 4
store i32 %x, i32* %1, align 4
%2 = load i32* %1, align 4
%3 = add nsw i32 1, %2
ret i32 %3
}
```

Obrázek 2.1: Ukázka IR kódu

2.1.2 Průchody

Průchod (orig. pass) je softwarový modul, který pracuje nad kódem v LLVM IR. Průchody se dají rozdělit na analytické a transformační: analytické kód nijak nemodifikují, ale počítají nějakou užitečnou informaci; transformační ze vstupního kódu generují jiný. Typickým příkladem transformačního průchodu je eliminace mrtvého kódu (dead code elimination, DCE). Nástroj využívající LLVM si může zvolit, které průchody budou spuštěny; také je možné spouštět je ručně pomocí nástroje opt, dodávaného spolu s LLVM.

2.1.3 Využití

V současné době existují nástroje pro software model checking (model checkery), které přijímají model v jazyce LLVM IR. Mezi ně patří například Divine [6]. Využití tohoto jazyka přináší celou řadu výhod. Zaprvé, pro velké množství překladačů do LLVM IR je model checker méně závislý na programovacím jazyku, v němž je napsaný ověřovaný software. Dále, před vlastní model checking lze zařadit již napsané průchody, které nástroji poskytnou užitečné informace nebo zrychlí model checking samotný.

Vztah k této práci

Z uvedených důvodů budeme i v této práci jako vstupní jazyk využívat právě LLVM IR. Nicméně protože sémantika tohoto jazyka není definovaná formálně a v oblasti model checkingu se obvykle pracuje s přechodovými systémy (jako jsou Kripkeho struktury [7]), v této práci bude výhodné pracovat s jazykem popisujícím přechodový systém.

2.2 Posix threads

Jazyk LLVM IR sám neposkytuje podporu pro explicitní paralelismus. Té je možné docílit pouze použitím specializovaných knihoven. Knihovna Posix threads (zkráceně pthreads) je standardizovanou knihovnou s rozhraním pro jazyk C, která obsahuje funkce pro manipulaci s vlákny.

2.2.1 pthread_create()

Nejzajímavější funkcí z této knihovny je `pthread_create`[8] (viz obrázek 2.2.1), která spustí funkci, jež dostala jako argument parametru `start_routine`, v nově vytvořeném vlákně. Funkce dále uloží ID vlákna do proměnné `*thread` (nesmí být `null`), argument parametru `arg` předá jako parametr funkci `start_routine` a o úspěchu či neúspěchu informuje volajícího prostřednictvím návratové hodnoty.

Protože podle normy POSIX.1-2001 je `pthread_t` neprůhledný (opaque) datový typ, prototyp této funkce v jazyce LLVM IR je platformově závislý. Na obrázku 2.2.1 se nachází prototyp `pthread_create`, jak je k dispozici v systému Fedora 21 x86_64 s GNU libc verze 2.20 (pthreads jsou součásti glibc).

reference

```
int pthread_create ( pthread_t *thread,
                     const pthread_attr_t *attr,
                     void *(*start_routine) (void *),
                     void *arg                      );
```

Obrázek 2.2: Prototyp funkce `pthread_create` v jazyce C

```
declare i32 @pthread_create (
        i64*, %union.pthread_attr_t*, i8* (i8*)*, i8* )
```

Obrázek 2.3: Prototyp funkce `pthread_create` v jazyce LLVM IR

2.3 NTS

Jazyk NTS (Numerical Transition Systems) je jednoduchý jazyk s formalizovanou sémantikou [9], sloužící pro popis numerických přechodových systémů. NTS mohou modelovat libovolný software [9] a protože software je obvykle strukturován do menších částí, NTS obsahuje konstrukce pro hierarchickou i paralelní

kompozici systémů. Následuje stručný popis tohoto jazyka, zájemci o preciznější popis mohou nahlédnout do [9].

Zmínit globální proměnné

2.3.1 BasicNts

Základní jednotkou NTS je BasicNts, což je samostatný přechodový systém, sestávající z proměnných, řídících stavů a přechodů mezi nimi. Stavy mohou být označeny jako iniciální, finální a chybové. Přechody jsou tvaru $s_i \xrightarrow{R} s_j$, kde R je přechodové pravidlo, strážící přechod. Přechod může být vykonán pouze v případě, že je přechodové pravidlo naplněno.

2.3.2 Přechodová pravidla

Přechodová pravidla mohou být dvojího druhu: volací pravidlo (například $(p1', p2') =$ `factorize`(x)) slouží k zavolání jiného BasicNts, předání parametrů volanému a uložení výsledků volání, zatímco formulové pravidlo (například $d' * d' = x$) obsahuje formuli predikátové logiky prvního řádu, jejíž splnění umožňuje přechod do cílového stavu přechodu.

2.3.3 Formule

Formule mohou být kvantifikované a jsou složeny pomocí logických spojek z atomických propozic a jiných formulí, přičemž za atomické propozice jsou považovány zejména booleovské termy a výsledky porovnání termů, navíc také `havoc` (viz 2.3.7). Nutno poznamenat, že ve formuli se mohou vyskytovat i hodnoty proměnných, platné v cílovém stavu. Tyto jsou označené znakem `'`. Jak je vidět u předchozího příkladu, je tak možné vytvořit formuli, která požaduje, aby hodnota proměnné y v příštím stavu byla odmocninou hodnoty proměnné x ve stavu současném.

2.3.4 Termy

Termy jsou již na syntaktické úrovni rozděleny na aritmetické a booleovské. Mezi booleovské termy patří booleovské konstanty, booleovské proměnné a jiné termy, pospojované obvyklými logickými operacemi. Obdobně, aritmetické termy jsou reálné a celočíselné konstanty a proměnné, pospojované aritmetickými operacemi z množiny $\{+, -, *, /, \%\}$. Jazyk explicitně vyžaduje, aby všechny subtermy každého termu měly stejný datový typ, jako celý term.

2.3.5 Typový systém

Jazyk NTS je vybaven třemi skalárními datovými typy: Int, Bool, Real. Datový typ `Int` reprezentuje matematické celé číslo, jeho doménou je tedy množina $\mathbb{Z}$; datový typ `Real` reprezentuje matematické reálné číslo, tedy jeho doménou je množina $\mathbb{R}$; nakonec typ `Bool` nabývá pouze hodnoty z množiny $\{$`true`, `false`$\}$. Jazyk dále podporuje type pole hodnot libovolného skalárního aritmetického typu.

2.3.6 Sémantika

Konfigurací systému se nazývá dvojice $\langle q, v \rangle$, kde q je jedním z řídících stavů a v valuace proměnných (tedy funkce, která každé proměnné přiřazuje hodnotu). Přechod z konfigurace $\langle q_1, v_1 \rangle$ do konfigurace $\langle q_2, v_2 \rangle$ je možný, pokud existuje přechodové pravidlo $q_1 \xrightarrow{F} q_2$ s následující vlastností: formule vzniklá z F nahrazením proměnných p hodnotami $v_1(p)$ a nahrazením proměnných ve tvaru p' hodnotami $v_2(p)$ je tautologií. Formálnější popis je k dispozici v [9].

2.3.7 Havoc

Havoc je speciální atomická propozice, jejíž účelem je zamezit samovolné modifikaci proměnných neuvedených ve formuli přechodového pravidla. Ve své podstatě se jedná o syntaktickou zkratku.

$$\texttt{havoc}\,(v_1, v_2, \ldots, v_k) = \bigwedge_{v \in V \setminus \{v_1, v_2, \ldots, v_k\}} v = v\prime \tag{2.1}$$

Její význam si můžeme ukázat na následujícím příkladu: uvažme přechodový systém s proměnnými x, y typu int, konfiguraci $s_1 = \langle q_1, v_1 \rangle \wedge v_1(x) = 0 \wedge v_1(y) = 3$, konfiguraci $s_2 = \langle q_2, v_2 \rangle \wedge v_2(x) = 1 \wedge v_2(y) = 5$ a přechod $q_1 \xrightarrow{F} q_2$. Pokud $F \equiv x' = x + 1$, pak je možné s využitím uvedeného pravidla přejít z konfigurace q_1 do konfigurace q_2, protože formule $1 = 0 + 1$ je tautologií. Tento přechod ale v rozporu s očekáváním modifikuje proměnnou y. Naopak, pokud $F \equiv x' = x+1 \wedge \texttt{havoc}(x)$, potom uvedené pravidlo nelze použít pro přechod z q_1 do q_2. Po dosazení a expandování `havoc` totiž vznikne formule $1 = 0+1 \wedge 3 = 5$, která je nesplnitelná.

2.3.8 Paralelismus

Jazyk NTS umožňuje paralelní běh libovolného konečného počtu vláken. Ke každému vláknu je přiřazen *vstupní bod* (entry point), což je BasicNts, který je vykonáván v kontextu daného vlákna. Paralelní NTS obsahuje specifikaci, tvořenou seznamem dvojic `vstupní bod [ počet vláken ]`. Identifikátory vláken jsem vláknům přiřazeny vzestupně od nuly, a to ve stejném pořadí, v jakém jsou zadány ve specifikaci. Uvážíme-li příklad 2.4, paralelní NTS obsahuje $N+1$ vláken, přičemž vstupním bodem vláken s `tid` $\in \{0, \ldots, N-1\}$ je BasicBlock `worker_nts` a vstupním bodem vlákna s `tid` $= N$ je BasicBlock `main_nts`.

