

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 .

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:

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

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 ostatní, 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 součástí 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, řídicích stavů a přechodů mezi nimi. Řídicí 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, a s_i, s_j řídicí stavy. 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 $(p1', p2') = \text{factorize}(x)$) 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 označeny 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 $\{\text{true}, \text{false}\}$. 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 řídicích stavů a v valuace proměnných (tedy funkce, která každé proměnné přiřazuje hodnotu). Přechod z konfigurace $c_1 = \langle q_1, v_1 \rangle$ do konfigurace $c_2 = \langle q_2, v_2 \rangle$ je možný, pokud existuje přechodové pravidlo $t = \left(q_1 \xrightarrow{F} q_2 \right)$ 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. Tuto situaci budeme zapisovat jako $c_1 \xrightarrow{t} c_2$. 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.

$$\text{havoc}(v_1, v_2, \dots, v_k) = \bigwedge_{v \in V \setminus \{v_1, v_2, \dots, v_k\}} v = v' \quad (2.1)$$

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 $F \equiv x' = x + 1 \wedge \text{havoc}(x)$, 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 nespílitelná.

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 jsou 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 $\text{tid} \in \{0, \dots, N - 1\}$ je BasicBlock **worker_nts** a vstupním bodem vlákna s $\text{tid} = N$ 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

```

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

producent {
    initial si;
    i : integer;
    si -> s1 { i' = 0 && havoc(i) }
    s1 -> s1 { c = false && c' = true &&
               G' = i && i' = i + 1 &&
               havoc(c, G, i) }
}

consument {
    initial si;
    sum, x : integer;
    si -> s1 { sum' = 0 && havoc(sum) }
    s1 -> sh { c = true && x' = G && havoc(x, G) }
    sh -> s1 { sum' = sum + x && c' = false &&
               havoc(sum, c) }
}

```

Obrázek 2.5: Příklad kódu v jazyce NTL

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 `s1 → s1` se může uskutečnit pouze v případě, že globální proměnná `c` je nastavena na hodnotu `false`, a sám tuto proměnnou nastaví na `true`. Naopak, přechod `s1 → s1` 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 `s1 → sh` v `consument` ničí hodnotu uloženou v `G`, což ale nevádí, protože `G` není nikdy čtena.

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], úprava POR pro jazyk NTS je popsána v kapitole 4.

2.4.1 Přebodový systém

Stavový přebodový systém je čtveřice $\langle S, T, S_0, L \rangle$, kde S představuje množinu stavů, $T \subseteq \mathcal{P}(S \times S)$ množinu přebodových relací (dále zkráceně jen přebodů), $S_0 \in S$ označuje iniciační stav a $L : S \rightarrow \mathcal{P}(AP)$ je značkovácí funkce, která každému stavu přiřazuje množinu atomických proposic. Jako syntaktickou zkratku za $\langle s_1, s_2 \rangle \in \alpha$, kde $\alpha \in T$, zavádíme $s_1 \xrightarrow{\alpha} s_2$.

Přebod $\alpha \in T$ se nazývá deterministický, pokud pro každý stav $s \in S$ existuje nejvýše jeden stav $s' \in S$ takový, že $s \xrightarrow{\alpha} s'$ je splněno. Pro takový přebod můžeme zkráceně psát $s' = \alpha(s)$. Zatímco v [7] jsou uvažovány pouze přebodové systémy s deterministickými přebody, v této práci uvažujeme přebody obecně nedeterministické.

Definice 1. Přebod $\alpha \in T$ je povolený ve stavu $s \in S$ (píšeme $\alpha \in \text{enabled}(s)$) právě tehdy, když existuje stav s' tak že $s \xrightarrow{\alpha} s'$

Definice 2. Cesta ze stavu s_0 v přebodovém systému je (potenciálně nekonečná) posloupnost $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$ taková, že $\forall i. (s_i \xrightarrow{\alpha_i} s_{i+1})$

2.4.2 Nezávislost a komutativita přebodů

Definice 3. Binární relace $I \subseteq T \times T$ nazýváme relací nezávislosti, pokud je symetrická, antireflexivní a pro libovolnou dvojici přebodů $(\alpha, \beta) \in I$ povolených ve stavu $s \in S$ platí:

$$\begin{aligned} \forall s_2 \in S. (s \xrightarrow{\beta} s_2 \Rightarrow \alpha \in \text{enabled}(s_2)) \wedge \\ \forall s_2, s_3 \in S. (s_1 \xrightarrow{\beta} s_2 \xrightarrow{\alpha} s_3 \Rightarrow \exists s'_2 \in S. (s \xrightarrow{\alpha} s'_2 \xrightarrow{\beta} s_3)) \end{aligned} \quad (2.2)$$

Odpovídající definice v [7] požaduje (kromě symetrie a antireflexivity), aby pro každou dvojici přebodů $(\alpha, \beta) \in I$ povolených ve stavu $s \in S$ platilo:

$$\alpha \in \text{enabled}(\beta(s)) \wedge \alpha(\beta(s)) = \beta(\alpha(s)) \quad (2.3)$$

Lemma 1. Pro přebodové systémy bez nedeterministických přebodů jsou obě definice ekvivalentní.

