Generatív programok helyessége

Doktori értekezés tézisei
2013

Pataki Norbert
patakino@elte.hu

Témavezető: **Dr. Porkoláb Zoltán, egyetemi docens**
Eötvös Loránd Tudományegyetem, Informatikai Kar,
1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doktori Iskola
Doktori program: Az informatika alapjai és módszertana
Az iskola vezetője: Dr. Benczúr András
A program vezetője: Dr. Demetrovics János akadémikus

---

1. Bevezetés

A *C++ Standard Template Library (STL)* a generikus programozási paradigmán alapuló könyvtárak mintapéldája [1]. Professzionális C++ program elképzelhetetlen a szabványkönyvtár részét képező STL alkalmazása nélkül. Az elegáns kialakítású könyvtár használata csökkenti a klasszikus C és C++ hibák lehetőségét, növeli a kód minőségét, karbantarthatóságát, érthetőségét és hatékonyságát [20].

Ugyanakkor a könyvtár alkalmazása nem garantál hibamentes kódot, sőt a könyvtár generikus megközelítése miatt új típusú hibalehetőségek keletkezhetnek. Ezeknek egy részét a fordítóprogram nem ellenőrzi és futási időben sem feltétlenül derül ki a kód hibás jellege. Nem megdöbbentő, hogy ilyen hibák nagy számmal előfordulnak C++ nyelven írt programok implementációjában [5]. Kutatásaim középpontjában ezen hibalehetőségek leküzdése áll mégpedig úgy, hogy az STL hatékonysága és rugalmassága megmaradjon.

2. A C++ Standard Template Library

A C++ Standard Template Library (STL) egy generikus programozási paradigmán alapuló könyvtár, mely része a C++ szabvány könyvtárának. Az STL kihasználja a C++ sablonok lehetőségeit, így egy bővíthető, hatékony, mégis flexibilis rendszert alkot. Az STL alapvető komponensei: konténerek (containers), algoritmusok (algorithms), iterátorok (iterators), allokátorok (allocators), funktorok (functors), átalakítók (adaptors).

A *konténerek* (pl. `vector`, `set`, `map`, stb.) alapvető feladata az adatok memóriában történő elhelyezése, tárolása és a memória konzisztensen tartása. Az *iterátorok* egy egységes interface segítségével definiálják a memóriában elhelyezett elemek elérését. Az *algoritmusok* a használt iterátorok típusa alapján paraméterezhetőek, így ezek konténer-független függvénysablonok gyakori feladatokra, mint például keresés, rendezés, másolás, számlálás. Az iterátorok garantálják az algoritmusok és a konténerek függetlenségét. Az STL egyik fontos komponense a *funktor*. Funktorok segítségével felhasználói kódrészleteket lehet hatékonyan végrehajtani a könyvtáron belül: funktorok definiálhatnak rendezéseket, predikátumokat, vagy valamilyen műveletet, amit végre szeretnénk hajtani az elemeken. Az *allokátorokat* eredetileg a memóriamodellek absztrakciójaként fejlesztették ki. A memóriaallokálás testreszabásához az összes szabványos STL konténer ad megoldást: az utolsó sablon paraméter az allokátor típusát definiálja. Van default értéke, de másik típus is megadható helyette.

3. Motiváció

A dolgozatban bemutatom azokat a nehézségeket, amelyekkel a programozóknak szembe kell nézniük az STL használatakor. Ismertetem azokat a problémákat, amelyeket az STL hibás használata okozhat. Ezek a hibák okozhatnak nehezen értelmezhető fordítási hibaüzeneteket, nem portábilis kódot, hibás futási eredményeket, memória szivárgást, korrupttá vagy inkonzisztenssé váló adatszerkezeteket illetve szükségtelen hatékonyságromlást. A felsorolt problémák egy részére dolgozatom megoldást kínál, más problémák további kutatások tárgyai.