```
instances worker_nts[N], main_nts[1];
```

Obrázek 2.4: Specifikace paralelně vykonávaných vláken v jazyce NTS

2.3.9 Příklad

Na obrázku 2.5 se nachází příklad jednoduchého paralelního systému s jedním producentem a jedním konzumentem, reprezentovanými BasicNts `producent` a `consument`. Po deklaraci globálních proměnných `G` a `c` je uvedena formule, kterou musí splňovat iniciální konfigurace (viz 2.3.6) paralelního systému. Systém bude

obsahovat dvě vlákna, přičemž vlákno s `tid` = 0 bude vykonávat kód BasicNts `producent` a vlákno s `tid` bude vykonávat kód `consument`.

BasicNts `producent` obsahuje lokální proměnnou `i`, která je přechodem z iniciálního stavu `si` nastavena na hodnotu 0 a každým dalším přechodem inkrementována. Přechod `sl` → `sl` se může uskutečnit pouze v případě, že globální proměnná `c` je nastavena na hodnotu `false`, a sám tuto proměnnou nastaví na `true`. Naopak, přechod `sl` → `sl` v `consument` může být vykonán pouze v případě, že `c` = `true`. Činnost celého systému je taková, že `producent` postupně do proměnné `G` ukládá zvyšující se hodnoty, které `consument` sčítá.

Mimochodem, již zmíněný přechod `sl` → `sh` v `consument` ničí hodnotu uloženou v `G`, což ale nevadí, protože `G` není nikdy čtena.

```
G : int;
c : bool;
init G = 0 && l = false;
instances producent[1], consument[1];

producent {
    initial si;
    i : integer;
    si -> sl { i' = 0 && havoc(i)              }
    sl -> sl { c  = false && c' = true  &&
               G' = i     && i' = i + 1 &&
               havoc(c, G, i)                  }
}

