A metaprogramozás elmélete és alkalmazásai

Zólyomi István

2008. november 20.

Tartalomjegyzék

1. Bevezetés (2-3 oldal) 6
1.1. A dolgozat felépítése 7

2. Generatív programozás 9
2.1. Az objektum-orientált programozás 10
2.1.1. A módszertan előnyei 10
2.1.2. Korlátok és hiányosságok 10
2.2. A generatív módszertan 10
2.2.1. Generikus programozás 10
2.2.2. Aspektus-orientált programozás 11
2.2.3. Szubjektum-orientált programozás 11
2.2.4. Jellemző-orientált programozás 11
2.2.5. Szándékalapú programozás 11
2.2.6. Szakterület-specifikus beágyazott nyelvek 11
2.2.7. Metaprogramozás 11
2.3. A generatív programozás nyelvi támogatása (5 oldal) 12
2.3.1. Kezdeti nyelvi támogatások 12
2.3.2. Funkcionális nyelvek 12
2.3.3. ADA 12
2.3.4. Java 12
2.3.5. C# 12
2.3.6. C++ 12
2.3.7. D 13

3. Metaprogramozás 14
3.1. Definíciók 15
3.2. Metaadatok 16
3.2.1. Hagyományos adatok 17
3.2.2. Típusok 17
3.2.3. Elemi vizsgálatok 17
3.2.4. Típusleírók 18
3.2.5. Szintaxisfák 19
3.3. Programtranszformációk 20
3.3.1. Metaadatok olvasása 20
3.3.2. Fordítás 21
3.3.3. Kódgenerálás 21
3.3.4. Bővítés 22
3.3.5. Kiterjesztés 23
3.3.6. Átszervezés 23
3.3.7. Teljes átalakítás 24
3.4. Alkalmazások 24
3.4.1. Információ kinyerése 25
3.4.2. Fordítóprogramok 25
3.4.3. Speciális (grafikus, beágyazott és többszintű) nyelvek 26
3.4.4. Konverzió, sorosítás, adattárolás 26
3.4.5. Kommunikáció, szolgáltatások elérése 27
3.4.6. Optimalizálás 28
3.4.7. Átszervezés 29
3.4.8. Típusrendszer átalakítása 29
3.5. Metaprogramozási környezetek 30
3.5.1. A C++ nyelv sablonjai 30
3.5.2. A D nyelv sablonjai 32
3.5.3. Virtuális gépek (Java és C#) 34
3.5.4. Ruby és más szkriptnyelvek 35
3.5.5. MetaML és MetaOCaml 36
3.5.6. Stratego/XT 37

4. Típusok tulajdonságainak vizsgálata 38
4.1. A probléma leírása . . . 39
4.2. Javasolt megoldás . . . 40
4.2.1. A C++ metaprogramozás felhasznált eszközei . . . 41
4.2.2. Predikátumok megvalósítása . . . 46
4.2.3. Predikátumok kompozíciója . . . 52
4.2.4. Vizsgálatok eredményének felhasználása . . . 53
4.2.5. Értékelés . . . 55
4.3. Kapcsolódó művek . . . 56
4.3.1. Kitekintés más nyelvekre . . . 57
4.3.2. C++ nyelvű megközelítések . . . 58

5. A típusrendszer kiterjesztése 61
5.1. A probléma leírása . . . 61
5.1.1. Formális leírás . . . 65
5.2. Lehetséges megközelítések . . . 66
5.2.1. Hagyományos öröklődés . . . 66
5.2.2. Virtuális öröklődés . . . 67
5.2.3. Szignatúrák . . . 67
5.2.4. Aspektusok . . . 68
5.2.5. Strukturális altípusosság . . . 68
5.3. Megoldás . . . 69
5.3.1. Típuslisták . . . 69
5.3.2. Osztályok kompozíciója . . . 71
5.3.3. A strukturális altípusosság megvalósítása . . . 73
5.3.4. Egy továbbfejlesztett megvalósítás mutatókkal . . . 78
5.4. Összegzés . . . 82
5.5. Kapcsolódó eredmények . . . 82

6. Szerializáció (20 oldal) 83
6.1. A probléma leírása . . . 83
6.1.1. XML séma . . . 83
6.2. Lehetséges megközelítések . . . 83

6.3. Megoldás 84
6.3.1. Típusmodell megadása 84
6.3.2. Metaadatok generálása 86
6.3.3. Osztályok generálása 86
6.3.4. Sorosító algoritmus 86

7. Related (5-8 oldal) 87

8. Összegzés (TODO 3 oldal) 88

Ábrák jegyzéke

5.1. Példa több interfész megvalósítására 64
5.2. Példa a C++ STL könyvtárából 65
5.3. A többszörös öröklődéssel felépített osztályhierarchia 73
5.4. A mutatókkal megvalósított osztályhierarchia 79
5.5. A CPtrSet működési elve 81

1. fejezet

Bevezetés (2-3 oldal)

A generatív programozás egy tudományos szemmel mérve új, feltörekvő paradigma, mely a programozási folyamat minél hatékonyabb automatizálását tűzte ki célul maga elé. A hagyományosnak mondható objektum-orientált módszertan lehetséges kiterjesztéseit vizsgálja, különböző alparadigmái más-más módszerrel próbálják a célt elérni: az aspektus-orientált programozás egyszerűen leírható programtranszformációkat végez, a generikus programozás minél általánosabb célú, absztraktabb, paraméterezhető programkomponenseket próbál megalkotni. Az alparadigmák közül legáltalánosabb az automatizált kódgenerálást és átalakítást előnyben részesítő metaprogramozás módszertana, mivel a többi paradigma ennek speciális aleseteként is értelmezhető. A dolgozat célja a generatív programozás, ezen belül is elsősorban a metaprogramozás előnyeinek, új lehetőségeinek és lehetséges alkalmazásainak vizsgálata.

A dolgozat több alapproblémát mutat be, ahol a generatív programozási technikák, elsősorban a generikus- és metaprogramozás használata jelentős előnyökkel jár az alapjául szolgáló objektum-orientált programozási technikákhoz képest. A dolgozatban túlnyomórészt használt nyelv a C++, melynek sablon (template) nevű eszköze meglehetősen erős kifejezőerővel bír, különösen a hasonló kategóriájú nyelvek eszközeihez képest. Más nyelvek generic nevű konstrukciójához hasonlóan támogatást nyújt a generikus programozáshoz, egyéni felépítésének köszönhetően azonban önmagában is Turing-teljes nyelvet alkot a fordító programozására, ezzel lehetővé teszi a metaprogramozást is.

1.1. A dolgozat felépítése

Ezen bevezető után egy áttekintést adok a generatív programozásról. A 2. fejezetben külön kitérek a céljaira, különböző elméleti irányzataira, valamint ezek megvalósulására napjaink széleskörűen használt programozási rendszereiben. A 3. fejezetben a generatív irányzatok közül részletesebben bemutatom a metaprogramozás módszertanát, kiemelve a C++ template metaprogramozást. Alapelveinek és lehetőségeinek ismertetése után kitérek a metaprogramozás jelenlegi és esetleges jövőbeli alkalmazásaira. Az átfogó bevezetés után kutatásaim három fontosabb eredményét ismertetem publikációim alapján.

A 4. fejezetben egy egyszerű metaprogramozási modellt vázolok fel előző publikációimra [1, 2] építve, mely elsősorban a programkód fordítási idejű önvizsgálatát támogatja. Ez alapvető eszköz a fordítóprogramba ágyazott kódgenerátorok létrehozásához, melyek többnyire ilyen vizsgálatokkal szerzett információkat használnak fel működésük során. Elemi vizsgálatok és különböző kompozícióik segítségével egy általános kódvizsgáló rendszer építhető a fordítóprogramon belül, ezzel széleskörű lehetőségeket biztosítva a metaprogramozás számára. A fejezetben megvizsgálom a rendszer C++ nyelvbe illeszthetőségének lehetőségeit a template-metaprogramozás eszközeivel, valamint módszereket adok a megvalósításhoz. A modell végül a jelenleg formálódó C++0x szabványban bevezetésre kerülő concept nevű nyelvi elemhez hasonló eszközt ad, ám annál természetesebben illeszkedik a már meglevő eszközökhöz, és kevesebb nyelvi kiterjesztést igényel.

Az 5. fejezetben a széleskörűen használt objektum-orientált nyelvek öröklődés alapú típusrendszerének néhány korlátját mutatom be, különös tekintettel az explicit módon jelölt altípusosság anomáliáira. Ismertetek néhány példát, melyek alapvetően a többszörös öröklődésből, vagy több interfész megvalósításából származnak, és a programozási környezetek gerincéül szolgáló rendszerkönyvtárak tervezésénél is komoly problémákat okoznak. Ezekre egy jó megoldást nyújthat a több nyelvben használt, ám a gyakorlatban széleskörűen még nem elterjedt strukturális altípusosság [33] alkalmazása, mely implicit leszármazási szabályokra épül. Korábbi kutatásaimra [4, 5] építve bemutatok egy C++ nyelvű, template metaprogramozáson alapuló megoldást, mellyel a meglevő szabályok mellé kiegészítő típuskonverziók biztosíthatók a strukturális konverziók megvalósítására. Ezáltal a meglevő típusmodell

kiegészítéseként a strukturális altípusosság természetes módon, nyelvi kiterjesztések nélkül a rendszerhez illeszthető.

A 6. fejezetben a futási idejű önleirást (reflection) biztosító típusok előnyeit vizsgálom [3] alapján, elsősorban különböző típusmodellek kölcsönös megfeleltetése és adatok más típusmodellre konvertálása szempontjából. Alkalmazásukkal lehetővé válik a különböző típusrendszerrel rendelkező nyelveken megírt, más adatreprezentációval dolgozó rendszerek közötti teljesen automatizált kommunikáció és adatkonverzió. A megoldás általános, kizárólag a típusok önleírásán alapul, formátumonként egyetlen általános algoritmussal dolgozik bármilyen egyéb kód generálása nélkül. Ezen elvekre építve bemutatok egy módszert, mellyel önműködő sorosítás (serialization) biztosítható C++ nyelvű adatok és XML Schema leírással rendelkező XML dokumentumok között. A megoldás hasonló a C# és a Java nyelveken rendelkezésre álló könyvtárakéhoz, ám ezekkel ellentétben az önleíró típusokat alapvetően nem támogató C++ nyelvhez készült. Felépítéséből adódóan a megoldás viszonylag kis memóriaköltségű, ezáltal kiválóan alkalmazható beágyazott vagy szűkebb erőforrásokkal rendelkező rendszerekben.

Saját eredményeim ismertetése után a 7. fejezetben kitekintek a témában mások által publikált kutatásokra is, bemutatva a területen elért eredményeket és megoldatlan problémákat, valamint a további kutatás lehetséges irányait.

2. fejezet

Generatív programozás

A programok fejlesztésének és karbantartásának magas költségének nem elhanyagolható része származik abból, hogy a programozási munkának még mindig jelentős részét teszik ki az ideális esetben automatizálható, ám kényszerűségből manuálisan végzett feladatok. Ilyenek például az erőforrások kezelése (memória, hardverek), a folyamatok párhuzamosítása, vagy akár csak egy osztályhierarchia áttervezése. Nemrégiben, a memóriát kezelő szemétgyűjtő rendszerek elterjedése előtt még ilyen feladat volt a dinamikusan foglalt memóriaterületek következetes felszabadítása, valamint az elkövetett hibáinkból származó memóriaszivárgások megszüntetése is. Ezen feladatok megvalósítására nehéz általános és hatékony módszert adni, ezért ezek többnyire a programozóra hárulnak. Ilyenkor nem csak saját megoldást kell terveznünk a feladat megoldására, hanem jelentős munkát igényel a megtervezett megoldás adott programozási rendszerbe illesztése is. Általában rengeteg redundáns információt kell szinkronban tartanunk, gondoljunk csak a memóriafoglalások és felszabadítások konzisztensen tartására. Természetes igény, hogy az ilyen monoton, kevesebb gondolkodást igénylő, nagy hibalehetőségű feladatokat automatizáljuk, hogy a programozás folyamata közben a feladat tényleges megoldására koncentrálhassunk.

A programozási munka megkönnyítésére számos kísérlet történt. Ebben a fejezetben a jelenleg legszélesebb körben elterjedt objektum-orientált programozás korlátainak bemutatása mellett az azt meghaladó különböző irányzatok, módszertanok áttekintése olvasható. Külön foglalkozom ezek megvalósulásával, gyakorlati

alkalmazásával a fontosabb programozási rendszerekben.

2.1. Az objektum-orientált programozás

(? mi is az a módszertan (paradigma), hogyan fejlődött, történelem ?)

csak röviden, sok hivatkozással

(TODO válogatni hivatkozasokat, kijelolni egy formalizmust, amit OO alatt ertunk mi is, pl. odersky, cardelli vagy bertrand meyer)

2.1.1. A módszertan előnyei

emberi gondolkodáshoz közel áll, egységbezárás, adatrejtés, modularizáció, karbantarthatóság, újrafelhasználás, stb

kiforrott modszertan, konyvtarak, tervezes [30]

2.1.2. Korlátok és hiányosságok

egydimenzios osztalyhierarchia sokszor nem eleg, ortogonalis feature-ok hozzaadasa (aspectek, mixinek, subject/feature oriented),

expression problem [51]

negativ variability [52]

2.2. A generatív módszertan

mi is az tulajdonkepp [35]

milyen területeken nyújt többet, mint az OO

2.2.1. Generikus programozás

mixin ide tartozik, vagy inkabb az aspectekhez?

generic a típusbiztos, fordítási idejű változata a leszármazáson alapuló megoldásoknak

paraméterezhető általános kódrészletek

Java és C# generic: type erasure, kódméret spórolás, statikus tagok nincsenek

C++ template: code bloat, lusta példányosítás, statikus tagok, teljes specializálás

2.2.2. Aspektus-orientált programozás

[31]

2.2.3. Szubjektum-orientált programozás

[36]

2.2.4. Jellemző-orientált programozás

Jellemző (feature)

[37]

2.2.5. Szándékalapú programozás

Szándékalapúság (intentional)

bővebben [35]-ben a II. rész 11. fejezete tárgyalja

TODO: 2006-os OOPSLA Simonyi-cikket meghivatkozni

2.2.6. Szakterület-specifikus beágyazott nyelvek

ez tenyleg ide tartozik?

2.2.7. Metaprogramozás

részletesebben a 3. fejezetben

2.3. A generatív programozás nyelvi támogatása (5 oldal)

2.3.1. Kezdeti nyelvi támogatások

ide mi kerul?

2.3.2. Funkcionális nyelvek

generikus fuggvenyekkel adott altalanos algoritmusok

2.3.3. ADA

generikus programozás támogatása [53]

több tekintetben fejlettebb az újabb nyelveknél

típus mellett adatok és függvények is lehetnek paraméterek

paraméterek megkötései

nehézkesebb használat a példányosítások miatt

2.3.4. Java

generikus programozás

típustörlés (type erasure)

paraméterek megkötései

2.3.5. C#

hasonló a Java nyelvhez

paraméterek megkötésére több eszköz

2.3.6. C++

A C++ nyelv sablonjainak egy rendkívül széleskörű és kimerítő tárgyalását adja [21].

generikus programozás támogatása

többi nyelv kifejezőerejénél jóval erősebb

Részletesebben a metaprogramozásról szóló részben olvasható, lásd 3.5.1.

2.3.7. D

A C++ nyelv nem kompatibilis továbbfejlesztésének tekinthető.

Részletesebben a metaprogramozásról szóló részben olvasható, lásd 3.5.2.

3. fejezet

Metaprogramozás

A programozás sok évtizedes története alatt a kifejlesztett rendszerek mérete és bonyolultsága folyamatosan nőtt. A munka megkönnyítésére egyre fontosabbá vált az újrafelhasználható könyvtárak, komponensek fejlesztése, melyhez a programkód széleskörű paraméterezhetősége és maximális absztrakciója nyújtott segítséget. A metaprogramozás ennek az absztrakciónak egy igen magas foka: a programok transzformációinak automatizálása magasabb szintű, programokat kezelő programokkal történik, ebből származik a *meta* elnevezés is. Az ilyen *metaprogramok* már nem csak "hagyományos" adatokkal dolgoznak, bemenetük része egy program, melyen működésük során különböző átalakításokat végeznek. A metaprogramok egy speciális esetének tekinthetők például a fordítóprogramok, melyek egy más, ekvivalens jelentésű reprezentációra (többnyire gépi kódra) alakítanak át bemenetként kapott programokat. Ez egy nagy múltú, jól ismert és kiváló szakirodalommal bíró terület, ezért a dolgozat elsősorban a nem ekvivalens átalakításokkal foglalkozik.

Hogy a dolgozat tárgyát jobban megérthessük, először a metaprogramok pontosabb meghatározására és egy egységes terminológiára van szükség, melyet 3.1 alatt tárgyalok.

Ezután a metaprogramozás alapelveit és alapvető eszközeit mutatom be kevésbé formálisan. Ahhoz, hogy egy programot képesek legyünk feldolgozni, átalakítani, szükségünk van arra, hogy információt nyerjünk ki a bemenetként kapott programról. A legtöbbször már ez is komoly gondot jelent, sok esetben a bemenő program tulajdonságainak csak egy töredékéről kapunk információt. Részletesebben 3.2 alatt

foglalkozom velük.

A kinyert metaadatok feldolgozásával tudnunk kell valamilyen hasznos működést is végezni. Ez jelentheti új programkód létrehozását, a bemenő program kiterjesztését, vagy egyéb átalakítását. Ezeket összefoglaló néven programtranszformációknak nevezzük, a témát 3.3 tárgyalja.

Ezután 3.4 alatt a metaprogramozás alkalmazásának különböző területeit járom körül.

Végül 3.5 alatt bemutatom a szélesebb körben elterjedt programozási nyelvek metaprogramozási képességeit.

3.1. Definíciók

Ahhoz, hogy a metaprogramozást részletesen tudjuk tárgyalni, először is be kell vezetnünk néhány alapvető meghatározást. Ezek a definíciók később nem szolgálnak alapul formálisan megfogalmazott, precíz matematikai tételekhez, céljuk kizárólag a dolgozat terminológiájának pontosítása. Következésképp a definíciók sem formális, inkább közérthetőbb, egyszerű szöveges alakúak.

A meghatározások megadásánál a [69] által leírt fogalmakat és programmodellt fogjuk felhasználni. Eszerint a program transzformációk sorozata, mely az *állapottér* nevű halmaz elemein dolgozik, és *állapotátmenet*ek segítségével változtatja meg az állapottér komponenseit. Véges számú determinisztikus állapotátmenet esetén a változásokat leíró függvények kompozícióját *programfüggvény*nek nevezzük, vagyis a programfüggvény a program bemenete és a hozzá tartozó kimenet közti közvetlen leképzést adja meg.

Metaprogram. Egy programot metaprogramnak nevezünk, ha állapotterének legalább egyik komponense a programok halmaza, vagy annak egy nem üres részhalmaza. Szemléletesebb módon leírva a program bemenetének és kimenetének legalább egyike tartalmaz egy programot.

Metaadat. A metaprogram programfüggvényének argumentumait metaadatoknak nevezzük. Másképpen fogalmazva ez a metaprogram bemeneteinek összességét jelenti.

Feldolgozás. Ha a metaadatoknak része egy program, akkor a metaprogramot programfeldolgozónak nevezzük. A feldolgozott program nem feltétlenül végez hasznos tevékenységet, hiszen a *Skip* vagy *Abort* is lehet paraméter. Értelmes feldolgozás azonban ezeken is végezhető, ilyen lehet például a program bonyolultságának mérése.

Kódgenerátor. Egy metaprogramot kódgenerátornak nevezünk, ha a hozzá tartozó programfüggvény értékkészletének része a programok halmaza. Más szóval kimenetének része egy program. Ez a program nem feltétlenül valamilyen átalakítást végző algoritmus, az is megengedett, hogy kizárólag ennél egyszerűbb programrészeket, például típusok vagy változók definícióját tartalmazza.

Programtranszformáció. Egy metaprogram programtranszformációt hajt végre, ha egy bemenő programot kiértékelve egy kimenő programot generál. A triviális eseteket, vagyis üres bemenő program feldolgozását vagy üres kimenő program generálását is megengedjük, vagyis a feldolgozás és kódgenerálás uniójáról van szó. Ezáltal gyűjtőfogalomként használhatjuk az összes lehetséges programokon végzett műveletre.

Valódi transzformáció. Olyan programtranszformáció, mely nem egy új, különálló kimeneti programot hoz létre a bemenet alapján, hanem magát a bemeneti programot változtatja meg. Másképp fogalmazva a bemeneti a metaprogram futása után felülíródik a kimenet eredményével.

3.2. Metaadatok

A metaadatok jelentősen különbözhetnek egy hagyományos program bemenetétől, biszen köztük már teljes programok is megjelenhetnek. A metaprogramnak képesnek kell lennie ezen programok tulajdonságait és felépítését felderíteni. Ez jelentheti paraméterül kapott típusok vizsgálatát, a programban szereplő konstansok, típusok felsorolását és szerkezetük vizsgálatát, vagy akár a bemeneti program kifejezésfáinak bejárását is. Sajnos a metaprogramozási környezetek többsége ennek csak egy töredékét biztosítja.

A metaprogramok sokszor rendkívül korlátozott információhalmazzal kénytelenek dolgozni. Ez különösen a metaprogramozást inkább mellékesen támogató rendszerekre igaz, melyek többnyire egy fordítóprogram belsejében futnak, és a bemenő programnak a fordítóprogram által felépített reprezentációjáról próbálnak adatokat kinyerni, mint például a C++ (lásd 3.5.1) nyelv sablon metaprogramjai.

A következőkben a metaadatokat fogjuk kategorizálni. A közönséges programok által is elérhető hagyományos adatoktól indulunk, minden új kategória az előzőnek egy bővítése, általánosítása lesz, míg a végén elérjük a programok teljeskörű leírását.

3.2.1. Hagyományos adatok

A metaadatok legalapvetőbb megjelenési formája a metaprogramnak biztosított valamilyen hagyományos bemenet. Ilyen lehet egy egész szám, karakterlánc vagy akár ezek listái. Például megadhatunk egy konstans egész számot egy fordítási idejű prímkiértékelőnek, lásd [47]. Ez a metaprogramok működéséhez elengedhetetlenül szükséges, de önmagában általában kevés, hiszen az ilyen adatok hagyományos programokkal jóval egyszerűbben és hatékonyabban feldolgozhatók. Hasznos olyan kódgenerátorok (lásd 3.3.3) esetén lehet, melyek kimenete kizárólag ilyen egyszerű adatokon alapul.

3.2.2. Típusok

A típusparaméterek jelentik a legegyszerűbb eszköz, mellyel a közönséges programok kifejezőereje már meghaladható. Ha a metaprogram bemenete egy típus is lehet (például a feldolgozott nyelv egy osztálya), már lehetőségünk van típussal paraméterezett kódot készíteni, mely a generikus programozás (lásd 2.2.1) fő eszköze. Ez az absztrakciós szint teljesen elfogadott és széles körben elterjedt, sablon néven (*generic* vagy *template)* a legtöbb modern programozási nyelv támogatja (lásd 2.3).