Důkaz. Formule 2.2 i 2.3 se vyjadřují o stejné množině přebodů, pro důkaz lemmatu stačí dokázat ekvivalenci těchto formulí pro deterministické přebody. Protože $\beta \in \text{enabled}(s)$ je deterministická, existuje právě jeden stav s_2 takový, že $s \xrightarrow{\beta} s_2$, a to $s_2 = \beta(s)$. Formulí 2.2 tedy můžeme přepsat jako

$$\begin{aligned} \alpha \in \text{enabled}(\beta(s)) \wedge \\ \forall s_3 \in S. (\beta(s) \xrightarrow{\alpha} s_3 \Rightarrow \exists s'_2 \in S. (s \xrightarrow{\alpha} s'_2 \xrightarrow{\beta} s_3)) \end{aligned} \quad (2.4)$$

Protože požadujeme $\alpha \in \text{enabled}(\beta(s))$ a uvažujeme pouze deterministické přebody, opět existuje pouze jeden takový stav s_3 , a to $s_3 = \alpha(\beta(s))$. Vhodný stav s'_2 také může být pouze jeden, a to právě $\alpha(s)$. Formulí můžeme opět zjednodušit na

$$\alpha \in \text{enabled}(\beta(s)) \wedge \alpha(s) \xrightarrow{\beta} \alpha(\beta(s)) \quad (2.5)$$

a protože relace I je symetrická, musí také platit $\beta \in \text{enabled}(\alpha(s))$, tedy výraz $\beta(\alpha(s))$ je dobře definovaný, díky čemuž můžeme formuli zjednodušit do podoby

$$\alpha \in \text{enabled}(\beta(s)) \wedge \beta(\alpha(s)) = \alpha(\beta(s)) \quad (2.6)$$

ve které je ekvivalence s 2.3 zřejmá. \square

Technika, použitá v [7], tedy partial order reduction s využitím dostatečných (ample) množin, spočívá v tom, že pro každý stav přechodového systému je spočítána množina přechodů $\text{ample}(s) \subseteq \text{enabled}(s)$, která má tu vlastnost, že přechody mimo $\text{ample}(s)$ nejsou „důležité“. Formálněji řečeno je třeba, aby ke každé cestě z iniciálního stavu přechodového systému existovala vhodným způsobem ekvivalentní¹ cesta z téhož stavu, která nevyužívá žádné přechody mimo ample . Při procházení přechodového grafu pak není třeba uvažovat přechody mimo ample , což zmenšuje počet stavů, které je nutné prozkoumat během model checkingu.

Definice 4. *Přechod je neviditelný, pokud nikdy nezmění platnost atomické propozice z množiny předem vybraných propozic.*

2.4.3 Nutné podmínky

Kniha [7] uvádí několik podmínek, postačujících pro volbu vhodného $\text{ample}(s)$.

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

C1 Každá cesta v úplném explicitním přechodovém grafu paralelního systému, začínající ve stavu s , má následující vlastnost: pokud přechod β , který se vyskytuje po cestě, závisí na nějakém přechodu $\alpha \in \text{ample}(s)$, pak se před jeho výskytem vyskytuje nějaký přechod $\alpha' \in \text{ample}(s)$.

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

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

2.4.4 Výpočet $\text{ample}(s)$

Podmínku C1 není snadné ověřit; [7] dokonce uvádí, že problém ověření podmínky C1 pro jeden stav je alespoň tak těžký, jako problém dosažitelnosti stavu v celém přechodovém systému. Z tohoto důvodu byly navrženy heuristiky, jejichž cílem je pro stav s efektivně zvolit $\text{ample}(s)$ tak, aby byly podmínky (nebo alespoň podmínka C1) jistě splněny. Tyto heuristiky silně závisí na použitém modelu výpočtu a proto budou v tomto textu popsány až v sekci 4 přímo pro jazyk NTS.

¹Jedná se o „kottavou“ (stuttering) ekvivalenci, viz [7].

Kapitola 3

Překlad LLVM na NTS

3.1 Omezení LLVM

3.1.1 Pole

Nepodporuji (zatím). Přidat podporu 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`.

