Metaprogramok alkalmazása erősen típusos objektum-orientált rendszerek kiterjesztésére

Zólyomi István

Doktori értekezés
2009

Doktori értekezés

Metaprogramok alkalmazása erősen típusos objektum-orientált rendszerek kiterjesztésére

Zólyomi István

Témavezető: Dr. Porkoláb Zoltán

Eötvös Loránd Tudományegyetem
Informatika Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

ELTE Informatika Doktori Iskola
Az informatika alapjai és módszertana doktori program

Iskola- és programvezető: Dr. Demetrovics János

Budapest
2009

Tartalomjegyzék

1. Bevezetés **11**

1.1. A dolgozat felépítése 12

2. Metaprogramozás **15**

2.1. Definíciók 16

2.2. Metaadatok 17

2.2.1. Hagyományos adatok 17

2.2.2. Típusok 17

2.2.3. Elemi vizsgálatok 18

2.2.4. Típusleírók 19

2.2.5. Szintaxisfák 19

2.3. Programtranszformációk 20

2.3.1. Metaadatok olvasása 20

2.3.2. Fordítás 21

2.3.3. Kódgenerálás 21

2.3.4. Bővítés 22

2.3.5. Kiterjesztés 22

2.3.6. Átszervezés 23

2.3.7. Teljes átalakítás 23

2.4. Alkalmazások 24

2.4.1. Információ kinyerése 24

2.4.2. Hagyományos alkalmazások 25

2.4.3. Speciális (grafikus, beágyazott és többszintű) nyelvek 25

2.4.4. Konverzió, sorosítás, adattárolás 26

2.4.5. Kommunikáció, szolgáltatások elérése 27

2.4.6. Optimalizálás 27

2.4.7. Átszervezés 28

2.4.8. Típusrendszer vizsgálata és átalakítása 28

2.4.9. Aktív könyvtárak 29

2.5. Metaprogramozási környezetek 29
2.5.1. A C++ nyelv sablonjai 30
2.5.2. A D nyelv sablonjai 31
2.5.3. Virtuális gépek (Java és C#) 33
2.5.4. Ruby és más szkriptnyelvek 34
2.5.5. MetaML és MetaOCaml 35
2.5.6. Stratego/XT 36

3. Típusok tulajdonságainak vizsgálata 37
3.1. A probléma leírása 38
3.2. Megoldás 39
3.2.1. Felhasznált metaprogramozási eszközök 39
3.2.2. Saját eredmények 42
3.2.3. Predikátumok megvalósítása 44
3.2.4. Predikátumok kompozíciója 50
3.2.5. Vizsgálatok eredményének felhasználása 51
3.3. Kapcsolódó munkák 52
3.3.1. Kitekintés más nyelvekre 53
3.3.2. C++ nyelvű megközelítések 54
3.4. Összegzés 56

4. A típusrendszer kiterjesztése 59
4.1. A probléma leírása 59
4.1.1. Formális leírás 64
4.2. Lehetséges megközelítések 64
4.2.1. Hagyományos öröklődés 65
4.2.2. Virtuális öröklődés 65
4.2.3. Aspektusok 65
4.2.4. Szignatúrák 66
4.2.5. Strukturális altípusosság 66
4.3. Megoldás 67
4.3.1. Típuslisták 67
4.3.2. Osztályok kompozíciója 68
4.3.3. A strukturális altípusosság megvalósítása 71
4.3.4. Egy továbbfejlesztett megvalósítás mutatókkal 75
4.4. Kapcsolódó munkák 78
4.5. Összegzés 79

5. Sorosítás és távoli szolgáltatások **81**
5.1. A probléma leírása 81
5.2. Kapcsolódó munkák 82
5.3. Megoldás 83
5.3.1. Alapelvek, áttekintés 83
5.3.2. Kódgenerátor 85
5.3.3. Metaadatok 87
5.3.4. Típusmodell 88
5.3.5. Sorosító metaprogram 91
5.3.6. Műveletek generálása távoli szolgáltatásokhoz 92
5.4. Összegzés 94

6. Összegzés **95**
6.1. A dolgozat eredményei 96
6.2. További kutatási irányok 97

A. Összefoglalás **99**

Ábrák jegyzéke

4.1. Példa több interfész megvalósítására . . . 61
4.2. A kifejezésprobléma . . . 62
4.3. Példa a C++ STL könyvtárából . . . 63
4.4. A többszörös öröklődéssel felépített osztályhierarchia . . . 70
4.5. A mutatókkal megvalósított osztályhierarchia . . . 76
4.6. A CPtrSet működési elve . . . 78

5.1. Az MXDB felépítése . . . 84
5.2. Az MXDB működési elve . . . 85

1. fejezet

Bevezetés

Az objektum-orientált programozás kipróbált, több évtizedes múlttal rendelkező, bevált paradigma, mely egyértelműen napjaink legelterjedtebben használt fejlesztési módszertanává vált. Megannyi erénye mellett nyilvánvalóan számos korláttal is rendelkezik, melyek megoldása az idők folyamán új elveket és módszertanokat ihletett. A dolgozat célja az objektum-orientált típusrendszerek problémáinak vizsgálata, a típusrendszer kiterjesztése, illetve alkalmazhatóságának javítása különböző módszerekkel.

A lehetséges utódok közül legígéretesebb módszertannak talán a generatív programozás tűnik, mely tudományos szemmel mérve még új, feltörekvő paradigma. Az objektum-orientált módszertant nem elveti, hanem annak kereteit új eszközökkel próbálja meghaladni. A generatív programozás csak célját tekintve egységes paradigma: a programozási folyamat minél hatékonyabb automatizálását tűzi ki célul. Módszereit tekintve azonban rendkívül sokrétű, 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 eseteként is értelmezhető. A dolgozat a generatív programozás, ezen belül is elsősorban a metaprogramozás előnyeire, lehetőségeire alapoz.

A dolgozat több alapproblémát mutat be, ahol a metaprogramozás használata jelentősen képes javítani az alapjául szolgáló objektum-orientált típusrendszeren. A dolgozatban túlnyomórészt használt objektum-orientált nyelv a C++, melynek sablon (template) nevű eszköze komoly kifejezőerővel bír, és lehetővé teszi a fordítási idejű metaprogramozást. Noha több más nyelv (például a Java vagy C#) is hasonló mértékben elterjedt, ráadásul a szabványos C++ nyelvet jóval meghaladó könyvtártámogatással rendelkezik, a C++ alapvető eszközkészlete rendkívül erős, néhány kivételtől eltekintve más objektum-orientált nyelvek minden eszköze közvetlenül támogatott, esetleg szimulálható.

1.1. A dolgozat felépítése

A 2. fejezetben modellt adok a metaprogramozás módszertanára. Alapelveinek és lehetőségeinek ismertetése után kitérek a metaprogramozás alkalmazásaira, valamint a főbb metaprogramozási rendszerek bemutatására is. Az átfogó bevezetés után kutatásaim fontosabb eredményeit ismertetem publikációim alapján.

A 3. fejezetben publikációimra [1, 2] építve a metaadatok egy elsőrendű logikán alapuló modelljét vázolom fel, mely a programkód fordítási idejű önvizsgálatát támogatja. Segítségével elemi vizsgálatok és különböző kompozícióik alapján egy általános kódvizsgáló rendszer építhető a fordítóprogramon belül, ezzel széleskörű lehetőségeket biztosítva a metaprogramozás számára. A fejezetben megmutatom a rendszer C++ nyelvbe illeszthetőségének lehetőségét a template-metaprogramozás eszközeivel, valamint megadom a fő alkotóelemek megvalósítását. A modell végül a jelenleg formálódó C++0x szabványban bevezetésre kerülő concept nevű nyelvi elemhez hasonló eszközt ad, ám jobban támogatja a vizsgálati eredmények felhasználását metaprogramokban, és nem igényel nyelvi kiterjesztést.

A 4. fejezetben a széleskörűen használt objektum-orientált nyelvek öröklődés alapú típusrendszerének korlátait mutatom be, különös tekintettel az explicit módon jelölt altípusosság anomáliáira. Ismertetek néhány esetet, melyek alapvetően a többszörös öröklődésből, vagy több interfész megvalósításából adódnak, és a programozási környezetek gerincéül szolgáló rendszerkönyvtárak tervezésénél is komoly problémákat okoznak. Ezekre megoldást nyújthat a több nyelvben használt, ám a gyakorlatban széleskörűen még nem elterjedt strukturális altípusosság [41] alkalmazása, mely implicit leszármazási szabályokra épül. Korábbi kutatásaimra [5, 6] építve bemutatok egy C++ nyelvű, sablon-metaprogramozáson alapuló megoldást, mellyel a meglevő típusainkhoz kiegészítő típuskonverziók nyújthatók a strukturális konverzió megvalósítására. Ezáltal a meglevő típusmodell kiegészítéseként a strukturális altípusosság természetes módon, nyelvi kiterjesztések nélkül a típusrendszerhez illeszthető.

Az 5. fejezetben [3, 4] alapján bemutatom a futási idejű önleirást (reflection) biztosító típusok előnyeit 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. Bemutatok egy ezen elvekre épülő keretrendszert, mely sorosítást (serialization) és távoli eljáráshívást (remote procedure call) biztosít önleíró objektum-orientált programnyelvbeli adatok és sémaleírással rendelkező XML dokumentumok között. A megoldás a használat szempontjából emlékeztet a C# és a

Java nyelveken eleve rendelkezésre álló sorosító könyvtárakra, ám más alapokon nyugszik, továbbá az önleíró típusokat alapvetően nem támogató C++ nyelvhez készült. Felépítéséből adódóan a megoldás kis memóriaköltségű, ezáltal kiválóan alkalmazható beágyazott vagy szűkebb erőforrásokkal rendelkező rendszerekben.

2. 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, a dolgozatban ilyen átalakításokkal nem foglalkozunk.

Hogy a dolgozat tárgyát jobban megérthessük, először a metaprogramok pontosabb meghatározására és egy egységes terminológiára van szükség, melyet 2.1 alatt tárgyalok. Ezután a metaprogramozás alapelveit és alapvető eszközeit mutatom be.

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 2.2 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 2.3 tárgyalja.

Ezután 2.4 alatt a metaprogramozás alkalmazásának különböző területeit járom körül. Végül 2.5 alatt bemutatom egyes, szélesebb körben elterjedt objektum-orientált vagy különleges, metaprogramozás szempontjából fontos programozási nyelvek, rendszerek meta-

programozási képességeit.

2.1. 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 [79] által leírt fogalmakat és elméleti modellt fogjuk felhasználni. Eszerint a program *állapotátmenet*ek (elemi transzformációk) sorozata, ezek értelmezési tartománya és értékkészlete is az *állapottér* nevű halmaz. 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. Metaadat alatt általában adatok leírását értik, a fogalom alapvetően nem kötődik a programozáshoz. Mivel mi a programozási szempontból érdekes metaadatokkal foglalkozunk, ezt leszűkítjük, és csak a programok leírásával, vagyis metaadataival foglalkozunk. Mi a metaprogram programfüggvényének argumentumait, vagyis a metaprogram bemeneteinek összességét nevezzük metaadatoknak.

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* programok is lehetnek paraméterek. 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 vagy annak egy részhalmaza. Más szóval kimenetének része egy program. A dolgozatban ennél megengedőbbek leszünk, és kódgenerátornak nevezzük azokat a metaprogramokat is, melyek kimenete csak egy kisebb programrészlet, például típusok vagy változók definíciója.

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

2.2. Metaadatok

A metaadatok jelentősen különbözhetnek egy hagyományos program bemenetétől, hiszen 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 például fordítás közben hajtódnak végre, és a bemenő programnak a fordítóprogram által felépített reprezentációjáról próbálnak adatokat kinyerni, mint például a C++ (lásd 2.5.1) 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 teljes leírását.

2.2.1. 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 [57]. 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 2.3.3) esetén lehet, melyek kimenete kizárólag ilyen egyszerű adatokon alapul.

2.2.2. Típusok

Ha a metaprogram bemenete egy típus is lehet (például a feldolgozott nyelv egy osztálya), már lehetőségünk van típussal paraméterezett függvényeket vagy osztályokat készíteni,

mely a generikus programozás paradigmájának alapja. A típusparaméterek jelentik az egyik legegyszerűbb eszközt, mellyel a felhasználható programkonstrukciók kifejezőereje jelentősen növelhető [1]. Mivel a paraméterezett programrészekhez számtalan különböző típusparamétert rendelhetünk, a létrehozható példányok száma nem korlátos, maga a példányosítás viszont rendkívül egyszerű. Ezzel a programok bonyolultsága, vagyis a programozói munka jelentősen csökkenthető. Ez az absztrakciós szint teljesen elfogadott és széles körben elterjedt, sablon néven (*generic* vagy *template)* a legtöbb modern programozási nyelv támogatja (lásd 2.5).

2.2.3. 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 típusok elemi tulajdonságait vizsgálják, s ezek kombinációját fordítási idejű feltételekben felhasználva döntenek. A szóba jöhető elemi tulajdonságok rendkívül változatosak lehetnek, az alábbiakban összegyűjtve néhány példa olvasható a leggyakrabban használt tulajdonságokról:

- 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. Változáskövetést támogató típusok esetén megtudhatjuk a verziót is.

- Változók esetében lekérdezhetjük a típust, továbbá annak minden fenti tulajdonságát. Adattagok esetén emellett megtudhatjuk a tartalmazó objektum kezdetéhez képest számított relatív címet (offset) is.

- Függvények esetében egy elemi lekérdezés megadhatja a szignatúrát, vagy információt nyújthat a hívási konvenciókról. Tagfüggvények esetén eldönthetjük, hogy

[1] A programrészek típussal paraméterezése a kétszintű nyelvtanokhoz rendkívül hasonló elven alapul. A nyelvtanok esetén bizonyított a nyelv kifejezőerejének növekedése, típussal paraméterezett programok esetén azonban továbbra sem lépjük túl a Turing-gépek lehetőségeit. Esetünkben a program leírásában alkalmazható nyelvi szabályok, konstrukciók kifejezőereje nő, vagyis jóval több funkcióval bíró, bonyolultabb program írható ugyanolyan terjedelemben.

dinamikus kötéssel rendelkezik-e, felüldefiniálható-e, vagy módosítja-e az objektumot, melynek tagfüggvénye.

Ezek az információk általában nem könnyen hozzáférhetők, a C++ nyelvhez a Boost könyvtár [26] próbálja az elemi vizsgálatok minél nagyobb részhalmazát megvalósítani. Legtöbbször azonban a lentebb olvasható típusleírókba beépülve láthatjuk őket.

2.2.4. 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 (reflection) jóval többet nyújt az elemi tulajdonságoknál, hiszen tulajdonságok kizárólag már ismert szimbólumokról kérdezhetők le. Önleírással azonban maguk a szimbólumok is felderíthetők, mivel 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 2.5.3), vagy a szkriptnyelvek szinte mindegyike (lásd a Python inspect modulját, vagy a Ruby objektumait 2.5.4 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 (a típus neve) helyett a valódi típust kapjuk meg. Ez tehát a típusleírásnak egy jóval erősebb formája, hiszen a futási idejű leírásokat könnyen előállíthatjuk a fordítási idejűekből (például a fordítási időben kiolvasott adatokat változókba és struktúrákba helyezzük el, ezek futás közben is olvashatók). Az önleírásból kinyert adatokat azonban használhatjuk metaprogramok paramétereiként is, ezzel jelentősen megnövelve azok kifejezőerejét. Ráadásul egy metaprogram a típusbiztosság minden előnyével dolgozik, hiba esetén futási idő helyett még fordítási időben kapunk hibát. Bár megvalósítására több kiterjesztés létezik különböző nyelvekhez (pl. OpenC++ [66]), kevés nyelvben van hozzá közvetlen támogatás, ilyen például Lisp ([72]), vagy a D nyelv (2.5.2).

2.2.5. 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 sablonok specializációját támogató nyelveken alkalmazható, például a C++ (2.5.1) vagy D (2.5.2) nyelv. Alapötlete szerint a fa belső csúcsai speciális osztálysablonokkal 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 eredményül kapott kifejezésfa specializált sablonok segítségével bejárható, feldolgozható, maga a kiértékelő művelet megvalósítása is erre épül. Részletesen [18] tárgyalja, alkalmazásai általában a programkód hatékonyságát növelik (lásd 2.4.6).

2.3. 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.

2.3.1. 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 2.4.1).

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.

2.3.2. Fordítás

Bár a fordítóprogramok (2.4.2) több évtizeddel öregebbek a metaprogramozás paradigmájánál, mégis szépen illenek annak elméletébe. A fordítás során egy magas absztrakciós szintű jelölésrendszer[2], 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.

A fordítás rendkívül összetett lépés, szükség van hozzá a bemenő program összes metaadatának elérésére, másrészt ezek teljes elemzésére és feldolgozására is. Bár a fordítás meglehetősen bonyolult, metaprogramozási kifejezőereje minimális, hiszen csak ekvivalens átalakításokat végez. A fordítás nevezhető a transzformációk legelterjedtebb formájának, gyakorlatilag minden számítástudománnyal foglalkozó képzés részét képezi, a dolgozatban nem is tárgyaljuk bővebben.