3.2.3. Elemi vizsgálatok

Típussal paraméterezhető programkód készítésénél sok esetben érdemes a paraméter tulajdonságaitól függő implementációt készíteni, mivel ezzel gyakran jelentősen

javítható a program idő- vagy tárhatékonysága. Használhatunk például különböző memóriafoglalási stratégiákat kis- és nagyméretű típusok allokálása esetén, a tárolt elemek mérete és száma szerint választhatunk a tároláshoz optimális konténer osztályt, vagy összehasonlítást támogató típusok esetén tárolhatjuk az elemeket rendezve, így optimalizálva a keresésre. A lehetséges döntési szempontok mindig alkalmazásfüggők, ám legtöbbször hasonló logika szerint épülnek fel. Általában a paraméter típus elemi tulajdonságait (traits) kérdezik le, s ezek kombinációját fordítási idejű feltételekben felhasználva döntenek. A szóba jöhető elemi tulajdonságok rendkívül változatosak lehetnek, az alábbiakban összegyűjtve a leggyakrabban használt tulajdonságok olvashatók:

- Típusok esetében ilyen lehet a típus egy példánya által igényelt memória mérete bájtokban. Eldönthetjük egy adott típusról, hogy van-e egy meghatározott nevű és típusú adattagja vagy metódusa, absztrakt típus-e, altípusa-e egy másik típusnak, netán konvertálható-e valamilyen formátumra. Verziózott típusok esetén megtudhatjuk a verziót is.
- Változók esetében lekérdezhetjük a típusát, vagy a méretét bájtokban. Adattagok esetén lekérdezhetjük a relatív címét (offset) a tartalmazó objektum kezdetéhez képest.
- Függvények esetében egy elemi lekérdezés megadhatja a szignatúrát, vagy információt nyújthat a hívási konvenciókról. Tagfüggvények esetén eldönthetjük, hogy dinamikus kötéssel rendelkezik-e, felüldefiniálható-e, vagy módosítja-e az objektumot, melynek tagfüggvénye.

Önmagukban ritkán támogatottak, a C++ nyelvhez a Boost könyvtár [18] próbálja ezek minél nagyobb részhalmazát megvalósítani. Legtöbbször azonban a lentebb olvasható típusleírókba beépülve láthatjuk őket.

3.2.4. Típusleírók

Egy típus összes tagjának teljes elemi tulajdonságleírásaiból alkotott halmazt típusleírónak nevezzük. Ez a függvény- és adattagokat, valamint a beágyazott típusok leírását egyaránt tartalmazza. Általában a típusok önleírásával valósítják meg, vagyis minden típus teljeskörű információt biztosít a saját belső szerkezetéről.

Az önleírás jóval többet nyújt az elemi tulajdonságoknál, hiszen tulajdonságok kizárólag már ismert szimbólumokról kérdezhetők le. Önleírással azonban maguk a szimbólumok is felderíthetők, hiszen bejárhatjuk a tartalmazott tagok vagy beágyazott típusok halmazát.

A típusok önleírásának megvalósítása két alapvetően különböző formában történhet. Gyakoribb a futási idejű (dinamikus) önleírás, ezt különböző néven a legtöbb interpretált vagy virtuális gépen futó környezet támogatja, például a Java és a C# reflection nevű szolgáltatása (lásd 3.5.3), vagy a szkriptnyelvek szinte mindegyike (lásd a Python inspect modulját, vagy a Ruby objektumait 3.5.4 alatt). Lényege, hogy a teljes önleírás a program futása közben, csak olvasható adatként áll a program rendelkezésére.

A fordítási idejű (statikus) önleírás esetén a tulajdonságok fordítási idejű konstansokként olvashatók, típusok esetében pedig karakterláncok helyett valódi típust kapunk. Ez a típusleírásnak egy jóval erősebb formája, hiszen a futási idejű leírásokat könnyen előállíthatjuk a fordítási idejűekből (például a fordítási időben kiolvasott adatokat futás közben is olvasható változókba és struktúrákba helyezzük el). Az önleírásból kinyert adatokat azonban használhatjuk metaprogramok paramétereiként is, ezzel jelentősen megnövelve azok kifejezőerejét. Ráadásul a típusbiztosság minden előnyével dolgozhat a metaprogram, hiba esetén futási idő helyett még fordítási időben kapunk hibát. Bár megvalósítására több kiterjesztés létezik különböző nyelvekhez (pl. OpenC++ [56]), kevés nyelvben van hozzá közvetlen támogatás, ilyen például Lisp ([62]), vagy a D nyelv (3.5.2).

3.2.5. Szintaxisfák

A számítógépes programok egyik ábrázolási formája az absztrakt szintaxisfa, a fordítóprogramok általában ilyen formára alakítják a szöveges bemenetüket. Ez az ábrázolási forma a programról jóval több információt nyújt, mint az egyszerű forráskód, hiszen a szintaxisfa a szöveg elemzésével épül fel, a különböző nyelvi elemek jelentésének és egymás közti kapcsolatainak figyelembevételével. Teljeskörű programinformációként ez jelenti a metaadatok legmagasabb szintjét.

Egy program szintaxisfájának elérése elvétve támogatott, különösen ritka a gyakorlatban használt programozási környezetekben. Mivel egyes alkalmazásokban a

fa elérése elengedhetetlen, ezt a közvetlen támogatás helyett sokszor kerülőúton oldják meg. Egyik ilyen megoldás a sablonokkal felépített kifejezésfák (expression template) alkalmazása, mely a sablonspecializációkat támogató nyelveken alkalmazható, például a C++ (3.5.1) vagy D (3.5.2) nyelv. Alapötlete szerint a fa belső csúcsai speciális sablonosztályokkal modellezhetők, a csúcsok gyermekei pedig a sablonok paraméterei. A műveleteket végző függvények és operátorok nem közvetlenül a művelet eredményét adják vissza, hanem a művelet által létrehozott kifejezésfának megfelelő típust, melynek külön kiértékelő művelete állítja elő az eredményt. Ilyen típusra egy példa a *Multiply<Matrix, Add<Matrix,Matrix> >* típus, mely a mátrixokkal végzett $A * (B + C)$ alakú művelet kifejezésfáját írja le a C++ sablonjaival. Az eredményként kapott kifejezésfa típusa sablonspecializációkkal bejárható, feldolgozható, maga a kiértékelő művelet működése is erre épül. Részletesen [10] tárgyalja, alkalmazásai általában a programkód hatékonyságát növelik (lásd 3.4.6).

3.3. Programtranszformációk

Ahogyan a metaadatok hozzáférésének, az azt felhasználó metaprogramozási műveleteknek is különböző szintjei vannak. A műveletek aszerint kategorizálhatók, hogy milyen mértékben változtatják meg a programot. Az alábbiakban az átalakítást nem igénylő műveletektől kezdve fokozatosan eljutunk majd a teljes átalakításig, a metaadatokhoz hasonlóan itt is minden új szint az előző egy kiterjesztése, általánosítása. Természetesen a lehetséges átalakítások mértékével együtt a metaprogram kifejezőereje is párhuzamosan nő. Minden szinthez felsoroljuk a hozzá tartozó legfontosabb alkalmazásokat is.

3.3.1. Metaadatok olvasása

Első szintünk az üres transzformáció, ez az átalakítások triviális esete. Valóban, egy program metaadatainak olvasása nem jár semmiféle átalakítással vagy mellékhatással, azonban már elégséges lehet egyes programfeldolgozást végző metaprogramok működéséhez. Az ilyenek csak feldolgozó, de nem kódgenerátor metaprogramok, többnyire statisztikákat állítanak össze, vagy a feldolgozott program különböző tulajdonságait vizsgálják, például bonyolultságot mérnek (lásd 3.4.1).

A legtöbb metaprogram azonban ennél bonyolultabb tevékenységet végez: általában programkódot is készít a kinyert metaadatok alapján, tehát kódgenerátor típusú metaprogram. Az általuk használt átalakítások kifejezőerejét a lentiekben vizsgáljuk.

3.3.2. Fordítás

Bár a fordítóprogramok (3.4.2) több évtizeddel öregebbek a metaprogramozásnál, mégis szépen illenek a metaprogramozás elméletébe. A fordítás során egy magas absztrakciós szintű jelölésrendszer[1], programnyelv segítségével megadott program feldolgozásával egy más nyelvű programot generálunk, mely a feldolgozottal teljesen egyenértékű. A generált program nyelve mindig alacsonyabb szintű, általában egy processzor gépi kódja vagy egy virtuális gép hordozható bájtkódja, tehát absztrakciót a fordítók nem végeznek. Ritkábban valamilyen közbülső formára, például a gépközeli C nyelvre fordítanak. A fordítás során a program szemantikájának a lehető legpontosabban meg kell maradnia, tehát egy jelrendszerek közti ekvivalens transzformációról van szó. Természetéből adódóan a fordítás semmilyen módon nem befolyásolja a bemeneti programot, a fordított kód ilyen szempontból tőle teljesen független.

A fordítás rendkívül összetett lépés. Szükség van hozzá a bemenő program összes metaadatának elérésére, hiszen enélkül nem is érthető meg a bemenet. Másrészt az adatok teljes elemzésére és feldolgozására is szükség van, hiszen

A fordítás nevezhető a transzformációk legelterjedtebb formájának, gyakorlatilag minden számítástudománnyal foglalkozó képzés részét képezi. A dolgozatban ezért nem tárgyaljuk bővebben a témát.

3.3.3. Kódgenerálás

Az átalakítások ezen formájánál megszűnnek a fordítás szemantikai korlátozásai, azonban a függetlenségi kitétel megmarad. Ez az utolsó szint, ahol tiltott a valódi

[1] A jelölésrendszer nem feltétlenül szöveges programnyelv alapú, elterjedt például a grafikai ábrázolás is. Ezek közül talán a folyamatábrák a legegyszerűbbek és legismertebbek, de ide sorolhatjuk az osztálydiagramokat is. Egyes módszertanok és fejlesztőkörnyezetek teljesen grafikus fejlesztést tesznek lehetővé, ilyen például a modellvezérelt programépítés (modell driven architecture).

transzformáció, vagyis működése során a feldolgozott programkód nem változhat meg, csak tőle teljesen független kód jöhet létre. E transzformációs szint elnevezése szándékosan azonos a metaprogram kategóriájának nevével, mivel a kódgenerátor metaprogramok számára ez az átalakítási szint engedélyezett.

A gyakorlatban széles körben elterjedt átalakítási forma. Alkalmazása gyakori sémaleírások (Xml, Sql, stb) alapján történő konverziós köd generálásánál (lásd 3.4.4). Egy másik alkalmazása során szolgáltatások leírása, vagy függvénydefiníciói alapján generálunk a szolgáltatás távoli elérésére kódot, elrejtve ezzel a kommunikációs bonyolultságot a szolgáltatás használata során (lásd 3.4.5). Ezek az alkalmazások általában jóval egyszerűbbek a fordításnál, és nincs hozzájuk szükség minden metaadatra, szintaxisfákat általában nem dolgoznak fel.

3.3.4. Bővítés

Az első, valódi transzformációt lehetővé tevő szint, vagyis segítségével nem csak új kódot generálhatunk, hanem megváltoztathatjuk a bemeneti programot is. A bővitésekhez tartoznak mindazon (összefoglaló néven nem intruzív) átalakítások, melyek új forráskóddal egészítik ki a programot, a már meglevő programkód definíciójának megváltoztatása nélkül. Ez jelentheti új konstansok, változók felvételét, vagy új típusok, algoritmusok hozzáadását is. Ez a transzformációs szint már magában, magasabb szintek bevonása nélkül is komoly kifejezőerővel bír. Gondoljunk csak arra, mennyi programozási nyelv biztosít önmagában is valamilyen lehetőséget a programkód viselkedésének, vagy már létező definícióinak megváltoztatására[2]nem intruzív programkód segítségével!

Gyakran használt programtranszformációs technika. Ilyen például konverziós algoritmusok készítése a típusok más formátumra alakításához (lásd 3.4.4), például adattárolás céljából vagy egy másik program által használt reprezentációra. A programnyelvek gyakran nyújtanak rá közvetlen támogatást, ide sorolhatók például a típussal paraméterezett kódok.

[2]Ilyenek például a C nyelv makrói, a C++ névterei, globális operátorai és sablonspecializációi, az AspectJ aspektusai, a C# bővítő metódusai (extension method), vagy akár a Ruby osztálydefiníciói. Mind képesek már létező definíciók bizonyos mértékű megváltoztatására, azonban ezt kizárólag új kód hozzáadásával érik el. Az ilyen eszközök maguk is a metaprogramozás magasabb szintű transzformációinak beépített nyelvi támogatását jelentik.

3.3.5. Kiterjesztés

A kódgenerátorok kifejezőerejét tovább bővíthetjük, ha eltöröljük azt a kikötést, hogy a generált kódrészlet kizárólag elszigetelt, különálló állomány lehet. Ehelyett valamely eddig is létező definíció kibővítése is megengedett, mellyel már a valódi átalakítások, vagyis a tényleges programtranszformációt végző metaprogramok szintjére értünk. Ilyen transzformációkkal legtöbbször osztályokhoz adunk hozzá új adattagokat, metódusokat, a kiegészített típus altípusait képezve ezzel. Képesek lehetünk akár metódusok törzsébe is beszúrni utasításokat.

Több ilyen átalakításokat támogató keretrendszer is létezik, ezek közül elsősorban a Java nyelv (lásd 3.5.3) kiterjesztésére épülő megoldások népszerűek. Ilyen az AspectJ [32] aspektus-átszövő mechanizmusa, vagy az MJ [55] osztályátalakítása (class morphing).

Ennek a transzformációnak egy korlátozott formája a programozási nyelvekben általában osztályok közti leszármazással megvalósított kódújrafelhasználás. Ennek során az eredeti típus nem módosul, a változások által egy új altípust képzünk. Ezen kívül is gyakran láthatunk rá közvetlen nyelvi támogatást, ilyen például a típusok önleírásának (3.2.4) automatikus biztosítása.

3.3.6. Átszervezés

A transzformációk ezen szintjén már a legtöbb korlátozás megszűnik az átalakításokkal kapcsolatban. Egyetlen kikötés, hogy az átalakítások eredményeképp a program kívülről megfigyelhető működése nem változhat meg. Ilyen átalakítások például egy osztály tagjainak átnevezése, kódrészek általánosítása paraméterezésének bővítésével, ismétlődő kódrészletek függvénnyé alakítása vagy a futás során elérhetetlen kódrészletek törlése. Segítségével javítható a kód strukturáltsága vagy teljesítménye, alkalmazásainak leírását lásd 3.4.7 alatt.

A korlátozás jellegéből adódóan kivitelezése rendkívül nehéz. A legtöbb programtranszformációval ellentétben a szintaktika elemzése nem elégséges, szemantikai elemzés szükséges a változások hatásainak meghatározására. Mivel a szemantikus elemzés egy adott nyelvre sem könnyű feladat, a megvalósítások általában nyelvfüggők, és jelentős korlátokkal bírnak.

3.3.7. Teljes átalakítás

A kikötésektől mentes programtranszformációk szintje, lehetővé téve a programód tetszőleges átalakítását. Ilyen átalakítás lehet például, ha a programkódunk által használt egyik (például a grafikus megjelenítésért felelős) könyvtárat lecseréljük egy hasonló funkcionalitású, ám eltérő elvek alapján felépített másik könyvtárra.

Megvalósításához nem feltétlenül szükséges szemantikai elemzés, hiszen itt nincs kikötés a program viselkedésével kapcsolatban. Ezen a szinten már nem is csak a metaprogramozási rendszer megvalósítása okoz gondot. Nehéz az alkalmazás, vagyis a szükséges átalakítások meghatározása is, gondoljunk csak a könyvtár lecserélésének problémájára.

A metaprogramozás más módszerekkel ellentétben még nem rendelkezik kellően megalapozott elméleti háttérrel és gyakorlati tapasztalattal. A szabad átalakításokat támogató rendszerek ritkák, mivel fejlesztésük egy teljes módszertan kidolgozását is igényli, az eltérő, kiforratlan módszerek miatt alkalmazásuk pedig nehézkes. Ennek ellenére megvalósítására már vannak ígéretes próbálkozások. Ilyen például C++ nyelven az OpenC++ [56] nevű metaprogramozási rendszer. A későbbiekben születtek nyelvfüggetlen metaprogramozást támogató rendszerek is, például a Stratego/XT (lásd 3.5.6).

3.4. Alkalmazások

A következőkben a metaprogramozás gyakorlati alkalmazásait tekintjük végig. Ezek száma még viszonylag kevés, mely több okra vezethető vissza. Legfontosabb talán a megalapozott metaprogramozási módszertanok hiánya, valamint ebből következően a metaprogramozási eszközök rendkívüli változatossága és kiforratlansága. Részben erre vezethető vissza, hogy a metaprogramozás alkalmazásával nagyságrendekkel bonyolultabbá válik a programfejlesztés. A szoftveriparban többnyire gyanús "mágiaként" tekintenek rá, valamint még irodalma és oktatása is gyengének mondható, ez pedig súlyosan korlátozza az elterjedését. A tendenciák azonban ígéretesek, hiszen a gyermekbetegségek ellenére a metaprogramozás terjedőben van, az alkalmazások száma növekszik.

3.4.1. Információ kinyerése

A szoftvermetrikák a program kódjának valamilyen mennyiségi vagy minőségi jellemzőjét mérő alkalmazások, eredményük egy egyszerű mérőszám. Leggyakoribb felhasználásuk a kód bonyolultságának mérése, mely alapján következtetni lehet a fejlesztés hatékonyságára, továbbá költségbecslés adható esetleges átalakításokra, valamint a hasonló technológiára épülő rendszerek fejlesztésére. A mérés egyik legfőbb akadálya, hogy a bonyolultság erősen szubjektív fogalom, ezért már az is nehezen határozható meg, pontosan mit is célszerű mérni. A legegyszerűbb, ennek ellenére jó hatékonyságú metrika a programsorok számát méri. Ez a módszer viszont azonnal hatástalanná válik, ha ismeretében a programozók szándékosan tömörítik vagy széthúzzák a program sorait, hiszen az eredmény ezzel tetszőleges irányban befolyásolható. Természetesen léteznek ennel jóval hatékonyabb módszerek is. Egy általános, nyelvfüggetlen bonyolultsági metrikát [67] mutat be.

Más metrikák a feladat bonyolultságát, a hibákat vagy a programozók teljesítményét próbálják megbecsülni. Ezek azonban nem metaprogramokon (sokszor nem is programokon) alapulnak, ezért nem térünk ki rájuk bővebben.

Legtöbbször azonban jóval többet szeretnénk kinyerni a kódból egyszerű mérőszámoknál. Számos alkalmazás állít elő például programozói dokumentációt a forráskódba ágyazott megjegyzések alapján (javadoc, doxygen, ddoc, stb). Egyes eszközök képesek a kód értelmezésével (reverse engineering) grafikus osztályhierarchia-ábrázolást adni, vagy különböző (például UML) formátumú szoftvertervet is készíteni.

3.4.2. Fordítóprogramok

A metaprogramozás gyermekbetegségei alól az egyetlen üdítő kivételt a fordítóprogramok jelentik, mivel sok évtizedes múlttal és erős elméleti háttérrel rendelkeznek. Ennek részletezése nem tartozik a dolgozat témájához, számtalan kiváló minőségű alkotás található a téma irodalmában. Az olvasó számára talán [68] érhető el legkönnyebben.

3.4.3. Speciális (grafikus, beágyazott és többszintű) nyelvek

A szakterület-specifikus (domain specific) nyelvek az általános célú programozási nyelvekkel ellentétben csak egy szűk, speciális alkalmazási terület problémáinak megoldására alkalmasak. Ennek következményeképpen kiemelkedően magas absztrakciós szinten képesek dolgozni. Jellemzően a terület szakértői számára könnyítik meg a számítógépes munkát, tőlük ugyanis nem várható el, hogy szakterületük mellett egyúttal a számítástudomány terén is kiemelkedő tudással bírjanak.

A régebbi, logikai alapú szakértői rendszerek, makró vagy programkönyvtár alapú megoldások mellett megjelentek a más programozási környezetbe ágyazott nyelvek is. Ezeknek közös vonása, hogy egy metaprogram a beágyazott nyelvet egy előzetes lépésben a beágyazó nyelvre fordítja, ezután pedig már a hagyományos fordítási lépés következhet. Ennek előnye, hogy TODO: micsoda?

TODO: példák (MetaCase, Eclipse GEMS?)

A másik kiemelendő alkalmazási területet azok a kódgenerátorok jelentik, melyek lehetővé teszik a grafikus tervezést. Számos eszköz képes UML diagramok alapján a programkód fő vázát elkészíteni, vagy akár fordítva, diagramokra fordítani a kész programkódot. Hasonló elvek alapján a modellvezérelt programépítés (modell driven architecture) már nemcsak a váz, hanem a teljes megvalósítás generálását is lehetővé teszi. Ilyen modellvezérelt elven működik például az AndroMDA [70] nevű fejlesztőkörnyezetet.

TODO: rendesen megirni

A többszintű nyelvek lehetővé teszik egyes programrészek kiértékelésének elhalasztását.

TODO: kesleltetett kiszamitast, tobbszintu nyelveket bepakolni a kategorizalasba, talan a kod kiterjesztesehez kerulne?

TODO tetszőleges számú szint lehetséges

TODO ez a kiterjesztett funkcionális megfelelője a kétszintű fordítási idő, futási idő modellnek

3.4.4. Konverzió, sorosítás, adattárolás

A program adatainak más formátumra alakítása a metaprogramozás egyik leggyakoribb alkalmazása. A konverzió mindig az adatok formátumának valamilyen leírása

alapján történik. Az átalakítás célja változatos lehet, a relációs adatbázisbeli (SQL formátumú) tárolástól kezdve, az olvasható (például XML vagy vizuálisan megjeleníthető) formátumra alakításon át, egészen a különböző formátumokat (például más helyiérték-bájtsorrendet) használó rendszerek közötti kommunikációig terjedően.

Két alapvetően különböző megközelítése van. Egyik esetben a metaprogram az adatok típusának programnyelvi definíciója adott, ennek alapján konvertálunk más formátumra. Ez tipikusan nem intruzív módszer, mivel a célformátum adatsémáját határozza meg a metaprogram a típusleírók alapján (lásd 3.2.4). A másik esetben a célformátum adatsémája adott, ezt szeretnénk a programozási nyelven egyszerűen kezelni. Ekkor a metaprogram általában a sémaleírás alapján programnyelvbeli típusokat generál, ez az adatséma szolgál egyben a típusleíróként is, ennek használhata már intruzív. Ez utóbbi módszerre mutat egy hatékony példát a dolgozat 6. fejezete.