3.1.3 Instrukce

Překládací nástroj v současné době podporuje jen velice omezenou množinu instrukcí. Mezi ně patří instrukce pro zápis do a čtení z paměti (`load` / `store`), instrukce volání a návratu z funkce (`call/ret`), instrukce skoků a větvení (`br`), porovnávací instrukce (`icmp`, pouze nad neznaménkovými operandy), instrukce pro sčítání čísel (`add`) a alokaci paměti na zásobníků (`alloca`). 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 vyvíjena dále, je to jedna z věcí, která by mohla být volitelná (tedy jestli vrátet 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 bezznaménkové aritmetiky, výsledné formule by byly složitější. Tato instrukce přiřadí do proměnné

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

'b' hodnotu 1 právě pokud x je znaménkově menší (signed less than, 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.

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

ukázat jak na to

zmínit sémantiku operaci nad BitVector

Někde zmínit, že LLVM IR využívá právě dvojkový doplněk

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 $y \leq 127$.
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 spočítatelnou 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. Návrátová hodnota funkce `pthread_create` musí být ignorována.
5. Funkce, předaná jako argument parametru `start_routine`, nesmí používat svůj parametr.
6. Návrátová 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 překladač 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 informo-

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 logické 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 booleovským 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>

$\langle literal \rangle ::= \langle id \rangle \mid \text{tid} \mid \langle numeral \rangle \mid \langle decimal \rangle$

$\langle aop \rangle = '+' \mid '-' \mid '*' \mid '/' \mid '\%'$

$\langle bop \rangle = '\&' \mid '|' \mid '->' \mid '<->'$

$\langle op \rangle = \langle aop \rangle \mid \langle bop \rangle$

$\langle term \rangle ::= \langle literal \rangle \mid \langle term \rangle \langle aop \rangle \langle term \rangle \mid \langle term \rangle \langle bop \rangle \langle term \rangle \mid '(' \langle term \rangle ')'$

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 Vlastní 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 bloku nemění tok řízení a po jejím vykonání je vykonávána instrukce následující, můžeme každý základní blok s N vnitřními instrukcemi přeložit jako N nových řídicích 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 instrukce 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 pře-
kladačů?

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 ne-
možnost 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 této


```

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

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 strážěn 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, podrobnější verze je k dispozici v příloze na obrázku A.1.

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

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

Do
appendixu
přidat
skutečnou
verzi s
funkcí
`thread_create`

3.5 Inliner

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

Kde je in-
liner? Ne-
patří k
libNTS?

Btw vůbec
neřešíme
memory
modely a
předpo-
kládáme
sekvenční
konzis-
tenci.

```

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() }
    // Volitelná 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, zjednodušeno

Kapitola 4

Partial Order redukce pro NTS

Cílem jednoho z nástrojů, vyvíjeného jako součást této práce, je převést paralelní přechodový systém v jazyce NTS na (co nejmenší) ekvivalentní sekvenční systém v témže jazyce. Vytvořený nástroj¹ k tomuto účelu využívá techniku, zvanou Partial Order Redukce, jejíž obecný popis se nachází v kapitole 2.4. Převod je možné provést i bez použití této techniky, nicméně počet stavů výsledného přechodového systému je v takovém případě veliký.

4.1 Terminologie a definice

Jazyk NTS používá odlišnou terminologii od té, která je použita pro popis Partial Order redukce v [7] a v sekci 2.4. V kontextu jazyka NTS je hojně používán pojem *konfigurace*, který označuje dvojici *řídícího stavu* a *valuace* proměnných. Oproti tomu přechodový systém používaný kontextu POR neobsahuje žádné proměnné, a tak každý jeho *stav* vlastně odpovídá *konfiguraci* programu v jazyce NTS. V této části textu budeme používat pojem *řídící stav* pouze ve svém původním významu, pojem *konfigurace* pak bude rozšířen i na popis *stavů* přechodového systému, jak je známe z 2.4. Samotný termín *stav* používán nebude, protože jeho použití by mohlo být mírně zavádějící.

Dále pak jazyk NTS používá termínu *přechodové pravidlo*, které je speciálním případem přechodů ze sekce 2.4. V následujícím textu budeme používat zejména pojmu přechodového pravidla. Funkci a *enabled* přenášíme do kontextu NTS tak, že pro *konfiguraci* $c = (Q, v)$ paralelního NTS, *enabled(c)* označuje množinu *přechodových pravidel*, která jsou v konfiguraci *c* povolena. Obdobným způsobem přenášíme i další funkce, zejména *ample*.

Protože předpokládáme, že přechodový systém v jazyce NTS, který míníme sekvencializovat, je plochý, můžeme si dovolit pracovat s jednodušší definicí konfigurace paralelního systému, než jaká je k dispozici v [9].

