Hatékony metaprogramozás

Sipos Ádám
programtervező matematikus szak
Témavezető: Porkoláb Zoltán

ELTE-IK, 2004

Tartalomjegyzék

Abstract

A generatív programozás napjaink egyre népszerűbb programozási paradigmája. Egyes részei (aspektus-orientált programozás, szándék-alapú programozás és generikus programozás) már kiléptek a kísérleti stádiumból, és elfogadott technológiává válnak. A C++ template metaprogramozás is most éli ezt a szakaszát. A template metaprogramok használata során a futási idejű algoritmusok egy részét fordítási idejű tevékenységgel helyettesítjük, és segítségükkel hatékonyabb kódot (expression templates), könnyebben bővíthető könyvtárakat és fordítási idejű program-adaptációt hozhatunk létre.

Ugyanakkor a template metaprogramozásban még nem alakultak ki széles körben elfogadott programozási módszertanok, eszközök, könyvtárak, amelyek a generatív programozás ezen ágát támogatnák. Az egyes megoldások gyakran ad-hoc jellegűek, a hibajavítások heurisztikusak, ezáltal a projekt ráfordítások, fejlesztési költségek nehezen becsülhetőek.

Dolgozatunk célja áttekintő elemzés a template metaprogramozás jelenlegi helyzetéről és azon módszerek ismertetése, melyek segítségével hatékony és megbízható metaprogramokat írhatunk. Kiemelten vizsgáljuk a metaprogram-projektek életciklusának teszteléssel, hibakereséssel és javításával kapcsolatos részeit, melyeken lényeges kutatások eddig még nem történtek. Mivel ez teljesen szűz kutatási terület, dolgozatunk másodlagos célja, hogy felhívja a figyelmet erre a kritikus területre és prototípus-jelleggel eszközöket adjunk a felhasználó kezébe, melyek további fejlesztések alapjai lehetnek.

Definiáljuk a metaprogramok helyességét és a specifikációtól való eltérés lehetséges fajtáit. Ismertetjük azokat a módszereket, amelyek a fejlesztés során támogatnak bennünket a specifikációnak megfelelő programok írásában. Javaslatot teszünk egy új konstrukciójú static assert megvalósítására, ami bővebb információt szolgáltat a jelenleg használatosaknál.

A legkorszerűbb módszerekkel sem zárható ki, hogy a fejlesztés során hibát kövessünk el. Ezért a dolgozatban a template metaprogramok hibakeresési lehetőségeit is tárgyaljuk. Áttekintjük, miért nem működnek a futási idejű programvizsgálati eszközök. Helyettük két javaslattal élünk, melyeket demonstratív céllal implementáltunk is: módosítottuk az egyik népszerű C++ fordítót, hogy nyomkövetési információkat biztosítson a példányosuló template-ekről. Ezáltal lehetővé válik a példányosítások nyomkövetése non-intrusive módon. Ez a legtöbb esetben életbevágó információ a fejlesztő számára, mely eddig elérhetetlen volt. Másik módszerünk a C++ nyelvi szabvány keretein belül maradva, speciális nyelvi konstrukciót használva írat ki információt a példányosulásokról fordítási időben.

1. Template-ek, metaprogramok

1.1. Template-ek

A *template* a C++ nyelv nagy kifejezőerejének egyik legfontosabb összetevője, melyet viszont sokszor félreértenek, rosszul, vagy egyáltalán nem is használnak. E fejezetben rövid áttekintést adunk róluk.

A template a kód-újrafelhasználás, illetve a magasabb absztrakciós szintek használatának elősegítésére került a nyelvbe. Programozás során gyakran előfordul, hogy különböző osztályok, vagy algoritmusok váza megegyezik, csak az általuk felhasznált típusok különbözőek. Ilyen például egy egész számokat, illetve egy kutyákat tartalmazó lista adatszerkezet. Kevés nyelv ad lehetőséget a kézenfekvő absztrakció leírására. C++ban a *template*-ek segítségével hozhatunk létre egy olyan konstrukciót, amely e típusok közös részét egy sablonná formálja, és a hivatkozott típust mint argumentumot kezeli.

Az előnyök nyilvánvalóak: azzal, hogy csak egyszer implementálunk bizonyos konkrét típusok közös műveleteit, illetve algoritmusait, csökkenthetjük a hibák megjelenésének valószínűségét és könnyíthetjük a kód követését. C++ban következőképpen nézhet ki egy típusokkal paraméterezhető struktúra definíciója:

```
template <class T>
class list
{
public:
    list();
    void insert(const T& x);
    T first();
    void sort();
    .
    .
    .
};
```

Template paraméternek nevezzük a template kulcsszó után két "kisebb" és "nagyobb" jel (*angle bracket*) között a `class` vagy `typename` kulcsszavakkal prefixelt változónevek listáját. Jelen esetben `T` az egyetlen template paraméter. `T` leírásával hivatkozhatunk arra a típusra, amelyből álló listát később használni szeretnénk. Egyelőre természetesen ez a típus ismeretlen számunkra, bármelyik beépített, vagy felhasználói típus lehet.

