Kockavilág program

A program a kockavilág problématerében logikai következtetések ellenőrzésére alkalmas. Készítők: Kalmár Dániel (C++ program, dokumentáció), Németh Boldizsár (esszé, algoritmus tervek).

Fontosabb file-ok:

- txt/Kockavilag_Algterv_MSC.pdf: a feladatkiírás
- txt/Kockavilag.pdf: a kockavilág formális leírása
- txt/essze_v2.pdf: Boldizsár esszéje
- txt/rules.txt: rövid szövegfile a szabályok/axiómák egyszerűsített leírásával
- checker: a program maga

A programról

A program C++ nyelven írodott, Linux alatt készült és volt tesztelve. Fordításhoz 4.8.2-es gcc ajánlott és boost programkönyvtár szükséges. A “checker” könyvtárban a make parancsot kiadva fordul le, a “main” programot önmagában futtatva minden beállított teszt lefut. A fontosabb osztályok/file-ok:

- BaseCube: összefogó ősosztály a Cube és CubeSymbol típusoknak
- Cube: konkrét kocka típus (A, B)
- CubeSymbol: szimbolikus kocka típus (X, Y)
- Rule: ősosztály a szabályok definiálásához
- World: adott kockavilágot magába foglaló osztály, ez tartalmazza a kockák, kocka szimbólumok és szabályok listáját
- Model: egy adott állapot a konkrét kockák elrendezése szerint
- Interpretation: a szimbolikus kockák egy interpretációja
- Checker: az állítást ellenőrző algoritmus, amely az állapotteret járja be
- Test: különböző tesztelési függvények

A szabályok

Minden szabály a Rule osztályból származik, implementációjuk nagyon egyszerű. Minden szabály kaphat paraméterül akár valódi kockát, akár kocka szimbólumot. Jelenleg a következő szabályok léteznek:

- NegatedRule: adott szabály logikai negációja
- OnRule: On(x, y) szabály
- OnTableRule: T(x) szabály
- NothingOnTopRule: E(x) szabály
- AboveRule: A(x, z) szabály
- EqualsRule: x = y szabály

Látszik, hogy két fontos logikai szabály hiányzik: az ÉS és a VAGY implementációja, amik lehetővé tennék sokkal komplexebb szabályok leírását. Ezekre a program létrehozásához és vizsgálatához nem volt szükség. Több szabály megadása esetén azok automatikusan ÉS kapcsolatban lesznek.

A program helyessége

Bár a programhoz készültek tesztek, formális helyességének teljes vizsgálata egy igen komplex feladat lenne, amire a gyakorlatban nincs lehetőség. Azonban érdemes átnézni, hogyan kapcsolódik a program a formális szabályrendszerhez.

Egyenlőség axiómák (1 - 3)

Ezek teljesülése a Cube típus == operátorából következik. Amennyiben ezt megfelelően implementáljuk (az == operátor ekvivalenciareláció), a kockák egyenlősége az axiómáknak meg fog felelni.

Szabályok tulajdonságai (4 - 21)

Ezek a tulajdonságok a szabályok megfelelő implementációjából következnek.

Egyértelműség, oszlopok tulajdonságai (22 - 23)

Ezek az állapotteret bejáró algoritmus működéséből következnek. Az algoritmus felelőssége az hogy ne építsen “hurok” oszlopot, ne tegyen egy kockára többet párhuzamosan, stb.

A használt algoritmus

Az implementáció egy egyszerű brute-force algoritmust használ (állapottér bejárás visszalépéses kereséssel). Amennyiben ellenpéldát talál a tételhez, azonnal leáll. Az állapottér bejárás eleje az összes lehetséges interpretáció előállításával kezdődik, majd minden interpretációhoz minden modellt megvizsgál.

Az algoritmus a modellek előállításához a Boldizsár esszéjében leírt “Fix sorrendű elhelyezés” módszert alkalmazza. Egyesével végigmegy a kockákon és az aktuális kockát elhelyezi egy meglévő oszlopban valahova, vagy új oszlopot indít belőle. Amikor a kockák végére ért, az egy levél állapot. Minden ilyen levél állapotot létrehoz.

Amikor az algoritmus elér egy levelet, ott egyesével teszteli a szabályokat. Az (I => O) következtetést egyszerűen a (!I v O) logikai formulával teszteli, ahol I az összes bemeneti szabály ÉS kapcsolatban, O pedig az összes kimeneti szabály ÉS kapcsolatban.

Implementációs részletek

A programban a belső kocka típus `unsigned long` (előjel nélküli egész), amin az egyenlőség implicit módon helyesen működik. Ez az egész szám egy-az-egyben megfeletethető egy-egy stringnek, ami az adott kocka beírt nevét adja meg a felhasználónak a szebb interakció érdekében.

Adott modellben a kockák helyzetét map (szótár) objektum tárolja, ami azt adja meg, melyik kocka alatt melyik van. Amennyiben egy kocka nem szerepel ebben, úgy az az asztalon van. A gyorsabb működéshez van még egy segéd map, ami ennek a fordítottja, és az adott kocka fölötti kockát tárolja.

Az interpretációban szintén egy map objektum tárolja, hogy melyik szimbólumhoz melyik kocka lett rendelve.

Tesztek

A programban a bizonyítandó állítások megadásához létrehoztam egy egyszerű leírónyelvet macro-k segítségével. Bár sok helyen lehetne még bővíteni és szépíteni ezt, a jelenlegi céloknak megfelel.