Definice 5. *Řídící stav plochého paralelního NTS s k vlákny je k-tice $Q = \langle q_0, \dots, q_{k-1} \rangle$, kde q_i je řídící stav BasicNts, který je vykonáván ve vlákne s $tid = i$.*

¹<https://github.com/h0nzZik/llvm-nts-PartialOrderReduction>

Pro každý řídicí stav q nějakého BasicNts označíme množinu přechodů, vedoucí ze stavu q , jako $outgoing(q)$. Tuto definici pak rozšíříme na řídicí stav Q plochého paralelního systému následovně:

$$outgoing(Q) = outgoing(q_0) \cup \dots \cup outgoing(q_{k-1}) \quad (4.1)$$

Definice 6. Konfigurace c plochého paralelního přechodového systému jazyka NTS je dvojice $c = \langle Q, v \rangle$, kde Q je řídicí stav daného systému a v značí valuaci (globálních i všech lokálních) proměnných.

Definice 7. Pro libovolné přechodové pravidlo t a konfigurace $c_1 = \langle Q_1, v_1 \rangle$, $c_2 = \langle Q_2, v_2 \rangle$, $Q_1 = \langle q_{1,1}, \dots, q_{1,m} \rangle$, $Q_2 = \langle q_{2,1}, \dots, q_{2,m} \rangle$ paralelního systému, zápisem $c_1 \xrightarrow{t} c_2$ rozumíme, že je možné provést přechod ze stavu c_1 do stavu c_2 pomocí přechodového pravidla t . To může nastat právě tehdy, pokud existuje i takové, že

$$q_{1,i} \xrightarrow{t} q_{1,j} \wedge \bigwedge_{j \neq i} (q_{1,j} = q_{1,i}) \quad (4.2)$$

Zápisem

$$c_1 \xrightarrow{t_1} c_2 \xrightarrow{t_2} \dots \xrightarrow{t_{m-1}} c_m$$

zkracujeme konjunkci

$$c_1 \xrightarrow{t_1} c_2 \wedge \dots \wedge c_{m-1} \xrightarrow{t_{m-1}} c_m$$

4.2 Heuristika

Jak již bylo řečeno v sekci 2.4.4, heuristika výpočet *ample* množin je závislá na zvoleném modelu výpočtu. Model výpočtu použitý v [7] se NTS podobá, proto heuristiky tam uvedené tvoří základ heuristiky pro NTS, jak je popsána v této sekci.

4.2.1 Nezávislost

Definice relace nezávislosti umožňuje jistou flexibilitu při výběru vhodné relace. Za nezávislé budeme považovat ty páry přechodů $\langle t_1, t_2 \rangle$, 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.

4.2.2 Ample sety

Porušení podmínky C1 znamená, že v úplném explicitním přechodovém grafu paralelního NTS existuje posloupnost $c_0 \xrightarrow{\alpha_0} \dots \xrightarrow{\alpha_{m-1}} c_m \xrightarrow{\beta} s_{m+1}$, $c_j = \langle Q_j, v_j \rangle$ taková, že $\beta \in enabled(c_l)$, β je závislé na nějakém přechodovém pravidlu ve zvoleném $ample(c_0)$ a všechna α_j jsou nezávislá na všech pravidlech v $ample(c_0)$. Potom, jak uvádí [7], mohou nastat dvě situace.

- Pravidlo β pochází z vlákna i (tedy $\beta \in outgoing(q_{m,i})$). Protože α_j jsou nezávislá na $ample(c_0)$, musí pocházet z jiného vlákna než i , tedy $q_{0,i} = q_{m,i}$. Protože $\beta \notin ample(c_0)$, tak $\beta \notin enabled(c_0)$. V tom případě se ale konfigurace c_0 a c_m liší ve valuaci některých globálních proměnných, které jsou používány přechodovým pravidlem β , a tedy nějaké vlákno s $tid \neq i$ obsahuje přechodové pravidlo, modifikující proměnnou, která je používána pravidlem β .
- Pokud $\beta \notin outgoing(q_{m,i})$, tedy pravidlo β pochází z vlákna jiného než i , nějaké jiné vlákno než i obsahuje pravidlo, které je závislé na nějakém pravidle z $ample(s_0)$.

V obou případech tedy musí nějaké jiné vlákno obsahovat přechodové pravidlo, které s nějakým pravidlem z $ample(c_0)$ sdílí proměnnou, která je modifikována jedním z nich. To se dá snadno ověřit, pokud si například předem pro každé vlákno napočítáme množinu proměnných v něm použitých. Přírozonou možností volby $ample(c_0)$ je množina všech povolených přechodů nějakého vlákna, tedy $enabled(c_0) \cup outgoing(q_{0,i})$. Pokud žádná taková volba neuspěje, konfiguraci plně expandujeme, tedy položíme $ample(c_0) = enabled(c_0)$.