consument {
        initial si;
        sum, x : integer;
        si -> sl { sum' = 0 && havoc( sum )              }
        sl -> sh { c = true && x' = G && havoc(x, G) }
        sh -> sl { sum' = sum + x && c' = false &&
                   havoc(sum, c)                         }
}
```

Obrázek 2.5: Příklad kódu v jazyce NTL

2.4 Partial Order Reduction

Partial Order Reduction je technika pro redukci stavového prostoru (paralelních) programů, která využívá komutativity paralelně vykonávaných přechodů. Pokud lze efektivně ověřit, že stav programu po vykonání přechodů t_1 a t_2 nezávisí na jejich pořadí a ověřovaná vlastnost není citlivá na jednotlivé přechody, není třeba uvažovat některé možné běhy. Technika použitá v této práci vychází z techniky popsané v [7].

2.4.1 Nezávislost a komutativita přechodů

Na rozdíl od [7] nevyžadujeme, aby přechodová pravidla byla (datově) deterministická. Všechny přechody v NTS již jsou kontrolně deterministické. Nezávislost přechodů tedy definujeme mírně odlišně.

Definice 1. *Binární relace I na přechodech paralelního NTS nazýváme relací nezávislosti, pokud je symetrická, antireflexivní a splňuje následující požadavek: Pro libovolné přechody $(t_1, t_2) \in I$ a libovolné stavy (konfigurace v NTS) s_1, s_2, s_3 takové, že $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3$ existuje stav s'_2 takový, že $s_1 \xrightarrow{t_2} s'_2 \xrightarrow{t1} s_3$.*

Pro NTS bez nedeterministických přechodů je tato definice shodná s definicí v [7].

Definice 2. *Přechod t je povolený ve stavu (konfiguraci) s (píšeme $t \in \texttt{enabled}(s)$ právě tehdy, když existuje stav s' tak že $s \xrightarrow{t} s'$*

Je zřejmé, že pokud pro stav $s = \langle (\rangle q, v)$ a přechod t platí $t \in \texttt{enabled}(s)$, pak $t \in \texttt{outgoing}(q)$. Stavy takové, že vybraný $\texttt{ample}(s) = \texttt{enabled}(s)$, označujeme jako plně expandované.

Technika, použitá v [7], tedy partial order reduction s využitím dostatečných (ample) množin, spočívá v tom, že pro každý stav (konfiguraci) paralelního systému je spočítána množina přechodů $\texttt{ample}(s) \subseteq \texttt{enabled}(s)$, která má tu vlastnost, že přechody mimo $\texttt{ample}(s)$ nejsou "důležité". Formálněji řečeno je třeba, aby ke každé cestě v úplném explicitním přechodovém grafu paralelního systému existovala (určitým způsobem) ekvivalentní cesta v tomtéž grafu, která nevyužívá žádné přechody mimo `ample`. Při konstrukci odpovídajícího sekvenčního systému pak není třeba uvažovat přechody mimo `ample`, což zmenšuje velikost výsledného sekvenčního systému.

2.4.2 Nutné podmínky

Kniha [7] uvádí několik podmínek, postačují pro zachování korektnosti. Pro připomenutí, ta kniha pracuje pouze s deterministickými přechody.

C0 $\texttt{ample}(s) = \emptyset \Rightarrow \texttt{enabled}(s) = \emptyset$

C1 Každá cesta v úplném explicitním přechodovém grafu paralelního systému, začínající ve stavu s, má následující vlastnost: pokud přechod t_2, který se vyskytuje po cestě, závisí na nějakém přechodu $t_1 \in \texttt{ample}(s)$, pak se před jeho výskytem vyskytuje nějaký přechod $t'_1 \in \texttt{ample}(s)$.

C2 Pokud s není plně expandovaný, pak každý přechod z $\texttt{ample}(s)$ je neviditelný . zadefinovat

C3 Nesmí existovat cyklus, který obsahuje přechod povolený v některém z jeho stavů a neobsažený v žádném z `ample` stavů cyklu.

2.4.3 Jednoduchá heuristika

Nezávislost

Definice relace nezávislosti umožňuje jistou flexibilitu při výběru vhodné relace. Za nezávislé budeme považovat ty páry přechodů (t_1, t_2), které splňují obě následující podmínky:

- Přechody t_1 a t_2 patří ke dvěma různým vláknům.
- Neexistuje proměnná, sdílená oběma přechody t_1 i t_2 a modifikovaná alespoň jedním z nich.

Ample sety

Pro řídící stav $q_0 = \langle q_{0,0}, \ldots, q_{0,k-1} \rangle$ spočítáme $A(q_0)$ následujícím způsobem. Pro každé $0 \leq i < k$ zkusíme položit $A(q_0) = \texttt{outgoing}(q_{0,i})$. Pokud zkoušená množina splňuje podmínky C0 až C3, použijeme jí. Pokud žádná z vyzkoušených množin není použitelná, položíme $A(q_0) = \texttt{outgoing}(q_0)$.

Popsat pro obyčejnou POR. Navíc funkci A definujeme až u statické POR

Porušení podmínky C1 znamená, že v úplném explicitním přechodovém grafu existuje posloupnost $s_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{m-1}} s_m \xrightarrow{\beta} s_{m+1}$, $s_j = (q_j, v_j)$ taková, že $\beta \in \texttt{enabled}(s_l)$, β je závislý na nějakém přechodu ve zvoleném $\texttt{ample}(s_0)$ a všechny α_j jsou nezávislé na všech přechodech v $\texttt{ample}(s_0)$. Potom, jak uvádí [7], mohou nastat dvě situace.

- Přechod β pochází z vlákna i (tedy $\beta \in \texttt{outgoing}(q_{m,i})$). Protože α_j jsou nezávislé na $\texttt{ample}(s_0)$, musí pocházet z jiného vlákna než i, tedy $q_{0,i} = q_{m,i}$. Protože $\beta \notin \texttt{ample}(s_0)$, tak $\beta \notin \texttt{enabled}(s_0)$. V tom případě se ale stavy s_0 a s_m liší ve valuaci některých globálních proměnných, které jsou používány přechodem β, a tedy nějaké vlákno s $tid \neq i$ obsahuje přechod, modifikující proměnnou, která je používána přechodem β.
- Pokud $\beta \notin \texttt{outgoing}(q_{m,i})$, tedy přechod β pochází z vlákna jiného než i, nějaké jiné vlákno než i obsahuje přechod, který je závislý na nějakém přechodu z $\texttt{ample}(s_0)$.

V obou případech tedy musí nějaké jiné vlákno obsahovat přechod, který s nějakým přechodem z $\texttt{ample}(s_0)$ sdílí proměnnou, která je modifikována jedním z nich. To se dá snadno ověřit, pokud si například předem pro každé vlákno množinu proměnných v něm použitých.

Kapitola 3

Překlad LLVM na NTS

3.1 Omezení LLVM

3.1.1 Pole

Nepodporuji (zatím). Přidat podporů některých operací nad poli by ale nemusel být problém

3.1.2 Pointery

Překládací nástroj zatím obecně neumí pracovat s ukazateli, zejména nepodporuje pointerovou aritmetiku. Nicméně protože jazyk LLVM pointery hojně využívá, a to zejména pro práci s globálními proměnnými a lokálními zásobníkově alokovanými proměnnými, překládací nástroj umí korektně přeložit několik speciálních případů použití ukazatelů. Mezi ně patří

- čtení a zápis globálních proměnných pomocí instrukcí `load` a `store`,
- čtení a zápis proměnných alokovaných pomocí instrukce `alloca` a
- předávání parametrů funkci `thread_create`.

Ale s proměnnými se pracuje zkrze pointery!

3.1.3 Instrukce

Také nepodporuji znaménkovou aritmetiku a hromadu instrukcí. Například znaménkové přetečení při add neošetřuji, přestože je to principiálně možné. Vede to k složitým formulím. Pokud ale bude překladová utilita vyvíjená dále, je to jedna z věcí, která by mohla být volitelná (tedy jestli vracet nedefinovaný výsledek nebo přejít do err stavu nebo pokračovat jako nic). S tím souvisí Poison a Undef values.

3.1.4 Znaménkovost

Jak číselné datové typy v jazyce LLVM IR, tak přidaný BitVector typ v jazyce NTS jsou bezznaménkové. Nicméně některé LLVM instrukce (například ICMP) mohou interpretovat použité proměnné znaménkově. Přestože je principiálně

možné vyjádřit sémantiku znaménkových operací pomocí bezznamnékové aritmetiky , výsledné formule by byly složitější. Tato instrukce přiřadí do proměnné

ukázat jak na to

```
%b = icmp slt i8 %x, 10
```

zminit semantiku operaci nad BitVector

'b' hodnotu 1 právě pokud x je znaménkově menší (signed less then, slt) než hodnota 10, tedy právě pokud x bude mít hodnotu v rozsahu -128 až 9. Tento rozsah, vyjádřený v dvojkovém doplňkovém kódu , odpovídá neznaménkovým hodnotám v rozsahu 0 ... 9 nebo 128 ... 255.

Někde zmínit, že llvm ir využívá právě dvojkový doplněk

```
%b = icmp slt i8 %x, %y
```

Pokud jsou ovšem obě hodnoty neznámé, porovnání je obtížnější. Výsledkem operace bude 1, právě pokud bude splněna jedna z následujících podmínek:

1. Hodnota proměnné x je záporná a hodnota proměnné y je kladná, tedy v unsigned aritmetice x $>$ 127 a y $<=$ 127.
2. Obě proměnné mají shodné znaménko a zároveň x je neznaménkově menší než y.

Jsou i jiné způsoby, jak v neznaménkové aritmetice vyjádřit chování této instrukce, ale nenašel jsem žádný elegantnější nebo jednodušší. Z tohoto důvodu současná verze překládacího nástroje nepodporuje překlad znaménkových operací do aritmetiky neznaménkových bitvektorů.

3.2 Omezení Posix Threads

Přestože knihovna posixových vláken poskytuje velkou množinu operací s vlákny (`pthread_create`, `pthread_join`, `pthread_exit`, `pthread_cancel`, ...) a některá synchronizační primitiva (`pthread_mutex_lock`), v současné době z této knihovny podporujeme pouze funkci `pthread_create`.

3.2.1 Funkce `pthread_create`

Některá omezení, která klademe na použití funkce `pthread_create`, plynou zejména z toho, že téměř nepodporujeme práci s ukazateli. Funkce je použitelná za dodržení následujících podmínek:

1. První argument je ukazatel se snadno spočitatelnou hodnotou. Přesněji řečeno, argument by měl být typu `llvm::GlobalVariable` nebo `llvm::AllocaInst`.
2. Argument parametru `start_routine` musí být přímo funkce, nikoliv jiný funkční ukazatel. Přesněji řečeno, argument by měl být typu `llvm::Function`
3. Všechny ostatní argumenty funkce `pthread_create` musí být null.
4. Návratová hodnota funkce `pthread_create` musí být ignorována.

5. Funkce, předaná jako argument parametru `start_routine`, nesmí používat svůj parametr.

6. Návratová hodnota této funkce musí být vždy null.

Těmto omezením odpovídá například fragment kódu z obrázku 3.1.