A sorosítás fontos alkalmazási területét az egyre szélesebb körben használt objektum-orientált adatbázisok jelentik, melyek az első megközelítést használják. A Java és .Net alapú megoldások (például a db4o [44]) előnye, hogy teljesen automatikusan, bevatkozás nélkül képesek beilleszteni az objektumok tárolásához szükséges utasításokat a program lefordított bájtkódjának átalakításával (lásd 3.5.3) . Az automatizáltság mellett nagy előnye, hogy a forráskód bonyolultságát sem növeli semmilyen generált adatbáziskezelő programkód, minden változás később, a virtuális gép kódjában történik. Hátránya, hogy egy esetleges hibajelenség esetén a hiba felderítése jóval nehezebb, mivel gépi kód alapján kell dolgozni.

3.4.5. Kommunikáció, szolgáltatások elérése

A számítástechnika alapvető feladata a feladatok elosztása és a munkavégzés párhuzamosítása. Ehhez a számítógépes rendszerek közti kommunikációra, más feldolgozóegységek szolgáltatásainak elérésére van szükség. Lehet szó fizikailag távol levő rendszerek eléréséről, melyek eltérő hardver vagy szoftvereszközökkel rendelkezhetnek, vagy egyazon környezetben futó folyamatok közti kommunikációról is. A kommunikáció megkönnyítésére és automatizálására számtalan technológia született, az elterjedtebb protokollok és architektúrák között említhető az RPC, DCOM, Corba, a Java RMI, DBus vagy a SOAP [71]. A megvalósításra felhasznált eszközök rendkívül változatosak, egyszerű függvénykönyvtáraktól (RPC) kezdve ab-

sztrakt interfészleíráson alapuló kódgeneráláson át (Corba) egészen a szinte teljes automatizálásig (RMI, SOAP) terjednek.

A más nyelven írt programkód elérése sokszor akkor is nehézkes, ha a kód ugyanazon a számítógépen fut. Ennek megoldására ad egy módszert a .Net (lásd 3.5.3) keretrendszer, mely a különböző nyelveket egységes formátumú bájtkódra fordítja, mely egyes nyelvek (pl. a C++) esetében jelentős korlátozásokkal is együtt jár. Egy másik módszer a kommunikációs kód automatikus generálása, így működik például a SWIG [58], mely C és C++ nyelvű programkód elérését támogatja kb. 20 másik nyelven.

3.4.6. Optimalizálás

A teljesítménykritikus alkalmások fejlesztése egy olyan speciális terület, ahol a hatékonyság mindent más szabályt felülír. A máshol jó okkal alkalmazott tervezési elvek, absztrakciók, fejlesztési módszerek itt mit sem érnek, ha teljesítménybeli hátránnyal járnak. A megoldás bonyolultsága ezzel szemben nem elrettentő erő, ha teljesítménynövekedéshez vezet. Emiatt a metaprogramozás ezen a területen elfogadott és elterjedt megoldásnak számít.

Ideális optimalizációs eljárás lehetne, melynek során a program függvényhívásait és kifejezéseit egy előfeldolgozó (meta- avagy supercompiler) helyettesítené be és redukálja minimálisra, a funkcionális nyelvekben alkalmazotthoz hasonló módszerrel. Ismert azonban, hogy tetszőleges program redukálása exponenciális időben megoldható probléma. Az optimalizálást ezért érdemes az emberek által írt programok jellegzetességeit figyelembe véve végezni, mely polinomiális időben is megoldható. Az eredményképp kapott programkód legtöbbször nehezebben érthető az eredetinél, ám bizonyítottan hatékonyabb. Bár ez az optimalizációs eljárás régi, kidolgozott elmélettel [59, 60, 61] bír, megvalósításai még ma is ritkák és kezdetlegesek. Ennek oka bonyolultsága mellett a teljes átalakítást (lásd 3.3.7) támogató eszközök hiánya is.

Az optimalizáció sokkal könnyebb, ha nem az egész programra vonatkozik, vagy nem teljes. Ezt számtalan metaprogramozásra épülő optimalizáló alkalmazás használja ki. Főleg numerikus számításoknál használják gyakran a kifejezéssablonokat (lásd 3.2.5), segítségükkel kiküszöbölhetők azok a feleslegesen létreho-

zott ideiglenes objektumok, melyek az.objektum-orientált módon megvalósított kifejezésfák kiértékelésekor jönnek létre. Ilyen numerikus számításokat végző könyvtár például a Blitz++ [11] vagy az LTL [12]. Sablonokkal felépített kifejezésfát használnak EBNF formájú kifejezések elemzésére is a Boost Spirit [19]könyvtárában, vagy a reguláris kifejezések gyorsítására például a Boost Xpressive könyvtárában [20]. Hasonló feladatokhoz D nyelven már kifejezés-sablonokra sincs szükség, mivel ott a kifejezésfa felépíthető egyszerű sztringek fordítási idejű feldolgozásával [49] is.

3.4.7. Átszervezés

A szoftverek fejlesztése során gyakran előre nem látható, fejlesztés közben felmerülő szempontok alapján kell átszervezni a programkódot. Ilyenkor az átalakítás inkább technikai, mintsem tartalmi, hiszen a program működésének, szemantikájának megtartása mellett történik, a program minőségének javítása céljából. Fontos szempont, hogy az optimalizálással ellentétben a programkódnak ember által könnyen érthetőnek, jól olvashatónak kell maradnia.

A kód átszervezésének (refactoring) leggyakoribb célja a bonyolultság csökkentése és karbantarthatóság javítása, módszereit [50] taglalja. Ennek kézi végrehajtása nehéz, munkaigényes és sokszor gépies folyamat, mely komoly hibaforrást is jelent, automatizálása természetes igényként merül fel. Mára számos fejlesztői környezet támogatja, azonban a megvalósítás nehézsége miatt a támogatás mindig az adott rendszerre szabva és többnyire komoly korlátozásokkal. A feladat jellegéből adódóan nem a fordítóprogramok, hanem a fejlesztői környezetekhez tartozó kódszerkesztők valósítják meg.

A kód átszervezése révén a hatékonyság is növelhető, ha szemantikailag egyenértékű, de kisebb teljesítményköltségű nyelvi elemekre térünk át. Szinte minden fordítóprogram rendelkezik valamilyen optimalizálóval, mely a gépi kódon végzi ezt a feladatot. A supercompilation elnevezésű módszer a forráskód nyelvi elemzésével optimalizál, lásd 3.4.6.

3.4.8. Típusrendszer átalakítása

Sok esetben kényelmesebbé vagy megbízhatóbbá tehetjük a nyelvet, ha változtatunk a típusrendszerén, vagy kiegészítjük a jelenleg meglevőt. Alkalmazásai rendkívül

széles körűek, az objektumok tulajdonjogainak nyomon követésétől kezdve az optimalizáció segítésén át egészen az automatikus helyességbizonyításig terjednek. Megvalósítása nehéz, mivel nem csak programok, hanem a nyelv átalakítását is igényli, az ehhez szükséges teljes átalakítást (lásd 3.3.7) azonban elvétve támogatja metaprogramozási rendszer. Következésképp legtöbbször egy programnyelv kiterjesztéseként, speciális fordítóprogrammal adják meg, ritkábban saját nyelvet definiálnak. Kevésbé bonyolult esetekben metaprogram is használható, a mi szempontunkból ezek a legérdekesebbek.

A meglevő típusrendszer metaprogrammal történő kiterjesztésére később (5. fejezet) láthatunk példát, ahol a strukturális altípusossághoz hasonló viselkedést valósítunk meg vele. Egy másik alkalmazását láthatjuk [22] alatt, mely metaprogram segítségével a C++ nyelv const kikötéséhez hasonló tetszőleges, új korlátozások bevezetését teszi lehetővé. A típusrendszer bővítését teszik lehetővé a Java annotációi és a C# attribútumai is (lásd 3.5.3).

3.5. Metaprogramozási környezetek

Az alábbiakban áttekintjük azokat az elterjedt programozási nyelveket és környezeteket, melyek támogatják a metaprogramozást. Ennek mértéke rendkívül különböző, ahogyan a metaprogramozásra felhasználható eszközök is változatosak.

TODO A metaprogramok futhatnak fordítóprogramok belsejében, mint C++, D, funkcionális nyelveken. Különálló futtató környezetük is lehet, virtuális gépek vagy interpreterek támogatásával...

3.5.1. A C++ nyelv sablonjai

Némi iróniával azt is mondhatjuk, hogy a jelenlegi C++ nyelven (lásd [24, 23]) végzett metaprogramozás véletlenül alakult ki. A nyelv sablon nevű eszközének tervezésekor fel sem merült a fordítóprogram manipulálása, mindössze a generikus programozás támogatására szánták. Ám a nyelv szabványának kialakítása közben rájöttek, hogy a sablonok kifejezőereje jóval nagyobb lett, mint arra eredetileg számítottak.

A terület újnak mondható, hiszen az első algoritmust, mely a C++ fordítópro-

gramban futott, Erwin Unruh készítette [47], és 1994-ben mutatta be. Az algoritmus fordítás közben hibákat generált, melyek a prímszámokat listázták egy megadott korlátig. Ez a definíció értelmében még csak nem is nevezhető igazi metaprogramnak, hiszen garantáltan fordítási hibával ért véget, így a fordításnak biztosan nem lehet használható kimenete. Ám már ebből is kitűnt, hogy a sablonok segítségével algoritmusok készíthetők.

Hamarosan az első valódi metaprogramok is megszülettek. Todd Veldhuizen képes volt az alapvető vezérlési szerkezeteket (fordítási idejű adatok, elágazás, ciklus, függvények) is reprodukálni a fordítóprogram futása közben [8] , ezzel megalkotta a sablonokkal végzett metaprogramozás alapjait. Később megmutatta azt is, hogy a sablonok Turing-teljes eszközt adnak [9] algoritmusok futtatására a fordítóprogramon belül. Megalkotta továbbá a sablonokkal felépített kifejezésfákat (expression template, lásd 3.2.5) is a numerikus számítások hatékonyságának javítására.

Azóta számos munka született a témában, kialakultak a metaprogramozási sablonkönyvtárak. Egyik legegyénibb és leghasznosabb alkotásnak talán Andrei Alexandrescu munkája [13] mondható, mely számos generikus és metaprogramozási újítás mellett egy könnyen használható metaprogram-könyvtárat is adott a típusokból összeállított listák, mint metaadatok kezelésére. Ezt a megvalósításra épít a 5. fejezet megoldása is.

Napjainkban a sablonokkal történő metaprogramozás egy elfogadott megoldássá nőtte ki magát, ám korántsem nevezhető megbízhatónak, stabilnak. A fejlődés azonban ígéretes, a jelenleg legjobb és legszélesebb körben használt metaprogramozási könyvtár, a boost::mpl [17] szabadon hozzáférhető, nyílt forráskóddal rendelkezik. A C++ szabványkönyvtárához hasonló programozási felületet nyújt a fordítási idejű algoritmusok számára, leírását lásd [14].

A sablonokkal való metaprogramozás azonban még így is rendkívül nehézkes. Egyik nagy hátránya, hogy a sablonparaméterekről semmilyen kikötést nem tehetünk, így metaprogramjaink gyengén típusosak. Mivel a fordítóprogram típusellenőrzései nélkül a metaprogram hibái csak futása közben derülnek ki, fejlesztésüknél kizárólag az alapos tesztelésre hagyatkozhatunk. A hibák ráadásul általában nehezen érthető példányosítási hibák hosszú és nehezen olvasható példányosítási útvonalak kíséretében, melyekből többnyire csak közvetve deríthető ki a hiba tényleges oka.

Többek között ezt a sablonok gyenge típusosságának problémáját is javítani kívánja a nyelv hamarosan megjelenő, jelenleg C++0x kódnéven futó szabványa [25] a concept nevű nyelvi eszköz [26] bevezetésével. A Java, C# vagy Eiffel nyelvekkel ellentétben a megoldás nem a programozó által megadott öröklődési szabályokra épül, hanem a fordító által automatikusan eldöntött megfelelési szabályokra. A típusparaméterek megkötései mellett a concept map nevű kiterjesztés segítségével lehetőség lesz a fordító megfelelési szabályainak kiegészítésére is.

A C++ nyelvű metaprogramozás másik problémája a módszer kiforratlansága. Mivel a sablonokat eredetileg nem metaprogramozásra tervezték, csak mellékhatásokkal dolgozik direkt nyelvi támogatás helyett, Bár a sablonok kifejezőereje funkcionálisan elegendő tetszőleges algoritmus megalkotására, a hatékony fejlesztéshez ennél jóval többre volna szükség, ahogyan a mai programozási módszerek is távol állnak már a Turing-gépektől. A kifejezőerőt szintén jelentősen megkönnyíti majd az új nyelvi szabvány, többek közt a sablonparaméterek változó hosszúságú listáinak kezelésével, egy jól felépített módszertan kialakulásához azonban még rengeteg további kutatás szükséges.

A nyelv további megoldatlan problémája a metaprogramok rendkívül bonyolult és nehézkes fejlesztése is. Ez legfőképp a szabadon elérhető, jó minőségű nyelvi elemzők, ezáltal az erre a célra készített segédprogramok és programozási környezetek hiányából fakad. Bár kutatások folynak a témában (például [6]), nemcsak megfelelő metaprogram debugger nem létezik a futás nyomon követésére, de nincs semmilyen szabványos eszköz legalább üzenetek kiíratására sem. A metaprogramok alkalmazásának akadálya ezen kívül a fordítóprogramok jelenlegi kapacitása is, hiszen a legtöbb fordítóprogram sajnos exponenciális költségnövekedéssel reagál a metaprogram bonyolultságának lineáris növekedésére.

A problémák egy részét igyekszik orvosolni a C++ nyelv új, jelenleg C++0x [25] kódnéven formálódó szabványa. Azonban a C++ nyelv sablonjai még az újításokkal együtt is csak részhalmazát adják a D nyelv (3.5.2) metaprogramozási eszközeinek.

3.5.2. A D nyelv sablonjai

Amint azt már a neve is jelzi, a D programozási nyelvet (lásd [48]) alapvetően a C++ nyelvből kiindulva tervezték. A C++ nyelv C kompatibilitási alapelvével szakítva az

alapoktól kezdve teljesen újratervezték a nyelvet, immár okulva a C++ korlátaiból és hibáiból. Mindez még nagyon friss technológia, a nyelv és fordítóprogramjai még messze nem mondhatók kiforrottnak, inkább kísérleti stádiumban vannak. Továbbra is sok változás történik a specifikációban, a fordítóprogramokban is folyamatosan javítják a hibákat. Ennek ellenére napjainkra már komolyan feltörekvő nyelv vált belőle, kiterjedt és lelkes programozói közösséggel. Számos modern eszközt építettek a nyelvbe (szemétgyűjtés, *delegate*, stb), azonban a hangsúly most nem ezen van.

A nyelv tervezésekor már a metaprogramozást is eleve fontos szempontnak tartották. A C++-ban legfeljebb csak mellékhatások segítségével kifejezhető konstrukciók többségére a D már közvetlen nyelvi támogatást biztosít, de számos jelentős újdonságot is tartalmaz. Ez nagyságrenddekkel megnöveli a nyelv kifejezőerejét és csökkenti a metaprogramok bonyolultságát. A D nyelv biztosítja a típusok futási idejű önleírását, ami nem virtuális gép vagy interpreter segítségével futtatott programok esetében korántsem magától értetődő. Az önleírást a beépített *object* osztály közvetlenül támogatja, ez minden más osztály implicit ősosztálya. Lehetővé teszi továbbá a függvényparaméterek lusta kiértékelését, mellyel a többszintű programozáshoz (3.4.3) ad támogatást.

A futási idejű eszközöknél a nyelv jóval erősebbekkel is rendelkezik. Egyik ilyen fontos eszköz a fordítóprogram fordítási idejű kódkiértékelési képessége, mely teljesen automatizáltan működik. Ez nem csak az aritmetikai és sztringeken végzett műveletekre vonatkozik, a fordítóprogram egyszerűbb függvények esetében a függvényhívások eredményét is képes fordítás közben kiértékelni és az eredményt behelyettesíteni[3]. Segítségével nem csak a futási idejű hatékonyság növelhető, hanem egy egyszerű függvény is azonnal metaprogramként futtatható. Az automatizáltság az egyszerűbb esetekben megtakaríthatja azt a jelentős erőfeszítést, mely a legtöbb környezetben a metaprogramok írásával jár. Amennyiben azonban bonyolultabb programelemekkel dolgozunk, már D-ben is eleve metaprogramokhoz tervezett eszközökhöz kell nyúlnunk. Az automatizálás erejét azonban az is mutatja, hogy egy Boost Xpressive (lásd [20] vagy 3.4.6) könyvtárhoz hasonló, reguláris kifejezéseket fordítási időben kiértékelő algoritmus [49] képes kifejezés-sablonok (lásd 3.2.5) használata nélkül, kizárólag sztringekkel leírt reguláris kifejezések alapján

[3]Ennek természetesen szigorú feltételei vannak a függvény paramétereire és az általa felhasználható műveletekre nézve.

működni.

A D nyelv bővelkedik a fordítási idejű döntéseket segítő eszközökben, számos hagyományos konstrukció megtalálható a metaprogramok szintjén is. Ilyen a fordítási idejű nyomkövetés (*pragma msg*, *static assert*), elágazás (*static if*), lista adatszerkezet (type tuple), listabejárás fejelem és maradék elvén (variadic template arguments), vagy a kifejezések fordítási idejű vizsgálata (*is* expression). A C++-hoz képest az általánosítások és bővítések miatt jelentősen nőtt a sablonok kifejezőereje is. Lehetőség van továbbá sablonpéldányként vagy fordítási időben kiértékelhető sztringként megadott forráskódot beszúrni a forráskódba (*mixin*).

A dolgozat írása közben megjelent a nyelv legújabb, 2.0 verziója is. Ebben már a típusok fordítási idejű teljes önleírása [4] (*trait*) és a sablonparaméterek megkötései (template constraint) is bemutatkoznak. Ezek az újítások a 4. fejezetben és [1] alatt korábban leírt C++ nyelvű megoldáshoz hasonlóan valósultak meg D nyelven.

3.5.3. Virtuális gépek (Java és C#)

A Java [73] és a C#/.Net [72] hasonló alapelveik és eszközeik miatt együtt kerülnek tárgyalásra. Mindkettő erősen típusos, de virtuális gépi kódra fordított nyelv. A programot futtató virtuális gép használatával az erősen típusos nyelvek biztonságát ötvözhetjük a szkriptnyelvek rugalmasságával, mellyel biztosítható a futtatókörnyezet megváltoztathatósága, így a metaprogramozás támogatása is.

A virtuális gépek egyik fontos előnye a dinamikus osztálybetöltés támogatása. Ez lehetővé teszi új osztályok biztonságos és automatizált hozzáadását egy olyan alkalmazáshoz, mely fordításakor ezek az osztályok még ismeretlenek voltak. Ezáltal válik lehetővé, hogy dinamikusan generált kódot is futtathassunk a program leállítása nélkül. Egy másik fontos előnyt jelent a típusok futási idejű önleírása (3.2.4), mely alapján hozzáférhetünk más osztályok adattagjaihoz és metódusaihoz.

A Java és C# nyelvek támogatják a sablonokhoz hasonló, de kisebb kifejezőerejű generic-eket (lásd 2.2.1).

Az attribútum-nyelvtanok [68] elmélete ihlette a C# attribútum és a Java nyelv annotáció nevű elemeit. Ezek lehetőséget adnak nemcsak a típusok metaadatainak közvetlen specifikálására, de ezen metaadatokat elemző és feldolgozó kód megadására

[4] Ez a futási idejű önleírásnál jóval erősebb eszköz, lásd 3.2.

is, ezzel ideális eszközt nyújtanak a nyelv típusrendszerének átalakítására (lásd 3.4.8). A metaadatok feldolgozása során természetesen programkód is generálható, ennek segítségével valósul meg például a sorosítás és adattárolás (3.4.4), valamint a távoli szolgáltatások automatizált elérése (3.4.5) ezen nyelvek alapkönyvtáraiban.

A virtuális gépek a hardvereszközöknél jóval magasabb szintű, saját formátumú gépi kódot használnak, melyre számos programnyelvet fordíthatunk. Ezáltal lehetővé válik, hogy más nyelvű elemeket a típusrendszerünkbe illesszünk, például más nyelven írt tetszőleges osztályt példányosítsunk és használjunk. A .Net keretrendszer létrehozásánál a nyelvek közti átjárhatóság eleve fontos szempont volt, így jelenleg már több tucat nyelv képes együttműködni az általa használt CLR (Common Language Runtime) nevű kód segítségével. A Java virtuális gépére is több nyelv fordítható, valamint léteznek az együttműködést támogató többnyelvű rendszerek is, például a JPython és JRuby.

Mivel a virtuális gépi kód tartalmazza a típusok információit, lehetséges a már lefordított bájtkód automatizált átalakítása (bytecode instrumentation). Ezt az újabb virtuális gépek már közvetlenül is támogatják, lehetővé téve a metaprogramozás igen magas szintjét. Segítségükkel válik lehetővé adatbáziskezelőkben az objektumok automatikus tárolása (például [44]), a teljesítmény mérésének (profiling) automatizálása, vagy a szerződés alapú programozás (design by contract) támogatása (lásd [74]).

3.5.4. Ruby és más szkriptnyelvek

A szkriptnyelvek közül a Ruby metaprogramozási lehetőségeit tekintjük át, mivel ezen nyelv rendelkezik a legerősebb eszközökkel. A leírt alapelvek nagy része azonban más nyelvekre (például Python, Perl, stb) is érvényes.

A szkriptnyelvek a gyenge típusosság biztonsági hátrányaiért cserébe rendkívüli rugalmasságot nyújtanak, a virtuális gépeknél (3.5.3) említett metaprogramozási eszközök mellett számos további áll rendelkezésünkre. A környezet legtöbb eleme vizsgálható a program futása közben, az objektumok teljes körű önleírása mellett listázható például a futás közben létező összes objektum is, vagy lekérdezhető az osztályhierarchia és a hívási verem is. Számos elem nemcsak vizsgálható, de meg is változtatható. A változókat, függvényeket átnevezhetjük és újat definiálhatunk a

helyükre, ennek segítségével tetszőleges kód módosítható. Ezzel többek között könnyen használhatunk az aspektus-orientált (2.2.2) programozáshoz hasonló stílust, mely a kód újrafordítása nélkül, menet közben használható. Tetszőleges kódot hozzáadhatunk egy objektumhoz vagy egy teljes osztályokhoz (ezáltal az összes objektumához), sőt, értesítést kérhetünk, ha egy osztályhoz például új metódust definiáltak, vagy leszármaztak belőle.

A szkripnyelvek erős szövegfeldolgozási képességekkel rendelkeznek, valamint tetszőleges karakterlánc programkódként való kiértékelését is lehetővé teszik futás közben. Ez lehetővé teszi akár teljes programkönyvtárak futás közbeni generálását is. A dinamikus kód futtatását nagyban megkönnyíti, hogy szkriptnyelveken nincs szükség kölön lépésre a programkód fordításához, mely egyébként szükséges lenne.