Ahhoz, hogy ebből a típus-sablonból konkrét típust hozzunk létre, *példányosításra* (*instantiation*) van szükség. Ez történhet a fordító által, vagy explicit módon is. A fordító akkor példányosít egy template-et, ha az abból létrejövő típusra szükség van. Például:

```
int main()
{
    .
    .
    .
    list<int> li;
    li.insert(1928);
}
```

A `list` template mögött angle bracketek között felsorolt konkrét típusok a class template *argumentumai*. E programsorhoz érve fogja a fordító behelyettesíteni a template törzsébe a konkrét típusokat és lefordítani a létrejövő kódot (feltéve, hogy előtte nem hivatkoztunk már valahol `list<int>`-re).

Explicit példányosításra példa a következő részlet:

```
template class list<kutya>;
```

Ekkor a fordítót utasítjuk, hogy az adott ponton (mely csak a globális névtérben lehet) hozza létre az adott típust, ha addig nem tette meg. Ezzel növehetjük a program hatékonyságát (több ilyen explicit példányosítást összegyűjtve a fordítónak nem kell megszakítania egy-egy kifejezés fordítását, hogy létrehozza a benne levő típust) és a *template metaprogramozásnak* (ld 1.2) is hasznos eszköze lehet.

Általában egy template típus-argumentumairól semmit sem tudunk előre. A template-ekkel szemben az egyik legfőbb kritika az, hogy semmilyen kikötést nem tesznek a későbbi típus-argumentumokra a fejlécükben. Igy csak jóval később derülhet ki, hogy adott típus megfelel-e a vele szemben implicit módon támasztott követelményeknek. Ilyen lehet példánkban, hogy definiált legyen rajta egy rendezés, mely a `sort` függvényhez valószínűleg szükséges lesz.

Egyes programozási nyelvek, pl az Ada a fenti probléma megoldására az ún sablon-szerződés modellt követik. Ennek segítségével előírást adhatnak a template argumentumra fordítási idejű hibát okozva, ha a paraméter megszegi a "szerződést", akkor is, ha a szerződésszegő függvényt nem használjuk.

Vegyük azonban észre, hogy a sablon-szerződés modell egyben erős megszorítással is jár. Kizárja ugyanis annak lehetőségét, hogy a sablonnak csak

azt a részét használjuk, melyet az argumentum teljesít. A C++ nyelv létrehozásakor Stroustrup szándékosan vetette el a sablon-szerződés modellt [5].

Ugyanabból a templateből legyártott két típus csak akkor egyezik meg, ha minden template argumentum megegyezik. Tehát

```
template <class T>
class A
{
    int n;
};
```

`A<int>` típus soha nem lehet egyenlő `A<kutya>` típussal, annak ellenére, hogy ugyanabból a sablonból generálja őket a fordító.

A C++ nyelv egyedülálló az ún. *specializáció* és *részleges specializáció* lehetőségének bevezetésében. Tegyük fel, hogy valamely típusra (legyen ez most az `int`), melyből listát szeretnénk készíteni, találunk egy hatékonyabb, a típusra specifikus implementációt. Ekkor megtehetjük, hogy a `list` template-et specializáljuk a típusra a következőképp:

```
template<>
class list<int>
{
public:
    list();
    void insert(const int& x);
    int first();
    void sort();
    .
    .
    .
    // itt egy teljesen más implementáció, reprezentáció lehet
};
```

Ne feledjük, hogy a template-ek csak sémák arra nézve, hogy majd a későbbiekben hogyan csináljon újabb, immár futtatható kódot belőlük a compiler. Tehát mikor explicit módon definálunk a `class list<int>`-et, akkor már nem egy sémát, hanem egy konkrét típust definiálunk. Igy egyrészről szabad kezet kapunk, hogy mi kerül a specializáció törzsébe, tehát nem köt

minket az eredeti template definíció.[1] Másrészt, ezek után egy `list<int>`... kifejezés kiértékelésekor a már meglévő, általunk definiált típust fogja használni, előnyben részesítve azt a sémával szemben.