2.3.3. Kódgenerálás

E transzformációs szint elnevezése szándékosan azonos a metaprogram kategóriájának (2.1) nevével, jelezvén, hogy a kódgenerátor metaprogramok ezen az átalakítási szinten dolgoznak. A kódgenerálás alkalmazásai általában jóval egyszerűbbek a fordításnál, és nincs hozzájuk szükség az összes metaadatra, például szintaxisfákat tipikusan nem dolgoznak fel.

A kódgenerálás a gyakorlatban széles körben elterjedt átalakítási forma. Alkalmazása gyakori sémaleírások (Xml, Sql, stb) alapján történő konverziós kód generálásánál (lásd 2.4.4). Egy másik alkalmazása során függvénydefiníciók vagy a szolgáltatások absztrakt leírása alapján generálunk kódot a szolgáltatás távoli elérésére, így ezt a feladatot a szolgáltatást használóknak már nem kell megoldania (lásd 2.4.5).

[2] 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 modell-vezérelt programépítés (modell driven architecture).

A kódgenerátorok másik elterjedt fajtáját az elemzőprogram-generátorok jelentik. Ezek bemenetként egy nyelv formális leírását kapják, kimenetként pedig egy olyan programot készítenek, mely képes az adott nyelv teljes elemzését elvégezni. Ezek a fordítóprogramokhoz hasonlóan erős elméleti háttérrel rendelkeznek, és általában kitérnek rájuk a formális nyelvek oktatásánál.

2.3.4. Bővítés

Gyakran van szükségünk arra, hogy egy korábban megírt kódot annak módosítása nélkül kiegészítsünk, viselkedését megváltoztassuk. A bővítés ezt a transzformációt teszi lehetővé. A bővítésekhez 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 nem intruzív programkód segítségével! Ilyenek például a C nyelv makrói [22], a C++ névterei, globális operátorai és specializálható sablonjai, az AspectJ [39] aspektusai, a C# bővítő metódusai (extension method [11]), vagy akár a Ruby osztálydefiníciói. Mind képesek már létező definíciók bizonyos mértékű megváltoztatására, azonban ezt kizárólag új kód hozzáadásával érik el. Vegyük észre, hogy az ilyen eszközök maguk is a metaprogramozás magasabb szintű transzformációinak beépített nyelvi támogatását jelentik.

Gyakran használt programtranszformációs technika. Számtalan célja lehet, ilyen például konverziós algoritmusok készítése a típusok más formátumra alakításához (lásd 2.4.4) adattároláshoz vagy egy más formátumot használó alkalmazásban történő felhasználáshoz.

2.3.5. Kiterjesztés

A metaprogramok kifejezőerejét tovább bővíthetjük, ha megengedjük, hogy a módosítás intruzív legyen, azaz megengedjük az eredeti kódrészlet kiegészítését, bővítését. 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 2.5.3) kiterjesztésére épülő megoldások népszerűek. Ilyen az AspectJ [40]

aspektus-átszövő mechanizmusa, vagy az MJ [65] osztályátalakítása (class morphing).

Ennek a transzformációnak egy korlátozott formája a programozási nyelvekben általában osztályok közti leszármazással megvalósított programkód-újrafelhasználás. Ennek során az eredeti típus nem módosul, a változások által egy új altípust képzünk. Ezen kívül is gyakran láthatunk rá közvetlen nyelvi támogatást, ilyen egyes nyelveken például a típusok önleírásának (2.2.4) automatikus biztosítása.

2.3.6. Átszervezés

Az átszervezések (refactoring) esetén a legtöbb korlátozás megszűnik az átalakítások módjával kapcsolatban. Egyetlen kikötés, hogy az átalakítások eredményeképp a program kívülről megfigyelhető működése nem változhat meg. Ilyen átalakítások például egy osztály tagjainak átnevezése, kódrészek általánosítása paraméterezhetőség bővítésével, ismétlődő kódrészletek függvénnyé alakítása vagy a futás során elérhetetlen kódrészletek törlése. Segítségével javítható a kód strukturáltsága, teljesítménye, vagy bonyolultsága, alkalmazásainak leírását lásd 2.4.7 alatt.

A korlátozás jellegéből adódóan kivitelezése rendkívül nehéz. A legtöbb programtranszformációval ellentétben a szintaktika elemzése nem elégséges, a fordításhoz hasonló 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, többnyire csak egyszerűbb, a fent említettekhez hasonló átszervezési feladatokat oldanak meg.

2.3.7. Teljes átalakítás

A kikötésektől mentes programtranszformációk szintje, lehetővé téve a programkód 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. Rendkívül nehéz az alkalmazás, vagyis a szükséges átalakítások pontos meghatározása is.

A szabad átalakításokat támogató rendszerek ritkák, mivel megfelelő módszertan híján fejlesztésük a módszer teljes kidolgozását is igényli. Régebben nyelvspecifikus rendszerek készültek, ilyen például C++ nyelven az OpenC++ [66] nevű metaprogramozási rendszer. A későbbiekben születtek nyelvfüggetlen metaprogramozást támogató módszerek és eszközök is, például a Stratego/XT (lásd 2.5.6).

Az eltérő alapelveken alapuló, még kiforratlan keretrendszerek miatt ilyen bonyolultságú metaprogramok gyakorlati alkalmazása ritka, és egyelőre nem is célszerű.

2.4. Alkalmazások

A következőkben a metaprogramozás gyakorlati alkalmazásait tekintjük végig. Ezek száma még viszonylag kevés, ami 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.

2.4.1. 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 ennél jóval hatékonyabb módszerek is. Egy nyelv- és paradigmafüggetlen bonyolultsági metrikát mutat be [77], generatív programokra szánt kibővítését [76] alatt olvashatjuk.

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.

2.4.2. Hagyományos alkalmazások

A metaprogramozás gyermekbetegségei alól kivételt a több évtizedes múlttal rendelkező, hagyományosnak mondható területek jelentik. Ilyenek a fordítóprogramok és elemzőprogram-generátorok, melyek erős nyelvelméleti háttérrel és kiforrott implementációval rendelkeznek. Ezek részletezése nem tartozik a dolgozat témájához, számtalan kiváló minőségű alkotás található a téma irodalmában. A széles választékból talán [78] érhető el legkönnyebben.

Szintén ide sorolhatjuk a hibakereső rendszereket (debugger). Ezek információt szolgáltatnak egy másik program forráskódjáról, nyomon követhetik annak futását, az újabb rendszerekben pedig menet közben meg is változtathatják a tárolt adatokat vagy magát a programot is.

2.4.3. 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.

Metaprogrammal megvalósított beágyazott nyelvekre klasszikus példa az AraRat rendszer [64], mely C++ kódba beágyazott SQL lekérdezéseket valósít meg. A megoldás előnye a klasszikus beágyazott módszerekkel szemben a fordítási idejű hibafelismerés, melyet az tesz lehetővé, hogy a sablonokkal felépített kifejezésfák által leírt SQL utasítások erősen típusos C++ definíciókra képződnek le (lásd 4.4).

Egy 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 további programrészek generálását is lehetővé teszi, ilyen például az AndroMDA [80] nevű fejlesztőkörnyezetet.

A többszintű programozási nyelvek magas szintű programgenerátorok, melyek erősen típusos módon teszik lehetővé programrészek részleges kiértékelését és a kódgenerálást. A

klasszikus programozási nyelvek esetében a program készítése, fordítása és végrehajtása három jól elkülöníthető lépés. A többszintű nyelvek esetében ezek egyetlen folyamatként zajlanak le. A többszintű nyelvek egy klasszikus példáját adja a MetaML (2.5.5) nyelv. Másik példát szolgáltat a Template Haskell [52], mely a Haskell funkcionális nyelv kiterjesztése. A Template Haskell lehetővé teszi típusbiztos fordítási idejű metaprogramok írását, mert segítségével a Haskell kódot, azaz a konkrét szintaxist absztrakt szintaxisfává alakíthatjuk, és vissza. A szintaxisfán azután hagyományos Haskell műveleteket végezhetünk el fordítási időben, vagy akár teljesen új kódot is létrehozhatunk.

2.4.4. 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 2.2.4). A másik esetben a célformátum adatsémája adott, ezt szeretnénk a programozási nyelven egyszerűen kezelni. Ekkor a metaprogram általában a sémaleírás alapján programnyelvbeli típusokat generál, melyek használata már intruzív. Ez utóbbi módszerre mutat egy hatékony példát a dolgozat 5. 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 nyílt forráskódú db4o [54]) előnye, hogy teljesen automatikusan, beavatkozás nélkül képesek beilleszteni az objektumok tárolásához szükséges utasításokat a program lefordított bájtkódjának átalakításával (lásd 2.5.3) . 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.

2.4.5. 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 [81]. A megvalósításra felhasznált eszközök rendkívül változatosak, egyszerű függvénykönyvtáraktól (RPC) kezdve absztrakt 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 2.5.3) 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 [68], mely C és C++ nyelvű programkód elérését támogatja kb. 20 másik nyelven.

2.4.6. Optimalizálás

A teljesítménykritikus alkalmazások 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 a teljesítmény jelentős növekedéséhez 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 [69, 70, 71] bír, megvalósításai még ma is ritkák és kezdetlegesek. Ennek oka bonyolultsága mellett a teljes átalakítást (lásd 2.3.7) 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 kifejezés-sablonokat (lásd 2.2.5), segítségükkel kiküszöbölhetők azok a feleslegesen létrehozott ideiglenes objektumok, melyek az objektum-orientált módon megvalósított kifejezésfák kiértékelésekor jönnek létre. Ilyen numerikus számításokat végző könyvtár például a Blitz++ [19] vagy az LTL [20]. Sablonokkal felépített kifejezésfát használnak EBNF formájú kifejezések elemzésére is a Boost Spirit [27] könyvtárában, vagy a reguláris kifejezések gyorsítására például a Boost Xpressive könyvtárában [28]. Hasonló feladatokhoz D nyelven már kifejezés-sablonokra sincs szükség, mivel ott a kifejezésfa felépíthető egyszerű karakterláncok fordítási idejű feldolgozásával [59] is.

2.4.7. Átszervezés

A szoftverek fejlesztése során gyakran előre nem látható, fejlesztés közben felmerülő szempontok alapján kell átszervezni a programkódot. Az átszervezés (refactoring) során 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 leggyakoribb célja a bonyolultság csökkentése és a karbantarthatóság javítása, módszereit [60] 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 2.4.6.

2.4.8. Típusrendszer vizsgálata és átalakítása

Sok esetben kényelmesebbé vagy megbízhatóbbá tehetjük a nyelvet, ha kiegészítjük a típusrendszerét, vagy akár nagyobb változtatásokat eszközölünk rajta. A saját típusrendszerek alkalmazása rendkívül széles körű, az objektumok tulajdonjogainak nyomon követésétől kezdve az optimalizáció segítésén át egészen az automatikus helyességbizonyításig rengeteg különböző célra használható. Egy típusrendszer átalakítása nehéz, mi-

vel legtöbbször nem csak a megírt programok, hanem a nyelv átalakítását is igényli, az ehhez szükséges teljes átalakítást (lásd 2.3.7) azonban elvétve támogatja metaprogramozási keretrendszer. Következésképp egy megváltoztatott típusrendszert legtöbbször egy programnyelv speciálisan módosított fordítóprogramjával adják meg, vagy saját nyelvet definiálnak. Sokszor azonban még jelenlegi, gyenge eszközeinkkel is képesek vagyunk a típusrendszer módosítását metaprogrammal megvalósítani, szempontunkból ezek a legérdekesebbek esetek.

A típusrendszer kiegészítésének egy érdekes alkalmazását láthatjuk [30] alatt, ahol Meyers metaprogramok segítségével a C++ nyelv *const* típusmódosítójához hasonló tetszőleges, új korlátozások bevezetését adja meg. A típusrendszer bővítését teszik lehetővé a Java annotációi és a C# attribútumai is (lásd 2.5.3).

A dolgozatnak a típusrendszerek vizsgálata és átalakítása adja a fő irányvonalát. Először (3. fejezet) egy általános típusvizsgáló rendszert adunk meg. A meglevő típusrendszer metaprogrammal történő kiterjesztésére később (4. fejezet) láthatunk példát, a strukturális altípusossághoz hasonló viselkedést valósítunk meg vele. Végül (5. fejezet) egy olyan általános sorosító módszert adunk meg, mely a típusok metainformációinak feldolgozására épül.

2.4.9. Aktív könyvtárak

A programozási nyelvek fejlődésével párhuzamosan jelentek meg az egyre fejlettebb felhasználói könyvtárak. Az objektum-orientált paradigma elterjedésével ezek a könyvtárak is átalakultak: függvények halmaza helyett állapottal rendelkező osztályok, öröklődési hierarchiák jelennek meg. Az ilyen könyvtárak azonban még mindig passzívak: a könyvtár írója minden lényeges típusokkal és algoritmusokkal kapcsolatos döntést kénytelen meghozni a könyvtár írásakor. Bizonyos esetekben ez a korai döntéskényszer hátrányos.

Az aktív könyvtárak [15] olyan kódrészletek, melyek aktív szerepet játszanak fordítási vagy szerkesztési időben. Segítségükkel helyzetspecifikus döntéseket hozhatunk, módosításokat végezhetünk. Például kiválaszthatják a hívási környezettől függően a legmegfelelőbb adatszerkezetet vagy algoritmust. Esetenként ennél bonyolultabb feladatokat, mint például normalizálást vagy hibaellenőrzést is elvégezhetünk [16]. Az aktív könyvtárak lényege, hogy esetükben a könyvtár írója döntéseket halaszthat el, algoritmusokat delegálhat a könyvtár alkalmazásának idejére.

2.5. Metaprogramozási környezetek

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

metaprogramozási környezet filozófiája és az általa biztosított eszközök is változatosak. A metaprogramok futhatnak fordítóprogramok belsejében, mint C++ vagy D nyelven. Futhatnak egy másik programba beágyazva is a futtató környezet segítségével, mint például virtuális gépek vagy interpreterek esetén.

2.5.1. A C++ nyelv sablonjai

Némi iróniával azt is mondhatjuk, hogy a jelenlegi C++ nyelven (lásd [32, 31]) végzett metaprogramozás teljesen véletlenül alakult ki. Valóban, 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++ fordítóprogramban futott, Erwin Unruh készítette [57], 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, rekurzió, függvények) is reprodukálni a fordítóprogram futása közben [14] , ezzel megalkotta a sablonokkal végzett metaprogramozás alapjait. Később megmutatta azt is, hogy a sablonok Turing-teljes eszközt adnak [17] algoritmusok futtatására a fordítóprogramon belül. Megalkotta továbbá a sablonokkal felépített kifejezésfákat (expression template, lásd 2.2.5) 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 [21] 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 4. 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 [25] szabadon hozzáférhető, nyílt forráskóddal rendelkezik. A C++ szabványkönyvtárához hasonló programozási felületet nyújt a fordítási idejű algoritmusok számára, leírását lásd [22].

A sablonokkal való metaprogramozás azonban még így is rendkívül nehézkes. Egyik nagy hátránya, hogy a sablonparaméterekről semmilyen kikötést nem tehetünk, így meta-

programjaink 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 [33] a concept nevű nyelvi eszköz [34] 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 [9]), 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 [33] kódnéven formálódó szabványa. Azonban a C++ nyelv sablonjai még az újításokkal együtt is csak részhalmazát adják a D nyelv (2.5.2) metaprogramozási eszközeinek.

2.5.2. A D nyelv sablonjai

Amint azt már a neve is jelzi, a D programozási nyelvet (lásd [58]) 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 nagyságrendekkel 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 (2.4.3) ad támogatást.

A futási idejű eszközöknél a nyelv jóval erősebbekkel is rendelkezik. Egyik ilyen fontos eszköz a fordítóprogram fordítási idejű kódkiértékelési képessége (CTFE - Compile Time Function Execution). Ez nem csak az aritmetikai és sztringműveletekre 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. Ezzel megtakaríthatjuk azt a jelentős erőfeszítést, mely a legtöbb környezetben a metaprogramok írásával jár. Amennyiben 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 [28] vagy 2.4.6) könyvtárhoz hasonló, reguláris kifejezéseket fordítási időben kiértékelő algoritmus [59] képes kifejezés-sablonok (lásd 2.2.5) használata nélkül, karakterláncok automatikus fordítási idejű feldolgozásával 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*).

[3]Ennek természetesen szigorú feltételei vannak a függvény paramétereire és a függvénytörzsben felhasználható műveletekre nézve, a függvény például nem használhat mutatókat vagy kivételeket. Részletes leírása a nyelv specifikációjában (lásd [58] alatt) olvasható.

A dolgozat írása közben megjelent a nyelv legújabb, 2.0 verziója is, mely jelenleg a nyelv kísérleti ágát adja. Ebben már a típusok részleges fordítási idejű önleírása[4] (*trait*) és a sablonparaméterek megkötései (template constraint) is bemutatkoznak. Ezek az újítások a 3. fejezetben és [1] alatt korábban leírt C++ nyelvű megoldáshoz hasonlóan valósultak meg D nyelven.