```
void * f ( void * ) { ... return NULL; }
...
pthread_t t;
pthread_create ( &t, NULL, f, NULL );
```

Obrázek 3.1: Možné použití funkce `pthread_create`

3.2.2 Funce `pthread_join`

Funkce `pthread_join` zatím není v aktuální verzi implementována, nicméně její implementace by měla být poměrně přímočará, jak uvidíme později v sekci . Pokud bude překládací nástroj dále vyvíjen, funkce `pthread_join` se pravděpodobně implementace dočká.

TODO

3.2.3 Funce `pthread_exit`

Naopak, funkce `pthread_exit` implementována není a možná ani nebude, a to ze dvou důvodů.

1. Její absence příliš nevadí, protože její význam je stejný jako návrat z hlavní funkce vlákna.

2. Tato funkce příliš drasticky mění tok řízení, zejména není-li zavolána z hlavní funkce vlákna, nýbrž z funkce vnořené. Její zavolání v jazyce NTS by mělo způsobit opuštění všech zásobníkových oken, což není v současné implementaci překladu funkcí snadno proveditelné.

Druhý důvod by nebyl problém v případě, že bychom překládali již zploštělý LLVM IR kód. Tato možnost je však není zcela v souladu s rozhodnutím 2. Z druhého důvodu také není implementována ani funkce `pthread_cancel`.

Jop, ale mohli bychom překládat funkce tak, že by BasicBlocky měly jednu speciální návratovou proměnnou navíc, tedy procedury by vracely jednu hodnotu a funkce dvě. Speciální návratová hodnota by mohla informovat režijní kód o podobných

3.3 Rozšíření NTS

3.3.1 Typ BitVector

Jak můžeme vidět v předchozích částech, typový systém jazyka NTS se od typového systému LLVM IR v mnohém liší. Zatímco v LLVM IR mohou stejný datový typ (Integer type, například i8) využívat jak aritmetické instrukce (binary instructions), tak bitové logické instrukce (bitwise binary instructions), jazyk NTS již na úrovni syntaxe podporuje logické operace pouze nad termy typu Bool. Pro překlad LLVM IR do NTS je tedy nezbytné cílový jazyk vhodným způsobem rozšířit. Možností je více:

1. Rozšířit logcké operace i nad datový typ Int. Jaký by ale byl význam například negace? Totiž, pro různou bitovou šířku dává negace různý výsledek. Budeme-li mít například proměnnou typu Int s hodnotou 5 (binárně 101), pokud se na ní budeme dívat jako na 4 bitové číslo (tedy 0101), dostaneme po negaci 10 (1010), pokud se na ní budeme dívat jako na 3 bitové číslo, dostaneme po negaci hodnotu 2 (binárně 010).

2. Zavést speciální operátory, které budou v sobě obsahovat informaci o uvažované bitové šířce.

3. Zavést datový typ BitVector<k>, který je vlastně k-ticí typu Bool. Na něm můžeme dělat logické operace bitově a aritmetické operace jako na binárně reprezentovaném čísle.

Volíme třetí způsob.

V syntaxi původního nts je rozlišováno mezi aritmetickým a booleovském literálem. Chtěli bychom nějak zapisovat bitvectorové hodnoty. Ve hře jsou dvě možnosti:

1. Pro zápis konstant typu BitVector<k> bychom mohli používat speciální syntaxi. Například zápis 32x"01abcd42"může představovat konstantu typu BitVector<32>, zapsanou v hexadecimálním tvaru. Toto řešení je navíc vhodné pro zápis speciálních konstant (jako 2^{31}) v lidsky čitelném tvaru.

2. Aritmetické literály mohou mít schopnost být Int-em nebo BitVector<k>-em podle potřeby (polymorfismus?).

Použita byla druhá možnost (také si jí tu podrobněji popíšeme). V případě potřeby můžeme do jazyka přidat i tu první, ale už ne kvůli potřebě rozlišovat mezi datovými typy, ale kvůli užitečnosti zapisovat některé konstanty hezky.

Syntaxi jazyka zjednodušíme tak, že definujeme jeden neterminál <term> místo dvou neterminálů <arith-term> a <bool-term>. S každým termem bude ale spojená sémantická informace, kterou je jeho datový typ. Definujeme následující skalární datové typy:

1. Int

2. BitVector<k>, $\forall k \in \mathbb{N}^+$

3. Real

Původní datový typ Bool chápeme jako zkratku za typ BitVector<1>

⟨*literal*⟩ ::= ⟨*id*⟩ | tid | ⟨*numeral*⟩ | ⟨*decimal*⟩

⟨*aop*⟩ = '+' | '-' | '*' | '/' | '%'

⟨*bop*⟩ = '&' | '|' | '->' | '<->'

⟨*op*⟩ = ⟨*aop*⟩ | ⟨*bop*⟩

⟨*term*⟩ ::= ⟨*literal*⟩ | ⟨*term*⟩ ⟨*aop*⟩ ⟨*term*⟩ | ⟨*term*⟩ ⟨*bop*⟩ ⟨*term*⟩ | '(' ⟨*term*⟩ ')'

3.3.2 Typovací pravidla

Abychom zajistili, že syntakticky stejný výraz může být typu BitVector nebo Int, definujeme typovou třídu Integral se členy BitVector<k> a Int. Typ výrazu je definován rekurzivně

TR1 Numerická konstanta <numeral> je libovolného typu 'a' z třídy Integral.

TR2 <decimal> je typu Real

TR3 tid je libovolného typu 'a' z třídy Integral

TR4 <id> je stejného typu, jako odpovídající proměnná

Mějme termy
a1 :: Integral a => a
a2 :: Integral b => b
b1 :: BitVector<k1>
b2 :: BitVector<k2>
i1 :: Int
i2 :: Int

Potom: a1 <aop> a2 :: Integral a => a a1 <op> b1 :: Bitvector<k1> a1 <aop> i1 :: Int b1 <op> b2 :: BitVector<maxk1,k2> i1 <aop> i2 :: Int

Tato pravidla platí i komutativně. Co se do nich nevejde, není typově správný výraz.

Tedy přidáváme datový typ BitVector, typové třídy a implicitní typová konverze.

Btw můžeme využívat anotace. Zatím je využíváme jenom trochu, ale dají se o nich vkládat i nějaké informace z LLVM nebo další postřehy.

3.4 Vlastni preklad

3.4.1 Funkce

Jak jazyk NTS, tak LLVM IR podporují určitou formu hierarchického členění programu na podprogramy. Zatímco u LLVM IR je základní jednotkou tohoto členění funkce, v případě jazyka NTS jde o BasicNts, který reprezentuje jednoduchý přechodový systém. Jak funkce, tak BasicNts mohou mít libovolný počet (vstupních) parametrů zvoleného typu, ašak zatímco funkce může vracet nejvýše jednu hodnotu, BasicNts může definovat libovolný počet výstupních parametrů. Funkce i BasicNts navíc mohou definovat lokální proměnné. Vzhledem k těmto podobnostem je celkem přirozené překládat funkce z LLVM IR na BasicNts.

3.4.2 BasicBlocky

Každá funkce se skládá z BasicBlocků. Protože žádná instrukce uvnitř základního blocku nemění tok řízení a po jejím vykonání je vykonávána instrukce následující, můžeme každý základní blok s N vnitřními instrukcemi přeložit jako N nových řídících stavů, kde n-tý stav odpovídá stavu programu po vykonání n instrukcí. Sémantika jednotlivých vnitřních instrukcí bude poté zachycena v přechodových pravidlech mezi jednotlivými stavy.

Terminující instrukce Nicméně poslední instrukce v základním bloku může změnit tok řízení, například ukončit vykonávání funkce nebo skočit na začátek jiného základního bloku. Třída takovýchto instrukcí se nazývá "terminační instrukce", protože tyto instrukce ukončují základní blok. Odpovídající přechody tedy nemusí vést do nového stavu, ale do již existujícího stavu jiného základního bloku.

V jazyce LLVM IR existuje instruce Phi, reprezentující Φ uzel v SSA jazyce. Tato instrukce se nachází pouze na začátku základního bloku a její výsledek je závislý na předchozím dokončeném základním bloku. Z tohoto důvodu jsou při překladu funkce základní bloky očíslovány a každá terminující instrukce ukládá číslo svého základního bloku do speciální proměnné. Samotná phi instrukce však v době psaní tohoto textu není v překládacím nástroji implementována.

Zmínit SSA

známe z teorie překladačů?

3.4.3 Paralelismus

Jak je vidět ze sekcí 2.2 a 2.3.8, jazyky LLVM IR a NTS implementují paralelismus velice odlišně. Asi nejvýznamějším rozdílem je, že použití pthreads v jazyce LLVM IR umožňuje vytvářet vlákna kdykoliv, a že pro množství programů není možné určit, kolik vláken nakonec použijí (problém zastavení). Existuje více způsobů, jak se se zmiňovanou rozdílností vypořádat, zde však uvádíme pouze dva nejvýraznější. K oběma uvedeným způsobům se vztahuje kód z obrázku 3.2, který je zde do jazyka NTS přeložen postupně oběma způsoby.

Je ta nemožnost a redukce na PZ zřejmá?

