Kapitola 1

Cíl práce

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.

Kapitola 2

Použité technologie

Proč volíme právě NTS? Jakou podmnožinu těch jazyků podporuji?

2.1 LLVM

2.1.1 Typový systém

2.1.2 Omezení

Nestaráme se o pointery, nepodporuji pole (zatím)

2.2 NTS

2.2.1 Popis originální verze

Potřeba říci, že máme datové typy Int, Bool, Real. Int, Real jsou matematické, s neomezeným rozsahem a (u Real) s neomezenou přesností.

2.2.2 Rozšíření

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*⟩ ‘)’

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.

2.2.3 Omezení

Zatím nepodporuji složitější operace s poli. Také nechápu, jak pracovat s parametry (par).

Pole

2.2.4 Možná budoucí rozšíření

Znaménkové datové typy

Kapitola 3

Jak na to?

3.1 Použitá terminologie

Velký obraz - jak se vypořádat s funkčními voláními? Jaké jsem měl možnosti?

Výsledkem trojice: llvm2nts, inliner, vlastní POR. Všechno pracuje nad knihovnou pro paměťovou reprezentaci NTS. Related work: Petrův parser.

Všechno ve formě knihovny - jednoduché rozhraní, snadno použitelné.

3.2 Architektura libNTS

- inspirováno LLVM

3.3 Překlad llvm na nts

Btw nepatří rozhodnutí o omezení vstupního jazyka sem?

3.3.1 Model paralelizace

Že tedy budu mít nějaký thread pool a funkci thread_create, která nějakému vláknu (nebo procesu?) přiřadí úlohu, jež bude dané vlákno vykonávat. Kromě thread poolu ještě poběží hlavní vlákno.

3.4 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.

3.5 POR

Pozor, POR je víc druhů. Uvézt chytrou knížku, Ample sety.

3.5.1 Jak to má fungovat

Nakonec jde jenom o to, zda proměnnou, kterou nějaký přechod používá, používá i jiné vlákno

3.5.2 Problém velkého procesu

Protože každé vlákno (kromě vlastního) může potenciálně vykonávat libovolnou úlohu, tak téměř každé vlákno může použít téměř každou proměnnou. Redukce by se zredukovala na pouhý test "používám globální proměnné"? Tedy je potřeba mít rozdělené stavy / přechody do úloh. O každé úloze spočítáme, jaké globální proměnné používá, a také, jaké jiné úlohy může aktivovat. Potom, pokud budeme znát řídící stav každého vlákna, můžeme zjistit, jaké úlohy běží a tedy i jaké globální proměnné jsou důležité.

3.5.3 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.

3.5.4 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.