Ruby nyelven a műveletek (például függvényhívások) objektumok közti egyszerű üzenetek, melyet a fogadó fél értelmez. A hívás előtt lekérdezhetjük, hogy a hívott fél elfogadja-e az üzenetet. Ha mégis elküldjük, és az értelmezés sikertelen (például nincs ilyen nevű művelet), kivétel váltódik ki. Ez a kivétel azonban elfogható, és tetszőleges módon, akár a kód átdefiniálásával is válaszolhatunk rá. A felsorolt eszközök segítségével lehetővé válik az is, hogy menet közben akkor generáljuk az objektumok eljárásainak megvalósítását, ha azokra valóban szükség van, ezzel jelentős mennyiségű memória takarítható meg.

Mivel a bármilyen programkód betölthető vagy menet közben átdefiniálható, ez a gyakorlatban akár a program teljes átalakítását is jelenti (3.3.7), mely a legmagasabb szintű metaprogramozási transzformáció. Amint látható, a szkriptnyelvek eszközeivel olyan összetett metaprogramozási feladatok is megoldhatók, melyekre a fordított nyelvek még nem képesek.

3.5.5. MetaML és MetaOCaml

Az OCaml nevű nyelv az ML nevű funkcionális nyelv objektum-orientált kiterjesztése. E nyelvek további kiterjesztései a MetaML [42] és a MetaOcaml [43] programozási nyelvek, melyek metaprogramozási eszközök használatát teszik lehetővé.

Többszintű nyelvek (lásd 3.4.3), ezt mindkét nyelvben három nyelvi konstrukció teszi lehetővé. A késleltetés (brackets) megjelöli a halasztott kiértékelésű kódrészleteket. A hivatkozás (escape) anélkül képes ezek értékére hivatkozni, hogy

kiértékelést kényszerítene ki, ezáltal további kifejezéseket építhetünk fel belőlük. Végül külön kiértékelés (run) művelet segítségével utasíthatjuk a programot, hogy generáljon kódot, mely kiszámítja a késleltetett kifejezések értékét.

A funkcionális nyelvek többszintű kiértékelésének leggyakoribb célja a program futási költségének javítása. Ez abból adódik, hogy a költség kiszámítási módja korántsem egyértelmű, sokszor csak a késleltetett programrészek kiértékelésének ideje számit. Ilyenkor célszerű minden lehetséges programrészt az első fázisban végrehajtani, hiszen ezek költsége ezen mérték szerint nulla.

3.5.6. Stratego/XT

A Stratego/XT [57] nevű metaprogramozási rendszer a programkifejezések átírását lehetővé tevő Stratego nyelv és az átírást futtató, valamint egyéb kisegítő funkcionalitásokkal rendelkező XT nevű keretrendszer együttese.

A rendszer működésének lényege, hogy tetszőleges nyelven írt forráskódot egy elemzőprogram (parser) egy belső reprezentációra (Annotated Term Format) alakít, melyen tetszőleges átírási lépések hajthatók végre, végul egy formázóprogram szöveges forráskódra alakítja vissza. Az elemzés és formázás nyelvfüggő, formátumleírás alapján működik. Néhány nyelvet már most is támogat (Java, C++, stb), és tetszőlegesen bővíthető.

Az átalakítások definíciója két részre bomlik: megkülönböztetünk átírási szabályokat és stratégiákat, mely a szabályok alkalmazásának hatókörét írja le. Az átírásokat a belső ábrázolási formátum segítségével definiálhatjuk, a reguláris kifejezésekhez hasonlóan a forráskód részkifejezéseinek illesztésével és cseréjével. Ez a nyelvfüggetlen módszer azonban sokszor nehezen olvasható és terjedelmes, ezért lehetőség van az átírások forrásnyelvű leírására is. Ezt a Metaborg nevű alrendszer teszi lehetővé, melyben leírhatjuk az átírások definíciójában felhasznált nyelvi elemek belső reprezentáció szerinti jelentését, melyből az illeszkedés és csere szabályai levezethetők.

4. fejezet

Típusok tulajdonságainak vizsgálata

A metaprogramok működése a bemenetként kapott programról kinyert metaadatokon és a programtranszformációkon alapul. A metaadatok (lásd **??**) közvetlen elérése azonban a legtöbb programozási környezetben rendkívül rosszul támogatott. Ez elmondható a legtöbb objektum-orientált nyelvről, így a C++ esetében is igaz.

Bár a C++ sablonjai jó kifejezőerővel bírnak programtranszformációk leírására, a metaadatok elérésének nehézsége gyenge pontját jelenti a C++ nyelvű metaprogramozásnak. Az egyetlen közvetlenül támogatott eszköz a *sizeof* operátor, mely egy típus bájtokban kifejezett méretét adja meg. Bár a fordítóprogramon belül nyilvánvalóan minden információ rendelkezésre áll, közvetlenül semmilyen más metaadat nem érhető el. A sablonok kifejezőerejét mutatja azonban az is, hogy néhány tulajdonságuk ügyes felhasználásával a típusok metaadatainak egy része mellékhatások kihasználásával mégis hozzáférhető. Erre alapul a fejezetben bemutatott metaprogramozási modell is.

További gondot okoz a vizsgálatok által kinyert eredmények felhasználása a metaprogramozás során. Sajnos a nyelv meglevő eszközeinek felhasználásával nehézkes még fordítási időben kijelölni egy adott eredményhez tartozó kódrészletet, különösen akkor, ha az külön definíciókat vagy más vizsgálati eredmények esetén érvénytelen kódot tartalmaz.

A fejezetben a probléma részletesebb bemutatása (4.1) után felépítek egy C++ nyelvű programkönyvtárat (4.2), mely lehetővé teszi a típusok egyes metaadatainak kinyerését. Mivel a könyvtár felépítése során kizárólag szabványos nyelvi elemeket

használok fel, a típusok teljes fordítási idejű leírásához (lásd 3.2.4) ez a módszer még nem elég erős[1]. Lehetséges lesz azonban a típusok egyes elemi tulajdonságainak vizsgálata (3.2.3). Bemutatok továbbá egy javasolt módszert, amely minimális nyelvi kiterjesztés segítségével jelentősen egyszerűsíti kódrészletek kiválasztását az elemi vizsgálatok eredményének alapján, megkönnyítve ezzel az eredmények felhasználását.

4.1. A probléma leírása

A C++ nyelv sablonjainak kifejezőerejéhez és alkalmazhatóságához nagyban hozzájárul a sablonok példányosításánál használt lusta stratégia: minden osztálysablonnak csak azok a tagjai példányosulnak, melyekre más kódrészlet hivatkozik. Mivel minden különböző paraméterrel rendelkező sablonpéldányt külön forráskódként kezel a fordítóprogram, ezek számának növekedésével az előálló programkód mérete folyamatosan nő. A sablonpéldányok nagy számából adódó méretrobbanást a lusta példányosítás jórészt orvosolni tudja.

A lusta példányosítás a sablonok felhasználhatóságát is jelentősen bővíti: példányosításnál szükségtelen a sablon teljes egészét lefordítani, kizárólag azokra a részekre van szükség, melyeket valóban használunk. Így előfordulhat, hogy adott paraméterekkel egy sablon egésze ugyan fordítási hibát adna, ám mivel mi csak egyes részeire hivatkozunk, ezért ezeket a hibákat elkerüljük, és a sablon többi részét gond nélkül használhatjuk. Gond nélkül használhatjuk például az alapkönyvtár *std::list* sablonját, mely *sort()* nevű, összehasonlítás alapú rendezést végző tagfüggvénnyel rendelkezik. A lista olyan elemekkel is működni fog, melyekre nincs összehasonlító operátor, mindaddig, amíg nem hívjuk meg ezt a rendezőfüggvényt. A lusta példányosítás teszi lehetővé, hogy sablonok segítségével kényelmes elágazást valósítsunk meg a C++ nyelvű metaprogramozáshoz: ügyes alkalmazásával elkerülhető az elágazás ki nem választott ágának példányosítása, így az általa tartalmazott kifejezéseknek elég csak a hozzá tartozó feltétel teljesülésekor esetén szemantikailag is helyesnek lennie (lásd a Boost metaprogram könyvtárának [17] elágazásait).

Ez a rugalmasság és kifejezőerő azonban jelentősen gyengíti a nyelv bizton-

[1] A fejezetben bemutatottak további kiterjesztésével lehetséges volna előállítani a típusok teljes fordítási idejű leírását is, ez azonban már túlhaladja a dolgozat kereteit és lehetőségeit.

ságát. Mivel a fordítóprogram csak példányosításkor végez teljes értékű szemantikai ellenőrzést, a hiba általában nem a sablon definíciójában, hanem jóval később, használatakor keletkezik. Ha például az említett *sort* függvény megvalósítása szemantikai hibát tartalmaz, az egészen a függvény első meghívásáig nem derül ki. Ez különösen programkönyvtárak fejlesztése esetén probléma, hiszen egy megbízható könyvtárnak garantálnia kell, hogy tesztjei a teljes könyvtár minden sorát lefedik. Ellenkező esetben nemcsak hibás működésű, hanem szintaktikailag helytelen kódot is tartalmazhat.

A sablonkönyvtárak típusbiztonságának növeléséhez tehát szükség volna arra, hogy megszoríthassuk a sablonparamétereket, erre azonban a nyelv semmilyen lehetőséget nem ad. Ez a nyelv megalkotásánál egy tudatosan választott kompromisszum volt (lásd [23]). A C++ nyelv megalkotója az ilyen megszorításokat kimondottan ellenezte, mivel lehetetlenné tennék a sablonok fentebb bemutatott nagymértékű rugalmasságát. Erre természetesen a megszorítások opcionális használata jelent megoldást.

A megszorítások hiányának orvoslására több speciális megoldás is született, ezeket 4.3.2 alatt ismertetem. Ezeknél azonban jóval általánosabb, metaprogramozáson alapuló megoldás is adható, mely a programkód önvizsgálatának segítségével képes megszorításokat kifejezni. Ennek részleteit 4.2 mutatja be.

4.2. Javasolt megoldás

Egy általános célú metaprogramozási rendszer könnyen megoldást adhatna megszorítások kifejezésére is. Ezért a 3. fejezetben felvázolt egyszerű metaprogramozási modell alapján felépítünk egy általános rendszert, mely a programkód vizsgálatára, vagyis a típusok tulajdonságainak kinyerésére alapul. A rendszer három alapvető alkotórésze a következő:

1. Alapvető, elemi típusinformációk kinyerése. Az elemi vizsgálatok segítségével egyszerű eldöntendő kérdéseket tehetünk fel a fordítóprogramnak tetszőleges típussal kapcsolatban. A kérdésekre logikai értéket kapunk eredményül. Mivel a vizsgálatok megfelelnek az elsőrendű logikában használt predikátum fogalmának, ezért elemi vizsgálatainkat a továbbiakban predikátumoknak is nevezzük.

2. Az elemi vizsgálatok eredményeinek kompozíciója, összetett kifejezések felépítése. Itt szintén célszerű az elsőrendű logika alapján dolgoznunk: a predikátumokból logikai műveletekkel építhetünk kifejezéseket, mi erre a C++ nyelv logikai operátorait fogjuk felhasználni.

3. Az összetett kifejezések eredményének felhasználása. Ilyen lehet a fordítás megállítása vagy egy fordítási idejű elágazás az eredmény alapján.

Mielőtt ezeket részletesen ismertetnénk, szükségünk lesz egy technikai jellegű kitérőre.

4.2.1. A C++ metaprogramozás felhasznált eszközei

Az elemi vizsgálatok megvalósításának alapját a C++ sablonjainak különleges alkalmazása teszi lehetővé. Mivel ezek várhatóan a legtöbb olvasó számára ismeretlenek, a megoldás technikai részleteinek ismertetése előtt szükség van ezen módszerek bemutatására. A módszerek bemutatása után már könnyen megérthetjük az elemi vizsgálatok működési elvét is.

4.2.1.1. Sablonparaméterek típusának kikövetkeztetése

A C++ nyelv lehetővé teszi, hogy függvénysablonok használatakor egyes esetekben elhagyhassuk a sablon típusparamétereinek kiírását. Ez olyankor lehetséges, ha a típusparaméterek mind szerepelnek a függvény szignatúrájában, és a konkrét paraméterek alapján a fordítóprogram képes ezeket kikövetkeztetni[2]. Például:

```
template <class T>
T twice(T t) { return t + t; }

twice(2); // --- Egyenértékû a twice<int>(2) hívással
```

Az utolsó sorban látható hívásnál nem kell kiírnunk az *int* típusparamétert. Mivel a megadott paraméter egész szám és a sablonbeli típusa T, ebből kikövetkeztethető, hogy a T típusparaméter *int*.

[2]A szabvány ezt a viselkedést csak függvénysablonok esetén írja elő, osztályok esetében erre csak a C++ új szabványának [25] bevezetésével lesz majd lehetőség.

A típusok automatikus kikövetkeztetése azért is rendkívül kényelmes, mert segítségével a sablonfüggvények a többi függvénnyel azonos módon hívhatók, és a túlterhelési szabályok lehetővé teszik a két függvénytípus közötti teljeskörű átjárhatóságot.

4.2.1.2. A SFINAE szabály

A sablonok használatának egyszerűsítésére a C++ nyelv engedékeny szabályokkal rendelkezik. A SFINAE (Substitution Failure is not an Error), ritkábban kétmenetes keresés (two phase lookup) nevű szabály azt írja elő, hogy túlterhelt sablonnevek esetén a sablonparaméterek sikertelen behelyettesítése egy adott definícióba önmagában még nem jelent hibát. Hiba csak akkor lép fel, ha a túlterhelt név összes lehetséges változatának alkalmazása egyaránt sikertelen volt. Lássunk erre egy példát:

```
template <class T>
typename T::iterator func(T t) { return t.begin(); }

void func(...) {}       // --- Minden paramétert elfogad

func( list<int>() );   // --- Az elsõ függvényt hívja meg
func(2);               // --- Nincs int::iterator, a másodikat hívja
```

A második függvény az ellipse nevű eszközt használja: a függvény a „...” jelöléssel tetszőleges számú és típusú paramétert elfogad. Az első függvény visszatérési értéke pedig a sablonparamétertől függ, mivel a sablonparaméter beágyazott *iterator* típusa. A típusok legtöbbje (például az *int* típus) nem tartalmaz ilyen definíciót, a szabványos könyvtár tároló osztályai (például a lista) viszont igen.

Mivel az ellipse precedenciája kisebb, az első függvényhívás a sablonparaméterek automatikus kikövetkeztetésének segítségével az első változatot hívja meg. A második hívás is először az első változatot próbálja példányosítani, ám egy egész számmal ez nem sikerülhet. A SFINAE szabály miatt azonban a fordítás nem áll meg hibával, hanem tovább keres, és meg is találja a második változatot, melyet már sikeresen meg is tud hívni. Látható, hogy a típuskikövetkeztetés segítségével a függvényhívások teljesen azonos módon működnek hagyományos és sablonfüggvények esetében, és a megfelelő változat kiválasztásában sincs látható megkülönböztetés[3].

[3]Egyetlen különbség, hogy a sablonfüggvények precedenciája az ellipse kivételével mindig kisebb,

4.2.1.3. Az *enable_if* sablon

Az *enable_if* osztálysablon a Boost könyvtár [15] része, a segítségével megvalósított döntési módszer részletes leírása megtalálható [46] alatt. A cél röviden egy fordítás közben kiértékelt egyparaméteres metafüggvényt megvalósítása, mely hamis logikai értékre üres halmazt ad eredményül, igaz értékre pedig az identitásfüggvényt. Mivel fordítási időben kiértékelt függvényeink és rájuk hivatkozó mutatóink C++ nyelven nincsenek, ezt a viselkedést kerülőúton kell megoldanunk. Az *enable_if* sablon megvalósításában két paramétert használ, első a logikai érték, a második egy típus, melyet igaz esetben eredményül ad. Programkódjának lényege a következő:

```
// --- Definíció
template <bool, class> struct enable_if {};
template <class T> struct enable_if<true,T> { typedef T Result; };

// --- Használat
enable_if<sizeof(int) == 4, int>::Result size;
```

Az első definíció a sablon általános alakja, mely nem tesz semmit, törzse üres. A második definíció az előző definíció részleges specializációja. Azokra az esetekre vonatkozik, melyekben a logikai érték igaz, a típusparaméter azonban továbbra is kötetlen. Törzsében *Result* néven hivatkozik a második paraméterére, ezzel definiálva az eredményt.

Az *enable_if* bárhol leírható, ahol típus szerepelhet, ám ha a logikai érték hamis, példányosítása a *Result* hivatkozás miatt a definíciójának hiányában sikertelen, ami legtöbbször fordítási hibához vezet. Fent látható egy példa is használatára: egy *int* típusú változót definiálunk, fordítási hibával jelezve, ha annak mérete az adott fordítóban feltételezésünkkel szemben mégsem 4 bájt. Ebben a formájában egy fordítási idejű *assert* kifejezést valósít meg, melyre kevéssé alkamas, [13] alatt erre egy jobb megoldás olvasható. A későbbiekben azonban a SFINAE segítségével egy másik függvényváltozatot választunk helyette, elkerülve ezzel a fordítási hibát.

így a fordító alkalmas illeszkedés esetén a sablon nélküli változatot részesíti előnyben.

4.2.1.4. Tulajdonságok meglétének eldöntése

A technikai háttér, melynek segítségével a fordítóprogram képes kiválasztani a megfelelő függvényváltozatot különböző tulajdonságok alapján, már adott. Mi a választás eredményéről azonban legfeljebb csak a program futása közben értesülünk, így ez a probléma még megoldásra vár.

A megoldásban a *sizeof* operátor lesz majd segítségünkre. Alkalmazásához először is két különböző méretű típust kell definiálnunk a hamis és igaz logikai értékek ábrázolására (a technika eredeti leírását [13] alatt):

```
typedef char No;                          // --- Hamis értékû típus
typedef struct { char dummy[2]; } Yes;    // --- Igaz értékû típus
```

Mivel a két típus mérete különböző, a *sizeof* operátor segítségével könnyen meg tudjuk különböztetni őket. Ezeket a típusokat függvények visszatérési értékeként használva el tudjuk dönteni, hogy két függvényváltozat közül melyik hívódott meg. Ez a következőképp törhénhet:

```
// --- A feltétel teljesülése esetén végrehajtódó ág
template <class T>
typename enable_if<sizeof(T) == 4, Yes>::Result sizeIsFour(T);

// --- Hibát megakadályozó mentõág, ha nem teljesül
No sizeIsFour(...);

// --- Példa a vizsgálat végrehajtására
const bool result = sizeof( sizeIsFour(0) ) == sizeof(Yes);
```

Előző példánknál maradva azt az eldöntendő kérdést akarjuk megválaszolni, hogy egy típus mérete 4 bájt-e. Az első függvény tetszőleges típusparaméterrel hívható, ilyenkor a függvényparamétere miatt a típusparaméter automatikusan kikövetkeztethető. A tulajdonság vizsgálatát a visszatérési értékbe kódoltuk az *enable_if* sablon segítségével[4]: ha a paraméter típus mérete nem 4 bájt, akkor a *Result* definiálatlan, így nem lehet példányosítani a függvényt. Ha azonban 4 bájt, akkor a függvény visszatérési értéke *Yes* típusú lesz. A második, No típusú visszatérési értékkel és kisebb precedenciával rendelkező függvényváltozat mentőágként

[4]A definícióban szereplő *typename* kulcsszó a programozó által a fordító számára kötelezően adandó segítség. A fordítóprogram ismeretlen típus esetén csak ennek segítségével képes megkülönböztetni a beágyazott típusokat osztályok adat- és függvénytagjaitól.

viselkedik, mivel tetszőleges paramétert elfogad. Ha az első függvény példányosítása sikertelen is, a második változat mindig példányosítható, megakadályozva ezzel a fordítási hibát.

Vegyük észre, hogy egyik függvényváltozatnak sincs törzse[5]. A függvénytörzs valójában teljesen felesleges, mivel a visszatérési érték típusának eldöntéséhez nem kell végrehajtanunk a függvényt, hiszen a típus kizárólag a deklarációk vizsgálata alapján is megállapítható, a *sizeof* operátor pedig pontosan így működik. Ezzel már könnyen megérthető az utolsó sorban végrehajtott vizsgálat működése. A függvénynek egy egész értéket átadva a fordítóprogram kikövetkezteti az *int* típust, az *enable_if* feltétele alapján kiválasztja a megfelelő függvényváltozatot, majd a *sizeof* operátorral meghatározza annak méretét. Ha ez megegyezik a *Yes* típus méretével, akkor az eredmény igaz. Látható, hogy az eredmény fordítási időben kiértékelt konstans.

Az eredmény vizsgálatának rövidítésére érdemes egy kisegítő típust és makrót definiálni:

```
// --- Kisegítõ típus definíciója
template <int> struct Evaluate;
template <> struct Evaluate<sizeof(No)>  { enum { Result = 0 }; };
template <> struct Evaluate<sizeof(Yes)> { enum { Result = 1 }; };

// --- Kisegítõ makró definíciója
#define EVALUATE(PARAM) Evaluate<sizeof(PARAM)>::Result

// --- Példa a fentiek használatára
const bool result = EVALUATE( SizeIsFour(0) );
```

A fentiekben először a *No* és *Yes* típusok méretéhez rendelünk hozzá egy (binárisan ábrázolt logikai) egész értéket. Ehhez először egy definíció nélküli, ezáltal nem példányosítható általános deklarációt adunk meg, majd ezt specializáljuk a két típusra, hozzájuk a 0 és 1 értékeket rendelve. Ezek a C++ implicit konverziós szabályainak segítségével közvetlenül logikai értékké alakíthatók, hiszen a hamis

[5]Ez abból a szempontból is szerencsés, hogy a változó számú függvényparaméter átvétele (ellipse) súlyos biztonsági problémákat hagyhat maga után a programban, ezért lehetőség szerint kerülik a használatát. Használatánál a paraméterek átvétele a függvénytörzsben történik, itt azonban ilyesmiről szó sincs. Mivel kizárólag a megfelelő függvényváltozat kiválasztásához használjuk, így alkalmazásával nem hagyunk biztonsági rést.

értéknek megfelel a 0, minden más egész szám, így az 1 is igaz értéket ad. Ez a definíció lehetővé teszi, hogy a vizsgálat végrehajtásakor ne kelljen minden alkalommal az eredményt a *Yes* típus méretéhez hasonlítani.

A makró abban segít, hogy a ne kelljen kiírnunk a *sizeof* operátor hívását és ne kelljen külön hivatkoznunk a *Result* tagra sem. A vizsgálat formája ezáltal jelentősen egyszerűsödött és könnyebben olvasható. Használatának végső alakját az utolsó sorban láthatjuk.