```
void * p1 ( void * ) { ... }
void * p2 ( void * ) { ... }

int main ( void ) {
        pthread_t t1, t2;
        pthread_create ( &t1, NULL, p1, NULL );
        pthread_create ( &t2, NULL, p2, NULL );
        /* ... */
        return 0;
}
```

Obrázek 3.2: Jednoduchý vícevláknový C program

1. Můžeme omezit množinu vstupních programů na ty, kde se všechny výskyty volání funkce `pthread_create` nachází v hlavní funkci před výskytem prvního cyklu. Takových výskytů bude konečně mnoho a protože každý z nich může být řízením navštívený nejvýše jednou, snadno získáme hodnotu nejvyššího počtu souběžně běžících vláken. Potom můžeme pro každý výskyt volání funkce `pthread_create` vytvořit jedno vlákno, jehož vstupní bod odpovídá předávané funkci. Pokud by se navíc první přístup na globální proměnné způsobený hlavní funkcí vyskytoval až po posledním volání `pthread_create`, mohli bychom spustit všechna vlákna paralelně s funkcí main. Na obrázku 3.3 je znázorněno, jak by mohl vypadat přeložený kód programu z obrázku 3.2.

2. Opačnou možností je podporovat programy, které mohou volat funkci `pthread_create` kdykoliv, a to i v právě vytvořeném vlákně. Princip

```
instances main[1], p1[1], p2[1];
en1, en2 : bool;
init en1 = false && en2 = false;
p1 {
        initial si;
        si -> s1 { en1 = true && havoc() }
        s1 -> ... // kod funkce p1
}
// p2 podobne jako p1
main {
        initial s0;
        s0 -> s1 { en1' = true && havoc ( en1 ) }
        s1 -> s2 { en1' = true && havoc ( en2 ) }
        s2 -> ... // kod funkce main
}
```

Obrázek 3.3: Přeložený program z obrázku 3.2 - první způsob

této varianty spočívá ve vytvoření N pracovních vláken a jednoho vlákna hlavního s tím, že všechna pracovní vlákna jsou v iniciální konfiguraci nečinná. Vstupní bod společný pro všechna pracovní vlákna pak obsahuje volací přechody pro zavolání každé funkce, která je v původním programu někdy předávána funkci `pthread_create`, avšak každý takový přechod je strážen formulí takovou, že z iniciální konfigurace paralelního systému není daný přechod povolený - vlákno tedy nemá žádný *efekt*. Každé volání funkce `pthread_create` se pokusí najít nečinné vlákno (při neúspěchu přejde do chybového stavu) a modifikuje globální proměnné tak, aby v onom vláknu došlo k povolení odpovídajícího přechodu. Naopak, po skončení funkce běžící v pracovním vláknu se vlákno opět vrací do nečinného stavu a je možné ho recyklovat. Zjednodušenou verzi této možnosti znázorňuje obrázek 3.4.

Množství skutečných programů se nevejde do množiny, podporované první možností. Právě z toho důvodu jsme se rozhodli zvolit možnost druhou.

Do appendixu pridat skutecnou verzi s funkci `thread_create`

Rozhodnutí 3. *Překládací nástroj neklade žádná omezení na umístění a četnost výskytů volání funkce* `pthread_create`.

3.5 Inliner

Používáme anotace, abychom zjistili, odkud jaká proměnná pochází. Nebo je to až u inlineru?

Kde je inliner? Nepatří k libNTS?

Btw vůbec neřešíme memory modely a předpokládáme sekvenční konzistenci.