Részleges specializációnak nevezzük az olyan specializációt, melyben konkrét típusokkal együtt template paraméterek ugyan továbbra is lehetnek, de ezekre tett szintaktikai megkötésekkel szűkítjük a lehetséges argumentumok körét. Például írhatunk egy `list` részleges specializációt a következőképpen:

```
template <class T>
class list<T*>
{
public:
    list();
    void insert(const T*& x);
    T* first();
    void sort();
    .
    .
    .
    // itt egy újabb implementáció és reprezentáció
};
```

A `T*`-ra történő specializálás jelenti azt, hogy amennyiben `list`-et valamilyen pointerrel példányosították éppen, akkor ezt a legújabb implementációt fordítsa be a compiler. Igy például írhatunk olyan destruktort ennek a részleges specializációnak, amely a mutatott objektumokat is törli, megakadályozva igy a memóriafolyást. Nyilván ennek a műveletnek csak akkor van értelme, ha speciálisan pointereket tárolunk.

Fontos tulajdonsága a template definícióknak, hogy két menetben fordítja le őket a compiler. Az első menetben a lexikális, szintaktikai és szemantikai elemzések lefutnak, de a fordító csak korlátozottan ellenőrzi a *függő-neveket* (*dependent name*; template paramétert tartalmazó kifejezés), hiszen a konkrét argumentum-értékeket nem ismeri, amíg egy példányosítás nem történik. Ekkor újrafordítja az érintett template-et, most már a paraméter helyébe a megfelelő típust írva, és ha sikerrel járt, kódot generál.

[1] Igy történhetett, hogy a *Standard Template Library* `vector<bool>` típusa, melynek explicit specializációját is megírták - részben a specializációt demonstrálandó, részben hatékonysági szempontok miatt -, teljesen máshogy működik, mint bármely más `vector` típus. Épp ezért gyakorlatilag használhatatlan.

Legalábbis ezt mondja ki a szabvány. Teljesen fordítófüggő ugyanis, hogy az elemzés milyen vizsgálatokra terjed ki. Az általunk vizsgált két korszerűnek mondható compiler, a Microsoft Visual C++ 7.1 (továbbiakban MSVC 7.1) és a GNU g++ 3.4 között nagy különbségek voltak már itt, az elemzés közben is. A MSVC 7.1 (és természetesen régebbi társa, MSVC 6 is) csak lexikális elemzést végzett a template definíció kiértékelésekor, majd a teljes szintaktikai és szemantikai elemzést csak az első példányosítás-kísérletnél hajtotta végre. Ez teljesen ellentmond a szabványnak és a helyesen működő g++ eljárásának.[2]

Újabb egyedülálló tulajdonsága a template-eknek, hogy nem csak típusokkal paraméterezhetők. Lehetnek egész értékek, függvénypointerek, sőt akár egy másik template is. A szabvány egyelőre nem támogatja a lebegőpontos számok template paraméterként való alkalmazhatóságát, de a legtöbb fordító tudja ezeket is kezelni.

Számos más módon használhatók még a template-ek, (például készíthetünk olyan class template-et, amely argumentumként kapott másik típusból örököl) ezek az alkalmazások nem képezik vizsgálataink tárgyát.

Meg kell még említenünk két olyan nyelvi elemet, melyek nélkül a metaprogramozás szinte elképzelhetetlen lenne, még ha elsőre nem is látszik jelentőségük.

A `typedef` kulcsszóval egy új álnevet adhatunk egy adott típusnak. A `typedef` nem hoz létre új típust, nem vizsgálja annak szintaktikai helyességét, tehát egy még definiálatlan, de már deklarált típusnak is lehet álnevet adni. Nemcsak a könnyebb áttekinthetőséget segíti egy `typedef` deklaráció (mint ahogy azt futási idejű programoknál megszokhattuk). Sokszor egy metaprogram működése teljesen azon múlik, hogy a programrészek tudnak-e egymás mezőire hivatkozni, esetleg úgy, hogy azokat explicite kifejezni nem is tudnák, hiszen nem ismert számukra az értékük.

A `enum` kulcsszóval pedig a már C-ből is jól ismert új felsorolási típust hozhatunk létre. A metaprogramok általában ebben tárolják az adatokat.[3]

Tisztáznunk kell a C++ template mechanizmusa és más objektum-orientált nyelvek – mint pl. a Java – un. *generic* megoldása közötti különbségeket. A C++ (és a hasonló elveken alapuló ADA és Eiffel)

[2]Ez templateket használó C++ programok esetében nagy problémát okozhat, amennyiben a programot mindkét platformon használni akarjuk, hiszen egy MSVC-n tökéletesen működő program teljesen hibás lehet g++ alatt, és ez akár a kód nagy részének újratervezését is jelentheti.

[3]Az adattárolás elképzelhető `const static` objektumokban is, de az `enum` használata hatékonyabb.

a template-eket egyfajta gyártási eljárásnak (sablonnak) tekinti. Amikor a template paraméterek helyén egy konkrét típus jelenik meg, a fordító a sablon alapján legyárt egy új kódrészletet (eljárást vagy osztályt). Ez a folyamat a példányosítás.