4.2.2. Predikátumok megvalósítása

Most már minden technikai eszköz rendelkezésünkre áll ahhoz, hogy képesek legyünk bizonyos elemi vizsgálatok végrehajtására. Ezek a vizsgálatok a következők:

- Egyszerű típusmegszorítások, többek között:
  - Típusmódosítók (const, volatile) megléte
  - Kategorizálás (lebegőpontos, skalár, stb. típus-e)
- Adott nevű beágyazott definíció létezése:
  - Függő típusokra (*typedef*-ek vagy beágyazott osztálydefiníciók)
  - Tagokra (adattagok és tagfüggvények)
- Beágyazott definíció típusának eldöntése:
  - Függő típusokra
  - Adattagokra (osztály- és példányszinten egyaránt)
  - Tagfüggvényekre (osztály- és példányszinten egyaránt)

Az első pontban szereplő egyszerű típusmegszorításokra a Boost Type Traits könyvtár (lásd 4.3.2.2) kiváló minőségű megvalósítást és dokumentációt biztosít, ezért ezekkel a dolgozat már nem foglalkozik. A többi pontra azonban még nem ismert megfelelő megoldás, ezért a továbbiakban rájuk koncentrálunk.

4.2.2.1. Adattagok típusának vizsgálata

Bár a fenti felsorolás sorrendje más, a megvalósítás logikája miatt érdemes ezen változtatni. Legegyszerűbb megvalósítással a létező nevek típusvizsgálata, ezen belül is az adattagok vizsgálata bír. A technikai részletek bemutatása alapján az alábbi programkód már magában, különösebb magyarázat nélkül is érthető lehet:

```
template <class VariableType>
struct Member
{
    // --- Osztályváltozó típusának vizsgálata
    static Yes Static(VariableType*);

    // --- Mentőág osztályváltozókra
    static No Static(...);

    // --- Példányváltozó típusának vizsgálata
    template <class Class>
    static Yes NonStatic(VariableType Class::*);

    // --- Mentőágak példányváltozókra
    static No NonStatic(...);
    template <class> No NonStatic(...);
};

// --- Egy példa a használatra
const bool result =
    EVALUATE( Member<int>::NonStatic( &list<string>::size ) );
```

A típusvizsgálatokat két részre osztjuk, külön vizsgáljuk az osztály és példányszintű tagokat. A statikus (osztályszintű) tagok vizsgálata egyben a globális, osztályhoz nem kötődő adatokra is használható. A statikus változókat vizsgáló kód egy várt típusú mutatót elfogadó függvényt és egy mentőágat tartalmaz, megvalósítás nélkül, ezeknek mindig egy egyszerű mutató típusú paramétert adunk át.

Az objektumszintű adattagok vizsgálata mindezt annyival egészíti ki, hogy az objektum típusát is a paraméterek közé kell vennünk. Ezért a *NonStatic* függvény egy típusparaméterrel (*Class*) rendelkező sablon, mely a függvényparaméter típusában is megjelenik, tehát a típusparaméter automatikus kikövetkeztetése lehetséges. A

függvényparaméterben látható szokatlan formájú *VariableType Class::** típusleírás azt adja meg, hogy a *Class* osztályon egy objektumán belülre mutató *VariableType* típusú objektumot várunk paraméterül.

Kövessük végig részletesen, mi is történik az utolsó sor meghívásakor:

1. Példányosulnak a *Member<int>* és *list<string>* osztályok.

2. Megkapjuk a *list<string>::size* tagjának címét. A C++ szabvány kikötése szerint ez egy tagfüggvény, tehát az eredmény egy tagfüggvény-mutató (member pointer) lesz.

3. Meghívódik a *Member<int>::NonStatic()* függvény. Ha a *Member* osztály típusparamétere megegyezik a paraméter által mutatott objektum típusával, akkor meghívható az első ág, különben a fordító a mentőágat választja. Mivel az elfogadó ág által várt paraméter *int list<string>::** típusú (vagyis mutató a *list<string>* típuson belül egy *int* típusú példányváltozóra), az átadott paraméter pedig egész szám helyett egy függvényre mutat, ezért a választás a mentőágra esik.

4. Az *EVALUATE* makró meghívja a *sizeof* operátort, mely megállapítja a kiválasztott ág által visszaadott típus méretét a függvény futtatása nélkül. A makró ezt összehasonlítja a *Yes* típus méretével, ennek alapján a fordító egy logikai értéket állít elő. Mivel a fordító a mentőágat választotta, az eredmény hamis lesz.

5. Egy konstans változóban eltároljuk a viszgálat eredményét, mely a továbbiakban tetszőleges fordítási időben kiértékelendő kifejezésben, például metaprogramok paramétereként is szerepelhet.

Ezzel tehát az objektum- és példányszintű változók típusvizsgálata egyaránt lehetségessé vált.

4.2.2.2. Tagfüggvények típusának vizsgálata

Bár a függvényszignatúrák típusához visszatérési érték, paraméterlista és egyéb módosítók (pl. *const*) is tartoznak, a függvények típusa alapvetően a változók típusával

azonos módon definiálható és használható C++ nyelven. Bármilyen meglepő, ennek segítségével a tagfüggvények típusának vizsgálata már magától adódik az adattagok vizsgálata alapján.

```
// --- A vizsgált függvénytípus definíciója
typedef size_type FuncType() const;

class ExampleClass {
   FuncType size; // --- Függvénydeklaráció a fenti rövidítéssel
};

// --- A size() függvény típusának megállapítása
const bool result =
    EVALUATE( Member<FuncType>::NonStatic( &list<string>::size ) );
```

Fent először rövidítésként egy nevet definiálunk a függvénytípushoz, melyet vizsgálni fogunk[6]. Az utolsó sorban végrehajtjuk a vizsgálatot, melynek alakja és működési is teljesen azonos az adatokknál bemutatottakkal. Mivel a *size* tagfüggvény valóban a vizsgált típussal rendelkezik, a kiértékelés ezúttal igaz értékkel tér vissza.

4.2.2.3. Beágyazott típusok létezésének vizsgálata

Létező beágyazott típusok eddigiekhez hasonló vizsgálata meglehetősen egyszerű és megoldott probléma (lásd Loki [13] könyvtár *IsSameType* vagy Boost Metaprogramming Library [17] *is_same_type*), ezért ezzel a továbbiakban nem foglalkozunk. Ezek a vizsgálatok azonban fordítási hibához vezetnek, ha a megadott nevű beágyazott típus nem létezik, ezért ezt is vizsgálnunk kell. Ennek eldöntése azonban már összetettebb probléma, szükségünk lesz hozzá az *enable_if* alkalmazására. A megoldás egy speciálisabb formája megtalálható [21] alatt is, de mi ennél általánosabbat adunk.

A megoldás megkívánja még egy apró technikai eszköz. a *Type2Type* típus bevezetését, mely szintén a Loki [13] könyvtár része. Egyparaméteres, üres törzzsel rendelkező egyszerű jelölőosztály, melyet függvényparaméterként fogunk használni.

[6] A *const* módosító használata első olvasásra furcsának tűnhet, mivel az csak tagfüggvények esetében értelmezhető. A C++ nyelv specifikációja azonban ezt a hasonló definíciók támogatása érdekében megengedi. Segítségével tagfüggvényeket definiálhatunk a függvényszignatúra kiírása nélkül, kizárólag a rövidítés felhasználásával, amint ez a példában is látható.

```
// --- Típusdefiníció
template <class T> struct Type2Type {};

// --- Példa a használatára
template <class T> void func(Type2Type<T>) { ... }
func( Type2Typ2<Matrix>() );
```

Bevezetésének célja, hogy automatikus típuskikövetkeztetéssel képesek legyünk típusparamétert átadni egy sablonfüggvény számára anélkül, hogy azt explicit módon ki kellene írnunk[7]. Erre azért van szükségünk, mert explicit típusparaméterrel a SFINAE szabályt már nem alkalmazhatnánk.

Az alábbi programkóddal azt dönthetjük el, hogy létezik-e egy osztálynak *iterator* nevű beágyazott típusdefiníciója:

```
// --- iterator beágyazott típussal rendelkezõ osztályok
template <class T>
typename enable_if<sizeof(typename T::iterator), Yes>::Result
checkIterator(Type2Type<T>);

// --- Mentõág
No checkIterator(...);

const bool result =
    EVALUATE( checkIterator( Type2Type< list<string> >() ) );
```

A megoldás az eddigiekhez hasonló alapelveken működik, lényeges különbség a *checkIterator* függvény visszatérési értékében látható. Itt az *enable_if* segítségével kötjük ki a paraméter típusnak azt a tulajdonságát, hogy rendelkeznie kell *iterator* nevű beágyazott típussal. Ha van ilyen, a függvény példányosítható, ha nincs, a mentőágat választja a fordító. A példában látható *list<string>* osztály esetében van, tehát igaz érték lesz a végeredmény. Az *enable_if* feltételében szereplő *sizeof* abban segít, hogy típusból logikai értéket készítsünk: mivel minden típus mérete legalább 1 bájt, így ha van *iterator* típus, mindig nullánál nagyobb értéket kapunk.

[7]Ha ez nem lenne követelmény, akkor a fenti példában a függvénysablonnak nem lenne szüksége paraméterre, valamint hívása is egyszerűen *func<Matrix>()* alakú lehetne.

Érdemes megemlíteni, hogy ugyanezt elérhetjük külön típus bevezetése nélkül is, ha a függvényben *Type2Type<T>* helyett egyszerűen *T** mutatót várunk paraméterként, és a függvényt egy megfelelő típusú nullpointerrel hívjuk meg *func(static_cast<T*>(NULL))* alakban. A *Type2Type* azonban könnyebben olvasható és elegánsabb, ezért inkább azt használjuk.

Ez C++ nyelven a beépített konverziós szabályok értelmében igaz logikai értéknek felel meg.

Ez a példa azonban csak az *iterator* típust vizsgálja, nekünk ennél általánosabb megoldásra van szükségünk. Mivel nevet C++ nyelven sablonokkal nem, kizárólag csak makrók segítségével adhatunk paraméterül, ezért a megoldáshoz ismét makrókat kell segítségül hívnunk.

```
// --- Makró a vizsgálat definíciójára
#define PREPARE_TYPE_CHECK(NAME) \
template <class T> \
typename enable_if<sizeof(typename T::NAME), Yes>::Result \
checkType_##NAME( Type2Type<T> ); \
\
No checkType_##NAME(...)

// --- Makró a vizsgálat meghívására
#define TYPE_IN_CLASS(NAME,TYPE) checkType_##NAME( Type2Type<TYPE>() )

// --- Példa a makrók használatára
PREPARE_TYPE_CHECK(iterator);
const bool result = EVALUATE( TYPE_IN_CLASS(iterator, list<string>) );
```

Az első makró az előző példa vizsgálatot végző függvényeinek névvel paraméterezett általánosítása[8]. Használatával egyszerűen definiálhatjuk a vizsgálatot végző függvényeket, lehetővé téve bármilyen általunk meghatározott típusnév vizsgálatát. Ezt a makrót értelemszerűen vizsgálatok elvégzése előtt, pontosan egyszer kell végrehajtani. A második makródefiníció mindössze a megfelelő függvényt hívja meg, és elrejti a Type2Type alkalmazását. A makrók alkalmazásával a teljes előző példa jóval egyszerűbben és olvashatóbban kifejezhető, amint az az utolsó sorokban látható.

4.2.2.4. Tagok létezésének vizsgálata

A beágyazott típusokhoz hasonlóan egy osztály tagjainak fent bemutatott vizsgálatai is fordítási hibával érnek véget, ha nem létezik megadott nevű tag. Ezért

[8]A makró sorainak végén levő visszaper (backslash) feltétlenül szükséges. Mivel a makródefiníciókat a sor vége lezárja, többsoros makrók definíciójában minden sor végén el kell kerülnünk a sortörést.

a tagok létezését is vizsgálunk kell, a vizsgálat azonban a típusok létezésének ellenőrzéséhez hasonlóan már könnyen megvalósítható. A különbség mindössze a név logikai feltétellé alakításában van.

Osztályok tagjaira tagmutatók állíthatók, melyek értéke garantáltan nem *null*. A C++ konverziós szabályai szerint minden nem *null* mutató automatikusan igaz logikai értékre konvertálható, ezzel pedig készen is vagyunk. Az *enable_if* feltétele így a következőre változik:

```
// --- Makró a vizsgálat definíciójára
#define PREPARE_MEMBER_CHECK(NAME) \
template <class T> typename enable_if< &T::NAME, Yes >::Result \
checkName_##NAME( Type2Type<T> ); \
\
No checkName_##NAME(...)

// --- Makró a vizsgálat meghívására
#define MEMBER_IN_CLASS(NAME,TYPE) checkName_##NAME( Type2Type<TYPE>() )

// --- Példa a makrók használatára
PREPARE_MEMBER_CHECK(size);
bool result = CONFORMS( MEMBER_IN_CLASS(size, MyContainer) );
```

Látható, hogy a vizsgálat minden egyéb részlete a típusoknál bemutatottakkal teljes mértékben megegyezik.

4.2.3. Predikátumok kompozíciója

Az eddig bemutatott elemi vizsgálatok egymástól teljesen független, diszjunkt tulajdonságok meglétét vizsgálták. Ezek önmagukban ritkán használatosak, gyakorlati alkalmazásokban ezekből többnyire valamilyen összetett kifejezést építünk fel. A hasonló eszközök általában csak a követelmények felsorolását támogatják, ezeket egy implicit *és* logikai művelettel kapcsolják össze. Ez azonban sok esetben kevés, ám könnyen kiterjeszthető. Mivel a vizsgálatok logikai értéket adnak eredményül, így ezekből a C++ nyelv logikai operátoraival természetes módon építhetünk fel tetszőleges összetett kifejezést.

Egyszerű példát hozva a *vagy* műveletre van szükségünk, ha azt akarjuk vizsgálni, definiált-e az összeadás operátor egy típusra. Mivel operátor C++ nyelven

definiálható tagfüggvényként és globális függvényként is, ezért mindkét esetet vizsgálnunk kell. Ezt például az alábbi módon tehetjük meg:

```
// --- Az "összeadható" tulajdonság vizsgálata a VAGY operátorral
template <class T> struct IsAddable
{
    enum { Result =
      CONFORMS( Function<T (const T&)>::NonStatic(&T::operator+) ) ||
      CONFORMS( Function<T (const T&, const T&)>::Static(&operator+) )
    };
};
```

A logikai negációra már ritkábban van szükségünk, de ennek használata sem elképzelhetetlen. Ilyen lehet például annak ellenőrzése, hogy egy osztály megfelel-e az egyke (singleton) tervezési mintának. Az ilyen osztálynak nem lehet publikus konstruktora, ami csak negációval fejezhető ki. Más esetekben is előfordulhat egy tulajdonság meglétét ellenőrző (jellemzően *HasX* vagy *IsX* alakú) vizsgálat negálása, bár nehéz előre látni az összes lehetséges hasznos kifejezést. Ilyen lehet, ha biztosítani akarjuk, hogy egy típus nem azonos egy másikkal (például bármilyen mutató, de nem *void**, ez a Boost Type Traits könyvtárának segítségével kifejezve *is_pointer<T>::value && ! is_same<T,void*>::value* alakban írható), vagy a paraméterül kapott típus nem tömb (*! is_array<T>::value*).

4.2.4. Vizsgálatok eredményének felhasználása

Eddigi eszközeinkkel a kvantorok kivételével minden elsőrendű logikai formulát ki tudunk fejezni. Mire is tudjuk felhasználni a kifejezések kiértékelésének eredményét?

Legegyszerűbb dolgunk akkor van, ha hibajelzéssel meg akarjuk szakítani a fordítást. Egyik fenti példánkat felhasználva garantálhatjuk, hogy egy mátrix elemtípusa összeadható:

```
// --- Példa a fordítási hiba kikényszerítésére
template <class T> class Matrix
{
    STATIC_CHECK(IsAddable<T>::Result, ELEMENT_TYPE_IS_NOT_ADDABLE);
    ...
};
```

A fenti példában a Loki könyvtár *STATIC_CHECK* nevű makróját használjuk a hiba kikényszerítésére, ha a feltétel nem teljesül.

Ennél azonban módszerünk jóval kifinomultabb lehetőségeket is biztosít. A vizsgálatok eredményét tetszőleges, C++ nyelven kifejezhető metaprogramban hasznosíthatjuk. Ilyen lehet a programban felhasznált típusok vizsgálatok eredményétől függő kiválasztása, például a Boost MPL fordítási idejű elágazásainak segítségével (*mpl::if_* és *mpl::if_c*). Ily módon algoritmusok is kiválaszthatók, nem csak típusok, hiszen kiválasztott osztályok esetében azok tartalmazhatnak statikus függvényeket is. Ez azonban a gyakorlatban már nagyon nehezen és körülményesen használható megoldás, helyette érdemes valamilyen közvetlen támogatást nyújtani.

A jelenlegi C++ nyelven erre sajnos nincs lehetőség, így megoldásként kénytelenek vagyunk egy egyszerű nyelvi kiterjesztést javasolni. Legyen minden sablondefinícióban a sablon törzse előtt megadható a *where* kulcsszó után a sablonparaméterekkel szembeni kikötések listája! A fordítóprogram pedig csak akkor engedje példányosítani a sablont, ha az teljes mértékben megfelel a kikötéseknek!

A sablonparaméterekkel szembeni kikötések tetszőleges, fordítási időben kiértékelhető logikai kifejezések lehetnek, melyet a fordítóprogramban már eddig is képes volt kezelni. Így a kiterjesztés megvalósítása nem igényel nagy erőfeszítést, mindössze a nyelvtan kiterjesztésével jár. Előző példánk ebben a formában a következőképp írható:

```
template <class T> class Matrix where IsAddable<T>::Result
{ ... };

// --- Egyenértékû behelyettesített változat
template <class T> class Matrix where
  CONFORMS( Function<T (const T&)>::NonStatic(&T::operator+) ) ||
  CONFORMS( Function<T (const T&, const T&)>::Static(&operator+) )
{ ... };
```

A második egyenértékű az elsővel: az *IsAddable* feltétel helyére annak megvalósítását fejtettük ki, hogy példát mutassunk az összetett logikai kifejezések feltételbe illesztésére.

Bár részleteiben eltérő, de hasonló alapokon nyugvó megoldást (lásd 4.3.2.3) javasol a C++ nyelv C++0x nevű, formálódó szabványa, valamint hasonló eszközt (4.3.1.3) vezetett be nemrégiben a D nyelv legújabb verziója is.

4.2.4.1. TODO esetleg valami megkötés alapú overloadingot kitalálni

4.2.5. Értékelés

A megoldás bemutatása után megkíséreljük azt objektíven értékelni. Ennek megfelelően egy felsorolás keretein belül összefoglaljuk a megoldás minden előnyét és hátrányát. Legfontosabb tulajdonságai a következők:

Szabványos. Megoldásunk legfontosabb előnye. Az opcionálisan javasolt *where* kulcsszó kivételével kizárólag szabványos C++ nyelvi eszközöket használtunk fel, ellentétben a többi hasonló megoldással, melyek mind nyelvi kiterjesztésen alapuló megoldások.

Széleskörű. Bár a bemutatott rendszer nem biztosít minden lehetséges típusú vizsgálatot, a legtöbb gyakorlatban felmerülő problémára megoldást ad. Kiváltképp igaz ez akkor, ha a fentiek (4.2.2) szerint rendszerünket a Boost Type Traits [18] könyvtár egy lehetséges kiterjesztéseként tekintjük.

A kifejezőerőt jelentősen növeli, hogy a megszorítások leírásában tetszőleges logikai operátort felhasználhatunk, ellentétben a legtöbb hasonló rendszerrel, mely implicit ÉS-t használ.

Rugalmas. A feltételek eredménye szabadon felhasználható. A kiértékelés ismeretében tetszőleges metaprogramozási akció végrehajtható, ellentétben a legtöbb hasonló eszközzel, mely kizárólag fordítási hibát képes kiváltani.

Bővíthető. Ha a rendelkezésre álló elemi vizsgálatokon kívül mégis továbbiak megvalósítására volna szükségünk, mindössze a szükséges predikátumokat kell megvalósítanunk, akár szabványosan, akár nyelvi kiterjesztésként. Ezután a predikátum eredménye kényelmesen, a többihez hasonlóan használható.

Láthatóság által korlátozott. A vizsgálatok megvalósítására használt módszer a fordítóprogram szabályainak betartásával dolgozik. Ezek a szabályok megakadályozzák, hogy egy osztály védett vagy privát tagjaihoz kívülről hozzáférjünk. A

legtöbb esetben ilyen információt nem használunk fel, ám ha erre mégis szükségünk lenne, a bemutatott módszer erre már nem alkalmas.

Túlterhelt tagokra nem alkalmazható. Túlterhelt függvényekre a C++ nyelv olyan szabályokkal rendelkezik, melyek esetünkben rosszul használhatók. Hívásuk egyszerű és kényelmes, de más esetekben, például a függvény címének lekérdezésekor ki kell segítenünk a fordítót. Típusmódosítókkal kiegészített pontos függvényszignatúrát kell adnunk, amely alapján a fordító ki tudja választani a megfelelő változatot. Mivel ez esetünkben ismeretlen, hiszen éppen ezt a típust próbáljuk kideríteni, ezért hiányában a fordítóprogram a többértelműség miatt hibát jelez.

Ezen szabványos viselkedés miatt a hiba feloldására valamilyen nyelvi kiterjesztéshez kell folyamodnunk. Mivel adattagokat nem terhelhetünk túl, rájuk a korlátozás szerencsére nem vonatkozik.

Összetett. Közvetlen nyelvi támogatás esetén a használat jóval természetesebb és egyszerűbb. Rövidebb ezen kívül az eszköz megértéséhez és alkalmazásához szükséges idő is. Ez természetszerű, hiszen a szabványos eszközök használata miatt lehetőségeink jóval korlátozottabbak, mint egy szabadon választott kiterjesztés esetén.

Túlterhelés nem megoldott. Ha különböző megszorítások segítségével túlterhelünk egy sablondefiníciót, megoldásunk nem biztosít lehetőséget a legspeciálisabb eset kiválasztására. Sablonspecializációk esetén ez a leszármazási információk alapján biztosított, elsőrendű logikai kifejezések esetén viszont automatikus tételbizonyításra lenne szükségünk, amely megoldatlan probléma.

4.3. Kapcsolódó művek

A saját eredmények ismertetése után áttekintjük a témához kapcsolódó munkákat és eredményeket. A közelmúltban a fenti megoldásra több cikk is hivatkozott, az ELTE kutatásai mellett (például [66]) több külföldi tanulmány, úgymint a Freie Universität Berlin és University of Auckland [64], illetve a EPITA Research and Development

Laboratory [63] kutatói által készített konferenciacikkek, továbbá a University of Auckland egy disszertációja [65].

Az alábbiakban azokat a különböző megközelítéseket és eredményeket ismertetjük, melyek hasonló problémák megoldására születtek, hangsúlyozzuk azok előnyeit és hátrányait is. Részletesebben elsősorban a C++ nyelven elért eredményekkel ismerkedünk meg, de a teljesség kedvéért kitekintünk más nyelvekre is.