2.5.3. Virtuális gépek (Java és C#)

A Java [83] és a C#/.Net [82] 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 (2.2.4), mely alapján hozzáférhetünk más osztályok adattagjaihoz és metódusaihoz.

A Java és C# nyelvek támogatják a sablonokhoz hasonló, de kisebb kifejezőerejű generic-eket. Segítségükkel lehetőség nyílik osztályok és metódusok paraméterezésére típusokkal. A C++ sablonjaival szemben lehetőséget biztosítanak a típusparaméterekkel szembeni kikötések megadására. Ilyen például egy kötelező ős vagy interfészmegvalósítás megadása, de C# nyelven a típus más tulajdonságaival vagy akár konstruktorokkal szemben is tehetünk kikötéseket. A sablonokhoz hasonló specializációra azonban semmilyen eszközt nem nyújtanak.

A Java kezdetlegesebb generic megvalósítása típustörléssel dolgozik, vagyis a típusparaméterek csak fordítás közben léteznek és ellenőrződnek, a virtuális gép kódjában már csak *Object* típus szerepel a helyükön. Ez többek között azzal a kellemetlen mellékhatással jár, hogy a különböző típusparaméterű generic-ek közös statikus tagokon osztoznak. A C# már nem így dolgozik, bájtkód szinten támogatja a sablonokat.

Az attribútum-nyelvtanok [78] 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 is, ezzel ideális eszközt nyújtanak a nyelv típusrendszerének átalakítására (lásd 2.4.8). A metaadatok feldolgozása során természetesen programkód is generálható, ennek segítségével valósul meg

[4]Ez a futási idejű önleírásnál jóval erősebb eszköz, lásd 2.2.

például a sorosítás és adattárolás (2.4.4), valamint a távoli szolgáltatások automatizált elérése (2.4.5) 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 [54]), a teljesítmény mérésének (profiling) automatizálása, vagy a szerződés alapú programozás (design by contract) támogatása (lásd [84]).

2.5.4. 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 (2.5.3) 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 [39] programozáshoz hasonló stílust, mely a kód újrafordítása nélkül, menet közben használható. Tetszőleges kódot hozzáadhatunk egy objektumhoz vagy egy osztályhoz (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 külön fordítási

lépésre a generált programkód futtatásához.

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 bármilyen programkód betölthető vagy menet közben átdefiniálható, ez a gyakorlatban akár a program teljes átalakítását is jelenti (2.3.7), mely a legmagasabb szintű metaprogramozási transzformáció. A szkriptnyelvek eszközeivel tehát olyan összetett metaprogramozási feladatok is megoldhatók, melyekre a fordított nyelvek általában nem képesek.

2.5.5. MetaML és MetaOCaml

Az OCaml nevű nyelv az ML nevű funkcionális nyelv objektum-orientált kiterjesztése. E nyelvek további kiterjesztései a MetaML [50] és a MetaOcaml [53] programozási nyelvek, melyek metaprogramozási eszközök használatát teszik lehetővé.

Többszintű nyelvek (lásd 2.4.3), ezt mindkét nyelvben három nyelvi konstrukció teszi lehetővé. A késleltetés (brackets, MetaOCaml jelölése .< >.) megjelöli a halasztott kiértékelésű kódrészleteket. A hivatkozás (escape, jele .~) anélkül képes ezek értékére hivatkozni, hogy azonnali kiértékelést kényszerítene ki, ezáltal további kifejezéseket építhetünk fel belőlük. Végül külön kiértékelés (run, jele *.!*) művelet segítségével utasíthatjuk a programot, hogy generáljon kódot, mely kiszámítja a késleltetett kifejezések értékét.

```
let rec power (n, x) =
    match n with
        0 -> .<1>. | n -> .<.~x * .~(power (n-1, x))>.;;

let power2 = .! .<fun x -> .~(power (2,.<x>.))>.;;
```

A fenti, MetaOCaml nyelvű programkód a hatványozás függvény kiszámítását optimalizálja. Az optimalizálás lényege, hogy a függvényhívás először a szorzásokból álló kifejezésfát építi fel, melyben így már nem lesznek ismétlődő, ezáltal többször kiértékelt részek. Felépítése után a kifejezésfa konkrét paraméterekkel optimálisabban kiértékelhető, a példa részletesebb leírását lásd [51] alatt.

A funkcionális nyelvek többszintű kiértékelésének leggyakoribb célja a program futási költségének javítása. Ez abból adódik, hogy a költség kiszámítási módja korántsem egy-

értelmű, sokszor csak a késleltetett programrészek kiértékelésének ideje számít. Ilyenkor célszerű minden lehetséges programrészt az első fázisban végrehajtani, hiszen ezek költsége ezen mérték szerint nulla.

2.5.6. Stratego/XT

A Stratego/XT [67] 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, végül 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.

Alkalmazásai sokrétűek, a hatékonyság növelésétől kezdve az automatizált dokumentáláson át egészen a logikai tételbizonyításig széleskörűen használható. Általánosságából adódóan rendkívül ígéretes rendszer, sokat hozzátehet a metaprogramozás elméletéhez és alkalmazásaihoz, az ilyen keretrendszerek jelenthetik a metaprogramozás jövőjét.

3. fejezet

Típusok tulajdonságainak vizsgálata

A metaprogramok működése a bemenetként kapott programról kinyert metaadatokon és a programtranszformációkon alapul. A metaadatok (lásd 2.2) közvetlen elérése azonban a legtöbb programozási környezetben rendkívül rosszul támogatott, mivel a nyelv tervezésekor a metaprogramozási célok nem szerepeltek az elsődleges követelmények között. 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. Ezen 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 (3.1) után felépítek egy C++ nyelvű programkönyvtárat (3.2), 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 2.2.4) ez a módszer még nem elég erős[1]. Lehetséges lesz azonban a típusok egyes elemi tulajdonságainak vizsgálata (2.2.3).

[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.

3.1. 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 elágazás ágainak csak a hozzájuk tartozó feltétel teljesülése esetén kell szemantikailag is helyesnek lenniük (lásd a Boost metaprogram könyvtárának [25] elágazásait).

Ez a rugalmasság és kifejezőerő azonban jelentősen gyengíti a nyelv biztonságát. 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 példányosításáig 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 [31]). 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 3.3.2 alatt ismertetem. Ezeknél azonban jóval általánosabb, metaprogramozáson alapuló megoldás is adható, mely a programkód önvizsgálatának segítségével képes megszorításo-

kat kifejezni. Ennek részleteit 3.2 mutatja be.

3.2. Megoldás

Egy általános célú metaprogramozási rendszernek megoldást kell adnia megszorítások kifejezésére is. Ezért a 2. 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.

3.2.1. Felhasznált metaprogramozási eszközök

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. Megjegyezzük, hogy a most bemutatott metaprogramozási technikák egyike sem saját eredmény, mások korábbi munkáját dícsérik.

3.2.1.1. Sablonparaméterek típusának kikövetkeztetése

A C++ nyelv lehetővé teszi, hogy függvénysablonok használatakor egyes esetekben elhagyhassuk a sablon típusparamétereinek kiírását. Ez olyankor lehetséges, ha a típusparamé-

terek 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); // --- Egyenértékű 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*.

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.

3.2.1.2. 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 első 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.

Az első függvényhívás a sablonparaméterek automatikus kikövetkeztetésének segítségével a két lehetséges változat közül az elsőt hajtja végre, mivel a nyelvi szabályok szerint ellipse-et tartalmazó függvényváltozatot csak más jelölt hiányában választhat a fordító. A második hívás is először az első változatot próbálja példányosítani, ám egy egész számmal

[2] A szabvány ezt a viselkedést csak függvénysablonok esetén írja elő, osztályok esetében erre csak a C++ új szabványának [33] bevezetésével lesz majd lehetőség.

ez nem sikerülhet. A SFINAE szabály miatt azonban a fordítás nem áll meg hibával, a fordító tovább keres, és meg is találja az ellipse-es második változatot, melyet 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].

3.2.1.3. Az *enable_if* sablon

Az *enable_if* osztálysablon a Boost könyvtár [23] része, a segítségével megvalósított döntési módszer részletes leírása megtalálható [56] 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 alkalmas, [21] alatt erre egy jobb megoldás olvasható. A későbbiekben azonban a SFINAE segítségével egy másik függvényváltozatot választunk helyette, elkerülve ezzel a fordítási hibát.

[3]Egyetlen különbség, hogy a sablonfüggvények precedenciája az ellipse kivételével mindig kisebb, így a fordító alkalmas illeszkedés esetén a sablon nélküli változatot részesíti előnyben.

3.2.1.4. Kiválasztott függvényváltozat kinyerése

A C++ függvénytúlterhelési szabályai, melynek alapján a fordítóprogram dönt, ismertek. Mi a választás eredményéről azonban legfeljebb csak a program futása közben értesülünk, a fordítóprogram erről semmilyen tájékoztatást nem ad. Egy ügyes metaprogrammal viszont ez az információ is kinyerhető.

A megoldásban a *sizeof* operátor lesz majd segítségünkre. Alkalmazásához először is két különböző méretű típust kell definiálnunk a hamis és igaz logikai értékek ábrázolására (lásd a technika eredeti leírását [21] 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 történhet:

```
Yes IsInt(int i);                // --- Egész számokra Yes
template <class T> No IsInt(T);  // --- Minden más típusra No

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

A megadott függvényváltozatok segítségével fordítási időben képesek vagyunk eldönteni, hogy egy átadott paraméter típusa egész szám-e. Ez azonban magában még kevéssé hasznos, de a módszerrel néhány hasznos feltételt is megadhatunk. Egyik a [21] alatt bemutatott *SUPERSUBCLASS* makró, melynek segítségével fordítási időben eldönthető, hogy altípusa-e egy osztály a másiknak.

3.2.2. Saját eredmények

A következő metaprogramozási technika saját eredmény, tudomásunk szerint nem volt [1] alattinál korábbi alkalmazása.

3.2.2.1. Tulajdonságok meglétének eldöntése

A 3.2.1.4 alatt bemutatott módszer hasznos alapot nyújt, azonban a vele kifejezhető feltételek száma rendkívül korlátozott. Ez annak köszönhető, hogy a C++ függvényszignatúrába ezen a módon kevés hasznos feltétel írható, mely hasznos, és teljesülésének sikertelensége esetén nem okoz fordítási hibát. A bemutatott technikai háttér segítségével azonban megalkotható egy olyan kódvizsgáló módszer, melyben tetszőleges, a nyelvi szabályok szerint kifejezhető logikai kifejezés írható. A módszer az alábbi:

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

// —— Hibát megakadályozó mentőág, ha nem teljesül
No sizeIsFour(...);

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

Előző példánknál maradva azt az miveleldöntendő 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 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. 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.

A függvénytörzs valójában teljesen felesleges volna, 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.

A módszer kifejezőereje a 3.2.1.4. alatt bemutatottnál jóval nagyobb, hiszen az *enable_if* feltételében összetett logikai kifejezések is megadhatók elemi, C++ nyelven kifejezhető logikai kifejezések és logikai operátorok segítségével. A módszer használata kicsit

[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.

kényelmesebbé tehető, az eredmény vizsgálatának rövidítésére még 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 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 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.

3.2.3. 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 3.3.2.3) kiváló minőségű megvalósítást és dokumentációt biztosít, ezért ezekkel a dolgozat már nem foglalkozik. A többi pontra még nem ismert megfelelő megoldás, ezért a továbbiakban az ezekkel kapcsolatos saját eredményekre koncentrálunk.

3.2.3.1. 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* osztály egy objektumán belüli, *VariableType* típusú mutatót 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 vizsgálat 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.

3.2.3.2. 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; // —— Tagfüggvény deklarációja típusrövidítéssel
};

// —— 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[5]. Az utolsó sorban végrehajtjuk a vizsgálatot, melynek alakja és működése is teljesen azonos az adatok vizsgálatánál bemutatottakkal. Mivel a *size* tagfüggvény valóban a vizsgált típussal rendelkezik, a kiértékelés ezúttal igaz értékkel tér vissza.

3.2.3.3. Beágyazott típusok létezésének vizsgálata

Létező beágyazott típusok eddigiekhez hasonló vizsgálata meglehetősen egyszerű és megoldott probléma (lásd Loki [21] könyvtár *IsSameType* vagy Boost Metaprogramming Library [25] *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 feltétlenül vizsgálnunk kell. A létezés eldöntése már összetettebb probléma, szükségünk lesz hozzá az *enable_if* alkalmazására. A megoldás egy speciálisabb formája megtalálható [29] alatt, de mi ennél általánosabbat adunk.

A megoldás megkívánja még egy újabb technikai eszköz, a *Type2Type* típus bevezetését, mely szintén a Loki [21] könyvtár része. Egyparaméteres, üres törzzsel rendelkező egyszerű jelölőosztály, melyet függvényparaméterként fogunk használni.

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

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

Bevezetésének célja, hogy automatikus típuskikövetkeztetéssel képesek legyünk típusparamétert átadni egy sablonfüggvény számára anélkül, hogy azt explicit módon ki kellene

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

írnunk[6]. 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 rendelkező osztályok
template <class T>
typename enable_if<sizeof(typename T::iterator), Yes>::Result
checkIterator(Type2Type<T>);

// --- Mentőág
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. 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(...)
```