Zmínit
ze volíme
přechody
z jednoho
procesu

4.3 Statická POR

Partial Order redukce bývá často implementována v model checkeru, který ji používá až během vlastní procedury model checkingu. Naším cílem je však provést redukci bez zkoumání stavového prostoru, jde nám pouze o sekvencializaci control flow. Tato varianta partial order redukce se nazývá statická [10].

To znamená, že nástroj nebude pracovat s konfiguracemi paralelního NTS, ale pouze s řídicími stavy, a z toho důvodu nebude přímo počítat $ample(c)$ konfigurace c . Pro řídicí stav paralelního NTS se pokusíme spočítat množinu hran z tohoto stavu vycházejících tak, aby tyto hrany při libovolné valuaci proměnných tvořili platnou $ample$ množinu. Formálněji, pro $Q = \langle q_0, \dots, q_{k-1} \rangle$ spočítáme $A(Q) \subseteq outgoing(Q)$ tak, aby $\forall v$ množina $A(Q) \cap enabled(\langle Q, v \rangle)$ byla platnou $ample$ množinou v konfiguraci $\langle Q, v \rangle$. V souladu s výše uvedenou heuristikou budeme za $A(Q)$ zkoušet použít $outgoing(q_i)$ nějakého vlákna i .

Zbývá tedy určit, kdy můžeme množinu $outgoing(q_i)$ za $A(Q)$. Každá odvozená $ample$ množina musí splňovat zmíněné podmínky 2.4.3 C0 až C3.

ad C0 Pro libovolnou valuaci v , pokud $enabled(\langle Q, v \rangle) \neq \emptyset$, pak $enabled(\langle q_i, v \rangle) \neq \emptyset$, tedy některé z přechodových pravidel v $outgoing(q_i)$ musí být povolené. Podmínku můžeme mírně zjednodušit (a zpřísnit) tak, že budeme požadovat, aby $outgoing(q_i) \neq \emptyset$ a navíc pro libovolnou² valuaci v nějaké přechodové pravidlo $t \in outgoing(q_i)$ bylo splnitelné. Tuto podmínku je možné vyřešit za použití externího solveru.

Zajímavá může být otázka, jak často může v praxi docházet k porušení této podmínky, pokud budeme uvažovat přechodové systémy, vzniklé překladem z LLVM IR. Pravidla, vzniklá překladem vnitřní instrukce BasicBlocku, budou splnitelná téměř vždy: tyto instrukce totiž obvykle zapisují spočítaný výsledek do registru nebo paměti. Pravidla, vzniklá z

²Tedy nejen takovou, že nějaké z pravidel nějakého vlákna je povolené, ale libovolnou.

instrukcí terminujících BasicBlock, se obvykle budou vyskytovat po skupinách, z nichž alespoň jedno bude splnitelné. Naopak, podmínce budou nevyhovovat zejména pravidla, čekající na změnu nějaké globální proměnné, která jsou obsažena ve vstupních bodech pracovních vlákních, nebo pravidla vzniklá možným překladem funkce `pthread_join` či funkce pro práci se synchronizačními primitivy.

Další možností je zesílit podmínku ještě více a požadovat, aby alespoň jedno z pravidel bylo povolené vždy. K potvrzení toho, že pravidlo splňuje takto zpřísněnou podmínku, stačí jednoduchá syntaktická analýza jeho formule. To pravidlo, jehož formule splňuje všechny následující požadavky, je jistě vždy povolené.

- Formule je složena výhradně z konjunkcí atomických proposic.
- Každá atomická proposice je buď havoc, relace nebo zápis do pole.
- Každá relace má jako jeden z termů čárkovanou (primed) proměnnou.
- Formule obsahuje havoc, který obsahuje všechny proměnné formule, které se v ní vyskytují čárkované (včetně případného pole, do něhož je zapisováno).
- Každá čárkovaná proměnná se ve formuli vyskytuje nejvýše jednou.

Opět, velká část vnitřních instrukcí se překládá na přechody s formulí, která těmto pravidlům vyhovuje

ad C1

ad C2

ad C3

Definice 8. 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 $R(P)$.