4.3.1. Kitekintés más nyelvekre

4.3.1.1. Virtuális gép alapú és interpretált nyelvek

Számos dinamikus programozási környezet létezik, melyek nem közvetlenül gépi kódot készítenek, hanem valamilyen magasabb absztrakciós szintű programleírással dolgoznak. Ezek maguk fordítják gépi kódra a betöltött programot, így mindenképp kénytelenek elemezni azt. Tárolják tehát a program szemantikájáról kinyert információt, melyre építve a típusleírók (3.2.4) már könnyen megvalósíthatók. Mindössze egy lekérdezőfelületet kell biztosítanunk a szemantikai információhoz, hogy az közvetlenül egy programnyelvből is elérhetővé váljon.

A fenti okok miatt a virtuális gépen futó nyelvek és programozási rendszerek, illetve a parancsértelmező alapú nyelvek (például a .NET rendszerű C# nyelv, Java, Smalltalk, Python, Ruby, stb.) általában biztosítanak lehetőséget a programkód futási idejű vizsgálatára. Sőt, a rendszerek dinamikus volta miatt általában a kódgenerálást, illetve a meglevő szemantikai információ módosítását is lehetővé teszik. Ez az információ viszont csak futási időben érhető el és használható fel, ezáltal az esetleges hibás működés kizárólag a program futása közben derülhet ki. Mivel a rugalmasságért a program biztonságával fizetünk, szükség van ennél statikusabb nyelvekre, fordítási idejű típusleírással. Mivel a dolgozat témájához ez áll közelebb, a továbbiakban kizárólag ilyen nyelvekkel foglalkozunk.

4.3.1.2. Ada

TODO

Ada attribútumok

korlátozott típusinformáció tetszőleges típusról

4.3.1.3. D

A D programozási nyelv (lásd [48] és 3.5.2) az Ada nyelvhez hasonló információt nyújt a típusok tulajdonságairól, ezeket *property*-nek nevezi. A nyelv újonnan megjelent 2.0 verziója az általam C++ nyelvhez javasolt megoldáshoz igen hasonló eszközöket és újításokat vezetett be a nyelvbe.

A fordítás közben lekérdezhető statikus típusinformáció (*traits*, lásd 3.5.2) jóval a tulajdonságok lehetőségein túl is támogatja elemi vizsgálatok végrehajtását. A 4.2.2 alatt bemutatott predikátumok mindegyike végrehajtható, az egyszerű típusmegszorításoktól kezdve (*isScalar*, *isAbstractClass*, stb) szimbólumok létezésének vizsgálatán át (*compiles*, *hasMember*, stb) a pontos típusvizsgálatig (*isSame*). A predikátumok szabadon kombinálhatók logikai operátorokkal, az eredményül kapott logikai érték kiértékelése természetesen szintén fordítási idejű.

A vizsgálatok eredményét egyszerűen felhasználhatjuk a sablondefiníciók után írt *if* megszorító kifejezés (template constraints) segítségével. A kifejezés paraméterét a fordító a sablon minden példányosításakor kiértékeli. Ha az eredmény hamis logikai érték, az fordítási hibát jelent.

Nemcsak predikátumok írhatók, hozzáférhető a fordítási idejű típusleirás is (*allMembers*, *getVirtualFunctions*, stb).

4.3.2. C++ nyelvű megközelítések

4.3.2.1. CCEL és Clean++

A CCEL (C++ Constraint Expression Language [45]) metanyelvet a C++ nyelvű programok megszorításainak leírására alkották meg. Segítségével kifejezhetők olyan megszorítások, melyeket a nyelv típusrendszerével nem lehet kifejezni. A lehetséges megszorítások jellege rendkívül sokrétű, kapcsolatos lehet osztályok tervezésével (például egy függvényt az osztály minden közvetett és közvetlen leszármazottja köteles felüldefiniálni), megvalósításával (mutató adattag esetén kötelező másoló konstruktor és értékadó operátor definiálása), illetve kódolási konvenciókkal (pl. minden osztálynév nagybetűvel kezdődik).

A megszorítások leírásához és ellenőrzéséhez természetesen szükségünk van a típusok metaadataira. Ezt egy objektum-orientált elven felépített metaosztály

könyvtár biztosítja. A metaosztályok 3.2.3 alatt leírtakhoz hasonló elemi vizsgálatok végrehajtását teszik lehetővé tetszőleges nyelvi konstrukciókon (például leszármazik-e egy osztály egy másikból, vagy virtuális-e egy tagfüggvény).

Az elemi vizsgálatok segítségével válik lehetségessé a megszorítások megfogalmazása. Ezek az elsőrendű logika alapelvei szerint fogalmazhatók meg. A változók különböző nyelvi konstrukciók (típusok, változók, függvények, stb) lehetnek, a támogatott elemi vizsgálatok (pl. *is_const()*) szolgálnak predikátumként. Az elsőrendű logika szabályainak megfelelően lehetőség van a kifejezések kvantálására is.

A CCEL nyelven leírt megszorításokat a Clean++ nevű rendszer ellenőrzi. Működése során először elemzi az ellenőrzött programot, és a kinyert információt egy adatbázisba menti. A megszorítások definícióját adatbázislekérdezésekké fordítja, és ennek segítségével végzi el az ellenőrzést.

Az eszköz előnye a segítségével kifejezhető megszorítások széles skálája. Hátránya, hogy a fordítóprogramoktól teljesen független, nem szabványos eszköz, továbbá csak hibajelzésre ad lehetőséget, a vizsgálatok eredményének kifinomultabb felhasználására nem.

4.3.2.2. Boost Type Traits

A Boost type traits [18] könyvtár lehetőséget biztosít bizonyos elemi vizsgálatok elvégzésére tetszőleges paraméter típuson. A vizsgálatok ebben a könytvárban is 3.2.3 alatti elveken nyugszanak. Az elvégezhető vizsgálatok széles skálán mozognak, azonban nyelvi kiterjesztés híján meglehetősen korlátozottak. Nem támogat például 4.2.2 alatt bemutatottakhoz hasonló adattagok vagy tagfüggvények létezését és típusát kikövetkeztető vizsgálatokat.

A type traits előnye, hogy különálló könyvtár, nem igényel semmilyen külső eszközt. További előnye, hogy a vizsgálatok eredménye fordítási idejű logikai konstans, így a metaprogramozás során közvetlenül felhasználható. Hátránya, hogy a vizsgálatok egy részének megvalósításához a könyvtár kénytelen fordítóprogramfüggő, nem szabványos eszközöket felhasználni. Másik hátránya, hogy önmagában nem ad jól használható megoldást, mivel C++ nyelven a fordítási idejű elágazások megvalósítása az esetek egy részében nehézkes, sok esetben pedig nem megvalósítható.

4.3.2.3. Concept

Új C++0x szabvány: concept-ek [25] hasonló elven dolgoznak
részletes összehasonlítás a concept-ekkel

5. fejezet

A típusrendszer kiterjesztése

A fordítóprogramon alapuló programozási nyelvek legtöbbje a program megbízhatóságának növeléséhez típusokat használ, ezáltal a fordítóprogram még a program fejlesztése közben képes kiszűrni az esetleges programozási hibák egy részét. Az objektum-orientált nyelvekben leggyakrabban használt típusrendszer az absztrakt adattípusokra (absztrakt ősosztály, interfész) épül, az altípusosság pedig a helyettesíthetőség elvén (Liskov substitution principle [28]) alapul. A helyettesíthetőség elvének gyakorlati alkalmazása a szerződésmodell alapú tervezés (design by contract [29]), mely a tesztelésnél és helyességbizonyításnál bír alapvető jelentőséggel. Az adattípusokon alapuló típusrendszer erejét az elméleti alapok mellett jól mutatja a rendszer több évtizedes sikeres gyakorlati alkalmazása is. A széleskörű használat azonban nem csak az elmélet használhatóságát igazolta, hanem természetes módon rámutatott annak korlátaira is. Ez a fejezet egy öröklődéssel kapcsolatos problémát jár körül és korábbi munkáimból [4, 5] kiindulva bemutat egy lehetséges megoldást, mely a szabványos C++ nyelv eszközeit használja fel.

5.1. A probléma leírása

Az interfészek és osztályok közti öröklődés az objektum-orientált programozás egyik alapköve, mely a kód újrafelhasználását és a programkomponensek fokozatos finomítását, részletezését hivatott elősegíteni. A gyakorlatban elterjedt objektum-orientált nyelvek mindegyike explicit módon jelölt öröklődést alkalmaz, ezzel

valósítja meg az újrafelhasználást, és ebből vezeti le az altípus relációkat. Explicit jelöléssel egy osztály csak akkor lesz egy ősosztály leszármazottja (illetve valósít meg egy interfészt), ha ezt a programkódban kijelentjük. Például Java nyelven:

```
class MyDerivedClass
extends MyBaseClass                        // --- õsosztály
implements Serializable, Cloneable         // --- interfészek
{
    ...
}
```

A programok karbantartásának és újrafelhasználhatóságának kritikus eleme a program alkotórészeinek, komponenseinek hatékony elkülönítése feladatok és felelősségek szerint (separation of concerns). Ez teszi lehetővé, hogy programjainkat ne az alapoktól kezdve, hanem a rendszerkönyvtárakban előre elkészített építőelemek, szolgáltatások felhasználásával készítsük el. Jól kidolgozott elmélete és módszertana számos forrásban megtalálható, például [39, 38].

Az alapelemekből építkezés során kódunk jellemzően több építőelemet, komponenst is felhasznál, vagyis sokszor több interfészt valósítunk meg egyszerre, esetleg több osztályból is öröklünk. Ez az egészen alapvető, egyszerűbb osztályoknál is gyakran előfordul. A Java 1.6 rendszerkönyvtárából véve egy egyszerű példát:

```
class String
extends Object
implements Serializable, CharSequence, Comparable<String>
{
    ...
}
```

Ha ilyen alapkomponensekből építkezünk, gyakran előfordul, hogy azok nem pontosan felelnek meg az igényeinknek. Nagyságrendekkel több munkát igényelne azonban ezeket nekünk megvalósítani, hiszen apróbb átalakításokat és finomításokat is végezhetünk a komponenseken. Ezt objektum-orientált rendszerekben legtöbbször leszármazással valósítjuk meg a lépésenkénti finomítás eszközével [40]. Végül az igényeinknek már pontosan megfelelő komponensekből állíthatjuk össze a kívánt működést biztosító programrészt, a konkrét esetnek megfelelően öröklődéssel vagy aggregációval.

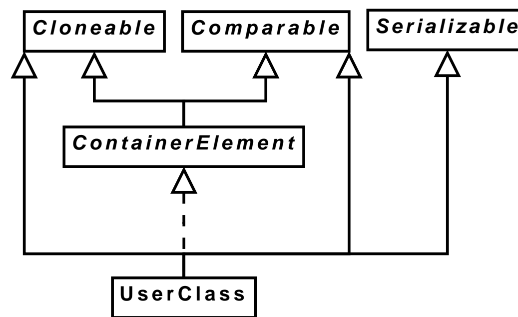
Több interfész megvalósításának egy másik példáját adják a különféle szolgál-

tatásokat nyújtó programkönyvtárak, melyek általában a kliens által átadott objektumokkal is dolgoznak. Az ilyen paraméterül kapott objektumokkal szemben támasztott követelményeket a szigorúan típusos nyelvek pontosan definiálják, általában interfész megvalósítását, esetleg ősosztályból való leszármazást kötve ki feltételként. Gyakran előfordul azonban, hogy egy paraméterrel szembeni kikötéseinket nem fedi le egyetlen építőelemként biztosított interfész vagy osztály sem, de ezeknek valamilyen halmaza már igen. A legtöbb programozási nyelv nem biztosít lehetőséget arra, hogy követelményként interfészek halmazát adjuk meg. Ilyen esetekben legtöbbször többszörös öröklődés [1] segítségével egy összetett interfészt származtatunk le az elemi interfészekből, ezzel elérve a kívánthoz hasonló viselkedést. Az így kapott összetett interfészek megvalósítandó interfészként szolgálhatnak az összes megszabott feltételnek megfelelő osztálynak. A megoldás egyetlen hibája, hogy az öröklődés explicit mivoltából adódóan előre, már az osztályhierarchia kialakításakor ismerni kellene az összes szóba jöhető feltételkombinációt. Mivel interfészekkel bármikor kiegészíthetjük a hierarchiánkat, a könyvtárak tervezésénél és legfőképp különálló programkönyvtárak együttműködésénél ez komoly problémákat okozhat.

A következő példában (5.1. ábra) a Java egyes alapvető interfészeit (Cloneable, Comparable, Serializable) implementáljuk egy osztályban (UserClass), illetve állítjuk össze egy kompozit interfésszé (ContainerElement). Példánkban feltételezzük, hogy egy konténerben tárolt elemek között keresnünk kell (összehasonlítás), és az elemekhez nem lehet írásra hozzáférni, ezért lekérdezéskor egy másolatot adunk vissza (klónozás). Ilyen konténert eredményezhet például a Repository tervezési minta [30] követése.

Az ábrán szaggatott nyíllal jelölve már fel is tűnik a probléma: hiába valósítja meg a kliens a UserClass osztályban mindhárom egyszerű interfészt, a ContainerElement interfésznek nem felel meg mindaddig, amíg nem fejezzük ki ezt explicit módon külön is. Bár osztályunk megfelel a konténer minden elvárásának, tervezésekor nem készítették fel a konténerrel való együttműködésre: mindaddig nem használhatók együtt, amíg az osztály nem származik le a ContainerElement interfészből. Ez nem elszigetelt és egyedi eset, hiszen könnyen előfordulhat, hogy jóval a UserClass

[1] Vegyük észre, hogy az interfészek közötti többszörös öröklődés jelentősen különbözik az osztályok közti többszörös öröklődéstől: előbbit például a Java nyelv támogatja, míg utóbbit már nem.



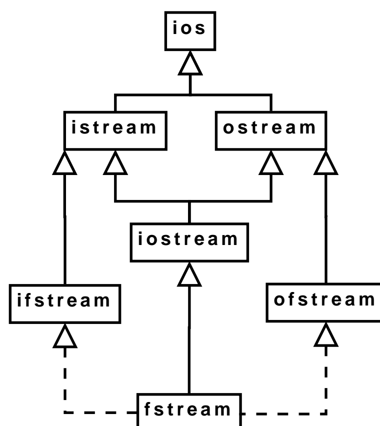
5.1. ábra. Példa több interfész megvalósítására

elkészülte után bővítették ki a rendszert a ContainerElement interfésszel. Emellett gyakran használunk együtt különböző felek által készített programkönyvtárakat, melyeket eleve képtelenség úgy tervezni, hogy a fentiek szerint minden más létező könyvtárral képesek legyenek együttműködni. Ha az osztály a sajátunk, a forráskód legtöbbször megváltoztatható, de egy másik fél által fejlesztett könyvtár vagy rendszerkönyvtár esetében ez a megoldás már nem jöhet szóba. Csak olyan megoldás fogadható el, amelyhez nincs szükség a már meglevő programkód módosítására, vagyis nem intruzív.

Végeredményben nincs lényeges különbség interfészek és osztályok közötti öröklődés esetén. A probléma szempontjából közömbös, hogy a UserClass interfész-e, mely nem származik le a ContainerElementből, avagy osztály, mely nem valósítja meg azt. Mindössze az eredmény fontos, miszerint semmilyen kapcsolatban nem állnak egymással. A megoldást a későbbiekben C++ nyelven adjuk majd, mely közvetlenül nem támogatja az interfészeket, de rendelkezik a hasonló funkcionalitást nyújtó "tisztán virtuális függvény" (pure virtual) nevű eszközzel. Ezért a továbbiakban általában csak az öröklést, leszármazást említjük, az interfészeket legtöbbször nem emeljük ki külön, kizárólag indokolt esetben.

Hogy lássuk, nem elszigetelt, egyedi jelenségről van szó, a következő 5.2. ábrán a C++ adatfolyamokat kezelő rendszerkönyvtárának alapja látható[2], ahol a leszár-

[2]Mivel az ifstream, ofstream és fstream osztályok pontosan ugyanazzal a funkcionalitással bővülnek ki a leszármazásnál. Egyes C++ implementációk (pl. g++ 2.95) ezért egy további osztályba (fileio) absztrahálták ezt a funkcionalitást, melyből mindhárom osztály leszármazik. Ebben az esetben az ábra értelemszerűen kiegészül. Más implementációk a fájlműveletek hozzáadását külön komponens hozzáadása nélkül, egyszerű kódismétléssel valósítják meg. A kódismétlés jól ismert karbantartási gondokat vethet fel, ezért általában nem használatos. Jól jelzi az eset súlyát, hogy



5.2. ábra. Példa a C++ STL könyvtárából

mazási probléma szintén felbukkan. Bár a szaggatott nyíllal jelölt leszármazási relációk kívánatosak lennének, a C++ szabvány a megvalósítás nehézségei miatt szándékosan [23] nem is írja elő azokat. Így az a meglepő helyzet áll elő, hogy ha egy ki- és bemenetet egyaránt támogató fájladatfolyam (fstream) objektumunk van, nem tudjuk azt paraméterül átadni függvényeknek, amelyek a fájlokat a be- és kimeneti műveletek közül kizárólag az egyiket igénylik, ezáltal ifstream és ofstream típusokkal dolgoznak. Tovább súlyosbítja a helyzetet az is, hogy az anomália nem csak a fájlokat kezelő folyamok esetében jelentkezik, hanem minden hasonló konstrukcióban, például a karakterláncokat kezelő folyamoknál is (istringstream, ostringstream és stringstream).

5.1.1. Formális leírás

A probléma formalizálását [40] által bemutatott modell alapján tesszük meg. A formalizáció során az interfészekra és osztályokra egységesen komponensekként (jelölése $k_i \in K$ komponenshalmaz) hivatkozunk, a lépésenkénti finomítás során a komponensekhez valamely tulajdonsággal, felelősséggel való kiterjesztésére funkcióbővítésként (jelölése $f_j \in F$ funkcióbővítések halmaza).

A funkcióbővítések olyan függvények, melyek komponenseket alakítanak át a hozzájuk tartozó tulajdonság hozzáadásával, $f : K \to K$. Egy k komponensre alka-

a legtöbb implementáció mégis ehhez az eszközhöz nyúlt. Ettől eltekintve a megoldás a működés szempontjából egyenértékű és szintúgy szabványos, az ábrán ez az egyszerűbb eset látható.

lmazott f funkcióbővítés jelölése $f \bullet k$. Komponensek egy halmazának kompozíciója (ahol $k = \{k_1, \ldots, k_n\}$) alatt a k_i interfészekből vagy osztályokból többszörös öröklődéssel történő leszármazását értjük, mely által k mindegyiküknek altípusa lesz. A funkcióbővítés művelete kommutatív, tehát

$$f_1 \bullet f_2 \bullet k = f_2 \bullet f_1 \bullet k$$

Továbbá a komponensek kompozíciójának műveletére nézve disztributív (lásd [40]), vagyis

$$f \bullet \{k_1, \ldots, k_n\} = \{f \bullet k_1, \ldots, f \bullet k_n\}$$

A fenti formalizmus segítségével a probléma egyszerűen megfogalmazható: az explicit módon jelölt altípusosság esetében f nem disztributív. Utolsó, adatfolyamos példánkat tekintve ez a következőképp látható be a fileio (lásd 2. lábjegyzet) funkcióbővítés felhasználva:

$$\begin{aligned} fstream &= fileio \bullet iostream = fileio \bullet \{istream, ostream\} \neq \\ &\quad \{fileio \bullet istream, fileio \bullet ostream\} = \{ifstream, ofstream\} \end{aligned}$$

A fejezet további részében a disztributivitás szimulációjára próbálunk megoldást nyújtani.

5.2. Lehetséges megközelítések

Az alábbiakban számbavesszük azokat a módszereket, melyek a probléma megoldására szóba jöhetnek, majd elemezzük azok előnyeit és hátrányait.

5.2.1. Hagyományos öröklődés

Az eddigiekben vázolt explicit altípusossággal rendelkező öröklődési modell úgy nyújthatna megoldást a problémára, ha sikerülne elérnünk, hogy leszármazottaink ne csak az eddig számításba vett ősosztályokból származzanak le. Számba kell ven-

nünk még mindazon összetett osztályokat is, melyek az eddigi ősosztályok variációival létrehozhatók, és ezekből is le kell származnunk. Sajnos az ősosztályok számának növekedésével azok lehetséges variációinak száma már exponenciálisan növekszik, tehát indokolatlanul sok osztályból kellene leszármaznunk végül a megoldás érdekében. Még ha rendelkeznénk is a fordítóprogramba épített automatizmussal minderre, a hagyományos öröklődés akkor sem nyújtana jó megoldást az extrém költségek miatt.

5.2.2. Virtuális öröklődés

A C++ módszere az ismételt öröklődés által felvetett problémák megoldására. Ilyen például a gyémánt alakú öröklődési anomália (diamond shape inheritance), mely a 5.2 ábrán is felbukkan. Ha egy D leszármazott osztály definíciójában egy B ősosztály neve mellett a virtual kulcsszó szerepel, a fordító figyel D mindazon további leszármazottaira, melyekben B ismételt ősként szerepelne. Ekkor a B-ben definiáltak a leszármazottban nem duplikálódnak minden alkalommal, ahol B ősosztályként szerepel, hanem garantáltan csak egyetlen példányban létezhetnek mindössze. A C++ nyelv az ismételt öröklődés problémáinak elhárítása mellett ezzel képes szimulálni az interfészeket, és támogatást nyújtani az absztrakt osztályokat és interfészeket felhasználó programozási módszerekhez, bővebben lásd [23, 24].

Sajnos a módszer jelentős hátrányokkal bír: a program megemelkedő processzor- és tárhelyköltsége mellett sajnálatos módon intruzív, hiszen beavatközást igényel a leszármazott osztály forráskódjába, emiatt nem nyújthat alapot egy általános megoldáshoz. Emellett nem oldja meg a hagyományos öröklődésnél fellépő exponenciális robbanást sem az ősosztályok számával kapcsolatban.

5.2.3. Szignatúrák

A szignatúrák a funkcionális nyelvekből származó (ML szignatúrák, Haskell típusosztályok), az interfészekhez valemelyest hasonló [34] konstrukciók; azok egy általánosításának is tekinthetők. A szignatúrákkal nemcsak osztályok tagfüggvényeire tehetünk kikötéseket, hanem adattagokra és beágyazott típusokra is, emelett az interfészekhez hasonlóan leszármazhatnak egymásból. Ráadásul egy osztály deklarációjában nem kell szerepelnie azon szignatúrák listájának, melyeknek az osztály

megfelel, tehát a megfelelés nem intruzív. Az interfészekhez hasonló megfelelési szabályokat alkalmazva a fordítóprogram implicit módon dönt, hogy egy osztály megfelel-e egy tetszőleges szignatúrának.

A szignatúrák egy C++ nyelvű megvalósítását bővebben [27] részletezi. A szignatúrák ideális megoldást nyújthatnának, ám sajnálatos módon nem szabványos eszközök egyetlen elterjedt objektum-orientált nyelvhez sem. A C++ nyelvű megvalósításnak létezett valaha egy prototípusa egy régi GNU C++ verzióhoz, efelett viszont vészesen eljárt már az idő.