[6]Ha ez nem lenne követelmény, akkor a fenti példában a függvénynek 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 *null* mutatóval hívjuk meg *func(static_cast<T*>(NULL))* alakban. A *Type2Type* azonban könnyebben olvasható és elegánsabb, ezért inkább azt használjuk.

```
// —— 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. 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.

3.2.3.4. 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 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.

3.2.3.5. A predikátumok használata

Az áttekinthetőség kedvéért az alábbiakban összefoglaljuk az eddig saját eredményként kialakított programozási felülelet, mely rendelkezésünkre áll a predikátumok használatára. Az összefoglalás a 3.1. táblázatban olvasható.

Predikátum hívása	Predikátum leírása
`TYPE_IN_CLASS( Name, Type )`	Adott *Name* nevű függő típus létezése a megadott *Type* típuson belül. A makró hívása a függő típusról semmilyen más információt nem nyújt, kizárólag a létezését dönti el. A predikátum használatát elő kell készíteni a *PREPARE_TYPE_CHECK(Name)* makró meghívásával. A makró függvényeket definiál, ennek megfelelő környezetben kell meghívni. A hívás a vizsgálatot elvégző függvényeket definiálja, a vizsgálat ezután tetszőlegesen használható.
`MEMBER_IN_CLASS( Name, Type )`	Mint az előző, adat- vagy függvénytag létezésére. Előkészítése a *PREPARE_MEMBER_CHECK(Name)* makró meghívásával.
`Member<T>::Static( &Type::Name )`	Statikus tag típusának vizsgálata. Egy *Type* típuson belül garantáltan létező (az előző hívással már vizsgált), *Name* nevű tag esetén a függvényhíváskor a tagra mutató pointert adjuk paraméterként. A predikátum igazat ad, ha az átadott tag statikus és típusa a megadott T. A megadott típus adattípus (pl. *int*) és függvénytípus (pl. *void(int, double)* kétparaméteres eljárás) egyaránt lehet.
`Member<T>::NonStatic( &Type::Name )`	Mint az előző, de dinamikus (nem statikus) tagok esetére.

3.1. táblázat. Saját predikátumok programozói felülete

3.2.4. 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*).

3.2.5. 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. Bár ez a gyakorlatban is működő

megoldás, azonban alkalmazása nehéz és körülményes, helyette érdemes volna valamilyen közvetlen támogatást nyújtani.

A jelenlegi C++ nyelven erre sajnos nincs lehetőség, így megoldásként egy egyszerű nyelvi kiterjesztést[7] javasolunk. Legyen minden sablondefinícióban a sablon törzse előtt megadható a *requires* 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 requires IsAddable<T>::Result
{ ... };

// --- Egyenértékű behelyettesített változat
template <class T> class Matrix requires
  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 3.3.2.4) javasol a C++ nyelv C++0x nevű, formálódó szabványa, valamint hasonló eszközt (3.3.1.3) vezetett be nemrégiben a D nyelv legújabb verziója is.

3.3. Kapcsolódó munkák

A saját eredmények ismertetése után áttekintjük a témához kapcsolódó munkákat. A közelmúltban a fenti megoldásra több cikk is hivatkozott, az ELTE kutatásai mellett (például [9]) több külföldi tanulmány, úgymint a Freie Universität Berlin és University of Auckland [74], illetve a EPITA Research and Development Laboratory [73] kutatói által készített konferenciacikkek, továbbá a University of Auckland egy disszertációja [75].

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.

[7] Hangsúlyozzuk, hogy a típusvizsgálatok kiterjesztés nélkül is teljes értékűek. A javasolt kiterjesztés az eredmények felhasználásához praktikus, de nem feltétlenül szükséges, kizárólag kényelmi szempontokat szolgál, a megoldást érdemben nem befolyásolja.

3.3.1. Kitekintés más nyelvekre

3.3.1.1. 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 (2.2.4) 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.

3.3.1.2. Ada

Az Ada nyelv attribútumai a típusokról nyújtott metaadatok egyik legrégebben használt formái. Az attribútumok ugyan csak korlátozott típusinformációt nyújtanak, ám szépen demonstrálják egy általános kódvizsgáló rendszer alapjait. Minden T típushoz léteznek róla információt szolgáltató, előre definiált attribútumok. Ilyenek például a típus példányai által lefoglalt memória méretét megadó *T'SIZE* vagy a típus ősét megadó *T'BASE*. A közvetlen nyelvi támogatás lehetővé teszi, hogy hasonló elvű megoldásunkkal ellentétben ezek az adatok ne egy független vizsgálat eredményei legyenek, hanem szigorúan a típushoz tartozzanak.

3.3.1.3. D

A D programozási nyelv (lásd [58] és 2.5.2) 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 2.5.2) jóval a tulajdonságok lehetőségein túl is támogatja elemi vizsgálatok végrehajtását. A 3.2.3 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 típusleírás is (*allMembers*, *getVirtualFunctions*, stb).

3.3.2. C++ nyelvű megközelítések

3.3.2.1. Szignatúrák

A szignatúrák [35] az interfészekhez hasonló, ám implicit altípusossággal dolgozó nyelvi eszközök (lásd 4.2.4). Használatukkal lehetővé válna a megszorítások egy korlátozott formájának megadása, ám a megoldás korántsem lenne általános és teljeskörű. Bizonyos típusú kikötéseket egyszerűen megadhatnánk, de a felsorolt elemek között mindig implicit logikai ÉS műveletet állna, más logikai műveletet nem használhatnánk.

3.3.2.2. CCEL és Clean++

A CCEL (C++ Constraint Expression Language [55]) 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 2.2.3 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 adatbázis-lekérdezésekké 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.

3.3.2.3. Boost Type Traits

A Boost Type Traits [26] könyvtár lehetőséget biztosít bizonyos elemi vizsgálatok elvégzésére tetszőleges paraméter típuson. A vizsgálatok ebben a könyvtárban is 2.2.3 alatti elveken nyugszanak. Az elvégezhető vizsgálatok széles skálán mozognak, azonban nyelvi kiterjesztés híján meglehetősen korlátozottak. Nem támogat például 3.2.3 alatt bemutatottakhoz hasonló adattagok vagy tagfüggvények létezését és típusát kikövetkeztető vizsgálatokat. Meglevő eszközei azonban jól használhatók, ezért a saját megoldás bemutatásánál több predikátumra nem is adtunk saját implementációt, hanem a Boost használatát javasoltuk.

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 már kénytelen fordítóprogramfüggő, nem szabványos eszközöket felhasználni.

3.3.2.4. Concept

A tervezés alatt álló, C++0x[8] kódnevű új nyelvi szabvány [33] számos tervezett újításának egyike a koncepció (*concept*), mely a szignatúrákra (3.3.2.1) emlékeztető nyelvi eszköz. Elveiben az interfészekhez hasonló, ám kizárólag sablonparaméterekkel szembeni kikötések meghatározására használható, egyszerű függvényparaméterek esetén nem alkalmazható.

Lehetővé teszi a típusok megfelelésének automatizált vizsgálatát, nincs szükség az interfészekhez hasonló explicit megfelelési definíciókra. Segítségével a 3.2.4 alatt bemuta-

[8] A kódnév *x* betűje a jövőbeli, jelenleg még ismeretlen kiadási évre utal. A szabványnak a jelenlegi tervek szerint 2009 folyamán kellene megjelennie, azonban ez várhatóan csúszni fog.

tottakhoz hasonló[9]kikötések írhatók. Lehetőség van az implicit szabályokat nem teljesítő típusok esetén explicit megfelelés (*concept map*) definiálására is.

A koncepciók a fent bemutatott, saját megoldáshoz hasonló eszközt nyújtanak ugyanazon probléma megoldására, melyet a nyelv kiterjesztésével érnek el. Ebből adódóan a koncepciók jóval könnyebben használhatók és természetesebben illeszkednek a nyelv többi eleméhez, ellenben nehezebben bővíthetők és a típusmegszorításokra specializáltak, tehát a metaprogramozást továbbra sem támogatják közvetlenül.

3.4. Összegzés

A fejezetben egy általános, az elsőrendű logika predikátumain alapuló kódvizsgáló rendszert, valamint annak C++ nyelvű megvalósítását mutattam be. A megvalósítás maga is C++ nyelven, a sablon-metaprogramozás segítségével készült. Fontos előnye tehát, hogy az opcionálisan javasolt *requires* kulcsszó kivételével kizárólag szabványos C++ nyelvi eszközöket használ fel, ellentétben a legtöbb megoldással, melyek nyelvi kiterjesztésen alapulnak.

Bár a bemutatott rendszer nem biztosít minden lehetséges típusú vizsgálatot, a legtöbb gyakorlatban felmerülő problémára megoldást ad. Ha a rendelkezésre álló elemi vizsgálatokon kívül mégis továbbiak megvalósítására volna szükségünk, mindössze a szükséges predikátumokat kell megvalósítanunk, ezt szükség esetén már nyelvi kiterjesztéssel is megtehetjük. Ilyenkor az új predikátum kényelmesen, a rendszer már meglevő részeihez illeszkedve használható.

A kifejezőerőt jelentősen növeli, hogy a megszorítások leírásában tetszőleges logikai operátort felhasználhatunk, ellentétben más, implicit ÉS kapcsolatra épülő megoldásokkal. A vizsgálatok végrehajtásával nyert eredmény szabadon felhasználható. Az eredményül kapott logikai érték ismeretében tetszőleges metaprogramozási akció végrehajtható, míg más, hasonló rendszerek általában csak fordítási hibát képesek kiváltani.

Megoldásunknak természetesen hiányosságai, hátrányai is vannak, ezek kiküszöbölése irányt ad a további kutatásnak, továbbfejlesztésnek. A hátrányok főképp a szabvány szabályainak teljes betartásából adódnak. Ezek megakadályozzák, hogy egy osztály védett vagy privát tagjaihoz kívülről hozzáférjünk. Ez általában nem gond, mivel a legtöbb esetben a kinyert információt a vizsgált típus definícióján kívül, számunkra ismeretlen típusok alkalmazásakor használjuk fel, hiszen belül egyszerűen elolvashatjuk a definíciót. Ilyenkor a védett vagy privát tagokhoz egyébként sem férünk hozzá, ezért többnyire nem

[9] A koncepció definíciójában szereplő kikötések implicit ÉS relációban állnak, azonban a típusparaméterrel szembeni kikötések felsorolásánál lehetőségünk van az *&&* és *!* logikai operátorok használatára is. A legutóbbi szabványjavaslat szerint *||* operátor nem használható, de a támogatott operátorok segítségével elvileg kifejezhető.

alkalmazunk ilyen vizsgálatokat. Ha erre mégis szükségünk lenne, megoldásunk erre már nem alkalmas.

A C++ nyelv túlterhelt függvényekre is olyan szabályokkal rendelkezik, melyek esetünkben előnytelenek. 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 éppen ezt a típust próbáljuk kideríteni, esetünkben e típus ismeretlen, hiányában a fordítóprogram többértelműség miatti hibát jelez. A hiba feloldására jelenleg nem tudunk szabványos módszert adni. Mivel adattagokat nem terhelhetünk túl, rájuk a korlátozás szerencsére nem vonatkozik.

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

További gondot okozhat, hogy a metaprogramok használata nehézkes, közvetlen nyelvi támogatás esetén a vizsgálatok végrehajtása jóval természetesebb és egyszerűbb lenne, továbbá rövidebb lenne az eszköz megértéséhez és alkalmazásához szükséges idő is. Ez sajnos szükségszerű, hiszen a szabványos eszközök használata miatt lehetőségeink jóval korlátozottabbak, mint egy szabadon választott kiterjesztés esetén.

1. Tézis. Definiáltam egy elsőrendű logikán alapuló, nem intruzív, univerzális típusvizsgáló (introspection) rendszert. C++ sablon-metaprogramok segítségével elkészítettem a rendszer egy konkrét megvalósítását, mely C++ nyelvű programkód típusvizsgálatára szolgál. A könyvtár az ISO/IEC 14882:1998 szabvány szerinti nyelvi eszközökre épül, ezért fordítófüggetlen és hordozható.

4. 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 [36]) alapul. A helyettesíthetőség elvének gyakorlati alkalmazása a szerződésmodell alapú tervezés (design by contract [37]), mely a tesztelésnél és helyességbizonyításnál bír alapvető jelentőséggel. A típusrendszer erejét az elméleti alapok mellett jól mutatja a több évtizedes sikeres gyakorlati alkalmazás is. A széleskörű használat azonban nem csak az elmélet használhatóságát igazolta, hanem természetes módon rámutatott annak korlátaira is.

A fejezetben azonosítjuk az objektum-orientált módszertan lépésenkénti finomítással kapcsolatos problémáit (4.1), majd az ok azonosítása után megadjuk annak formális definícióját (4.1.1). Ezután korábbi eredményeim [5, 6] alapján rátérünk egy metaprogramozáson alapuló megoldásra (4.3), mely kizárólag a szabványos C++ nyelv eszközeit használja fel. Végül áttekintjük a kapcsolódó irodalmat (4.4) és összefoglaljuk a látottakat (4.5).

4.1. 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                         // —— ősosztály
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 [47, 46].

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:

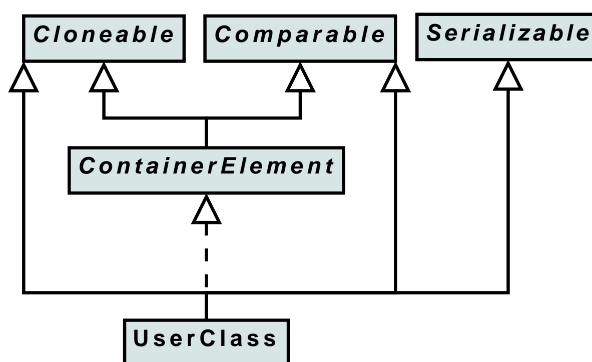
```
class String
extends Object
implements Serializable, CharSequence, Comparable<String>
{
    ...
}
```

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 [48]. 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 programkönyvtárak, melyek a kliens által átadott paraméterekkel, objektumokkal is dolgoznak. Az ilyen paraméterül kapott objektumokkal szemben támasztott követelményeket a szigorúan típusos nyelvek pontosan definiálják, általában interfész megvalósítását, vagy 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, és az így kapott összetett interfészt kell megvalósítania a paraméterként használt objektumok osztályának.

Ezt példázva (4.1. ábra) a Java egyes alapvető interfészeit (Cloneable, Comparable, Serializable) implementáljuk egy osztályban (UserClass), illetve állítjuk össze egy saját 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 [38] követése.

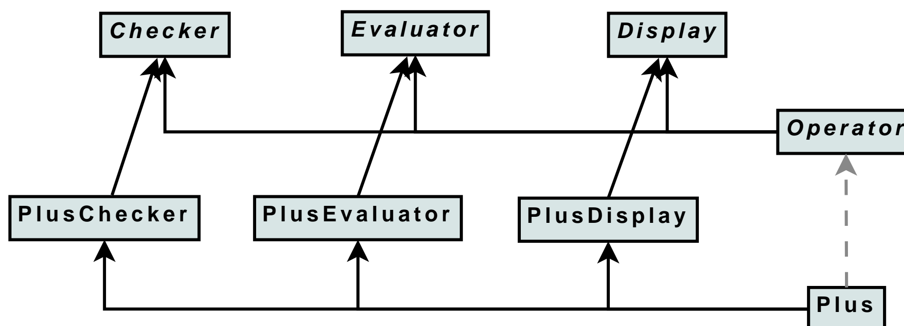


4.1. ábra. Példa több interfész megvalósítására

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 nincs a fordítóprogram által is felismert öröklődési kapcsolata, amíg nem fejezzük ki ezt explicit módon. 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 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.

A probléma szempontjából közömbös, hogy a UserClass interfész-e, melynek származnia kellene a ContainerElement-ből, avagy osztály, melynek meg kellene valósítania azt.

[1] Vegyük észre, hogy az interfészek közötti többszörös öröklődés jelentősen különbözik az osztályok közti többszörös öröklődéstől: előbbit a legtöbb interfészeket alkalmazó (pl. Java, C#, stb) nyelv támogatja, míg utóbbit általában már nem.



4.2. ábra. A kifejezésprobléma

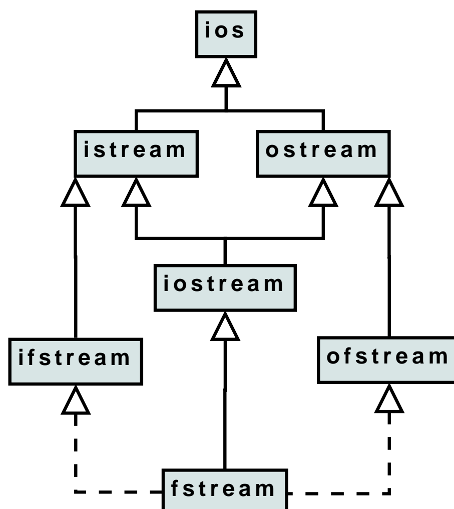
Az eredmény ugyanaz: semmilyen kapcsolatban nem állnak egymással, pedig ez intuíciónk és a gyakorlati alkalmazás szempontjából is kívánatos lenne. A megoldást a későbbiekben C++ nyelven adjuk majd, mely közvetlenül nem támogatja az interfészeket, de rendelkezik a hasonló funkcionalitást nyújtó "tisztán virtuális függvény" (pure virtual) nevű eszközzel. Ezért a továbbiakban általában csak az öröklést, leszármazást említjük, az interfészeket legtöbbször nem emeljük ki külön.

A fent vázolt alapprobléma egy klasszikus esetét adja [46], eredetéből adódóan kifejezésprobléma (expression problem) néven. Tegyük fel, hogy egy kifejezésfákat feldolgozó programot kell készítenünk. A kifejezésfák feldolgozásának több, alapvetően különböző funkcionalitása van, ilyen lehet például a megadott fa ellenőrzése (Checker), a fa kiértékelése (Evaluator), vagy a megjelenítés (Display). Az ehhez tartozó osztályhierarchia a mellékelt 4.1 ábrán látható. Ha az összeadás operátort az ábrán látható módon az egyes funkciók kompozíciójaként adjuk meg, nem fog fennállni a kívánt, szaggatott nyíllal jelölt kapcsolat.

A kifejezésprobléma nevet ennél általánosabb értelemben is szokták használni. Ilyenkor adottak típusaink és mindegyikükön végrehajtható műveleteink. A rendszert bármikor kiterjeszthetjük egy új típus hozzáadásával, melyre működnie kell az összes meglevő műveletnek, vagy egy új művelet hozzáadásával, melynek működnie kell az összes meglevő típussal. Ez nem csak a kifejezésekre jellemző eset, így épülnek fel többek között a modern grafikus és adattároló könyvtárak is, például a C++ és D nyelvek szabványos könyvtárai. A probléma abban jelentkezik, hogy egy ilyen hozzáadás a legtöbbször intruzív, ezért igen költséges, mivel az összes típus vagy függvénydefiníció megváltoztatását igényelheti. Az említett könyvtárak ezt többnyire sablonkönyvtárak segítségével, iterátorok és hasonló absztrakt koncepciók jól tervezett használatával képesek a problémát saját, speciális esetükben megoldani.

Hogy lássuk, valódi jelenségről van szó, a következő 4.3. ábrán a C++ adatfolyamokat kezelő rendszerkönyvtárának alapja látható[2], ahol a leszármazási probléma szintén

[2]Mivel az ifstream, ofstream és fstream osztályok pontosan ugyanazzal a funkcionalitással bővülnek



4.3. ábra. Példa a C++ STL könyvtárából

felbukkan. Bár a szaggatott nyíllal jelölt leszármazási relációk kívánatosak lennének, a C++ szabvány a megvalósítás nehézségei miatt szándékosan [31] nem is írja elő azokat. Így az a meglepő helyzet áll elő, hogy ha egy ki- és bemenetet egyaránt támogató fájl-adatfolyam (fstream) objektumunk van, nem tudjuk azt paraméterül átadni függvényeknek, amelyek a fájlokon csak be- vagy kimeneti műveleteket végeznek, 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).

A leírt jelenség az öröklődés az objektum-orientált típusrendszerek egy jellemző problémája. Az anomáliának angolul a chevron-shape inheritance [5, 6] nevet adtuk, illeszkedve egyúttal a jól ismert diamond-shape inheritance (gyémánt alakú öröklődés) problémájához is. A chevron magyar fordítása nehéz, jelentése "ék alakú rendfokozati csík”, esetleg karpaszomány vagy szarufa. A fordítás nehézsége miatt kivételesen az angol elnevezést használjuk majd, vagy a „lépésenkénti finomítás problémája” néven hivatkozunk rá.

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 a legtöbb implementáció mégis ehhez az eszközhöz nyúlt. A kódismétlés a funkcionalitás és szabványosság szempontjából nem különbözik a leszármazáson alapuló megoldástól, ezért az ábrán egyszerűbb esetként a kódismétléses hierarchia látható.

4.1.1. Formális leírás

A chevron alakú öröklődés problémáját a Batory [48] által kidolgozott formalizmus segítségével definiáljuk. A formalizáció során az interfészekre és osztályokra egységesen komponensekként (jelölése $k_i \in K$ komponenshalmaz) hivatkozunk, a lépésenkénti finomítás során a komponensek valamely tulajdonsággal, felelősséggel való kiterjesztésére funkcióbővítésként (jelölése $f_j \in F$ funkcióbővítések halmaza).

A funkcióbővítések olyan függvények, melyek komponenseket alakítanak át egy adott programfunkció hozzáadásával, $f : K \to K$. Egy k komponensre alkalmazott 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 leszármazást 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 funkcióbővítésnek a komponensek kompozíciójának műveletére nézve disztributívnak kell lennie, 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 altípusosságon alapuló, öröklődéssel megvalósított funkcióbővítés általában nem disztributív. Az állításra nem célunk kimerítő formális bizonyítást adni, mivel fentebb több ellenpéldát is láthattunk igazolására. Utolsó, adatfolyamos példánk esetén ez a következőképp látható be a fileio (lásd 2. lábjegyzet) fájlkezelő funkcióbővítést felhasználva:

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

A fejezet további részében a disztributivitás problémájára próbálunk megoldást nyújtani.

4.2. Lehetséges megközelítések

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

4.2.1. Hagyományos öröklődés

Az eddigiekben vázolt explicit altípusossággal rendelkező öröklődési modell úgy nyújthatna megoldást a chevron-problémára, ha sikerülne elérnünk, hogy osztályaink a lehetséges ősosztályok minden lehetséges variációjával létrehozható összetett osztályból is leszármazzanak. Ha nem így lenne, egy osztály (pl. a már bemutatott *fstream*) őseinek vagy interfészeinek bármikor felbukkanhatna egy olyan variációja (pl. *ifstream*), melyet nem valósított meg, pedig egy új felhasználási helyen ezt várnák tőle. Az explicit leszármazási szabályok miatt pedig ezt már csak intruzívan tudja megtenni, vagyis az osztály utólagos módosítása árán, ami sokszor nem lehetséges.

A lehetséges variációk száma $\Theta(2^n)$, ahol n az ősosztályok és megvalósított interfészek száma. Jól látható tehát, hogy az igényelt osztályok száma exponenciálisan növekszik, ami a gyakorlatban elfogadhatatlan bonyolultságot jelent. Így indokolatlanul sok osztályból kellene leszármaznunk a megoldás érdekében. Még ha rendelkeznénk is a fordítóprogramba épített automatizmussal minderre, a hagyományos öröklődés akkor sem nyújtana jó megoldást az intruzív jellege miatt.

4.2.2. Virtuális öröklődés

A C++ módszere az ismételt öröklődés által felvetett problémák megoldására. Ilyen például a gyémánt alakú öröklődési anomália (diamond shape inheritance), mely a 4.3 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 mindössze egyetlen példányban létezhetnek. 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 [31, 32].

Sajnos a módszer jelentős hátrányokkal bír: a program megemelkedő processzor- és tárköltsége mellett sajnálatos módon intruzív, hiszen beavatkozást 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.

4.2.3. Aspektusok

Az aspektus-orientált programozás [39] egy általános eszközt nyújt osztályok és eljárások definíciójuktól független kiterjesztésére. A módszertan lehetővé teszi, hogy általunk

meghatározott illesztési pontokhoz (pointcut) valamilyen kiegészítő kódrészletet (aspect) szőjünk hozzá (weave). Előnye, hogy ehhez az eredeti forráskódnak nem kell rendelkezésünkre állnia, tehát lehetőséget ad nem intruzív típusmódosításra. Tudományos szemmel nézve széleskörűen elfogadott és elterjedt [40] megoldásnak nevezhetjük.

Megfelelő aspektusok osztályokba szövésével azok kívülről is kiterjeszthetők úgy, hogy a továbbiakban megvalósítsanak egy új interfészt. Ez ideális lehetne számunkra, ám az aspektusok leírásának formája miatt minden új interfész hozzáadása esetén módosítanunk kell 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.

4.2.4. Szignatúrák

A szignatúrák a funkcionális nyelvekből származó (ML szignatúrák, Haskell típusosztályok), az interfészekhez valamelyest hasonló [42] konstrukciók; azok egy általánosításának is tekinthetők. A szignatúrákkal nem csak osztályok tagfüggvényeire tehetünk kikötéseket, hanem adattagokra és beágyazott típusokra is, emellett az interfészekhez hasonlóan leszármazhatnak egymásból. Ráadásul egy osztály deklarációjában nem kell szerepelnie azon szignatúrák listájának, melyeknek az osztály megfelel, tehát a megfelelés implicit, nem intruzív. Az interfészekhez hasonló megfelelési szabályokat alkalmazva a fordítóprogram önállóan dönt, hogy egy osztály megfelel-e egy tetszőleges szignatúrának.

A szignatúrák egy C++ nyelvű megvalósítását bővebben [35] részletezi. A szignatúrák megoldást nyújthatnának problémánkra, 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.

4.2.5. Strukturális altípusosság

A strukturális altípusosság [41] 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 megoldásra váró chevron-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 [49]. A fejezet további részében a strukturális altípusossághoz hasonló viselkedést igyekszünk a C++ nyelvbe illeszteni a sablonokkal végzett metaprogramozás segítségével.

4.3. 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.

4.3.1. 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 [43] leírja, az első általános célú típuslista megvalósítás a Loki nevű programkönyvtárban [21] található, jelenlegi legfrissebb formája a Boost metaprogramozást támogató könyvtárában [25] 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 2.4.6) 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 definícióját:

```
// --- A kisegítő 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 egyenértékű 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 sablonpéldány), a C++ előfordítója ezt a vesszők mentén széthasítva több elemnek fogja értelmezni. Ez könnyen megkerülhető, ha a szóban forgó típusra egy egyszerű névvel hivatkozunk, például *typedef* segítségével.

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.

4.3.2. Osztályok kompozíciója

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

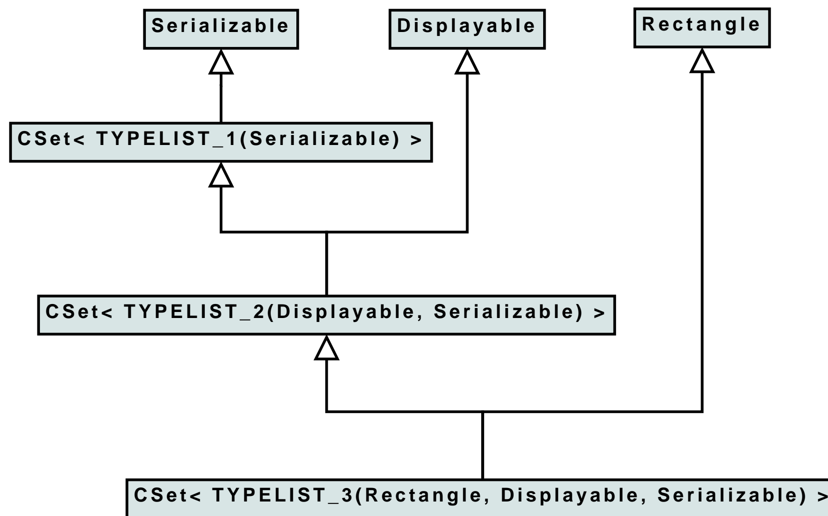
Lássunk egy példát az algoritmus működésére! Három elemi funkcionalitást megvalósító osztályból készítünk kompozíciót. Egyik egy téglalap geometriáját írja le (*Rectangle* osztály, a síkidomokat leíró *Shape* leszármazottja), a másik a képernyőn való megjelenítésért felelős (*Displayable*), a harmadik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

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

objektum típusának osztálydiagramja az ábrán (4.4) 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;
```



4.4. ábra. A többszörös öröklődéssel felépített osztályhierarchia

Az ábrán jól látható, hogy adott típuslistához a lista hosszával megegyező számú új osztály jön létre. Bár különböző listák példányosítása esetén több ilyen osztályhierarchia is létrejön, ezek mennyisége listánként mindig lineáris, így a létrehozott osztályok száma $\Theta(km)$, ahol k a típuslisták száma, m pedig a listaelemek átlagos száma. Az igény szerinti, lusta példányosításnak köszönhetően tehát a létrejött osztályok száma a gyakorlatban lineáris. A típuslisták összes lehetséges kombinációjának példányosításával előállítható típusok maximális száma a konvencionális objektum-orientált megközelítések (lásd 4.2) exponenciális osztályszámával egyezik meg. Ugyanakkor a mi esetünkben csak az alkalmazások által valóban hivatkozott kombinációk példányosulnak, szemben a konvencionális esettel, ahol kénytelenek vagyunk az interfészek összes lehetséges kombinációját előre elkészíteni. Az emberek által készített alkalmazásokban ennek jellemzően csak elenyésző töredéke példányosul.

A CSet sablon használata igen hasonló a típuslistákéhoz, így a definíciók megkönnyítésére érdemes az ott leírtakhoz hasonló makrókat bevezetni. Ezáltal a kompozíció 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 egyszerűbb alakja
CSET_3(Rectangle, Displayable, Serializable) window;
```

4.3.3. 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 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 osztályra 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 4.4 ábrán látható módon. A rekurziót megszakító, egyelemű listákra specializált *CSet* konverziós függvényei a fentihez hasonlók, 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 része 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 (4.1 és 4.3) 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.

4.3.4. 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:

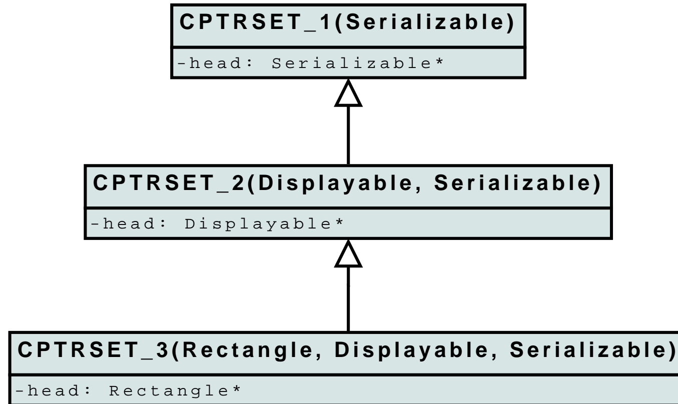
```
// --- Elődeklaráció általános paraméterre
template <class ListOfTypes> struct CPtrSet;

// --- 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 rámutató adattagot tárol. 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*, ..., *CPTRSET_N* makrókat a típusdeklarációk megkönnyítésére. A korábbi példában bemutatott *window* objektum típusának (4.4 ábrán látható) leszármazási hierarchiája a *CPtrSet* segítségével jelentősen egyszerűsödik a 4.5 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



4.5. ábra. A mutatókkal megvalósított osztályhierarchia

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:
    // --- Kisegítõ 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 (4.6) olvasható le.

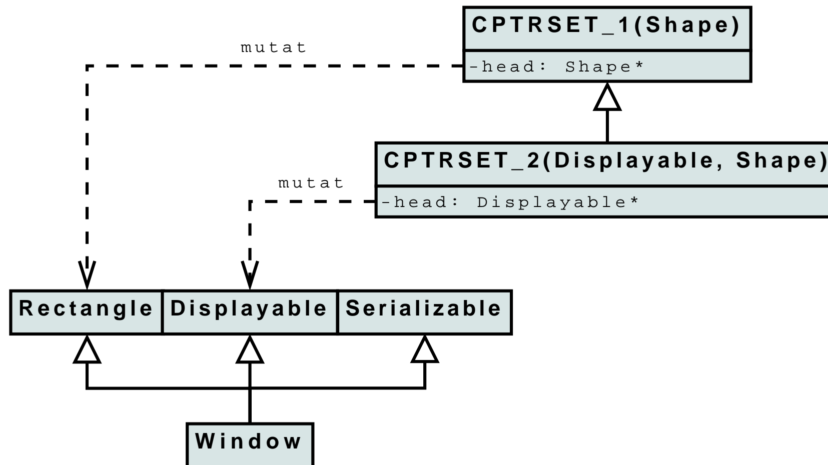
```
// --- Kirajzoló eljárás
typedef CPTRSET_2(Displayable, Shape) BorderType;
void drawBorder(BorderType border) { ... }

// --- 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 a többi 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 elpusztul, a mutatókra érvénytelenné válnak, használatuk meghatározatlan viselkedést eredményez. Ez azonban nem tér el a C++ nyelv mutatóinak alapvető tulajdonságaitól, így nem nevezhető hibásnak.



4.6. ábra. A CPtrSet működési elve

Másik kényelmetlenség, hogy a *CPtrSet* 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 baloldali 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.

4.4. Kapcsolódó munkák

Metaprogramozáson alapuló módszerekkel más esetekben is jelentősen javíthatunk a típusrendszeren. A C++ nyelv *const* típusmódosítójához hasonló altípusképzésre [30] alatt olvashatunk egy egyszerű elveken alapuló, általános megoldást. Felhasználásával tetszőleges, saját típusmódosítókat definiálhatunk (ExceptionSafe, ThreadSafe, Reviewed, stb), melyek meglétét a típus felhasználásának helyén explicit jelöléssel kényelmesen vizs-

gálhatjuk. A megvalósítás sablon-metaprogramozáson alapul, a 4.3.1 alatt bemutatott típuslistákhoz hasonló eszközökkel dolgozik.

A sablonokkal felépített kifejezésfákra (lásd 2.2.5) építve ad módszert [13] az XML sémák típuskonstrukcióinak C++ nyelvbe illesztésére. A kifejezésfával leírt sémát a reguláris kifejezések elméletén alapuló metaalgoritmusokkal dolgozza fel, amely lehetővé teszi a kifejezésfák ekvivalens átalakításait. A bemutatott megoldásunkhoz hasonló módon, konstruktor-sablonok segítségével egyéni konverziós szabályokat definiál, melyekkel pontosan a kívánt konverziókat teszi lehetővé, ezzel erős típusbiztonságot ér el.

Hasonló problémát old meg az AraRat [64] nevű sablon-metaprogram könyvtár is, mely C++ nyelvbe ágyazott SQL lekérdezésekre ad típusbiztos módszert. Meg kell azonban jegyeznünk, hogy ez a könyvtár a probléma bonyolultsága miatt már kénytelen nyelvi kiterjesztésekhez folyamodni.

A lépésenkénti finomítás problémájának megoldására talán legígéretesebbnek a jelenleg C++0x kódnéven tervezési fázisban levő új C++ nyelvi szabvány látszik. A szabvány újításai közül a sablonparaméterek kikötéseinek leírására kitalált concept [34] nevű nyelvi eszköz a szignatúrákhoz (4.2.4) hasonló tulajdonságokkal rendelkezik. Mivel implicit megfelelési szabályokkal rendelkezik, ezért kiváló megoldást nyújthat a C++ típusrendszerének többszörös öröklődéséből származó problémáira. Szabványos nyelvi eszköz lesz, ezért a szignatúrákkal ellentétben várhatóan széles körben alkalmazható megoldást ad majd.

4.5. Összegzés

A fejezetben megmutattam az erősen típusos objektum-orientált nyelvek típusrendszerének korlátait a lépésenkénti finomítás során jellemzően előálló osztályhierarchiákra nézve, konkrét, alkalmazott programkönyvtárakból vett példákkal igazolva a probléma gyakorlati jelentőségét. Megvizsgáltam a lehetséges megoldásokat az esetleges előnyökkel és hátrányokkal együtt. Az explicit megfelelési szabályokból adódóan a rendelkezésre álló nyelvi eszközök szinte mindegyike exponenciális számú osztály létrehozását igényli a megfelelő megoldás érdekében, vagy más hátrányai miatt nem alkalmazható széleskörűen.

Sablon-metaprogramozási eszközök segítségével megmutattam, hogy C++ nyelven megoldás adható a problémára, ha a nyelvet új típuskonverziós szabályokkal egészítjük ki, melyek a strukturális altípusosság elve alapján működnek. A megoldást két lépésben mutattam be, hátrányai miatt finomítva egy kezdeti változatot. A javasolt módszer implicit megfelelési szabályaiból és lusta példányosulásából adódóan az általános megoldáshoz nem szükséges exponenciális számú osztályt előre létrehozni, azok mindig csak igény szerint, automatikusan példányosulnak.

A megoldás előnye, hogy kizárólag szabványos eszközökkel bővíti ki a nyelv típusrend-

szerét. Az implementáció azonban sajnos még a finomítás után sem tökéletes. Könnyen fordítási hibához juthatunk például a többértelműség miatt, ha a típuslistában többszörösen szerepel ugyanaz az elem. Ez ugyan hibás felhasználásnak mondható, de például a boost::mpl [25] típuslistakezelő algoritmusainak segítségével a könyvtár hasonló esetekre elvileg felkészíthető.

2. Tézis. Megmutattam a jelenlegi objektum-orientált nyelvek típusrendszerének korlátjait osztályok lépésenkénti finomításának esetében. Szabványos sablon-metaprogramok segítségével a strukturális altípusosságon alapuló, kiegészítő konverziós szabályokat vezettem be a C++ nyelv típusrendszerébe, megoldást adva a lépésenkénti finomítás problémájára. A módszerrel elkerülhető a létrehozandó interfészek számának kombinatorikus robbanása.