```
instances worker[2], main[1];
sel[2] : int;
init sel[0] = 0 && sel[1] == 0;

worker {
        initial si;
        si -> s1 { sel[tid] == 1 && havoc() }
        si -> s2 { sel[tid] == 2 && havoc() }
        s1 -> sf { p1() }
        s2 -> sf { p2() }
        // Volitelna recyklace
        // sf -> si { sel'[tid] = 0 && havoc(sel) }
}
p1 { ... } p2 { ... }
main {
        initial s0;
        s0 -> s1 { sel'[0] = 1 && havoc(sel)  }
        s1 -> s2 { sel'[1] = 2 && havoc(sel)  }
        s2 -> ... // kod funkce main
}
```

Obrázek 3.4: Přeložený program z obrázku 3.2 - druhý způsob, zjednoduseno

Kapitola 4

Statická Partial Order redukce

4.1 Obecně k sekvencializaci

V rámci zachování jednoduchosti v této části textu používáme o něco jednodušší definice než v [9], zejména vypouštíme definici valuace proměnných.

Definice 3. *Řídící stav plochého paralelního NTS s k vlákny je k-tice* $q = \langle q_0, \ldots, q_{k-1} \rangle$*, kde* q_i *je řídící stav BasicNts, který je vykonáván ve vlákně s* `tid` $= i$*.*

S každým řídícím stavem q paralelního NTS je spojena množina přechodů $\texttt{Outgoing}(q)$, vedoucích z tohoto stavu.

$$\texttt{outgoing}(q) = \texttt{outgoing}(q_0) \cup \ldots \cup \texttt{outgoing}(q_{k-1}) \tag{4.1}$$

Definice 4. *Stav s paralelního přechodového systému je dvojice* $s = \langle q, v \rangle$*, kde v je řídící stav daného systému a v značí valuaci (globálních i všech lokálních) proměnných.*

Referenční manuál označujuje právě definovaný stav jako "konfiguraci".

Formálněji, odkaz do manuálu

Definice 5. *Pro libovolný přechod t a stavy* s_1, s_2 *paralelního systému, zápisem* $s_1 \xrightarrow{t} s_2$ *rozumíme, že je možné dostat se ze stavu* s_1 *do stavu* s_2 *pomocí přechodu t. Zápisem*

$$s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{m-1}} s_m$$

zkracujeme konjunkci

$$s_1 \xrightarrow{t_1} s_2 \wedge \ldots \wedge s_{m-1} \xrightarrow{t_{m-1}} s_m$$

4.2 Použití POR bez znalosti dat - statické

Navazuje na POR vysvětlenou dříve. Jak tedy POR upravit tak, aby fungovalo staticky.

Uvedené podmínky se vztahují k explicitnímu CFG, zatímco náš nástroj generuje symbolický CFG a o proměnných si mnoho informací neudržuje. Protože $\texttt{ample}(s) \subseteq \texttt{enabled}(s)$, nemůžeme generovat ample sety přímo: museli bychom totiž rozhodnout, zda je nějaký z přechodů v daném stavu povolený. Náš nástroj namísto toho pro *řídící* stav (tedy nikoliv konfiguraci) q paralelního systému spočítá množinu $A(q) \subseteq \texttt{outgoing}(q)$ tak, že pro libovolný stav (konfiguraci) $s = (q, v)$ můžeme položit $\texttt{ample}(s) = A(q) \cap \texttt{enabled}(s)$.

Problém s C0 Pokud zvolíme A neprázdné tak, že je dosažitelný stav, kdy žádný přechod z A nebude proveditelný, ale nějaký přechod mimo A ano, pak dojde k porušení C0. Toto je ale problém pouze pro obecné NTS, protože přechody překládané z LLVM IR jsou vždy proveditelné.

To není tak docela pravda - viz například podmíněné skoky. Ale dost možná je pravda, že vždycky bude alespoň jeden přechod z daného stavu uskutečnitelný.

Definice 6. *Pro libovolný přechod t (tedy i přechod s volacím pravidlem) zápisem $W(t)$ rozumíme množinu proměnných, které může přechod t změnit, a zápisem $R(t)$ množinu proměnných, které přechod t čte. Pro libovolné vlákno P zápisem $W(P)$ rozumíme množinu takových proměnných, že pro každou $v \in W(P)$ existuje přechod t vlákna P t.ž. $v \in W(t)$. Obdobně definujeme i $R(P)$.*

4.3 Problémy

4.3.1 Problém velkého vlákna

Ukazuje se, že pro zvolený model paralelismu 3.4.3 uvedená heuristika není příliš efektivní. Pro velkou množinu programů totiž napočítané množiny $W(P_i)$ a $R(P_i)$ obsahují mnoho proměnných, které ve skutečnosti vlákno i nepoužívá.

Problémová situace

Uvažme například příklad . Hlavní funkce vytvoří dvě vlákna, která používají vzájemně disjunktní množiny globálních proměnných. Po přeložení do jazyka NTS (se zvoleným modelem paralelismu) výsledné NTS obsahuje vygenerované BasicNts "__thread_pool_routine", které obsahuje přechody, volající přeložené funkce @p1 a @p2.

všechny, které lze vyjádřit v modelu "vytvoř vlákna na začátku", todo taky nekde popsat dalsi mozne modely

Po zploštění tedy thread_pool_routine obsahuje přechody, které odpovídají původním funkcím p1 a p2. Protože tento BasicNts je vykonáván v každém pracovním vlákně, v každém pracovním vlákně jsou také obsaženy všechny přechody přeložených funkcí p1 a p2. Potom napočítané množiny $W(P_0) = W(P_1) \supseteq \{G1, G2\}$. V situaci, kdy některé z vláken zapisuje do $G1$ (viz t1), potom nedojde k použití přechodu odpovídajícího t1 jako jednoprovkového ample setu, přestože žádný jiný přechod není na t1 závislý.

Možné řešení

reference

Pro zvolený model paralelismu se rozložení programu na vlákna zdá být příliš hrubým, protože mnohá vlákna budou mít totožné řízení (BasicNts, viz předchozí příklad). Rozložme tedy program jemněji na jednotky, kterým budeme říkat "úlohy".

Program to pojmenovava '__thread_poll_routine' - prijde opravit

```
@G1 = global i32 16;
@G2 = global i32 0;

define i8* @p1 ( i8* %x ) {
        ... ; some local calculations
t1:     store i32 42, i32* @G1;
        ret i8* null;
}

define i8* @p2 ( i8* ) {
        ... ; some more local calculations
t2:     store i32 14, i32* @G2;
        ret i8* null;
}

define void @main() {
        call i32 @pthread_create ( ... , @p1, ... );
        call i32 @pthread_create ( ... , @p2, ... );
        ret void;
}
```

Obrázek 4.1: Vlákna, využívající odlišnou sadu proměnných. Zjednodušeno.

Definice 7. *Pro plochý (i paralelní) přechodový systém N, množina úloh* $\texttt{Tasks}(N)$ *je množina taková, že libovolný řídící stav nějakého BasicNts z N leží v právě jedné úloze* $T \in \texttt{Tasks}(N)$.

Přechodové systémy, vzniklé překladem pomocí našeho nástroje z LLVM IR a následným zploštěním, obsahují právě dva použité BasicNts: první z nich odpovídá funkci `main` původního programu, druhý pak kódu vykonávaným vláknem z thread poolu. Pro účely této práce můžeme rozdělit jejich stavy do úloh následujícím způsobem.

1. Máme právě jednu hlavní úlohu $T_{\texttt{main}}$, právě jednu nečinnou úlohu $T_{\texttt{idle}}$ a jednu úlohu T_{f_i} pro každou funkci f_i, která je někde v původním LLVM IR programu předávána jako parametr funkce `pthread_create`.

2. Každý stav s hlavního BasicNts (tj. toho, který vznikl překladem funkce main) leží v $T_{\texttt{main}}$.

3. Všechny stavy, odpovídající stavům nezploštělého `__thread_pool_routine`, leží v $T_{\texttt{idle}}$.

4. Všechny ostatní řídící stavy jsou řídícími stavy zploštělého `__thread_pool_routine`, do kterého se dostaly během fáze zplošťování. Každý z těchto stavů přiřadíme úloze T_{f_i} takové, že daný stav pochází z funkce f_i.

Koncept úloh přirozeně rozšiřujeme i na přechody:

Definice 8. *Přechod t náleží úloze T (zapisujeme* $t \in \texttt{trans}(T)$*) právě tehdy, když zdrojový řídící stav s přechodu t (*$s = \texttt{from}(t)$*) náleží úloze T (*$s \in T$*).*