5.2.4. Aspektusok

Az aspektus-orientált programozás [31] egy általános eszközt nyújt osztályok és eljárások definíciójuktól független kiterjesztésére, bővebben lásd 2.2.2. A módszertan lehetővé teszi, hogy általunk meghatározott illesztési pontokhoz (pointcut) valamilyen kiegészítő kódrészletet (aspect) szőjünk hozzá (weave). Előnye, hogy ezt az eredeti forráskódnak ehhez nem kell rendelkezésünkre állnia, tehát nem intruzív. Tudományos szemmel nézve széleskörűen elfogadott és elterjedt [32] megoldásnak nevezhetjük.

Megfelelő aspektusok osztályokba szövésével azok kívülről is kiterjeszthetők úgy, hogy a továbbiakban megvalósítsanak egy új interfészt. Ez ideális lehetne számunkra, ám minden új interfész és új megvalósító osztály esetén ki kell egészítenünk az illesztési pontjainkat vagy aspektusainkat. Ez minden alkalommal felhasználói beavatkozást igényel, ennek automatizálására sajnos szintén nincs támogatás. Az aspektusok használatával tehát nem tettünk mást, minthogy ugyanazt a problémát az interfészek szintjéről az aspektusok szintjére toltuk át, ami ugyanolyan elfogadhatatlan, mint a többi alternatíva.

5.2.5. Strukturális altípusosság

A strukturális altípusosság [33] egy, a fentiektől gyökeresen eltérő altípusossági modell. Továbbra is használ öröklődést, ám azon csak a kód újrafelhasználása alapul, az altípusok levezetése ettől teljesen leválasztva, külön szabályok szerint működik. Strukturális altípusosság esetén az öröklődési hierarchia helyett a típusok felépítése,

strukturális megfeleltethetősége határozza meg az altípus relációkat. Ez az altípusosság implicit, vagyis nem követeli meg az altípus relációk külön meghatározását, azok a fordító által programozói jelölés nélkül is automatikusan kikövetkeztethetők.

Mivel a megoldani kívánt öröklődési anomália az altípusosság öröklődéshez kötéséből adódik, a strukturális altípusosság alkalmazása ideális megoldást nyújthat. Ez a típusrendszert azonban csak néhány ritkábban használt, többnyire funkcionális nyelv valósítja meg, például az Ocaml [41]. A fejezet további részében a strukturális altípusossághoz hasonló viselkedést igyekszünk a C++ nyelvbe illeszteni a sablonokkal végzett metaprogramozás segítségével.

5.3. Megoldás

A lehetséges megoldások elemzésénél láttuk, hogy a strukturális altípusosság szimulálása a legcélravezetőbb módszer. Az alábbiakban ezt a megoldást mutatjuk be C++ nyelven a sablon metaprogramozás segítségével. Először a felhasznált módszerek leírása olvasható, majd az ezekre épített megoldás és annak egy továbbfejlesztett változata következik.

5.3.1. Típuslisták

A típuslisták a metaadatok számára megalkotott tárolók. A metaadatok jelen esetben nem egyszerű konstansok, hanem a C++ nyelv típusai. Bár egy speciális formáját már [35] leírja, az első általános célú típuslista megvalósítás a Loki nevű programkönyvtárban [13] található, jelenlegi legfrissebb formája a Boost metaprogramozást támogató könyvtárában [17] érhető el.

A lista a funkcionális nyelvek adatszerkezeteihez hasonlóan rekurzív módon épül fel. A típuslista maga is a C++ nyelv egy típusa, ám hivatkozásokat tárol más típusokra. Alapelve és megvalósítása alapján a kifejezés-sablonok (lásd 3.4.6) közé sorolhatjuk. Megvalósítása egy sablonnal történik, melynek két paramétere a fejelem és a lista maradéka.

```
template <class H, class T>
struct Typelist
{
```

```
    typedef H head;
    typedef T tail;
};
```

Bár ellenőrzésére a fenti definíció semmilyen eszközt nem ad, közmegegyezés szerint az első paraméter (a lista feje) nem lehet beágyazott típuslista, a második paraméter (a lista maradéka) rekurzívan tartalmazza az összes további listatagot, További konvenció, hogy a lista végének könnyebb meghatározása végett mindig egy speciális listalezáró elem kerül az utolsó helyre. A sablon példányosítása, vagyis egy konkrét típuslista létrehozása az alábbi módon történik:

```
// --- A listalezáró definíciója
class NullType{};

// --- Konkrét típuslista definíciója
typedef Typelist<float, Typelist<double,
    Typelist<long double, NullType> > >
    FloatingPointTypes;
```

A fenti kódrészletből könnyen észrevehető, hogy a típuslisták definíciója nagyobb elemszám esetén túlzottan nehézkes. Ennek megkönnyítésére több módszer is létezik, de mivel a megoldáshoz nincs szükség a típuslisták minden finomságára, a legegyszerűbb is megfelel. Makrókat definiálunk a lista elemszáma szerint, melyek kiegyenesítik a lista rekurzív definicióját:

```
// --- A kisegítõ makrók definíciója
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >
...

// --- A fenti típuslista egyenértékû definíciója
typedef TYPELIST_3(float, double, long double)
    FloatingPointTypes;
```

A megoldás egyik hátránya, hogy a lista hosszát nekünk kell explicit megadnunk a lista definíciójában. A másik előnytelen tulajdonsága az, hogy ha a típuslista valamely elemében szerepel vessző (például ha a lista egyik tagja több paraméterrel rendelkező sablonpéldányok), a C++ előfordítója ezt a vesszők mentén széthasítva

több elemnek fogja értelmezni. Ez könnyen megkerülhető, ha a szóban forgó típusra egy *typedef* segítségével egy egyszerű névvel hivatkozunk.

Az említett hátrányok azonban nem olyan súlyosak, hogy a lenti megoldásban érdemes lenne valamelyik kifinomultabb, ám sokkal bonyolultabb megoldást használni. A Loki könyvtárban található típuslista alapvetően a fenti módon valósul meg.

5.3.2. Osztályok kompozíciója

A típuslisták az osztályok kompozíciójának megvalósításában nyújtanak segítséget. Osztályok egy halmazának kompozíciója alatt a továbbiakban egy olyan osztályt értünk, mely minden a halmaz minden elemének leszármazottja.

Az osztályok egy halmazát típuslistával adjuk majd meg. A lista és a halmaz típusszerkezet az elemek sorrendiségének kérdésében alapvetően különbözik. Ez azonban nem okoz gondot, mivel a leszármazásokat előállító algoritmus nem használja ki az elemek sorrendjét, egyszerűen csak bejárja a lista elemeit, ahogyan azt egy halmaz elemeinek esetében is megtehetnénk. Halmazt azonban jóval nehezebb (bár nem lehetetlen) előállítani metaprogramozás segítségével, ám jelen esetben semmilyen előnyünk nem származna belőle, ezért célszerű típuslistákat használni.

A kompozíció a típuslista rekurzív bejárásával történik. A bejárás során az aktuális elem mindig a lista fejeleme. Az algoritmus minden lépésében az aktuális listaelemből (osztályból) öröklünk, így az algoritmus végére előálló osztály minden listaelem leszármazottja lesz. A leszármazást az alábbi kóddal valósíthatjuk meg:

```
// --- Elődeklaráció általános paraméterre
template <class ListOfTypes> struct CSet;

// --- Specializáció tetszőleges típuslistára
template <class Head, class Tail>
struct CSet< Typelist<Head,Tail> > :
    public Head, public CSet<Tail> { ... };

// --- Specializáció egyelemű típuslistára
template <class Head>
struct CSet< Typelist<Head,NullType> > :
    public Head { ... };
```

A kompozíciót a *CSet* osztálysablon[3] végzi el. A sablon nevében a *Set* a már fentebb is említett halmazjellegre utal, vagyis a lista elemeinek sorrendje tetszőleges, nem befolyásolja a kompozíció tulajdonságait. A név *C* betűje a *class*, composite, collaboration és chevron szavakra utal, melyek együttesen jól jellemzik a kompozíciót.

A *CSet* általános deklarációjának egyáltalán nincs törzse, vagyis példányosítás esetén soha nem fordul le. A definíció azonban típuslistákra specializálva teljeskörűen meghatározza a törzset is. Ezzel biztosítható, hogy típuslistákon kívül semmilyen más paramétert ne fogadjon el a fordító a sablon paramétereként.

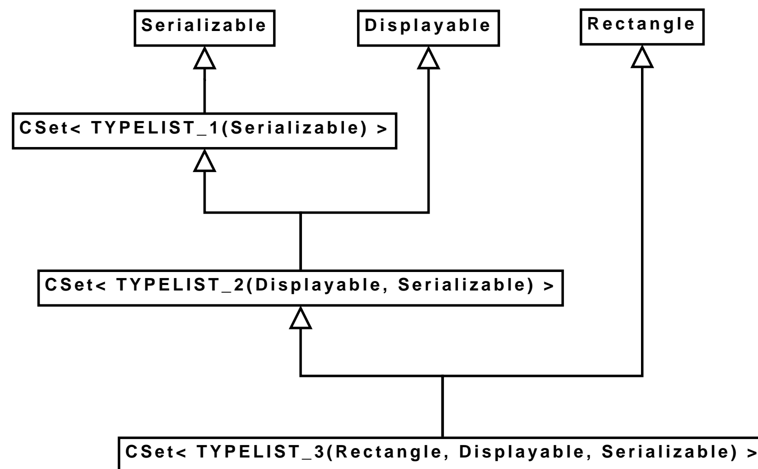
Az osztálysablon leszármazik a lista fejeleméből. illetve rekurzívan önmagából, de már csak a típuslista maradékával paraméterezve. Mivel a típuslisták konstrukciója kizárja a végtelen listákat, ezért a rekurzió a lista elemeinek bejárásával biztosan véget ér. A bejárás végét az egyelemű listákra adott specializáció biztosítja, mely már nem származik le a lista maradékából, mindössze a lista egyetlen eleméből, megszakítva ezzel a rekurziót.

Lássunk egy példát az algoritmus működésére! Három elemi funkcionalitást megvalósító osztályból készítünk kompozíciót. Egyik egy téglalap geometriáját írja le (*Rectangle* osztály, a síkidomokat leíró *Shape* leszármazottja), másik a képernyőn való megjelenítésért felelős (*Displayable*), a harmadik pedig az objektum sorosítást képes elvégezni (*Serializable*). Az osztálykompozíciót a lenti programkód végzi, az eredményül kapott objektum típusának osztálydiagramja az ábrán (5.3) látható.

```
// --- Osztálydefiníciók
class Shape { ... };
class Rectangle : public Shape { ... };
class Displayable { ... };
class Serializable { ... };

// --- A fenti osztályokból felépített összetett objektum
typedef TYPELIST_3(Rectangle, Displayable, Serializable) WindowTypes;
CSet<WindowTypes> window;
```

A CSet sablon használata igen hasonló a típuslistákéhoz, így a definíciók megkönnyítésére érdemes az ott leírtakhoz hasonló makrókat bevezetni. Ezáltal a kom-

[3] C++ nyelven a *struct* és *class* típuskonstrukciók kizárólag a tagok alapértelmezett láthatóságában különböznek, ezért beszélhetünk struktúrák esetében is osztályokról.



5.3. ábra. A többszörös öröklődéssel felépített osztályhierarchia

pozíció megadása teljesen természetessé válik, a típuslista elrejtésével a definíciók jelentősen egyszerűsödnek:

```
// --- Kényelmi makrók definíciója
#define CSET_1(T1) CSet< TYPELIST_1(T1) >
#define CSET_2(T1, T2) CSet< TYPELIST_2(T1, T2) >
#define CSET_3(T1, T2, T3) CSet< TYPELIST_3(T1, T2, T3) >

// --- Segítségükkel a fenti definíció egyszerûbb alakja
CSET_3(Rectangle, Displayable, Serializable) window;
```

5.3.3. A strukturális altípusosság megvalósítása

A típuslisták és kompozíció segítségével már bármilyen felépítésű, struktúrájú típust képesek vagyunk megalkotni. A megoldás utolsó lépése azoknak a konverziós szabályoknak a megfogalmazása és megvalósítása, melyekkel szimulálni lehet a strukturális altípusosságot. Ebben a C++ nyelv speciális konstruktorai és konverziós operátorai lesznek segítségünkre.

A C++ nyelv sajátos konverziós szabályokkal rendelkezik, különösen a függvényhívások esetén. Amennyiben egy függvényhívás aktuális paraméterei nem felelnek meg pontosan a függvény definíciójában szereplő argumentumok típusainak, a paraméterek konvertálhatósága esetén a fordító automatikus típuskonverziót hajt végre, ha ezzel pontos illeszkedést érhet el. Lássunk erre egy példát:

```
// --- Függvénydefiníció
void f(const void *ptr, double num) { ... }

// --- Függvényhívás konverziókkal
f("Hello world", 42);
```

A fenti függvényhívás egyik paramétere sem felel meg pontosan az argumentum elvárt típusának, azonban mindkét paraméter a nyelv beépített konverziós szabályai szerint a kívánt típusra (az egész szám lebegőpontosra, a karakterlánc pedig típus nélküli mutatóra) konvertálható). Ezen automatikusan konverzió segítségével a függvényhívás már szabályos.

Az ilyen automatikus konverziók nemcsak a nyelv beépített típusaira működnek, a felhasználó is adhat meg konverziós szabályokat saját típusaira. Erre szolgálnak a C++ konverziós, értékadó operátorai és az egyetlen paraméterrel rendelkező konstruktorai. Használatuk látható az alábbi példán:

```
struct MyType      // --- Típus definíciója
{
    int value;
    // --- Egy paraméteres konstruktor
    MyType(int i) { value = i; }
    // --- Konverziós operátor
    operator int() { return value; }
    // --- Értékadás operátor
    void operator = (int i) { value = i; }
};

MyType myObj = 3;      // --- Konstruktor hívása
int result = myObj;  // --- Konverziós operátor hívása
myObj = 42;            // --- Értékadó operátor hívása
```

Amint látható, a konverziós operátor biztosítja egy objektum más típusokra konvertálhatóságát, míg a konstruktorok és az értékadó operátorok együttesen képesek a más típusokból történő konvertálást megoldani.

Mivel C++ nyelven a konverziós operátorok és konstruktorok is nagyrészt csak szintaktikájukban különleges függvények, ezért a többi közönséges függvényhez hasonlóan maguk is lehetnek sablonok. Ezt felhasználva képesek vagyunk olyan konverziós függvényeket készíteni, melyek tetszőleges típusról képesek az adott osztá-

lyra konvertálni, amennyiben ez a nyelv szabályainak egyébként megfelel. Ezáltal automatizálható a konverziós függvények generálása, kiváltható a munkaigényes kézi megadás, mely jelentős hibatényezőt is jelent. Ezzel meg is kaptuk a *CSet* immár teljes funkcionalitással bíró formáját:

```
template <class Head, class Tail>
struct CSet< Typelist<Head,Tail> > :
    public Head, public CSet<Tail>
{
    // --- Kisegítõ típusrövidítések
    typedef CSet<Tail> Rest;
    typedef CSet< Typelist<Head,Tail> > ThisType;

    // --- Alapértelmezett konstruktor és destruktor
    CSet() : Head(), Rest() {}
    virtual ~CSet() {}

    // --- Konverziós konstruktor sablon
    template <class FromType>
    CSet(const FromType& from) : Head(from), Rest(from) {}

    // --- Konverziós értékadás operátor sablon
    template <class FromType>
    ThisType& operator = (const FromType& from)
    {
        Head::operator=(from); // --- Értékadás delegálása az
        Rest::operator=(from); // --- õsosztályok függvényeinek
        return *this;
    }
};
```

A fenti programkódban elég más típusokról *CSet*-re konvertálni, ezért a konstruktor és az értékadás operátor van definiálva. Mindkettő működési elve ugyanaz, mivel ugyanazt a tevékenységet végzik az objektum különböző életciklusaiban. A konverzió során a paraméterként kapott összetett objektum alapján először a fejelemhez tartozó adatrész kap értéket, majd rekurzívan a maradék adatrész. A függvényhívások láncolata tehát pontosan a leszármazási hierarchiát követi, például a korábban bemutatott 5.3 ábrán látható módon. A rekurziót megszakító, egyelemű listákra specializált *CSet* konverziós függvényei a fentihez nagyon hasonló, mindössze

a lista maradékára történő rekurzív hívásokat (*Rest* hivatkozások) kell törölnünk.

A C++ nyelv támogatja a függvénysablonok típusparamétereinek automatikus kikövetkeztetését a függvényhívás konkrét paraméterei alapján. Ez a fenti kód konverziós képességének fontos kiegészítőjét adja azáltal, hogy megszabadítja a programozót a pontos sablonparaméterek állandó meghatározásától, ezáltal a függvényhívások kiírásától a konverziók folyamán. Szemléltetésére egy egyszerű példa látható alább (természetesen nemcsak a nyelv beépített típusaira, hanem bármilyen általunk definiált típusra is ugyanígy működik):

```
// --- Függvénysablon
template <class T>
void operator << (T& t1, int i) { ... }

class MyType { ... };
MyType myObj;

// --- Az alábbi függvényhívások egyenértékûek:
myObj << 42;
::operator<< <MyType> (myObj, 42);
```

Mindezek ismeretében már könnyen megérthető a strukturális konverzió *CSet* által megvalósított működése. Tegyük fel, hogy a fent bemutatott, három komponensből összeállított ablak (*window*) objektumunkat szeretnénk használni az ablak keretét kirajzoló eljárásban, mely síkidomokat jelenít meg. Az eljárás paraméterének tehát megjeleníthetőnek (*Displayable*) kell lennie, továbbá síkidomnak (*Shape*) kell lennie, mely a téglalap (*Rectangle*) ősosztálya. Ezt demonstrálja az alábbi kód:

```
// --- Kirajzoló eljárás
typedef CSET_2(Displayable, Shape) BorderType;
void drawBorder(const BorderType &border) { ... }

CSET_3(Rectangle, Displayable, Serializable) window;
drawBorder(window); // --- Konverzió automatikus hívása
```

Jól látható, hogy a függvényparaméter típuslistája mind hosszában, mind az elemek sorrendjében eltér a paraméterként átadott objektumétól, a konverzió azonban így is működőképes. A rajzoló függvény meghívásánál a már ismertetett elvek szerint automatikusan meghívódik a *BorderType* konverziós konstruktora, egy ideiglenes, konvertált objektumot állítva elő a függvény *border* paraméterének. Futása

során először a *border* objektum *Displayable* típusú részé kap értéket a *window* objektum szintén *Displayable* típusú része alapján, majd a rekurzív hívás következik. Mivel ekkor a listának már csak egyetlen eleme maradt, a rekurziót megszakító specializáció hívódik meg. A *border* objektum *Shape* típusú része kezdeti értéket kap a *window* objektum *Rectangle* típusú darabja alapján, ami egy objektum egyszerű konverzióját jelenti az ősosztályára. Ez egy érvényes átalakítás és már a C++ alapvető konverziós szabályai szerint működik.

A konverzió azonban nem csak a *CSet* osztály segítségével felépített típusokra működik, a kiinduló objektum tetszőleges lehet. Tegyük fel, hogy a *window* objektumot valaki más már létrehozta egy közönséges típussal definiálva. A konverzió ezzel az objektummal is pontosan ugyanúgy fog működni:

```
// --- Nem CSet alapú közönséges típusdefiníció
struct Window : public Rectangle, public Displayable,
    public Serializable { ... };

Window window;
drawBorder(window); // --- A konverzió változatlan
```

A megoldás jól látható előnyökkel rendelkezik. A fent megvalósított automatikus konverzió kiterjeszti a C++ típusrendszerét, lehetővé téve ezzel a fejezet elején ismertetett probléma elegáns megoldását. Valóban, az ábrákon (5.1 és 5.2) szaggatott nyíllal jelölt leszármazások a strukturális konverzió automatikus szimulálásával azonnal előállnak.

Természetesen a megoldásnak hiányosságai és hátrányai is vannak. Legfontosabb hátránya, hogy bár a konverzió kiindulási objektuma tetszőleges, a céljának mindenképp a *CSet* típus segítségével kell előállnia. Ezáltal a megoldás intruzív, tehát már meglevő programkód esetén a kód átalakítása nélkül nem használható, mindenképp a kód megváltoztatását igényli. További hátránya, hogy a C++ típusrendszeréből adódóan absztrakt osztályokra a konverzió nem használható, hiszen nem hozhatunk belőlük létre példányt. Virtuális kötést használó osztályokon pedig az objektumok csonkolása (slicing) lép fel, vagyis az objektum elveszít minden, a dinamikus típusának megfelelő információt, kizárólag a statikus típus adatai maradnak meg.

Bár a megoldás továbbra is intruzív marad, a többi hátrány az alábbiakban a

megoldás átalakításával javításra kerül.

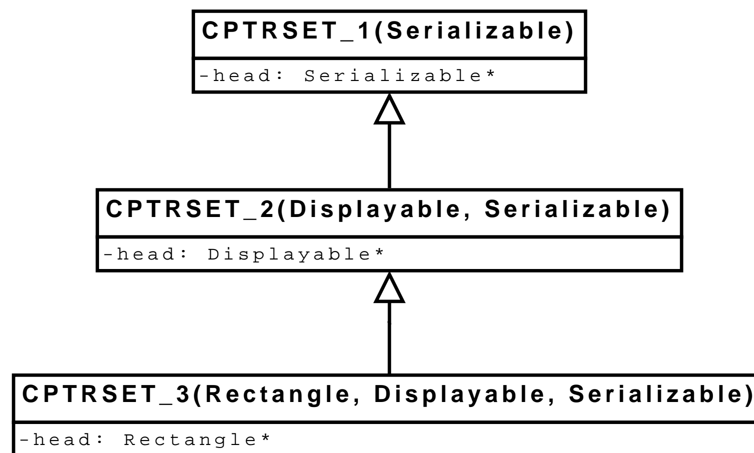
5.3.4. Egy továbbfejlesztett megvalósítás mutatókkal

A C++ nyelv az objektumok dinamikus típusának kezelését érték szerint tárolt változók esetén nem támogatja, kizárólag mutatók és hivatkozások (referenciák) esetén. Egy objektumra állított hivatkozás a későbbiekben már nem állítható másik objektumra, ez pedig megakadályozná a már bemutatott konverziós értékadó operátor megvalósítását. Következésképpen a javított megoldást mutatókra kell építeni.

