Možná, že strukturní prvek na nejvyšší úrovni by měl být "chapter", ne "section".

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.

2 Použité technologie

Proč volíme právě NTS? Jakou podmnožinu těch jazyků podporuji?

2.1 LLVM

2.1.1 Omezení

Nestaráme se o pointery, nepodporuji pole (zatím)

2.2 NTS

2.2.1 Rozšíření

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.

2.2.2 Omezení

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

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.