A dolgozatban ismertetem az algoritmusokkal kapcsolatos hibákat: másoló algoritmusokkal elkövethető, futás idejű hibákat okozó helytelen használatukat, a törlő algoritmusokban rejlő potenciális hibalehetőségeket. Megmutatom, hogy a `unique` algoritmus használta miért nem intuitív. Bemutatok olyan STL-es algoritmusokat melyeknek olyan előfeltétele van, amit nem ellenőriz a fordítóprogram, így használatuk hibákat rejthet. Ezenkívül megvizsgálom, hogy a `count` és `find` algoritmusok milyen környezetben viselkedhet hibásan.

A dolgozatban bemutatom a konténerekkel kapcsolatos jellegzetes hibalehetőségeket: az `auto_ptr`-eket tartalmazó konténereket (COAP-okat), a `vector<bool>` specializációjának okát és szabványnak ellentmondó működését. Ismertetem a `string` és a `vector` reallokációjával kapcsolatos problémákat, valamint az asszociatív konténerek hordozhatósággal kapcsolatos problémáit. Megvizsgálom, hogy a konténerek virtuális destruktorának hiánya milyen hibákat okozhat.

A dolgozatban átnézem az iterátorokkal kapcsolatos problémákat: az invalidálódással és a konverzióval kapcsolatban, valamint megmutatom a pointerek és az iterátorok összetévesztésével járó gondokat is.

Ismertetem az STL-es funktorokkal és allokátorokkal kapcsolatos hibalehetőségeket, ezenkívül a nehezen érthető fordítási hibákat és a fejállományokkal kapcsolatos, portolási hibákat okozó lehetőségeket is.

4. Célkitűzések

Azt a célt tűztem ki magam elé, hogy a programozók dolgát megkönnyítem az elkövethető hibák minél átfogóbb kiszűrésével. A szűrést segítsem mind *formális*, mind *szoftveres* eszközökkel. Egy kísérleti eszköz az STLlint [4], mely módosított fordítóprogram alapján működik, sokáig online elérhető volt, de működése nem váltotta be a hozzá fűzött reményeket, támogatása megszűnt. Az STLlint kizárólag fordítási idejű információk alapján mű-

ködött. Az én megoldásaim ezzel szemben a könyvtár implementációjának bővítésén, változtatásán alapulnak, szabványos fordítóprogramok használata mellett. Én is igyekeztem a lehetséges hibákat fordítási időben felderíteni és a C++ sablon konstrukciója segítségével fordítási figyelmeztetéseket generálni, de a megoldásaim egy része futási időben működik. Tehát céljaimat a következő prioritással lehet definiálni:

1. Az STL generikusságából adódó hibalehetőségek kiszűrése *fordítási időben, nem-intruzív* módon.

2. Az STL generikusságából adódó hibalehetőségek kiszűrése *fordítási időben, az STL implementáció módosításával.*

3. Az STL generikusságából adódó hibalehetőségek kiszűrése *futási időben, nem-intuzív módon, a szabványos aszimptotikus futási idők betartásával* (törekedve a minimális overhead-re).

4. Az STL generikusságából adódó hibalehetőségek kiszűrése *futási időben, az STL implementáció módosításával, a szabványos aszimptotikus futási idők betartásával* (törekedve a minimális overhead-re).

5. Az STL generikusságából adódó hibalehetőségek kiszűrése *futási időben, nem-intruzív módon, a szabvány aszimptotikus futási idő korlátainak megsértésével.*

6. Az STL generikusságából adódó hibalehetőségek kiszűrése *futási időben, az STL implementáció módosításával, a szabvány aszimptotikus futási idő korlátainak megsértésével.*

5. Az STL formális megközelítése

A C++ nyelv szabványa az STL specifikációját is tartalmazza. Sajnos az STL specifikációja informális [14]. Ez félreérthető és megnehezíti a helyesség vizsgálatát.

Kétféle formális eszközt mutattam be az STL specifikációjához: az első az általam kidolgozott technika, amely *elő- és utófeltételek* segítségével definiálja az STL-t. A technika a számítástudomány mai napig népszerű *Hoare módszerén* alapul, felhasználható STL-t használó programok, könyvtárak, valamint STL implementációk helyességének vizsgálatához. A másik technika *LaCert* nyelven írt *temporális logikai* eszközöket használó specifikáció. Ennek alapvető célja, hogy specifikációkból a LaCert fordítóprogram generálja az

STL alapú C++ kódokat kritikus pontokon. Ezen specifikáció elkészítésében részt vettem, de Dévai Gergely munkájának tartom.