```
instances __thread_pool_routine[2], main[1];
...
__thread_pool_routine {
        initial si;
        si  -> sr1 { ... }
        sr1 -> ss  { p1() }
        ss  -> si  { ... }
        si  -> sr2 { ... }
        sr2 -> ss  { p2() }
        ss  -> si  { ... }
}
```

Obrázek 4.2: Přechodový systém pracovního vlákna

Protože každý přechod má jeden zdrojový řídící stav, každý přechod náleží právě jedné úloze. Tímto způsobem můžeme o každé úloze říci, jaké globální proměnné používá.

Definice 9. *Zápis $W(T)$ rozšiřujeme na úlohy následujícím způsobem:*

$$v \in R(T) \Leftrightarrow \exists t, t \in \textbf{\textit{trans}}(T) \wedge v \in R(t)$$

$$v \in W(T) \Leftrightarrow \exists t, t \in \textbf{\textit{trans}}(T) \wedge v \in W(t)$$

Tedy úloha T čte globální proměnnou v právě tehdy, pokud existuje přechod t, který čte v a patří T. Podobně, T mění v právě pokud nějaký přechod t patřící T mění v. Dále, v každém řídícím stavu paralelního systému můžeme některé úlohy označit jako *aktivní*.

Definice 10. *Pro řídící stav $q = \langle q_0, \ldots, q_{k-1} \rangle$ paralelního systému definujeme množinu* $\textbf{\textit{active}}(q) = \{T \mid \exists i, 0 \leq i < k \wedge q_i \in T\}$

Jednou ze situací, která se hojně vyskytuje, je "přepnutí úlohy". V případě přechodových systémů, vzniklých překladem z LLVM, například často dochází k přepnutí $T_{\texttt{idle}}$ na nějakou T_{f_i}.

Definice 11. *Definujeme antireflexivní binární relaci nad úlohami $\rightarrow$ tak, že $T_1 \rightarrow T_2$ právě tehdy, když $T_1 \neq T_2$ a existují řídící stavy paralelního systému q_1, q_2, valuace v_1, v_2 a přechod t takové, že stav (konfigurace) $\langle q_1, v_1 \rangle$ je v paralelním systému dosažitelný, $T_1 \in \textbf{\textit{active}}(q_1) \wedge T_2 \in \textbf{\textit{active}}(q_2)$ a $\langle q_1, v_1 \rangle \xrightarrow{t} \langle q_2, v_2 \rangle$.*

Protože rozhodnout dosažitelnost nějakého stavu není snadné, budeme často pracovat volnější s relací "T_1 se může přepnout na T_2".

Definice 12. *Binární relace nad úlohami $\dashrightarrow$ je relací možného přepnutí, pokud je antireflexivní a pro libovolné dvě úlohy T_1, T_2 takové, že $T_1 \rightarrow T_2$, platí $T_1 \dashrightarrow T_2$.*

Nyní se můžeme vrátit k úvahám ze sekce 2.4.3. Přechod β jistě leží v nějaké úloze T_β. Pokud přechod $T_\beta \notin \texttt{active}(s_0)$, T_β musela být aktivována během sekvence přechodů α_j. Musí tedy existovat nějaká úloha T, kterou je možné po několika přepnutích přepnout až na T_β, zejména pak dvojice $\langle T, T_\beta \rangle$ musí ležet v tranzitivním uzávěru relace $\dashrightarrow$.

Jak může úloha způsobit přepnutí jiné úlohy na ještě jinou

4.3.2 Problém velkého pole

V případě, že bychom měli pole takové, že by na každou jeho pozici přistupoval nejvýše jeden proces, a procesů bychom měli mnoho, vyplatilo by se sledovat jeho jednotlivé buňky zvlášť. To ale neděláme. Btw jedno takové pole máme.

4.3.3 Problém závislosti na datech

Zda může nějaké vlákno běžet, závisí na datech. My se ale o data moc nestaráme (TODO: tohle je třeba ujasnit na začátku). Tedy nemůžeme vědět, že na začátku poběží jenom hlavní vlákno. Tedy zeserializovaný systém bude obsahovat běhy, jejihž podmínka cesty bude nesplnitelná. Obecně tohle řešit snadno nelze, ale pokud se omezíme na zjištění informace, zda nějaké vlákno z thread poolu může začít vykonávat nějakou úlohu, stačí nám sledovat pár zvolených proměnných. Na to máme dvě možnosti:

a) Analyzovat vykonávané přechody, zda modifikují naše vybrané proměnné. Předpokládáme, že většina formulí bude mít hezký tvar, a že tedy nemusíme vědět všechno na to, abychom některé mohli rovnou označit za nesplnitelné a o jiných prohlásit, že modifikují námi vybranou proměnnou jednoduchým způsobem.

b) Umět rozpoznat původně existující struktury i v přeloženém a zplacatělém přechodovém systému. Tedy musíme vědět, co jsou pracovní vlákna, co dělá __thread_create (a jak jí poznám) a další věci.

Kapitola 5

Implementace

5.1 Přehled existujících nástrojů

V době vzniku této práce již existovala řada nástrojů, pracující s jazykem NTS. Na webové stránce jazyka NTS[1] je seznam některých nástrojů, které NTS podporují. Za všechny jmenujme jenom nástroj Flata[2] (GitHub[3]), který slouží pro analýzu dosažitelnosti a terminace přechodového systému. Tento nástroj využívá knihovnu od stejného autora, obsahující i parser jazyka.

FlataC Dalším z nástrojů je FlataC[4], což je plugin do analyzéru Frama-C, které umí na NTS převést program v jazyce C. Flatac využívá knihovnu Ocaml-nts[5] od stejného autora, která je (jak název napovídá) napsaná v jazyce OCaml.

napsat jakou množinu umí

5.2 Volba implementačního jazyka a knihoven

Pro implementaci překládacího nástroje jsme se rozhodli zvolit jazyk C++; hlavním důvodem této volby byla skutečnost, že knihovna LLVM je sama implementována v C++. Přestože existují vazby LLVM na D, Rust, Haskell, OCaml i další jazyky, tyto mohou být vydávány se zpožděním oproti LLVM, a my jsme zamýšleli držet se nejnovější verze LLVM. Dále, na Fakultě Informatiky Masarykovy Univerzity probíhá vývoj nástroje pro model checking s názvem DiVinE, který je implementován v C++, a implementace knihovny pro paměťovou reprezentaci jazyka NTS v jazyce C++ může později umožnit snadnější přidání podpory NTS právě do nástroje DiVinE. V neposlední řadě hrála roli také zkušenost autora s vývojem v jazyce C/C++.

[1] `http://nts.imag.fr/index.php/Main_Page`
[2] `http://nts.imag.fr/index.php/Flata`
[3] `https://github.com/filipkonecny/flata`
[4] `https://github.com/fgarnier/flatac`
[5] `https://github.com/fgarnier/Ocaml-nts`

5.2.1 •

5.3 Knihovna libNTS