Ebben a megoldásban a fent bemutatott többszörös öröklődésre épülő hierarchia helyett egyszerű lineáris leszármazási láncot építünk, a kieső leszármazási relációt pedig aggregációval fogjuk helyettesíteni. Ezt a *CPtrSet* nevű osztállyal valósítjuk meg, melyben az elnevezés arra utal, hogy az osztály mutatók egy halmazával dolgozik:

```
// --- Elődeklaráció általános paraméterre
template <class ListOfTypes> struct CPtrSet;

// --- Specializáció tetszőleges típuslistára
template <class Head, class Tail>
class CPtrSet< Typelist<Head,Tail> > : public CPtrSet<Tail>
{
    Head* head;
    ...
};

// --- Specializáció egyelemű típuslistára
template <class Head> class
CPtrSet< Typelist<Head,NullType> >
{
    Head* head;
    ...
};
```

Mint látható, a *CPtrSet* nem leszármazik a típuslista aktuális típusából, hanem egy mutató adattagot tárol rá. A rekurzív leszármazási szerkezet továbbra is változatlan marad. Mivel a *CPtrSet* osztálysablon paraméterei továbbra is ugyanolyan típuslisták, ezért a fent ismertetett módon bevezethetjük a *CPTRSET_1*, ..., *CP-*



5.4. ábra. A mutatókkal megvalósított osztályhierarchia

TRSET_N makrókat a típusdeklarációk megkönnyítésére. A korábbi példában bemutatott *window* objektum típusának (5.3 ábrán látható) leszármazási hierarchiája a *CSetPtr* segítségével jelentősen egyszerűsödik a következő 5.4 ábrán láthatóra:

Az előző megoldás konverziója is alkalmazható marad. Az elv annyiban változik, hogy a konverzió során nem az aktuális objektumrész egésze íródik felül érték szerinti másolással, mindössze a mutatót állítjuk át a konvertálandó objektum egy megfelelő darabjára. Mivel a rekurzió és egyéb alapelvek nem módosulnak, ezért a megvalósítás további magyarázat nélkül is könnyen érthető. Itt is csak a többelemű listákra használt konverziót mutatjuk be, amiből az egyelemű listák algoritmusa a rekurzió (*Rest* hivatkozások) elhagyásával könnyen megkapható:

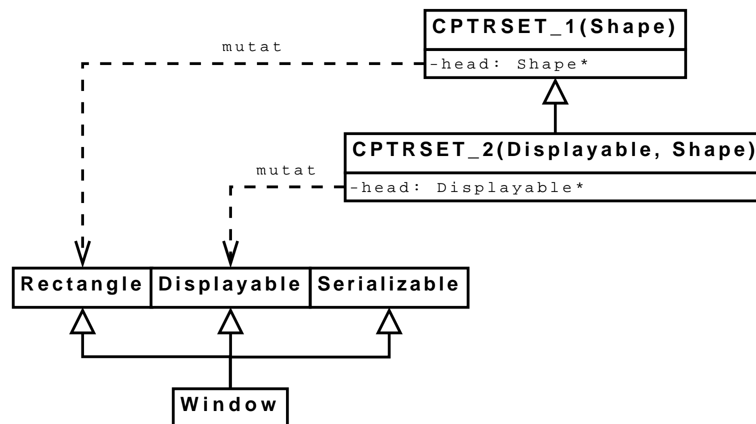
```
template <class Head, class Tail>
class CPtrSet< Typelist<Head,Tail> > : public CPtrSet<Tail>
{
    Head* head;

public:
    // --- Kisegítő típusdefiníciók
    typedef CPtrSet<Tail> Rest;
    typedef CPtrSet< Typelist<Head,Tail> > ThisType;

    // --- Alapértelmezett konstruktor és destruktor
    CPtrSet() : Rest(), head() {}
    virtual ~CPtrSet() {}
```

```
    // --- Konverziós konstruktor
    template <class FromType>
    CPtrSet(FromType& from) : Rest(from),
        head(& static_cast<Head&>(from) ) {}

    // --- Konverziós értékadás operátor
    template <class FromType>
    ThisType& operator = (FromType& from)
    {
        head = & static_cast<Head&>(from);
        Rest::operator= (from);
        return *this;
    }

    // --- Konverziós operátor a fejelemre
    operator Head& () const { return *head; }
    operator Head* () const { return head; }
};
```

A megvalósítás nagyon hasonló az előzőhöz, a mutató adattag mellett azonban két dologban alapvetően eltér. Mivel a *CPtrSet* már nem származik le a lista fejeleméből. ezért a fejelem típusára történő konverziót többé már végzi el magától a fordító. Az automatikus konverzió biztosításához a konverziós operátort kézzel kellett megadnunk. Másik technikai változás, hogy bizonyos esetekben (például ha a konvertálandó *from* objektum típusa maga is *CPtrSet*) a konstruktor és értékadó operátor működéséhez feltétlenül szükséges az explicit *static_cast* konverzió. Ennek az az oka, hogy mutatókra (a *&from* eredményére) már nem hívódna meg a konverziós operátor, a konverziós hívást ezért külön ki kell írni.

Az előző megoldás példáját követve itt is szemléltetjük, mi történik a konverzió során. Lenti példánkban a *BorderType* immár *CPtrSet* típusra van definiálva. A *drawBorder* eljárás emiatt annyiban változik, hogy már felesleges referencia szerint átvennie a paraméterét, hiszen a paraméter maga is mutatókat tartalmaz, azok másolása pedig nem költséges. A példa többi része teljesen azonos marad, a konverzió menete azonban kissé megváltozott. A konverzió működési elve az ábráról (5.5) olvasható le.

```
// --- Kirajzoló eljárás
```



5.5. ábra. A CPtrSet működési elve

```
typedef CPTRSET_2(Displayable, Shape) BorderType;
void drawBorder(BorderType border) { ... }

// --- Közönséges típusdefiníció
struct Window : public Rectangle, public Displayable,
    public Serializable { ... };

Window window;
drawBorder(window); // --- A konverzió változatlan
```

A *border* objektum létrejöttekor a *CPtrSet* sablon többelemű típuslistákra specializált konstruktora hívódik meg. A konstruktor rekurzívan meghívja a lista maradékával az ősének konstruktorát, majd a paraméterül kapott *window* objektumra állítja a *head* mutatót. Mivel a mutató típusa *Displayable,* ezért az objektum ezen típusú részére fog mutatni. A rekurzió során az ősosztály már az egyelemű listákra specializálódott változat lesz, konstruktora megszakítja a rekurziót. Az ősosztály a *Shape* típusú mutatóját az objektum *Rectangle* típusú részére állítja, mivel az a *Shape* leszármazottja.

Sajnos minden megoldáshoz hasonlóan ez is rendelkezik hátrányokkal. Továbbra is intruzív, ellenben a mutatók bevezetése két kisebb kényelmetlenséget is magával hoz. Egyik, hogy ha a hivatkozott objektum valahogyan valahogyan elpusztul, a mutatókra érvénytelenné válnak, használatuk meghatározatlan viselkedést eredményez. Ez azonban nem tér el a C++ nyelv mutatóinak alapvető tulajdonságaitól, így nem nevezhető hibásnak.

Másik kényelmetlenség, hogy a *CSetPtr* már nem származik le közvetlenül a típuslista osztályaiból, így rajta nem hívhatók közvetlenül azok műveletei. Mivel a konverziós operátorok taghivatkozások esetén (mint a pont vagy nyíl operátor hívása) nem hívódnak meg automatikusan az objektumra (az operátor baloldai paraméterére), ezért a hívás előtt szükség van egy explicit konverzióra. Például ha a *Displayable* osztály rendelkezik egy *draw* eljárással, a *drawBorder* törzséből azt a következőképpen érhetjük el:

```
void drawBorder(BorderType border)
{
    // --- A border.draw() fordítási hibát okozna
    Displayable &displayRef = border;
    displayRef.draw();
}
```

E kényelmetlenségeket azonban kompenzálja, hogy az előző megoldás minden előnye megmaradt, egyes hátrányai pedig megjavultak. A mutatók alkalmazásával elkerültük a konverzió során az objektumok érték szerinti másolását, ezáltal a csonkolásukat, megtartottuk az objektumok dinamikus típusát, ezáltal adattagjaikat és függvényeik dinamikus kötését is. A bemutatott megoldás kizárólag a C++ nyelv alapvető metaprogramozási eszközeit használta, így nem igényel semmiféle nyelvi kiterjesztést.

5.4. Összegzés

Láttuk, hogy a metaprogramozás hasznos eszköznek bizonyul a típusrendszer kiegészítése esetében.

TODO

5.5. Kapcsolódó eredmények

TODO

yossi gil sql [54]

szignatúrákhoz hasonlóan a concept-ek [26] is jól használhatók lesznek

6. fejezet

Szerializáció (20 oldal)

6.1. A probléma leírása

metaadatok alapján objektumok sorosítása gyakori metaprogramozási alkalmazás

XML séma alapján automatizált sorosítás támogatása ritka

Felhasználás: erre legfőképp az XML-alapú protokollokon alapuló hálózati szolgáltatásoknál (web service) van szükség

6.1.1. XML séma

TODO: irodalomjegyzek hivatkozas az XSD szabvanyra

alaptípusok ismertetése

konstrukciók ismertetése

6.2. Lehetséges megközelítések

gyakori az RTTI-vel rendelkező nyelveknél, pl. Java vagy C# sorosítási alapkönyvtárai

ezeknél osztályok leírása adott, ennek alapján különböző formátumokra sorosíthatunk

boost::serialization [16]: ervenytelen formatumu bemenet eseten elhasal

könyvtár C++ metaprogramokkal [7]

Java, C# serialization

Mi most az ellenkező irányból indulunk ki: XML sémaleírás adott, különböző nyelvekre sorosítunk

gsoap: kódgenerálás minden típushoz

TODO: Peti által emlegetett új rendszereket ideidézni, linkek:

http://www.codalogic.com/lmx/

http://codesynthesis.com

6.3. Megoldás

Xml Data Binding könyvtár linuxra és symbianra

symbian forráskód megtalálható: Nokia WSDL-to-C++ Wizard for S60 (TODO irodalomjegyzék bejegyzés és hivatkozás)

XSLT nyelvű (TODO: irodalomjegyzék hivatkozás) kódgenerátor

XML nyelvű séma leírás -> kódgenerátor -> XSD típusú önleírással rendelkező C++ adattároló típusok

XML dokumentum <- általános szerializációs algoritmus <-> generált C++ adattároló típusok

6.3.1. Típusmodell megadása

XML séma alaptípusainak és típuskonstrukcióinak megfeleltetése az adott nyelv típusrendszerébe

Alaptípusok. hozzárendelést lásd 6.1

Típuskonstrukciók. Konténer (sequence, all, choice)

Element, attribute

ComplexType (Extension, SimpleContent)

SimpleTypeRestriction / TypeList / TypeUnion

Példa generált osztályra. TODO

XSD alaptípus	Hozzárendelt C++ típus
Boolean	bool
Byte	signed char
Binary	std::string (base64 kódolással)
Duration	::Impl::Duration (saját megvalósítás)
Date	::Impl::DateTime (saját megvalósítás)
DateTime	::Impl::DateTime (saját megvalósítás)
Decimal	double
Float	float
Int	int
Long	long
QName	::Impl::QName (saját megvalósítás)
Short	short
String	std::string
Time	::Impl::DateTime (saját megvalósítás)
UnsignedByte	unsigned char
UnsignedInt	unsigned int
UnsignedShort	unsigned short
UnsignedLong	unsigned long long

6.1. táblázat. Az XSD alaptípusai és a hozzárendelt C++ típusok

6.3.2. Metaadatok generálása

séma alapján C++ nyelvű, futási idejű metaadatok generálása

TypedXmlData interfész ismeretése

pár szó, hogy a metaadatok miért struktúrák, miért nem objektumok

metaadatok formátuma (header fájlok kivonata, hierarchia)

példa XSD <-> metaadat megfelelésre

6.3.3. Osztályok generálása

esetleg rövid XSLT bevezető

kódgenerátor működési elvének szűkszavú ismertetése, esetleg néhány sor példakód

6.3.4. Sorosító algoritmus

kizárólag futási idejű, XSD alapú önleírással dolgozó általános sorosító (meta)algoritmus ismertetése

7. fejezet

Related (5-8 oldal)

TODO: ezt inkabb fejezetenkent lebontva volna erdemes csinalni, bar altalaban a generativ programozashoz is lehet ertelme kulon, altalanosabb fontos eredmenyeket ismertetni

Saját eredményeim ismertetése után a 7. fejezetben kitekintek a témában mások által publikált kutatásokra is, bemutatva a területen elért eredményeket és a lehetséges fejlődési irányokat.

Siek: G Type theory Szemantika

8. fejezet

Összegzés (TODO 3 oldal)

A dolgozat körüljárta a generatív programozási paradigmák, ezen belül is elsősorban a programátalakításokkal dolgozó metaprogramozás elméletét és alkalmazásait. Az elméleti bevezetés után saját munkáimat és eredményeimet ismertettem. Ezek a metaprogramozás legtöbb alapvető eszközével és alkalmazási területével foglalkoztak, bennük új modelleket, módszereket és lehetséges alkalmazási módokat mutattam be, melyek remélhetőleg hozzájárulnak majd a metaprogramozás további fejlődéséhez.

Mivel a metaprogramozás egy új, feltörekvő módszertan, így még nem rendelkezik kellőképpen kiforrott elméleti háttérrel. A programkód önvizsgálata (lásd **??**) a módszertan alapvető eszköze, ám alig akad olyan programozási nyelv vagy környezet, mely fordítási időben is támogatná. A 4. fejezet a támogatás elősegítését célozta meg. Bemutatott egy egyszerű elméleti modellt, és felvázolt hozzá egy lehetséges C++ nyelvű megvalósítást, mely csak minimális nyelvi kiterjesztést igényelt.

A metaprogramozás fontos alkalmazási területe a nyelvek típusrendszerének kiterjesztése, tulajdonságainak javítása. A 5. fejezet a C++ nyelv típusrendszerének strukturális öröklődést szimuláló kiterjesztésére mutatott be egy megoldást. A megoldás metaprogrammal automatizált kódgenerálásra épült, és kizárólag szabványos nyelvi eszközöket használt.

A metaprogramozás fontos alkalmazási területe az adatok sorosítása, melyre a típusok önleírásának (metaadatainak) felhasználásával egy jó általános megoldás adható. A típusok önleírása a virtuális gépen, illetve értelmezőn alapuló programozási

környezetekben (pl. Java, C#) többnyire adottság, így a sorosítás könnyebben megoldható, máshol nehezebb. A 6. fejezet a típusok önleírását alapvetően nem támogató C++ nyelven adott egy megoldást adatok automatikus, XML formátumú sorosítására. A megoldás hasznosságát tovább növelte, hogy kis erőforrásigénye miatt alkalmazása ideális lehet telefonok, kéziszámítógépek és egyéb, kisebb kapacítású rendszerek egymás közötti kommunikációjára.

Irodalomjegyzék

[1] István Zólyomi, Zoltán Porkoláb. Towards a General Template Introspection Library. Generative Programming and Component Engineering LNCS Vol. 3286 (2004) pp. 266-282.

[2] Zoltán Porkoláb, István Zólyomi. An anomaly of subtype relations at component refinement, and a generative solution in C++. MPOOL Workshop, ECOOP 2004, Oslo, pp. 39-44.

[3] Szabolcs Payrits, Péter Dornbach, István Zólyomi. Metadata-Based XML Serialization for Embedded C++. Proceedings of ICWS 2006, pp. 347-356

[4] István Zólyomi, Zoltán Porkoláb, Tamás Kozsik. An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming. Generative Programming and Component Engineering LNCS Vol. 2830 (2003) pp. 209-227.

[5] István Zólyomi, Zoltán Porkoláb. A Feature Composition Problem and a Solution Based on C++ Template Metaprogramming. Generative and Transformational Techniques in Software Engineering LNCS Vol. 4143 (2006) pp. 459-470.

[6] Zoltán Porkoláb, József Mihalicza, Ádám Sipos. Debugging C++ Template Metaprograms. Proceedings of GPCE 2006, The ACM Digital Library pp. 255-264.

[7] Yuriy Solodkyy, Jaakko Järvi, Esam Mlaih. Extending type systems in a library - type-safe XML processing in C++. In Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon, October 2006.

[8] Todd Veldhuizen. Using C++ template metaprograms. C++ Report Vol. 7 No. 4 (May 1995), pp. 36-43.

[9] Todd Veldhuizen. C++ Templates are Turing Complete. `http://ubiety.uwaterloo.ca/~tveldhui/papers/2003/turing.pdf`, 2003

[10] Todd Veldhuizen. Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[11] Todd Veldhuizen. Arrays in Blitz++. Proceedings of ISCOPE'98, Springer-Verlag, pp. 223-230.

[12] C. A. Gössl, N. Drory, J. Snigula. LTL - The Little Template Library. ASP Conference Proceedings 2004, Vol. 314, p. 456

[13] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)

[14] David Abrahams, Alexander Gurtovoy. C++ Template Metaprogramming Concepts, Tools and Techniques from Boost and Beyond. Addison-Wesley (2004)

[15] The Boost C++ Libraries. `http://www.boost.org`

[16] The Boost Serialization Library. `http://www.boost.org/libs/serialization`

[17] The Boost Metaprogramming Library. `www.boost.org/libs/mpl`

[18] The Boost Type Traits Library. `http://boost.org/libs/type_traits/index.html`

[19] Boost.Spirit Home. `http://spirit.sourceforge.net/`

[20] Boost.Xpressive: Advanced C++ Regular Expression Template Library. `http://boost.org/doc/libs/1_35_0/doc/html/xpressive.html`

[21] David Vandevoorde, Nicolai M. Josuttis. C++ Templates - The Complete Guide. Addison-Wesley (2002)

[22] Scott Meyers. Red Code, Green Code: Generalizing const. Northwest C++ Users Group, April 2007.

[23] Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley s(1994)

[24] Bjarne Stroustrup. The C++ Programming Language Special Edition. Addison-Wesley (2000)

[25] C++0x. C++ Standards Committee Papers. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/`

[26] Douglas Gregor, Bjarne Stroustrup. Specifying C++ concepts (Revision 1). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2081.pdf`

[27] Gerald Baumgartner, Vincent F. Russo. Implementing Signatures for C++. ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 19 Issue 1. 1997. pp. 153-187.

[28] Barbara H. Liskov, Jeannette M. Wing. A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, Vol. 16 No. 6 (Nov 1994), pp 1811-1841

[29] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall (1988)

[30] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. Proceedings of ECOOP, Finland. Springer-Verlag LNCS 1241, June 1997.

[32] The AspectJ Project. `http://www.eclipse.org/aspectj/`

[33] Luca Cardelli. Structural Subtyping and the Notion of Power Type. Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, January 1988. pp. 70-79.

[34] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. Proceedings of the 18th ACM SIGPLAN OOPSLA 2003, pp. 115-134.

[35] Krzysztof Czarnecki, Ulrich W. Eisenecker. Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)

[36] William Harrison, Harold Ossher. Subject-oriented programming: a critique of pure objects. Proceedings of 8th OOPSLA 1993, Washington D.C., USA. pp. 411-428.

[37] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. Technical Report, TR-CCTC/DI-35, GTTSE 2005, pp. 153-186.

[38] Harold Ossher, Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. IBM Research Report 21452, April, 1999. IBM T.J. Watson Research Center.

[39] Don Batory, Jia Liu, Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. Proceedings of the 9th European Software Engineering Conference, 2003.

[40] Don Batory, Jacob Neal Sarvela, Axel Rauschmayer. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, vol. 30, no. 6, pp. 355-371.

[41] Xavier Leroy. The Object Caml system, release 3.10. (May 2007) `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`

[42] Walid Taha, Tim Sheard. MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science, 2000, vol. 248, number 1-2, pp. 211-242

[43] MetaOcaml. `http://www.metaocaml.org/`

[44] Db4objects. `http://db4o.com`

[45] C. K. Duby, S. Meyers, and S. P. Reiss. CCEL: A metalanguage for C++. In USENIX C++ Conference, August 1992.

[46] Jaakko Järvi, Jeremiah Willcock, Andrew Lumsdaine: Concept-Controlled Polymorphism. In proceedings of GPCE 2003, LNCS 2830, pp. 228-244.

[47] Erwin Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.

[48] Walter Bright. D programming language. `http://www.digitalmars.com/d/`

[49] Walter Bright. Templates Revisited. `http://digitalmars.com/d/2.0/templates-revisited.html`

[50] Martin Fowler. Refactoring. Improving the Design of Existing Code. Addison-Wesley (1999)

[51] Mads Torgersen. The Expression Problem Revisited - Four New Solutions using Generics. Proceedings of ECOOP 2004, LNCS vol. 3086, pp. 123-146.

[52] James O. Coplien. Multiparadigm Design for C++. Addison-Wesley (1999)

[53] David R. Musser, Alexander A. Stepanov. A library of generic algorithms in Ada. Proceedings of ACM SIGAda 1987, pp. 216-225

[54] Yossi Gil, Keren Lenz. Simple and safe SQL queries with c++ templates. Proceedings of GPCE 2007, ACM, pp. 13-24

[55] Shan S. Huang, David Zook, Yannis Smaragdakis. Morphing: Safely Shaping a Class in the Image of Others. Proceedings of ECOOP 2007, pp. 399-424.

[56] Shigeru Chiba. OpenC++. `http://opencxx.sourceforge.net/`

[57] Stratego Program Transformation Language. `http://strategoxt.org/`

[58] SWIG (Simplified Wrapper and Interface Generator). `http://www.swig.org/`

[59] Valentin F. Turchin. A supercompiler system based on the language REFAL. SIGPLAN Not. 1979, vol. 14, num. 2, pp. 46-54.

[60] Valentin F. Turchin. The concept of a supercompiler. ACM Trans. Program. Lang. Syst. 1986, vol. 8, num. 3, pp. 292-325.

[61] Jens Peter Secher, Morten Heine Sørensen. On Perfect Supercompilation. Proceedings of Perspectives of System Informatics 1999, LNCS vol. 1755, pp 113-127.

[62] Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press (1991)

[63] Thierry Géraud, Roland Levillain. Semantics-Driven Genericity. A Sequel to the Static C++ Object-Oriented Programming Paradigm (SCOOP 2). Proceedings of International Workshop on Multiparadigm Programming with Object-Oriented Languages, 2008.

[64] Dirk Draheim, Christof Lutteroth, Gerald Weber. A Type System for Reflective Program Generators. Proceedings of GPCE 2005, pp. 327-341.

[65] Christof Lutteroth. AP1: A Platform for Model-Based Software Engineering. PhD Theses, University of Auckland, 2008,

[66] Zoltán Porkoláb, József Mihalicza, Ádám Sípos. Debugging C++ Template Metaprograms. Proceedings of GPCE 2006, pp. 255-264.

[67] TODO: Zoltán Porkoláb. Metrikák.

[68] Csörnyei Zoltán. Fordítóprogramok. Typotex, 2006 (ISBN 963 9548 83 9)

[69] Fóthi Ákos. Bevezetés a programozáshoz. ELTE Eötvös Kiadó 2007 (ISBN: 963 463 833 3)

[70] AndroMDA. `http://www.andromda.org/`

[71] SOAP Specifications. `http://www.w3.org/TR/soap/`

[72] Microsoft .Net Framework. `http://www.microsoft.com/net/`

[73] Java Developer Network. `http://java.sun.com/`

[74] jContractor: Bytecode instrumentation techniques for implementing design by contract in Java. In Proceedings of Second Workshop on Runtime Verification, 2002. Electronic Notes in Theoretical Computer Science: `http://www.elsevier.nl/locate/entcs`