4.4 Problémy

4.4.1 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 disjunktí 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 popsát další možné modely

reference

Program to pojmenujeme '_thread_poll_routine'

```

@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.

Po zploštění tedy `thread_pool_routine` obsahuje přechody, které odpovídají původním funkcím `p1` a `p2`. Protože tento `BasicNts` je vykonáván v každém pracovním vlákne, v každém pracovním vlákne 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 jednoprovokového ample setu, přestože žádný jiný přechod není na `t1` závislý.

Možné řešení

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".

Definice 9. Pro plochý (i paralelní) přechodový systém N , množina úloh $Tasks(N)$ je množina taková, že libovolný řídicí stav nějakého `BasicNts` z N leží v právě jedné úloze $T \in Tasks(N)$.

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_{main} , právě jednu nečinnou úlohu T_{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`.

```

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

2. Každý stav s hlavního BasicNts (tj. toho, který vznikl překladem funkce `main`) leží v T_{main} .
3. Všechny stavy, odpovídající stavům nezploštělého `__thread_pool_routine`, leží v T_{idle} .
4. Všechny ostatní řídicí stavy jsou řídicími 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 10. *Přechod t náleží úloze T (zapisujeme $t \in \text{trans}(T)$) právě tehdy, když zdrojový řídicí stav s přechodu t ($s = \text{from}(t)$) náleží úloze T ($s \in T$).*

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

Definice 11. *Zápis $W(T)$ rozšiřujeme na úlohy následujícím způsobem:*

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

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

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 patří T mění v . Dále, v každém řídicím stavu paralelního systému můžeme některé úlohy označit jako *aktivní*.

Definice 12. *Pro řídicí stav $q = \langle q_0, \dots, q_{k-1} \rangle$ paralelního systému definujeme množinu $\text{active}(q) = \{T \mid \exists i, 0 \leq i < k \wedge q_i \in T\}$*

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

Definice 13. Definujeme antireflexivní binární relaci nad úlohami \rightarrow tak, že $T_1 \rightarrow T_2$ právě tehdy, když $T_1 \neq T_2$ a existují řídicí 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, $T_1 \in \text{active}(q_1) \wedge T_2 \in \text{active}(q_2)$ 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 14. Binární relace nad úlohami $--\rightarrow$ je relací možného přepnutí, pokud je antireflexivní a pro libovolné dvě úlohy T_1, T_2 takové, že $T_1 \rightarrow T_2$, platí $T_1 --\rightarrow T_2$.

Nyní se můžeme vrátit k úvahám ze sekce 4.2.2. Přechod β jistě leží v nějaké úloze T_β . Pokud přechod $T_\beta \notin \text{active}(s_0)$, 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 $--\rightarrow$.

4.4.2 Problém velkého pole

Další z omezení použití POR souvisí s poli. Na pole je totiž nahlíženo jako na jednu proměnnou, a tak pokud dvě přechodová pravidla přistupují na různé prvky téhož pole, heuristika s nimi pracuje, jako kdyby přistupovaly na tutéž proměnnou. Taková situace dokonce nastává i v praxi v případě paralelních NTS získaných překladem programu v LLVM IR kódu. 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.

Jak může úloha způsobit přepnutí jiné úlohy na ještě jinou

Inliner a dynamicky alokované pole

Odkaz na appendix

4.4.3 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, jejichž 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 zplacatěném 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, pracujících s jazykem NTS. Na webové stránce jazyka NTS¹ je seznam některých nástrojů, které NTS podporují. Za všechny jmenujme jenom nástroj Flata² (GitHub³), 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 FlataC⁴, což je plugin do analyzáru Frama-C, které umí na NTS převést program v jazyce C. FlataC využívá knihovnu Ocaml-nts⁵ od stejného autora, která je (jak název napovídá) napsaná v jazyce OCaml.

napsat ja-
kou mno-
žinu 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[6]. V neposlední řadě hrála roli také zkušenost autora s vývojem v jazyce C/C++.

5.3 Knihovna libNTS

Jak je vidět z úvodu a předchozích kapitol, celá práce je složená z nástrojů pracujících s přechodovým systémem vyjádřeným v jazyce NTS. Z toho důvodu

¹http://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>

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, jejíž zdrojový kód je hostován na serveru GitHub⁶. Knihovna je psaná v jazyce C++ revize C++14 a nevyužívá RTTI.

5.3.1 Parser

Knihovna libNTS v sobě neobsahuje parser jazyka NTS. Petr Bauch však vyvíjí samostatný parser s názvem `nts-parser`⁷, který by měl v budoucnu umět z vybudovaného AST poskládat paměťovou reprezentaci přechodového systému 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. Záměrem bylo vytvořit obecnou knihovnu, která bude použitelná i mimo tuto práci, a její architektura byla v některých ohledech (například uživatelé proměnných, sekce 5.3.4) inspirována architekturou paměťové reprezentace LLVM IR.