Ezeket a specifikációkat felhasználhatjuk a STL-t használó programok és könyvtárak helyességének az ellenőrzésére, az esetleges hibák kiszűrésére, STL implementációk helyességének vizsgálatára [16]. A LaCert nyelvű specifikációk integrációjával olyan STL alapú kódok generálhatók, amelyek formálisan bizonyítottak [3].

1. Tézis. Eszközrendszert dolgoztam ki, amely alkalmas generikus programok formális specifikációjára. Az eszközrendszer alkalmasságát a C++ Standard Template Library alapvető komponenseinek formális specifikálásával mutattam meg. A módszer segítségével pontosabbá tehető a könyvtár definíciója és kisebb méretek esetén a program-helyességi vizsgálatokban is felhasználható.

A tézishez kapcsolódó fontosabb publikációim: [2, 3, 6, 14, 16].

6. Fordítás idejű megoldások

A fordítóprogramok nem tudnak figyelmeztetéseket (warning-okat) adni az STL (vélhetően) szemantikusan hibás alkalmazásakor [21]. A fordítási idejű megoldások egyik nagy előnye az, hogy a leforduló kódok futási ideje nem változik, így megmarad az STL hatékonysága és a specifikációt továbbra is betartja az implementáció.

Az STL biztonságos használatához adtam olyan eszközöket, amelyek fordítási időben ellenőrzik az STL használatát. Ha valamelyik konstrukcióról feltehető, hogy hibás viselkedése lehet futás közben, arról fordítási figyelmeztetést ad a fordítóprogram. Kidolgoztam egy eljárást, amellyel ,,tetszőleges" figyelmeztetések generálhatóak. Ehhez a viselkedéshez nem a fordítóprogramokat változtattam meg, hanem a könyvtár kódját módosítottam.

A believe-me mark-ok olyan *annotációk*, amelyek nem eredményeznek futás idejű aktivitást, csak az általam generált specifikus warning-ok letiltására terveztett kódrészletek. A figyelmeztetésekhez believe-me mark-okat adtam, ami segíthet a programozóknak végiggondolni a kód helyességét.

2. Tézis. Módszereket dolgoztam ki, melyekkel fordítási időben detektálhatjuk az STL bizonyos hibás használati eseteit. A módszer a könyvtár implementációjának C++ szabvány szerinti módosításán alapul, ezért megoldásaim minden szabványos C++ fordító használata esetén alkalmazhatóak. A módszerek a hibás példányosításokat (2.1), egyes algoritmusokat (2.2),

adaptálható funktorokat (2.3), az allokátorokat (2.4), a reverse iterátorokat (2.5) és a lusta példányosítást (2.6) érintik.

Hibás példányosítások	[10]
Algoritmusok	[15]
Adaptálható funktorok	[8]
Allokátorok	[13]
Reverse iterátorok	[13]
Lusta példányosítások	[9]

1. táblázat. A tézishez kapcsolódó fontosabb publikációim

7. Futás idejű megoldások

Sajnos fordítási időben nem áll minden információ rendelkezésünkre ahhoz, hogy eldönthessük, hogy egy potenciális hiba ténylegesen be fog-e következni. Futási időben több információ áll a rendelkezésünkre, hogy ezekre a kérdésekre válaszoljunk, jelezzük a felhasználó számára, esetleg korrigáljuk a rendszer viselkedését. Ilyen esetben az ellenőrzések futási időben értékelődnek ki, ami költséggel jár, a futási idő akár jelentősen megnőhet. Különböző módszereket, komponenseket dolgoztam ki, amelyek futási időben növelik a könyvtár használatának biztonságát.

3. Tézis. Módszereket dolgoztam ki, melyek segítségével futási időben lehet a C++ Standard Template Library egyes hibás használati eseteit detektálni illetve elkerülni, A módszerek az iterátor invalidációt (3.1), a másolás-biztos iterátorokat (3.2), a törlő iterátorokat (3.3), egyes algoritmusok előfeltételeit (3.4) és a funktorok használatát (3.5) érintik.