5. fejezet

Sorosítás és távoli szolgáltatások

5.1. A probléma leírása

A programok futása közben felhasznált adatok tárolható formátumra alakítása és későbbi pontos visszaállítása az informatika egyik alapvető feladata. Ezt az alkalmazás számos elnevezése, szinonimája is bizonyítja: mentés és töltés, sorosítás, szerializáció, perzisztencia (serialization, persistence, marshalling, deflation).

A sorosítás céljának megfelelően számtalan adatreprezentációs formátummal dolgozhatunk. Lehet célunk például a hatékony tömörítés, az írási és olvasási idők minimalizálása, a hibatűrés, változáskezelés, egyszerű feldolgozhatóság, a hordozhatóság vagy az emberek számára is könnyű olvashatóság. Ez utóbbi három célkitűzésnek megfelelően tervezték napjaink egyik legtöbbet használt, XML (Extensible Markup Language [85]) elnevezésű formátumát. Alkalmazásai széles skálán mozognak, egyszerű konfigurációs állományok tárolásától kezdve a hordozható szabványos adattároláson keresztül a hálózati szolgáltatások eléréséig (web services) terjednek. A fejezet az adatok XML formátumú sorosítására összpontosít, elsősorban a hálózati szolgáltatások elérésére céljából, a bemutatott megoldás azonban más formátumokra is alkalmazható.

A legegyszerűbb esetben a programozó egyesével, maga határozza meg, hogy a programban felhasznált típusokat hogyan kell sorosítani. Ez azonban nemcsak időigényes és gépies munka, melyben könnyű hibázni, hanem felesleges is, mivel a sorosítás remekül automatizálható. Az automatizált sorosítás rendszerint a típusok leírását, vagyis a metaadatait feldolgozó metaprogram segítségével valósul meg. Ez a metaprogram legtöbbször egy kódgenerátor, mely külön sorosító algoritmust készít minden feldolgozott típushoz, ahogyan azt legegyszerűbb esetben a programozó is tenné. Amint azt a javasolt megoldásban látni fogjuk, más megközelítés is lehetséges.

A fejezetben először a témához kapcsolódó munkákat ismertetjük (5.2). Ezután 5.3 alatt saját megoldásunkat mutatjuk be, végül 5.4 alatt összegezzük az eredményeket.

5.2. Kapcsolódó munkák

Egy XML dokumentum pontos formátuma XSD (XML Schema Definition [86]) dokumentumokkal írható le. Az ilyen sémaleírások metadokumentumok[1], hiszen maguk is XML dokumentumok. Mivel az XML formátum egyik célja az egyszerű számítógépes feldolgozhatóság, nem meglepő, hogy az XSD típusai általában megfeleltethetők programozási nyelvek típusainak. A sémaleírás és a programozási nyelvek típusrendszere nem teljes, de jelentős részben izomorf. A teljes izomorfia megvalósításához léteznek az XML típusrendszeréhez alkalmazkodó nyelvek [90], de hagyományos nyelvekhez is találhatunk újszerű típuskezelési technikákat [13].

Az XML formátum elterjedtsége miatt a típusok önleírását támogató, újabb programozási környezetek már beépített sorosító könyvtárral rendelkeznek, melyek az XML formátumot is támogatják, például a Java és a C#/.Net. Az ilyen módon sorosított XML dokumentumok sémája a programozási nyelv típusaitól függ: a programnyelvvel leírt típusok adottak, ennek alapján a könyvtár XSD sémát rendel hozzájuk, majd ennek megfelelően sorosít. Természetesen léteznek nem virtuális gép alapú sorosító rendszerek is, egyik legelterjedtebb például a C++ nyelvű Boost Serialization Library [24]. Mivel a sorosító könyvtárak nagy része formátumfüggetlen, az XSD sémából emiatt sem indulhatnak ki.

A hálózati szolgáltatások elérésénél azonban az ellenkező irányban kell kiindulnunk. Ilyenkor általában a szolgáltatás WSDL (Web Service Definition Language [87]) formátumú leírása adott, mely többek között tartalmaz XSD típusdefiníciókat is. Ilyenkor célunk az, hogy a szolgáltatást minél több programozási nyelv alól egyszerűen igénybe tudjuk venni. Ilyenkor célszerű a WSDL dokumentumban definiált típusokhoz a választott programozási nyelvbeli típusokat generálni. Ezeket majd a szolgáltatások eléréséhez használt SOAP (Simple Object Access Protocol [87]) formátumú kommunikációba építhetjük be az automatikus sorosítás segítségével.

A fenti elvek szerint dolgozik a legtöbb XML-sorosító vagy hálózati szolgáltatások elérését támogató könyvtár. Ilyen például az egyik legismertebb, nyílt forráskóddal rendelkező gSOAP [91], a Liquid XML [94], a Codalogic Lmx [92] vagy a Code Synthesis XSD [93] is. A felsorolt sorosító megoldások közös tulajdonsága, hogy minden típushoz külön sorosító programkódot generálnak, mely jelentős részét teszi ki a típusokhoz generált programkódnak. Mivel a könyvtárak használatához nincs szükségünk a teljes generált programkód megértésére, ezért ez a programozó szempontjából nem kritikus. Fontos azonban az erőforrások szempontjából, hiszen a felesleges programkód nem csak a futási időt ronthatja, hanem a szükséges tárhely mennyiségét is növelheti. A személyi

[1] A szabványleírásban példaként szerepel egy olyan XSD dokumentum, mely a szabványos XSD dokumentumok sémáját írja le, tehát a séma önleíró is.

számítógépeken napjainkra ez már jellemzően nem okoz problémát, ám a hordozható és beágyazott eszközök továbbra sem rendelkeznek bőséges erőforrásokkal. További fontos szempont, hogy a fenti megoldások általában speciálisan erre a feladatra készült, monolitikus felépítésűek.

5.3. Megoldás

Mi egy hasonló célú, ám a felsorolt megoldásoknál több szempontból előnyösebb módszert adunk a metaprogramozás segítségével. Erre a módszerre építve több platformra megvalósítottuk az Xml Data Binding nevű, teljes mértékben funkcionális kódgenerátort és programkönyvtárat a Nokia Research Center keretein belül.

Javasolt megoldásunk a felhasználó szempontjából nem sokban különbözik a fentebb felsoroltaktól, az általa biztosított programozási felületeket az 5.3. ábrán láthatjuk. A megvalósítás egyik fontos szempontja az egyszerűség és alkalmazhatóság a szűkösebb erőforrásokkal rendelkező eszközökön, például mobiltelefonokon. A másik szempont a megoldás általánossága és a megvalósítás moduláris szerkezete, elkülönített felelősségű, lecserélhető komponensekkel. A megoldás Symbian C++[2] platformra Nokia WSDL-to-C++ Wizard for S60 [8] néven letölthető és használható. A Metadata-based XML Data Binding for C++ (MXDB) nevű könyvtár ennek egy Linux alapú, több szempontból fejlettebb[3] változata, mely sajnos még jelenleg sem nyilvánosan hozzáférhető. A lentebb példaként adott leírások és forráskódok azonban mégis a Linux verzióból származnak majd, mivel a Symbian változat megoldásai a rendszer különleges felépítése és megszorításai miatt jóval bonyolultabbak és nehezebben érthetők.

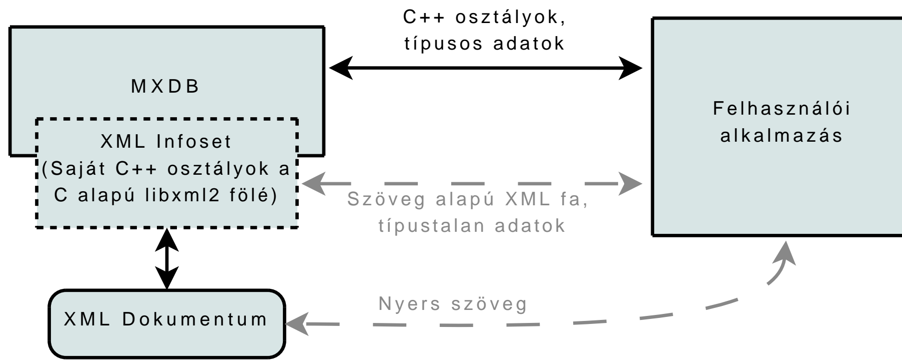
A tervezést Dornbach Péterrel és Payrits Szabolccsal közösen végeztük, míg a megvalósítás az én feladatom volt. Munkánk eredményeit [3, 4] alatt publikáltuk.

5.3.1. Alapelvek, áttekintés

A feladat erőforrásigénye jelentősen csökkenthető, ha a sorosítást végző programkód a lehető legegyszerűbb, ezáltal a fordítóprogram kis méretű tárgykódot tud belőle előállítani. Természetesen különféle programozási trükkök segítségével a generált kód mérete jelentősen csökkenthető. A lehetséges megoldásokat megvizsgálva azonban felismertük, hogy a sorosítás elvégzéséhez generált algoritmusra egyáltalán nincs szükség.

[2] Ez a szabványos C++ egy részhalmaza, nem támogatja például kivételeket (tehát ezáltal a szabványos könyvtárat sem), valamint a futtatott folyamatok preemptív ütemezésére sem képes.

[3] A Linux platformra készült változat kiforrottabb programkódja mellett tartalmaz egy kisegítő programkönyvtárat XML formátumú adatok kezeléséhez, valamint egy hálózati szolgáltatások biztosítására és használatára szolgáló keretrendszert is.



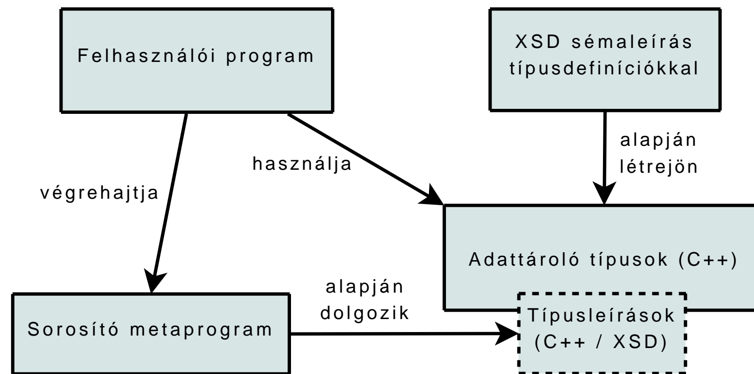
5.1. ábra. Az MXDB felépítése

A fent (5.2) bemutatott megoldások közös alapelve, hogy egy kódgeneráló metaprogram a sorosító algoritmust mindig egy-egy típus metainformációja (típusleírása) alapján, de minden típushoz teljesen különállóan készíti el. Ha egy metaprogram képes lenne ezt a kódot egy előzetes kódgenerálási fázis helyett a program futása közben előállítani, a sorosító algoritmus elkészítéséhez nem lenne szükség külön kódgenerálási fázisra. Ráadásul a program futása közben új programkódot létrehozni teljesen felesleges, hiszen az utasításokat a metaprogram közvetlenül maga is végrehajthatja. Ezáltal egy univerzális sorosító metaprogramot nyerhetnénk, mely mindössze a típusleírások alapján képes további programkód hozzáadása nélkül elvégezni a sorosítást. Az alábbiakban bemutatunk egy olyan módszert, mellyel ez az elképzelés megvalósítható.

Mivel az XSD dokumentumokban definiált típusokat C++ nyelvre kell átültetni, a rendszer alapja továbbra is egy kódgenerátor marad, mely esetünkben csak típusdefiníciókat készít, de algoritmusokat, függvényeket már nem. A kódgenerátor megvalósítására természetes választás az XML dokumentumok átalakítására megalkotott XSLT 2.0 (Extensible Stylesheet Language Transformations) [88] nyelv használata. Az XSLT segítségével az XML dokumentumok feldolgozása egyszerű, emellett a kódgenerálás rendkívül hordozhatóvá válik, mi például a Java alapú Saxon [95] könyvtárat használtuk a kódgenerátor futtatására. A későbbiekben bemutatott sorosító metaprogram is jól hordozható, mivel a könyvtár többi részéhez hasonlóan szabványos C++ nyelvű algoritmusként valósítottuk meg.

Sémaleírással rendelkező XML dokumentumok kezelésének menete az 5.2 ábrán látható módon történik. Röviden összefoglalva:

1. A séma típusainak leírása alapján a kódgenerátorral létrehozzuk a hozzájuk rendelt C++ nyelvű adattároló típusokat. Ezek célja, hogy a programozó a C++ nyelven megszokott adatszerkezeteken keresztül tudja manipulálni az adatokat. A kódgenerátor minden ilyen típusnak biztosítja saját XSD formátumú önleírását, ezáltal lehetővé téve a sorosító metaprogram működését.



5.2. ábra. Az MXDB működési elve

2. A generált típusok XML formátumú sorosításához mindössze egyetlen függvényhívásra van szükségünk. Adatok olvasása esetén létrehozunk a kívánt típusból egy üres objektumot, melyet aztán a *deserialize()* függvény adatokkal tölt fel. Írás esetén egy kitöltött objektum sorosítható a *serialize()* függvénnyel. A függvények mögött egyetlen, előre elkészített, általános sorosító metaprogram dolgozik, mely a konverziót kizárólag a típusok metaadatai alapján végzi.

Az áttekintés után lássuk részletesebben az egyes komponenseket!

5.3.2. Kódgenerátor

A kódgenerátor felelőssége, hogy az XML séma típusdefinícióiból C++ adattároló típusokat hozzon létre, melyek a felhasználó által a lehető legkönnyebben programozhatók. A kódgenerátor működési részleteinek teljes, részletes ismertetése meghaladja a dolgozat kereteit, ezért itt csak egy áttekintését adjuk.

A kódgenerátor futása során először feltérképezi a séma teljes definícióját, rekurzívan bejárva minden további importált sémafájlt. Ezekből összegyűjti az összes (nevesített vagy név nélküli) típusdefiníciót, majd összeállít belőlük egy speciális listát, melynek kiszámítása a felhasznált rekurziók miatt viszonylag költséges. A listában minden típus rekurzívan felsorolja az összes további függő típusát, melyre definíciójában hivatkozik. Eközben nyilvántartjuk a már bejárt típusdefiníciókat, így képesek vagyunk elkerülni a körkörös hivatkozások esetén fellépő végtelen rekurziót. A lista összeállítása után a legelső elem megtartásával kiszűrjük a listaelemek minden további ismétlését, ezáltal a típusdefiníciók függősége szerint (az esetleges kölcsönös függőségek kivételével) helyesen rendezett listához jutunk. Ennek célja az, hogy a C++ kód generálása folyamán minimalizálni tudjuk a típusdefiníciók sorrendjéből adódó hibák lehetőségét[4].

[4]Ez a kötelező tartalmazás esetén fontos, melyet a típushozzárendelésben egyszerű aggregációként kezelünk, ehhez pedig szükség van a tartalmazott típus méretének, így definíciójának ismeretére is. Köl-

A kódgenerátor ezek után bejárja a listát, és 5.3.4 alatt bemutatott típushozzárendelési szabályok alapján elkészíti annak C++ nyelvű megfelelőjét. Minden típus mellé eltárolja annak XSD formátumú eredeti sémaleírását is C++ nyelvű adatok formájában, ennek formátumáról bővebben 5.3.3 alatt olvashatunk. A típusdefiníciók a névütközések elkerülése céljából minden esetben a sémában megadott névterük alá kerülnek C++ nyelven is.

A generált kód felépítését legkönnyebben egy egyszerű (összetett sémakonstrukciók és névtér nélküli) példán keresztül érthetjük meg. Vegyük a következő típusdefiníciót, mely egy naptárbejegyzést ír le:

```
<complexType name="Appointment">
  <sequence>
    <element name="InstanceStartDate" type="dateTime"/>
    <element name="InstanceEndDate" type="dateTime" minOccurs="0"/>
    <element name="Text" type="string"/>
    <element name="Location" type="string"
             minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

Az egyszerűség kedvéért a definícióban elemi típusú, különböző multiplicitású tagokat adtunk meg. Az ebből generált típus deklarációja:

```
struct Appointment : public TypedXmlData
{
    const MetaType& MetaInfo() const;

    Appointment();

    DateTime InstanceStartDate;
    Nullable<DateTime> InstanceEndDate;
    String Text;
    std::vector<String> Location;
};
```

A *TypedXmlData* típusról és a *MetaInfo()* függvényről 5.3.3. alatt, a *Nullable* sablonról pedig 5.3.4. alatt olvashatunk bővebben. Ezeken kívül az osztály csak az alapértelmezett konstruktor és az adatok manipulálásához felhasznált adattagok[5] definícióját tartalmazza.

csönös függőség esetén legalább az egyik tag opcionális, különben véges dokumentum nem felelhet meg a sémának. Az opcionális tagokat viszont mutató típusra képezzük le, amihez csak egy elődeklarációra van szükség.

[5]Jegyezzük meg, hogy ha ragaszkodnánk az objektum-orientáltság adatrejtést előíró elvéhez, a jobb karbantarthatóság és biztonság érdekében beállító és lekérdező függvényeken keresztül érnénk el az adattagokat. Mi azonban a későbbiekben sem szándékozunk módosítani a generált osztályokon, ezért a függvények használata számunkra nem járna előnnyel, azonban ellenkezne célkitűzéseinkkel (egyszerűség és természetes használat), továbbá feleslegesen megnövelné a fordított tárgykód méretét is.

5.3.3. Metaadatok

Minden generált típus őse a *TypedXmlData* osztály, melynek egyetlen funkciója a típusleírás biztosítása. Az általunk használt típusleíró nem a megszokott módon, közvetlenül írja le a típust, hanem az XSD sémáján keresztül, közvetetten. A gyakorlatban ez azt jelenti, hogy a metainformációk nem C++ nyelvű típusinformációk helyett XSD nyelvű típuskonstrukciókat tartalmaznak. Ezek azonban a megfelelő típushozzárendelési szabályok segítségével leképezhetők a C++ típuskonstrukcióira (lásd 5.3.4). Vegyük észre, hogy ezt a leképzést a kódgenerálás során minden 5.2. alatt megemlített rendszer végrehajtja. Mi ezen annyit változtattunk, hogy a leképzést nemcsak a kódgenerátor használja fel, hanem egyes részleteit a sorosító metaprogram is (lásd 5.3.5).