5.3.3 Vlastnictví

Nejen z důvodu, že C++ neobsahuje garbage collector, je třeba nějak řešit vlastnictví objektů uvnitř knihovny. V rámci celé knihovny obecně platí, že každý prvek má právě jednoho vlastníka. Vlastníkem je ten prvek, který lexikálně obklopuje vlastněného. Tedy například vlastníkem přechodů a lokálních proměnných je vždy příslušný `BasicBlock`, vlastníkem podtermu 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 5.3.4). Koncept pamatování si svého vlastníka však není příliš zobecněný ani abstrahovaný.

5.3.4 Proměnné

Specifikace jazyka rozlišuje mezi čárkovanými (primed) proměnnými a nečárkovanými proměnnými. Protože se čárkované proměnné mohou vyskytovat pouze uvnitř přechodových pravidel a navíc každé čárkované proměnné odpovídá nějaké nečárkované proměnné, 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

Což takhle nakreslit obrázek, zachycující vlastnictví a uživatele proměnných?

⁶https://github.com/h0nzZik/libNTS_cpp

⁷<https://github.com/xbauch/nts-parser>

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.

5.3.5 Typové informace

Protože rozšíření jazyka NTS uvedená v sekci 3.3 znamenají zesložnění jazyka a jeho typového systému, je vhodné, aby libNTS uměla s typovou informací pracovat. V paměťové reprezentaci proto každý term nese informaci o svém datovém typu a už při vytváření termů a atomických propozic je ověřována typová korektnost (tedy jestli například nedochází k sčítání pole a reálného čísla). Knihovna nepodporuje přepočítání typové informace po změně nějakého termu (například v případě výměny proměnných), uživatel knihovny je tedy povinen měnit strukturu již používaných formulí a termů pouze tak, aby nedocházelo ke změně typovosti.

5.3.6 Uživatelská data

Dalším ze způsobů, jak knihovna libNTS ulehčuje práci svým uživatelům, jsou uživatelská data. Některé významné třídy (třeba třída **Variable**) totiž obsahují veřejně přístupnou členskou proměnnou **user_data**, která není knihovnou nijak používána. Aplikace využívající libNTS mohou do této proměnné uložit ukazatel na svá vlastní data, která jsou tak přístupná v konstantním čase (tedy rychleji než například při použití kontejneru **map**). Tato vlastnost je využívána jak ve zplošťovací proceduře, tak v nástroji pro partial order redukci.

5.3.7 Omezení

Knihovna libNTS v současné době nepodporuje celý jazyk NTS. Mezi chybějící vlastnosti patří práce s poli, zejména jejich přiřazení, reference polí a předávání polí jako parametrů funkcí. Tyto vlastnosti buď nejsou vůbec implementovány, nebo jsou netestovány. Další vlastnosti (anotace, zjišťování vlastníků prvků) jsou implementovány neúplně či nesystematicky. Takovýchto omezení by však mělo v čase ubývat, navíc se dá předpokládat, že do knihovny bude přibývat další funkcionality, která je v současné době implementována v externích nástrojích přímo.

Kapitola 6

Experimenty

6.1 Použití implementovaných nástrojů

Pro překlad a následnou partial order redukci programu v jazyce LLVM IR jsou potřeba následující nástroje:

- Hlavičkové soubory a knihovny LLVM verze 3.6
- Knihovna libNTS
- Překládací nástroj `llvm2nts`
- Nástroj implementující partial order redukci

6.1.1 LLVM

Přestože některé linuxové distribuce (například Fedora) mají LLVM ve svých oficiálních repozitářích, může být vhodnější zkompilevat si LLVM vhodné verze manuálně. Kromě oficiálního gitového repozitáře¹ projektu je možné využít jeho zrcadlo na serveru GitHub². Verze 3.6 se nachází ve větvi `release_36`. Pro kompilaci nástroje `llvm2nts` je použit CMake, je proto vhodné i LLVM kompilovat pomocí CMake. Návod³ k tomu je součástí oficiální dokumentace LLVM.

6.1.2 libNTS

6.1.3 Kompilace

6.1.4 Instalace?

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

¹<http://llvm.org/git/llvm>

²<https://github.com/llvm-mirror/llvm.git>

³<http://llvm.org/docs/CMake.html>

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 podmnoužiny NTS - tedy do intů. Celé LLVM tak nepůjde, ale jistá podmnoužina ano. Každopádně zatím se na tom nepracuje.