Jak je vidět z úvodu a předchozích kapitol, celá práce je složená s nástrojů pracujících s přechodovým systémem vyjádřeným v jazyce NTS. Z toho důvodu je vhodné, aby tyto nástroje používaly stejnou paměťovou reprezentaci. Aby mohly být nástroje do určité míry odděleny, tato reprezentace by měla být k dispozici ve formě knihovny.

Přestože již některé nástroje pro jazyk NTS existují, v době psaní této práce nám nebyla známa existence žádné NTS knihovny s vazbou na C++. Z toho důvodu vznikla knihovna libNTS[6] , jejíž zdrojový kód je hostován na serveru GitHub[7] . Knihovna je psaná v jazyce C++ revize C++14 a nevyužívá RTTI.

5.3.1 Parser

Knihovna libNTS v sobě neobsahuje parser jazyka NTS; tento vyvíjí Petr Bauch pod názvem nts-parser[8] a v budoucnu by mohl umět z vybudovaného AST poskládat paměťovou reprezentaci programu ve formátu knihovny libNTS.

5.3.2 Architektura

Architektura knihovny do velké míry odpovídá struktuře jazyka NTS. Celý přechodový systém se skládá ze seznamu vláken a jejich vstupních bodů, globálních proměnných a seznamu BasicNts, které obsahují zejména proměnné a přechody. Místy může architektura připomínat architekturu paměťové reprezentace LLVM IR, zejména pokud jde o

...

5.3.3 Vlastnictví

Nejen z důvodu, že C++ neobsahuje garbage collector, je třeba nějak řešit vlastnictví objektů. V rámci celé knihovny obecně platí, že vlastník prvku je vždy jeden: je jím právě ten prvek, jež ho lexikálně obklopuje. Tedy například vlastníkem přechodů a lokálních proměnných je vždy příslušný BasicBlock, vlastníkem podtermu či podformule je nadřazený term a podobně. Některé vlastněné prvky (zejména formule a termy) si svého vlastníka pamatují, což umožňuje procházet strom vlastnictví zdola nahoru. Tato vlastnost je užitečná zejména pro analýzu toho, jaké přechody či BasicNts využívají nějakou proměnnou (viz sekce 5.3.3). Koncept pamatování si svého vlastníka však není příliš zobecněný ani abstrahovaný.

Což takhle nakreslit obrázek, zachycující vlastnictví a uživatele proměnných?

Proměnné

Specifikace jazyka rozlišuje mezi čárkovanými (primed) proměnnými a nečárkovanými proměnnými. Protože se čárkované proměnné mohou vyskytovat pouze

[6] `https://github.com/h0nzZik/libNTS_cpp`
[7] `https://github.com/`
[8] `https://github.com/xbauch/nts-parser`

uvnitř přechodových pravidel a navíc každé čárkované proměnné odpovídá nějaké nečárkovaná proměnná, knihovna libNTS používá mírně odlišný koncept. Proměnné zde existují pouze v nečárkované variantě (`Variable`) a uvnitř přechodů jsou většinou používány ve formě takzvaných *referencí* (`VariableReference`). Každá reference je buď čtecí nebo zapisovací a tak odpovídá buď nečárkované proměnné (tedy proměnné v aktuálním stavu), nebo čárkované (proměnné v budoucím stavu).

Každá proměnná si také pamatuje seznam svých *uživatelů* (`VariableUse`), mezi které patří kromě `VariableReference` také `Havoc`, volací přechodové pravidlo (`CallTransitionRule`) a atomická propozice pro zápis do pole (`ArrayWrite`). Protože proměnné nejsou používány přímo, ale vždy přes `VariableUse`, je sledování uživatelů proměnných (na rozdíl od sledování vlastníků, viz 5.3.3) automatické.

5.3.4 Omezení

Zatím nepodporuji složitější operace s poli (přířazení polí, práci s referencemi, pole jako parametry funkcí). Parametry (par) jsou používány jako proměnné.

Kapitola 6

Experimenty

6.1 Použití implementovaných nástrojů

Tuto část textu bych rád viděl i v příslušných README.md

6.1.1 Získání

6.1.2 Kompilace

6.1.3 Instalace?

Kapitola 7

Závěr

Kapitola 8

Nepřiřazeno

8.1 Inlining

Jo teda nepodporuju rekurzi, a proto si můžu dovolit to, co dělám v inlineru - prostě tak dlouho zainlinovávám jednotlivé BasicNts, až mi nezůstané žádné volání.

8.2 Možná budoucí rozšíření

8.2.1 Znaménkové datové typy v NTS

Petr psal autorům NTS, a pokud si dobře vzpomínám, jeden z nich projevil přání, abychom mohli překládat llvm do stávající podmnožiny NTS - tedy do intů. Celé LLVM tak nepůjde, ale jistá podmnožina ano. Každopádně zatím se na tom nepracuje.

Obsah

Bibliografie

1 LATTNER, Chris. *The LLVM Compiler Infrastructure* [online]. 2014 [cit. 2014-11-07]. Dostupný z WWW: ⟨http://www.llvm.org⟩.

2 *LLVM Language Reference Manual*. [online]. 2014 [cit. 2014-11-07]. Dostupný z WWW: ⟨http://llvm.org/releases/3.5.0/docs/index.html⟩.

3 LATTNER, Chris. LLVM. In BROWN, Amy; WILSON, Greg (ed.). *The Architecture of Open Source Applications: Elegance, Evolution and a Few Fearless Hacks* [online]. 2014 [cit. 2014-11-09]. Dostupný z WWW: ⟨http://www.aosabook.org⟩.

4 ZHAO, Jianzhou; NAGARAKATTE, Santosh; MARTIN, Milo M. K.; ZDANCEWIC, Steve. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In. *POPL '12* [online]. 2012 [cit. 2014-11-09]. Dostupný z WWW: ⟨http://www.cis.upenn.edu/~stevez/papers/ZNMZ12.pdf⟩.

5 LATTNER, Chris. *clang: a C language family frontend for LLVM* [online]. 2014 [cit. 2014-10-18]. Dostupný z WWW: ⟨http://clang.llvm.org⟩.

6 BARNAT, Jiří; BRIM, Luboš; HAVEL, Vojtěch et al. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In. *Computer Aided Verification (CAV 2013)*. 2013, s. 863–868. LNCS.

7 EDMUND M. CLARKE, Jr.; GRUMBERG, Orna; PELED, Doron A. *Model Checking*. Cambridge, Massachusets a London, England: MIT Press, 1999. ISBN 0-262-03270-8.

8 THE LINUX MAN-PAGES PROJECT. *pthread_ create(3)* [online]. 2014 [cit. 2015-05-12]. Dostupný z WWW: ⟨http://man7.org/linux/man-pages/man3/pthread_create.3.html⟩.

9 IOSIF, Radu; KONEČNÝ, Filip; BOZGA, Marius. *The Numerical Transition Systems Library* [online]. [pravděpodobně 2009-2013] [cit. 2014-11-09]. 32 s. Dostupný z WWW: ⟨http://nts.imag.fr/images/b/b5/Ntslib.pdf⟩.