Invalid iterátorok	[12, 18, 19]
Másolás-biztonságos iterátorok	[11, 12]
Törlő iterátorok	[11, 12]
Algoritmusok előfeltétele	[12, 18, 19]
Funktorok	[8, 9]

2. táblázat. A tézishez kapcsolódó fontosabb publikációim

Hivatkozások

[1] Czarnecki K., Eisenecker, U. W.: *Generative Programming: Methods, Tools and Applications*, Addison-Wesley (2000).

[2] Dévai, G., Pataki, N.: *Towards verified usage of the C++ Standard Template Library*, In Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST) 2007, pp. 360–371.

[3] Dévai, G., Pataki, N.: *A tool for formally specifying the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **31** (2009), pp. 147–166.

[4] Gregor, D., Schupp, S.: *STLlint: Lifting static checking from languages to libraries*, Software: Practice and Experience **36(3)** (2006), pp. 225–254.

[5] Meyers, S.: *Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley (2001).

[6] Pataki, N.: *A C++ Standard Template Library helyességvizsgálata* (TDK Dolgozat), Országos Tudományos Diákköri Konferencia (2005).

[7] Pataki, N.: *C++ Standard Template Library by Ranges*, in Proc. of the 8th International Conference on Applied Informatics (ICAI 2010) Vol. 2., pp. 367–374.

[8] Pataki, N.: *Advanced Functor Framework for C++ Standard Template Library* Studia Universitatis Babeş-Bolyai, Informatica, Vol. **LVI(1)** (2011), pp. 99–113.

[9] Pataki, N.: *C++ Standard Template Library by Safe Functors*, in Proc. of 8th Joint Conference on Mathematics and Computer Science, MaCS 2010, Selected Papers, pp. 363–374.

[10] Pataki, N.: *C++ Standard Template Library by template specialized containers*, Acta Universitatis Sapientiae, Informatica **3(2)** (2011), pp. 141–157.

[11] Pataki, N.: *Advanced Safe Iterators for the C++ Standard Template Library*, in Proc. of the Eleventh International Conference on Informatics, Informatics 2011, pp. 86-89.

[12] Pataki, N.: *Safe Iterator Framework for the C++ Standard Template Library*, Acta Electrotechnica et Informatica, Vol. **12(1)**, pp. 17–24.

[13] Pataki, N.: *Compile-time Advances of the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando

Eötvös Nominatae, Sectio Computatorica **36** (2012), Selected papers of 9th Joint Conference on Mathematics and Computer Science MaCS 2012, pp. 341–353.

[14] Pataki, N., Dévai, G.: *A Comparative Study of C++ Standard Template Library's Formal Specification*, in Conference of PhD Students in Computer Science, CSCS 2008, Volume of extended abstracts, 2008, p. 48.

[15] Pataki, N., Porkoláb, Z.: *Extension of Iterator Traits in the C++ Standard Template Library*, in Proc. of the Federated Conference on Computer Science and Information Systems (FedCSIS 2011), pp. 919–922.

[16] Pataki, N., Porkoláb, Z., Istenes, Z.: *Towards Soundness Examination of the C++ Standard Template Library*, In Proc. of Electronic Computers and Informatics, ECI 2006, pp. 186–191.

[17] Pataki, N., Szűgyi, Z.: *C++ Exam Methodology*, Annales Mathematicae et Informaticae **37** (2010), pp. 211–223.

[18] Pataki, N., Szűgyi, Z., Dévai, G.: *C++ Standard Template Library in a Safer Way*, In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.

[19] Pataki, N., Szűgyi, Z., Dévai, G.: *Measuring the Overhead of C++ Standard Template Library Safe Variants*, Electronic Notes in Theoretical Computer Science (ENTCS) Vol. **264(5)** (2011), pp. 71–83.

[20] Stroustrup, B.: *A C++ programozási nyelv*, Kiskapu Kiadó (2000).

[21] Van Wyk, E., Borin, D., Huntington, P.: *Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions*, in Proc. of the Second International Workshop on Library-Centric Software Design (LCSD '06), pp. 35–44.