Příloha A

Ukázky


```

nts parallel;

tpl      : BitVector<32>
sel[20]  : BitVector<32>

init  tpl = 0 && forall i : BitVector<32>[0, 19] .
      sel[i] = 0 ;
instances worker[20], main[1];

thread_func { ... }

main {
id : BitVector<64>;
initial si;
si -> s2 { id' = thread_create ( 0 ) }
...
}

worker {
initial s_idle;
s_idle      -> s_running_1 { sel[tid] = 1 }
s_running_1 -> s_stopped   { thread_func () }
s_stopped   -> s_idle      { sel'[tid] = 0 }
}

thread_create {
in  func_id : BitVector<32>;
out ret_tid : BitVector<64>;
wt      : BitVector<32>;
initial si;
final   sf;
error   se;

si -> s1 { tpl = 0 && tpl' = tid + 1 && wt' = 0 }
s1 -> sh { sel [wt] = 0 && sel '[wt] = func_id + 1 }
s1 -> sn { sel[wt] > 0 }
sn -> sl { wt < 19 && wt' = wt + 1 }
sn -> se { wt >= 19 }
sh -> sf { tpl' = 0 && ret_tid' = wt }
}

```

Obrázek A.1: Skutečný paralelní NTS vygenerovaný z programu v jazyce LLVM IR. Pro jednoduchost byly pouze přejmenovány proměnné a vynechány havoc() z formulí. Vstupní parametr `func_id` v BasicNts `thread_create` určuje funkci, která má být spuštěna ve volném vlákně. Globální proměnné `tpl` (thread pool lock) slouží jako zámek k ochraně kritické sekce.

Obsah

1	Úvod	1
1.1	Motivace	1
1.2	Přístup	1
1.2.1	Funkční volání	1
1.2.2	Architektura	2
1.3	Popis následujících kapitol	2
2	Použité technologie	3
2.1	LLVM	3
2.1.1	LLVM IR	3
2.1.2	Průchody	4
2.1.3	Využití	4
2.2	Posix threads	5
2.2.1	pthread_create()	5
2.3	NTS	5
2.3.1	BasicNts	6
2.3.2	Přechodová pravidla	6
2.3.3	Formule	6
2.3.4	Termy	6
2.3.5	Typový systém	6
2.3.6	Sémantika	7
2.3.7	Havoc	7
2.3.8	Paralelismus	7
2.3.9	Příklad	7
2.4	Partial Order Reduction	8
2.4.1	Přechodový systém	9
2.4.2	Nezávislost a komutativita přechodů	9
2.4.3	Nutné podmínky	10
2.4.4	Výpočet ample(s)	10
3	Překlad LLVM na NTS	11
3.1	Omezení LLVM	11
3.1.1	Pole	11
3.1.2	Pointery	11
3.1.3	Instrukce	11
3.1.4	Znaménkovost	12
3.2	Omezení Posix Threads	12
3.2.1	Funkce pthread_create	12

3.2.2	Funce <code>pthread_join</code>	13
3.2.3	Funce <code>pthread_exit</code>	13
3.3	Rozšíření NTS	13
3.3.1	Typ <code>BitVector</code>	13
3.3.2	Typovací pravidla	15
3.4	Vlastní překlad	15
3.4.1	Funkce	15
3.4.2	<code>BasicBlocky</code>	15
3.4.3	Paralelismus	16
3.5	Inliner	17
4	Partial Order redukce pro NTS	19
4.1	Terminologie a definice	19
4.2	Heuristika	20
4.2.1	Nezávislost	20
4.2.2	Ample sety	20
4.3	Statická POR	21
4.4	Problémy	22
4.4.1	Problém velkého vlákna	22
4.4.2	Problém velkého pole	25
4.4.3	Problém závislosti na datech	25
5	Implementace	26
5.1	Přehled existujících nástrojů	26
5.2	Volba implementačního jazyka a knihoven	26
5.3	Knihovna <code>libNTS</code>	26
5.3.1	Parser	27
5.3.2	Architektura	27
5.3.3	Vlastnictví	27
5.3.4	Proměnné	27
5.3.5	Typové informace	28
5.3.6	Uživatelská data	28
5.3.7	Omezení	28
6	Experimenty	29
6.1	Použití implementovaných nástrojů	29
6.1.1	LLVM	29
6.1.2	<code>libNTS</code>	29
6.1.3	Kompilace	29
6.1.4	Instalace?	29
7	Závěr	30
8	Nepřiřazeno	31
8.1	Inlining	31
8.2	Možná budoucí rozšíření	31
8.2.1	Znaménkové datové typy v NTS	31
	Příloha A Ukazky	32

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, Massachusetts 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>.
- 10 KURSHAN, R.; LEVIN, V.; MINEA, M.; PELED, D.; YENIG, H. Static Partial Order Reduction. *TACAS 98*. 1998, s. 345–357.