A beépített tesztek kimenete lehet PASS vagy FAIL. A PASS jelentése az, hogy a tételbizonyítás eredménye az elvárt, kézzel definiált eredmény (lehet igaz vagy hamis is), tehát a teszt átment. A FAIL ennek ellentétje. Amennyiben a tétel hamis (függetlenül attól, hogy a teszt átmegy vagy nem), a program ellenpéldát is ad.

Amikor a program ellenpéldát talál, vagy bármilyen okból egy adott modellt ír ki, helymegtakarítás végett horizontálisan teszi azt meg. Minden egybefüggő kiírt string egy kockaoszlop, aminek a bal oldala az oszlop teteje, a jobb oldalon lévő | jelzés pedig az asztalt mutatja. Például:

```
AB| C|
```

két oszlopot jelöl: az egyikben A van B tetején, a másik C önállóan.

Egyszerű tesztek

Néhány egyszerűbb teszt csak arra szolgál, hogy a program helyességét és alapvető működését ellenőrízze. Ezek nem generálnak releváns kimenetet, mind PASS értéket ad.

Példák a feladatkiírásból

A következő két példa a formális leírás dokumentumból (Kockavilag.pdf) a két példafeladat implementációját és eredményét mutatja be a programban.

Első feladat

Két kocka van: A és B. Az A kocka nincs az asztalon, a B az asztalon van. Lássuk be, hogy A rajta van B-n.

A feladat leírása a programban:

```
TEST(
```

```
      IN(new OnTableRule(CUBE("B")))
      IN(new NegatedRule(new OnTableRule(CUBE("A"))))
      OUT(new OnRule(CUBE("A"), CUBE("B"))),
      true)
```

A TEST macro segít definiálni egy feladatot. Az IN az input szabályokat, az OUT az output szabályokat adja meg. A végén a true azt jelzi, igaz eredményt várunk, vagyis a tétel bizonyított.
A program kimenete:

```
PASS, steps: 3
```

Tehát átment a teszten (a tétel bizonyított) és három állapotot kellett ehhez bejárni.

Második feladat

Négy kocka van: A, B, C, D.
Input állítások:

1. Az A az asztalon van.
2. A B üres.
3. A C nem üres.
4. Legalább két kocka az asztalon van.

Output állítás:

1. B nincs a D-n.

A feladat leírása a programban:

```
TEST(
      IN(new OnTableRule(CUBE("A")))
      IN(new NothingOnTopRule(CUBE("B")))
      IN(new NegatedRule(new NothingOnTopRule(CUBE("C"))))
      IN(new OnTableRule(SYMBOL("X")))
      IN(new OnTableRule(SYMBOL("Y")))
      IN(new NegatedRule(new EqualsRule(SYMBOL("X"), SYMBOL("Y"))))
      OUT(new NegatedRule(new OnRule(CUBE("B"), CUBE("D")))),
      true)
```

A program kimenete:

```
7: FAIL, steps: 495, counter example: Model: BDC| A| , Interpretation: X -> C Y -> A
```

Tehát a program nemcsak hogy észrevette hogy a megadott leírással szemben a tétel mégsem igaz, de még ellenpéldát is adott.

Méret teszt

Ez egy egyszerű teszt függvény ami adott számú kockák állapotterét bejárja. Lehet vele vizsgálni a bejáró függvény sebességét és az állapottér méretét a kockák számának függvényében. A teszt nem használ input és output szabályokat és szimbolikus kockákat.

Kimenete:

```
World of size 1 -> 1 steps.
World of size 2 -> 3 steps.
World of size 3 -> 13 steps.
World of size 4 -> 73 steps.
World of size 5 -> 501 steps.
World of size 6 -> 4051 steps.
World of size 7 -> 37633 steps.
World of size 8 -> 394353 steps.
```

8-nál több kockával már nem érdemes tesztelni, mert túl sokáig tart az állapottér bejárása.

Két torony teszt

A Boldizsár esszéjében leírt “Két Torony” problémából épít egy világot és futtatja le rá az ellenőrzést. Paraméterül kapja n-t, a tornyok magasságát.

Kimenete:

```
Towers 2, steps: 9477
Towers 3, steps: 195703125
```

n = 4-re már túl lassú a program és nem fejeződik be időben.

Fejlesztési lehetőségek

A program továbbfejlesztésével két irányba lehetne elindulni. Először is sok osztály és függvény implementációja messze nem optimális - valamennyi tervezéssel és módosítással legalább 50%-al fel lehetne gyorsítani a programot csak ezzel. Ennek oka, hogy nagyon sok allokációt végez a program (minden modell és interpretáció létrehozásánál). Ezt el lehetne kerülni átmeneti tárolók okos használatával.

Másodszor is, ami fontosabb, hogy sok egyszerűbb vagy bonyolultabb algoritmikus optimalizációt lehetne alkalmazni. Ezeknek egy részéről Boldizsár esszéjében vannak részletek. Sajnos az egyszerűbb optimalizációk mind olyanok, hogy a leglassabb teszten (két torony) nem segítenek.

Ezek mellett a fent említett ÉS és VAGY szabályokat is implementálni lehetne. Ezekkel együtt a szabályrendszer már teljes lenne, és létre lehetne hozni egy valódi leíró nyelvet szabályok definiálására.