```
struct TypedXmlData
{
    virtual const MetaType& MetaInfo() const = 0;
    virtual ~TypedXmlData() {}
};
```

A típusleírón kívül az osztály mindössze egy virtuális destruktort tartalmaz, mely a leszármazottak biztonságos lebontását biztosítja. A *MetaType* tartalmazza a típus teljes XSD alapú leírását. Tárolását egy saját adatszerkezettel oldottuk meg, melynek definíciója a következő:

```
// --- Konstruktor helyettesítése
typedef TypedXmlData* (*AllocatorFunction)();

struct MetaType
{
    const char *nsUri;
    const char *localName;

    ConstructionType construct;
    AllocatorFunction allocFunc;

    bool isComplexType() const;
    const MetaComplexType& asComplexType() const;

    bool isSimpleTypeList() const;
    const MetaSimpleTypeList& asSimpleTypeList() const;

    ...
};
```

Minden típusleíró tartalmaz egy allokátort, mely paraméter nélküli konstruktorként szolgál az általa leírt osztályhoz. Erre azért van szükség, hogy a leírt típust annak for-

dítási idejű ismerete nélkül példányosíthassuk, ezt a típusleírók általában más nyelveken is támogatják. Tartalmazza a típus minősített (qualified) XSD-beli nevét, mely egy névteret meghatározó URI-ból (Uniform Resource Identifier) és egy benne definiált lokális névből áll. Biztosít továbbá segédfüggvényeket, melyek a különböző altípusokra (tehát egyes XSD típuskonstrukciókra) specifikus adatok lekérdezésére szolgálnak. Az objektum-orientáltság szerint ezt leszármazással kellene megvalósítanunk.

A metaadatok leírására több okból sem az objektum-orientált megközelítést választottuk. Metaadataink egyrészt statikus adatként vannak jelen a programban, melyek Symbian rendszereken konstruktorokkal nem, csak struktúra-inicializátorokkal tölthetők ki. Másrészt a beépített futásidejű típusvizsgálatok (*dynamic_cast*) C++ nyelven rossz hatékonyságúak, a fenti módszer ezen jelentősen javít.

A leszármazott típusok (pl. *ComplexType*) természetesen további adatokat is tartalmaznak, például a benne található adattagok (elemek és attribútumok) listáját, valamint az elemek sorrendezésének szabályait. Az adattagok leírását a *MetaDataHolder* típus tartalmazza, benne megtalálható az adat típusa, illetve a tartalmazó típuson belüli memóriacíme (offsetje) is.

A sémaleírás adatszerkezeteinek részletes bemutatásától terjedelmi okok miatt eltekintünk.

5.3.4. Típusmodell

Az XSD séma és a C++ nyelv típusai igen hasonlók, de korántsem egyformák. Ahhoz, hogy a sémadefiníció típusaiból C++ nyelvű típusokat generálhassunk, és segítségükkel az XML dokumentumok adatait kezelhessük, egyértelmű típushozzárendelésre van szükségünk. Mivel a megoldásunk alapvetően az XSD séma típuskonstrukcióit használja, és C++ nyelvű típusokhoz soha nem rendel XSD konstrukciókat, elég megadnunk az XSD típusok leképzését C++ nyelvre. A leképzés pontos definíciójához megadjuk egyrészt az alaptípusok hozzárendelési szabályait, majd a különböző típuskonstrukciókét, ezzel az összes lehetséges típust lefedjük.

Alaptípusok. Az egyszerű típusok (*simpleType*) hozzárendelése az összefoglaló 5.1. táblázatban látható.

Az alaptípusok hozzárendelési szabályait a kódgenerátor a többi kódtól függetlenül, egy fentihez hasonló, XML formátumú táblázatban tárolja. Ezen típushozzárendelési szabályok megváltoztatása tehát rendkívül egyszerű, csak a táblázat megfelelő elemeit kell módosítanunk. Mivel a többi hozzárendelési szabály már típuskonstrukciókat ad meg, és ezeket a jelenlegi megvalósításban táblázathoz hasonló deklaratív leírás helyett közvetlenül a kódgenerátor programkódja adja meg, megváltoztatásuk jóval nehezebb.

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`

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

Attribútumok és elemek. A generált programkód könnyebb átláthatósága kedvéért leképzésünk ebben az esetben nem izomorf. A C++ nyelvű adatreprezentáció szempontjából lényegtelen, hogy az XML dokumentumban egy adat külön elemként, vagy attribútumként tárolódik, ezért mindkét esetben egy egyszerű adattagra képezünk le.

Számosság. A sémadefinícióban megadott elemek előfordulásaik helyén tetszőleges multiplicitással rendelkezhetnek, melyet a *minOccurs* és *maxOccurs* attribútumok értéke határoz meg. Ilyenkor az általuk megadottnál kevesebb vagy több elem nem állhat a megadott helyen.

- Alapértelmezés szerint mindkét attribútum értéke 1, ilyenkor kötelező elemről van szó. Ehhez C++ nyelven legegyszerűbben a tartalmazás (aggregáció) rendelhető hozzá.
- Ha a *minOccurs* 0, a *maxOccurs* pedig 1, opcionális elemről beszélünk. Mivel a generált osztályokban alaphelyzetben nem mutatók segítségével, hanem közvetlenül, érték szerint tároljuk az adatokat, nem áll rendelkezésünkre speciális érték (pl. NULL) az üres érték jelzésére. Szükség van tehát a nem kitöltött adatok jelzésére. Ezt a *Nullable* nevű, saját könyvtári sablontípusban valósítottuk meg.
- Minden más esetben az elemet tömb típusra képezzük le, melynek szabványos C++

típusa az *std::vector*. Mivel a sorosítás általában az allokátor függvények segítségével dinamikusan hoz létre objektumokat, a vektor elemei legtöbbször referenciaszámlált mutatók. Ez alól egyedüli kivételt a táblázatban felsorolt alaptípusok jelentik, mivel ezek előre ismertek, kezelésükhöz nincs szükség allokátorokra, így egyszerűen érték szerint tárolódhatnak.

Konténerek. A sémaleírás az elemek háromféle csoportosítását ismeri: a sorozatot (*sequence*), a gyűjtemény (*all*) és az kiválasztást (*choice*). Utóbbi megfelel a programozási nyelvekben általában használt unió (*union*) típuskonstrukciónak, míg a sorozat a direkt szorzatnak (struktúra, rekord, néhol osztály), melyet C++ nyelven a struktúrák (*struct*) ábrázolnak. A C++ nyelv unió típusának korlátai miatt a kiválasztást inkább olyan opcionális (*Nullable*) tagokkal rendelkező struktúrára képezzük le, mely mindig pontosan egy kitöltött adattaggal rendelkezik. A gyűjtemény konstrukciót a programozási nyelvek közvetlenül nem támogatják, azonban a gyűjtemény is jól modellezhető struktúrával. Különbségük kizárólag sorosítás közben az XML dokumentumok olvasásánál, érvényességvizsgálatkor jelentkezik az elemek sorrendjének ellenőrzésénél, tehát a C++ nyelvű adattároló osztályokat nem érinti. A generált kódban a két konstrukció egyforma struktúrákra képződik le, kizárólag a metainformáció alapján különböztethetők meg.

Típuskonstrukciók. A legalapvetőbb egyszerű típusokat már összefoglaltuk az 5.1. táblázatban, de még adósok vagyunk a további *simpleType* konstrukciók leképzésével. A megszorított egyszerű típusokra (*restriction*) a hozzárendelés változatlan, csak a sorosító kód ellenőrzi a definiált megszorítások teljesülését. Az unió (*union*) a kiválasztás konténertípussal megegyező módon képződik le. Mivel a listák (*list*) elemei tetszőleges egyszerű típussal rendelkezhetnek (többek közt unió típusúak is lehetnek), ezért leképzési problémák miatt a listákat egyelőre nem támogatjuk, vagyis a dokumentumhoz hasonlóan karakterlánc típust rendelünk hozzájuk.

Az összetett típusokhoz (*complexType*) minden esetben egy új struktúra jön létre[6], akár egyszerű típusok kibővítéséről van szó (*simpleContent*), akár más valódi összetett típusokról (*complexContent*) van szó. Ha a valódi összetett típus kiterjesztés (*extension*), akkor a típust egyúttal a kiterjesztett típus leszármazottjaként is definiáljuk.

Összegzés. A fent megadott típushozzárendelési szabályok a végeredményt tekintve egyszerűek és könnyen használhatók. Léteznek 5.1. táblázatban látható egyszerű típusok

[6]Konténerek kizárólag összetett típusok definícióiban fordulnak elő, melyekhez szintén egy struktúrát rendeltünk. Így legtöbbször semmiféle funkcionalitással nem bíró, felesleges beágyazási szintekhez jutnánk. A könnyebb érthetőség és használhatóság kedvéért programkódunk az ilyen összetett struktúrákat összevonja (flattening), így a konténerek a generált programkódban csak a metainformációk szintjén maradnak meg.

(esetenként típusmódosítókkal, mint például *Nullable*, *vector*), valamint összetett típusok, melyek a *TypedXmlData* leszármazottjai, tehát nyilvános adattagokkal és önleírással rendelkező *struct*-ok.

5.3.5. Sorosító metaprogram

Mostanra már minden szabály és komponens adott a sorosítás működéséhez. Az ezt végző metaprogram végrehajtása a következő felületeken történhet:

```
// --- Beillesztés és kibontás XML dokumentumfához
void insertToDom(const TypedXmlData& object, XmlElement toNode);
void extractFromDom(TypedXmlData& object, XmlElement fromNode);

// --- Közvetlen sorosítás szöveges XML formátumhoz
std::string serialize(const TypedXmlData& object);
void deserialize(const std::string& xmlDocStr, TypedXmlData& toObject);
```

Az első esetben az objektumok sorosítása XML dokumentumok fa reprezentációjú, DOM (Domain Object Model [89]) alapú formátumára történik, paraméterei a sorosított objektum és az XML részfa gyökere[7]. Második esetben közvetlenül szöveges, karakterláncokkal ábrázolt XML dokumentumokra sorosítunk.

A felületekből látható a sorosítás elvének egy korlátja, miszerint az objektumot az olvasáshoz is előre kell példányosítanunk. Ez feltétlenül szükséges, mivel a sorosító algoritmusnak szüksége van az objektumok által biztosított pontos típusleírókra. Ez azonban esetünkben nem jelent különösebb megszorítást, hiszen mi szigorúan típusos C++ adatelérést biztosítunk az XML dokumentumokhoz, előre nem ismert típusokat pedig nem is kezelhetnénk szigorúan típusos felületeken.

A sorosítást végző futási idejű metaprogram a *TypedXmlData* típus 5.3.3. alatt bemutatott XSD formájú önleírása alapján dolgozik. Működésének alapelve hasonló a kódgenerátoréhoz, a típusinformáció alapján rekurzívan bejárja az objektum alkotórészeit, kódgenerálás helyett azonban az adott rész sorosítását végzi el. A típusok bejárása az XSD formátumú leírás alapján azért lehetséges, mert az XSD leírás alapján a metaprogram is elvégezheti az 5.3.4. alatt bemutatott leképzéseket, így pontosan ismerheti a feldolgozott típus szerkezetét.

A leképzéshez azonban szükség van arra is, hogy a metainformáció alapján típusosan hozzáférjünk a leírt adattagokhoz. Az XML dokumentum esetében a karakterlánc alapú hozzáférés az XSD leírás és a DOM interfész segítségével már adott. Közvetlen támogatás híján az adattagok elérését C++ nyelven kerülőúton kell megvalósítanunk, ezért az adattagok leírása a típus mellett tárolja az adattagot tartalmazó struktúrán belüli relatív

[7] Az *XmlElement* típusú paramétereket azért nem referencia szerint adjuk át, mert mindössze egy okos mutató típusról van szó, melynek másolása minimális költséggel jár.

címet (offset) is. Mutatóaritmetika segítségével így a tartalmazó címét a relatív címmel eltolva az adattagra állított típustalan (*void**) mutatót kapunk, az adattag pontos típusának ismeretében ez a kívánt típusra konvertálható. Alaptípusokra ez egyszerűen működhet az 5.1. táblázat alapján, összetett típusnál azonban a metaprogram nem ismerheti előre a generált típusokat. Ennél a pontnál kritikus, hogy minden összetett típus a *TypedXmlData* típus leszármazottjaként jön létre, így erre biztonságosan konvertálhatunk a sorosítás folyamán. A típus további szűkítése már nem szükséges, hiszen innen a típus önleírása rendelkezésre áll, az önleírás alapján pedig folytatódhat a rekurzió.

Az XSD leírás tárolásának további előnye, hogy a sorosítás garantálhatja az adatok tárolt formátumának érvényességét. A metaprogram a dokumentum írásakor a séma alapján annak pontosan megfelelő dokumentumot készít, olvasáskor pedig ellenőrzi a bemenetet, és nem megfelelő formátum esetén hibajelzéssel megszakítja az olvasást. Képes ellenőrizni többek között az elemek különböző sorrendezését (sorozat, gyűjtemény, kiválasztás), elemszámát, vagy az alaptípusok megszorításait.

A megvalósítás nehézségei miatt a sorosítás egyelőre nem támogatja a teljes XSD szabványt. A *simpleType* típusok hozzárendelési szabályainál (lásd 5.3.4.) megoldatlan problémaként említett lista és unió konstrukciókat a leképzés hiányossága miatt jelenleg a metaprogram sem képes megfelelően kezelni. Az ilyen típusú adatokat a sorosítás egyszerűen figyelmen kívül hagyja.

5.3.6. Műveletek generálása távoli szolgáltatásokhoz

Kódgenerátorunk a fentiekben bemutatottak mellett egy magasabb szintű problémára is megoldást ad, képes biztosítani távoli WSDL/SOAP [87] alapú szolgáltatások automatizált elérését is. Ezt azért képes megtenni, mert ezek a protokollok is az XML-t használják platformfüggetlen kommunikációra, így a probléma egy része visszavezethető a már megoldott sorosítási feladatra.

Távoli szolgáltatások WSDL formátumú leírásában leegyszerűsítve a szolgáltatás műveletei, valamint a hozzájuk tartozó bejövő és kimenő üzenetpárok vannak felsorolva. Ezen kívül tartalmaz információt az üzenetek kódolásáról és a hálózati (többnyire HTML) kommunikáció formájáról, ennek részletes tárgyalása azonban itt nem célunk. A szolgáltatás egy konkrét műveletének meghívásakor XSD sémaleírással megadott XML adatokat küldünk és kapunk vissza. Egyszerű példa egy szolgáltatás leírására:

```
<definitions ...>
  <types> <schema ...>
    <!-- A korábban bemutatott Agenda típus definíciójának betöltése -->
    <import namespace="..." schemaLocation="AgendaType.xsd"/>

    <complexType name="EmptyResponse"> <sequence /> </complexType>
```

```
    <element name="AddRequest" type="Appointment" />
    <element name="AddResponse" type="EmptyResponse" />
  </schema> </types>

  <message name="AddRequest">
    <part name="input" element="AddRequest"/> </message>
  <message name="AddResponse">
    <part name="output" element="AddResponse"/> </message>

  <portType name="Agenda"> <operation name="Add">
      <input message="AddRequest"/> <output message="AddResponse"/>
  </operation> </portType>

  <binding ...> <!-- Kódolási információk --> </binding>
  <service ...> <!-- Kommunikációs információk --> </service>
</definitions>
```

A kódgenerátor a szolgáltatás leírásából automatikusan elkészíti a kommunikációt elrejtő teljes kliens kódot és a kiszolgáló (server) kód vázát is. A Symbian platformra készített változatban a műveletek szinkron és aszinkron hívása is lehetséges, Linux alatt még csak a szinkron változatot támogatott.

A kommunikáció teljesen automatizált, a távoli szolgáltatások hívásához semmilyen kiegészítő programkódra nincs szükség. Szinkron híváskor a kliensben mindössze egy szolgáltatás helyét meghatározó URL (Uniform Resource Locator) függvényparamétert kell megadnunk a többi mellé. A fenti művelethez generált kliens függvényszignatúra lényege (névterek és néhány elnevezési konvenció elhagyásával) a következő:

```
auto_ptr<EmptyResponse> Add(const Appointment &request,
                            const Url &target);
```

A szolgáltatás által adott visszatérési értéknél az *auto_ptr* azt jelzi, hogy az eredményobjektum a függvénytörzsben újonnan jött létre, a tulajdonjogát a hívónak adja át. Ha a hívó másképp nem rendelkezik, a kódblokk végén az objektum automatikusan el is pusztul.

A kódgenerátor elkészíti a kiszolgáló (server) kód vázlatát is. A kiszolgálónak a kommunikációs részletekkel már nem, csak a szolgáltatást végző függvények törzsének kitöltésével kell foglalkoznia. A kiszolgálóhoz generált függvényszignatúra mindössze egy tranzakció függvényparaméterrel rendelkezik, melyen keresztül elérhetjük a kérés és a válasz adatait, esetünkben *Appointment* és *EmptyResponse* típusú objektumokat.

5.4. Összegzés

A fejezetben bemutattunk egy újszerű megoldást XML formátumú adatok sorosítására és típusbiztos elérésére, mely a típusok XSD alapú önleírására és egyetlen, általános sorosító metaprogramra épült. Ez a felépítés számos előnyt biztosít a problémára adott más megoldásokhoz képest.

Mivel nincs szükség külön sorosító algoritmusra minden egyes generált típushoz, kevesebb programkód jön létre. A kód így nemcsak tömörebb és könnyebben érthető, de más megoldásokhoz képest lefordított mérete és memóriaigénye is jóval kisebb (lásd [3, 4]), vagyis ideális szűk erőforrásokkal rendelkező platformok számára.

A sorosítás különleges működési elvének köszönhetően működése közben elvégzi az adatok ellenőrzését is, nincs hozzá szükség külön validációs fázisra vagy további függvényhívásokra.

A megoldás szerkezete moduláris, jól karbantartható, világosan különválasztja a különböző feladatköröket. Így például könnyen megváltoztatható az alaptípusok hozzárendelése, hozzáadhatók új típusok, lecserélhető a kódgenerátor vagy az XML dokumentumokat kezelő könyvtár. Mivel a típusok leírása nyilvános, azt szükség esetén akár más alkalmazásokban is felhasználhatjuk.

A fejezet az adatok XML formátumú sorosítására összpontosított, elsősorban a hálózati szolgáltatások elérésére céljából, mely a kidolgozott könyvtárak eredeti célkitűzése volt. A megoldás azonban semmilyen módon nem kötődik az XML formátumhoz: ha megadható az adatok sémaleírása, megfelelő típusleképzés és sorosító metaprogram kidolgozásával a módszer tetszőleges formátumra alkalmazható. A megoldás tehát formátum- és nyelvfüggetlen.

A fenti elvekre épülő, XML (SOAP/WSDL) formátumú megvalósítás a gyakorlatban is bizonyított, a Nokia az S60 platformra készült változatot WSDL to C++ Wizard néven hivatalos, szabadon letölthető [8] fejlesztőeszközeként adta ki 2006 folyamán.

3. Tézis. Nyelv- és formátumfüggetlen, moduláris módszert adtam sémaleírással rendelkező dokumentumok sorosítására. Leképzést definiáltam az XML sémaleírók típusrendszeréről a C++ nyelv típusrendszerére, valamint megvalósítottam egy leképzést elvégző kódgenerátort. Implementáltam a leképzésre épülő, XML dokumentumokat sorosító általános metaprogramot. Az elkészült könyvtárat a Nokia hivatalos fejlesztői eszközként adta ki S60 platformjára.

6. fejezet

Összegzés

A dolgozatban egy rövid bevezetés után áttekintést adtam a metaprogramozás alapfogalmairól és alkalmazásairól, majd saját munkáimat és eredményeimet ismertettem. Ezen munkák az erősen típusos objektum-orientált nyelvek korlátaival, illetve ezen korlátok metaprogramok segítségével történő meghaladásával foglalkoztak.

Mivel a metaprogramozás egy új, feltörekvő módszertan, így még nem rendelkezik kellőképpen kiforrott elméleti háttérrel és fejlesztőeszközökkel. A programkód metaadatainak elérése (lásd 2.2) a metaprogramozás alapvető eszköze, ám alig akad olyan programozási nyelv vagy környezet, mely ezt fordítás közben is támogatná. A 3. fejezet a támogatás elősegítését célozta meg. A metaprogramozás alapfogalmait, köztük a metaadatokat bemutató 2 fejezet alapján a 3. fejezetben megadtam egy általános, elsőrendű logikára épülő, általános típusvizsgáló rendszert. Megvalósítottam a C++ nyelvű programok alapvető típusvizsgálatait szabványos C++ nyelvű metaprogramok segítségével.

A metaprogramozás fontos alkalmazási területe a nyelvek típusrendszerének kiterjesztése, tulajdonságainak javítása. A 4. fejezetben először bemutattam az erősen típusos objektum-orientált nyelvek típusrendszerének gyengeségét a lépésenkénti finomítás módszerének alkalmazása esetén. Mivel ez a gyengeség a struktúrális altípusosság szabályainak alkalmazásával kiküszöbölhető, ezután C++ nyelven olyan automatikus konverziókat valósítottam meg, melyek a nyelv típusrendszerének kiterjesztésével a strukturális altípusosságot szimulálják. A megoldás sablon-metaprogramokkal automatizált kódgenerálásra épült, és kizárólag szabványos nyelvi eszközöket használt. Az igény szerinti kódgenerálás segítségével elkerülhető a hagyományos megoldások exponenciális osztályszámot eredményező bonyolultsága.

A metaprogramozás egy másik fontos alkalmazási területe az adatok sorosítása, melyre a típusok önleírásának felhasználásából kiindulva egy viszonylag jó, általános, formátumfüggetlen megoldás adható. A típusok önleírása a virtuális gépen, illetve értelmezőn alapuló programozási környezetekben (pl. Java, C#) legtöbbször adottság, ezek többnyire

rendelkeznek is szabványos sorosító könyvtárakkal. A könyvtárak közös jellemzője, hogy a nyelven megadott típusok alapján határozzák meg az elmentett adatok formátumát. A sorosítás más, a metaadatok elérését fordítási és futási időben sem támogató nyelveken jóval nehezebb. A 5. fejezetben bemutatott megoldás a másik irányt választotta, és a rendelkezésre álló adatleírásokhoz, sémákhoz készített programnyelvi típusokat. A megoldás nyelv- és formátumfüggetlen, ám a formátumot leíró séma típusainak leképzését igényli a programozási nyelv típusaira. Bemutattam a sorosítás működését XML formátumú adatokra, az algoritmust a típusok önleírását nem támogató C++ nyelven megvalósítva. A megoldás hasznosságát tovább növelte, hogy kis erőforrásigénye miatt alkalmazása ideális lehet telefonok, kéziszámítógépek és egyéb, kisebb kapacitású rendszerek egymás közötti kommunikációjára.

Remélem, hogy ezen eredmények hozzájárulnak majd a metaprogramozás fejlődéséhez, valamint további elterjedéséhez.

6.1. A dolgozat eredményei

1. Tézis. Definiáltam egy elsőrendű logikán alapuló, nem intruzív, univerzális típusvizsgáló (introspection) rendszert. C++ sablon-metaprogramok segítségével elkészítettem a rendszer egy konkrét megvalósítását, mely C++ nyelvű programkód típusvizsgálatára szolgál. A könyvtár az ISO/IEC 14882:1998 szabvány szerinti nyelvi eszközökre épül, ezért fordítófüggetlen és hordozható.

2. Tézis. Megmutattam a jelenlegi objektum-orientált nyelvek típusrendszerének korlátait osztályok lépésenkénti finomításának esetében. Szabványos sablon-metaprogramok segítségével a strukturális altípusosságon alapuló, kiegészítő konverziós szabályokat vezettem be a C++ nyelv típusrendszerébe, megoldást adva a lépésenkénti finomítás problémájára. A módszerrel elkerülhető a létrehozandó interfészek számának kombinatorikus robbanása.

3. Tézis. Nyelv- és formátumfüggetlen, moduláris módszert adtam sémaleírással rendelkező dokumentumok sorosítására. Leképzést definiáltam az XML sémaleírók típusrendszeréről a C++ nyelv típusrendszerére, valamint megvalósítottam egy leképzést elvégző kódgenerátort. Implementáltam a leképzésre épülő, XML dokumentumokat sorosító általános metaprogramot. Az elkészült könyvtárat a Nokia hivatalos fejlesztői eszközként adta ki S60 platformjára.

6.2. További kutatási irányok

Ahogy egyetlen programozási megoldás sem lehet minden szempontból tökéletes, így természetesen az előző fejezetekben bemutatottak sem azok. A kutatás legfontosabb további irányának mégsem ezek további javítását, fejlesztését gondolom. A kutatási munka során minduntalan olyan nehézségekbe, korlátokba ütköztünk, melyek messze nem a metaprogramozás határainak eléréséből, hanem újszerűségéből, gyermekbetegségeiből adódnak. Azt gondolom tehát, hogy a metaprogramozás vizsgálata és fejlesztése, módszereinek megalapozása lehet a további kutatások legfontosabb iránya.

A programozási paradigmák, módszerek tervezésének legfontosabb szempontjai a születő programok fejlesztésének és karbantartásának megkönnyítése, valamint a program megbízhatóságának, biztonságának növelése. A metaprogramozás esetén ilyenről gyakorlatilag szó sincs, az irodalom hiányossága a témában egészen megdöbbentő. Mivel a metaprogramozás leggyakrabban még mindig viszonylag ad-hoc jellegű megoldásokon alapul, súlyos módszertani hiányosságokkal rendelkezik, mely valószínűleg a megfelelő eszközök, illetve részben a megfelelő elméleti alapok hiányának tudható be. Fontosnak tartom tehát megtalálni azokat a módszereket, melyek segítségével jól tervezett, helyes és karbantartható metaprogramok írhatók.

Ennek fontos része volna egy jól használható programozási modell felállítása. Jelenleg léteznek imperatív (pl. OpenC++ [66]), funkcionális (EClean [10]), illetve deklaratív (Stratego/XT, lásd 2.5.6) paradigmán alapuló metaprogramozási rendszerek is. Nem ismert azonban, hogy ezek a paradigmák mennyire alkalmasak metaprogramozási feladatok elvégzésére, milyen előnyeik, hátrányaik vannak speciálisan a metaprogramozás esetén. Fontos lenne tehát az elméleti alapok felállítása.

Szintén számtalan kidolgozatlan területtel rendelkezik a metaprogramok fejlesztésének módszertana is. A hagyományos programok fejlesztésénél már ismert, régóta használt, jól bevált fogalmak, eszközök itt még ismeretlenek. Nincsenek metaprogramok bonyolultságot mérő metrikáink, automatizált tesztkörnyezeteink vagy metaprogramozást célzó fejlesztőkörnyezetek (integrated development environment), gondot okoz a metaprogramok nyomkövetése és teljesítménymérése is. Kutatások ugyan már folynak a témában, ezek azonban még csak igen korai fázisban vannak. C++ nyelvre nemrégiben készült el a nyomkövetés és teljesítménymérés kezdetleges formáját nyújtó TempLight [9] rendszer prototípusa, sok más környezetben még semmilyen eszköz nem áll rendelkezésre.

A. Függelék

Összefoglalás

In this thesis work I discuss shortcomings of most strongly typed object-oriented type systems. After identification of drawbacks, metaprogramming is applied to extend the type systems in order to overcome these drawbacks. Being the most widespread method recently, using C++ templates was a natural selection to implement metaprograms.

I defined a universal, non-intrusive type introspection system based on first order logic. I implemented type introspection for the C++ language. Showing the expressive power of C++ templates, I used only template metaprograms in the implementation. This introspection library conforms to the ISO/IEC 14882:1998 standard, thus it is compiler-independent and portable.

I described a problem that appears in most object-oriented type systems while using stepwise refinement. Similarly to diamond-shape inheritance, it can be described by a certain graphical pattern, thus we called it chevron-shape inheritance. Solving the chevron-shape inheritance problem usually needs an exponential number of classes or interfaces in conventional object-oriented type systems. I applied standard C++ template metaprograms for the automated generation of type conversions to extend the type system of the C++ language. These conversions implement typing rules similar to structural subtyping. As a result, I was able to give a solution with a linear number of generated classes.

I gave a general method independent of language and format for the serialization of schema-described data. I mapped the type system of XML Schema Descriptions to the C++ programming language and implemented a code generator in XSLT to perform this mapping. Based on the mapping, I implemented a general metaprogram to execute the serialization. The result is published as an official development tool for web services for the S60 platform by Nokia.

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] Szabolcs Payrits, Péter Dornbach, István Zólyomi. Metadata-Based XML Serialization for Embedded C++. International Journal of Web Services Research (IJWSR), megjelenés alatt.

[5] 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.

[6] 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.

[7] István Zólyomi, Zoltán Porkoláb. A generative approach for family polymorphism in C++. ICAI'04 5th International Conference on Applied Informatics, Ed. Lajos Csőke et al. Eger, 2004., pp. 445-454.

[8] Nokia WSDL-to-C++ Wizard for S60. `http://forumnokia.com/info/sw.nokia.com/id/5ddeb939-c4e4-4e64-8f25-282e1e86afed/Nokia_WSDL_to_Cpp_Wizard_for_S60.html`

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

[10] Ádám Sipos, Viktória Zsók, Zoltán Porkoláb. Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms. Studia Universitatis Babes-Bolyai Informatica LIII, 2008/2, Cluj-Napoca, 2008, pp. 55-66.

[11] Mihály Biczó, Krisztián Pócza, Zoltán Porkoláb. Runtime Access Control in C\# 3.0 Using Extension Methods. Proceedings of 10th Symposium on Programming Languages and Software Tools, 2007, pp. 45-60.

[12] Tamás Kozsik, Zoltán Csörnyei, Zoltán Horváth, Roland Király, Róbert Kitlei, László Lövei, Tamás Nagy, Melinda Tóth, Anikó Víg. Use cases for refactoring in Erlang. Lecture Notes in Computer Science (ISSN 0302-9743), vol. 5161, pp. 264-298.

[13] 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.

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

[15] Krzystof Czarnecki, Ulrich Eisenecker, Robert Glück, Daveed Vandevoorde, Todd Veldhuizen. Generative programming and active libraries. LNCS vol. 1766, 2000, pp. 25-39

[16] Zoltán Juhász, Ádám Sipos, Zoltán Porkoláb. Implementation of a Finite State Machine with Active Libraries in C++. Proceedings of GTTSE 2007, Lecture Notes in Computer Science 5235, pp. 474-488.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] 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.

[36] 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

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

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

[39] 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.

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

[41] 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.

[42] 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.

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

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

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

[46] 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.

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

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

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

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

[51] Walid Taha. A Gentle Introduction to Multi-stage Programming. Proceedings of GTTSE 2007, pp 260-290.

[52] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. ACM SIGPLAN Haskell Workshop 2002, pp. 1-16, ACM Press

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[73] 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.

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

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

[76] Ádám Sipos, Norbert Pataki, Zoltán Porkoláb. On multiparadigm software complexity metrics. Pu.M.A vol. 17 (2006), No 3-4, pp. 469-482.

[77] Ákos Fóthi, Judit Nyéky-Gaizler, Zoltán Porkoláb. The Structured Complexity of Object-Oriented Programs. Mathematical and Computer Modelling, Volume 38, Number 7, October 2003, ISSN 0895-7177, pp. 815-827 (13).

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

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

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

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

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

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

[84] 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`

[85] W3C World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/2006/REC-xml-20060816/

[86] W3C World Wide Web Consortium. XML Schema. Specifications and Development. http://www.w3.org/XML/Schema#dev

[87] W3C World Wide Web Consortium. Web Services. Recommendations. http://www.w3.org/2002/ws/#documents

[88] W3C World Wide Web Consortium. XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/

[89] W3C World Wide Web Consortium. Document Object Model. http://www.w3.org/DOM/DOMTR

[90] Gavin Bierman, Erik Meijer, Wolfram Schulte. The essence of data access in C omega. Proceedings of ECOOP 2005, LNCS 3586, pp 287-311.

[91] Robert A. van Engelen, Kyle Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002), pp. 128-135.

[92] Codalogic LMX W3C XML Schema to C++ Data Binding. http://www.codalogic.com/lmx/

[93] Code Synthesis XSD: XML Data Binding for C++. http://codesynthesis.com

[94] Liquid XML Data Binding. http://www.liquid-technologies.com/Product_XmlDataBinding.aspx

[95] Saxon: The XSLT and XQuery Processor. http://saxon.sourceforge.net/

[96] Erik Ernst. Family Polymorphism. Proceedings ECOOP 2001, LNCS 2072, pp 303-326

[97] Erik Ernst. gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[98] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys, Volume 17, Number 4, December 1985, pp. 471-522