A Java (és a C#) *generic* módszere más elveken alapul. Tekintsük az alábbi Java forráskód-részletet:

```
public String loophole(Integer x) {
    List<String> ys = new LinkedList<String>();
    List xs = ys;
    xs.add(x); // figyelmeztetés fordítás közben
    return ys.iterator().next();
}
```

Formálisan ez a kód igen hasonló a megfelelő C++ megoldáshoz. Míg azonban a C++ fordító új kódot generálna a `LinkedList String` argumentumú példányosításából, a fenti Java forrásból az alábbi generic-mentes kód jön létre:

```
public String loophole(Integer x) {
    List ys = new LinkedList();
    List xs = ys;
    xs.add(x); // figyelmeztetés fordítás közben
    return (String) ys.iterator().next();
}
```

Látható, hogy a típusinformációk elvesznek, és az összes template argumentum a közös bázisosztályra, `Object` típusúra konvertálódik. Ezt a műveletet nevezzük *type erasure*-nek. A Java generic garantálja a típusbiztonságot: nem helyezhetünk el `Integer` objektumokat egy `LinkedList<String>`-ben, de ha áttérünk az `xs` listára, ez már nem áll fenn tovább.

Miután a generic nem példányosít, nem áll módunkban specializációt sem írni, igy a C++ -ban szokásos módon nem is metaprogramozhatunk Javaban (ld 1.2). Mindamellett a C++ template metaprogramozás egyes résztechnikáit sikeresen implementálták Java nyelven is [2].

Összefoglalva tehát a template-ekkel egyfajta gyártási eljárást definiálhatunk a fordító számára, amelyből az egy adott tulajdonságokkal rendelkező típust tud generálni, majd ebből a típusból már objektumokat tud

létrehozni. Nem csak class template-k léteznek, hanem struct és union, valamint függvény template-ek[4] is. A metaprogramozás eszközei legnagyobbrészt struct és a class template-ek, melyekkel a dolgozatban részletesen foglalkozunk.
A függvény template-eket ilyen célra nem használjuk, de ha egy programban több függvény template hivja egymást, ugyanolyan bonyolult lehet az esetleges hibák megtalálása, mintha egy metaprogramot próbálnánk debuggolni.

1.2. Metaprogramok

A C++ template-ek fentebb ismertetett tulajdonságait a típussal való paraméterezés igénye hívta elő. A fő cél a kód duplikálásának elkerülése, az egyszerűbben karbantartható program volt. A fentieknek megfelelő template specifikáció ezek alapján 1994-ben bekerült a nyelv szabványtervezetébe.

Még ebben az évben történt, hogy a szabványbizottság elé került egy Erwin Unruh által készített speciális program [14]. A program fordítása közben a compiler egy `int` paraméterű template példányosítását végezte el egy megadott felső határig `1`-től indulva, majd egy típuskonverziós hibára hivatkozott egy error formájában. Ám ez az error csak azon létrejövő típusokra generálódott, melyek argumentuma prím volt. Ez a működés nem kis megdöbbenést keltett a C++ alkotóiban, hiszen a statikus típusrendszer hirtelen "életre kelt" és egy működő metaprogramot hozott létre.[5]

Mint említettük, a fordító abban az esetben fog példányosítani egy sablon alapján, ha egy adott kifejezés, vagy deklaráció kiértékelésében számára ismeretlen típus van, és a típus létrehozható valamelyik template definícióból. Érezhető, hogy megfelelően megfogalmazott deklarációkkal meghatározhatjuk, hogy egy adott template a program mely pontján példányosuljon.

Az alábbiakban megmutatjuk, hogyan vezérelhetjük template-definíciók segítségével a fordítóprogram tevékenységét. Tekintsünk egy példát. A faktoriális-számítás egy klasszikus megoldása a rekurzió használata:

[4] A függvény template-eket szokás a *parametrikus polimorfizmus* eszközeként megjelölni. Itt arról van szó, hogy maguk a típusok is paramétereivé válnak egy függvénynek, így generikus módon implementálhatunk algoritmusokat

[5] Az Interneten megtalálható, a programot implementáló 2 forráskód közül egyiket sem tudtuk működésre birni a 3 vizsgált compiler közül egyikkel sem. Ennek oka a nyelv specifikációjában azóta történt jelentős változás.

```
int Factorial(int n)
{
    if (n==1) return 1;
    return n*Factorial(n-1);
}

int main()
{
    int r=Factorial(5);
}
```

Látható, hogy a `Factorial` függvény rekurzív módon fogja kiszámítani a `Factorial(5)` értéket, önmaga többszöri hívásával, majd mikor `n` értéke 1 lesz, visszatér 1-el és visszafelé összeszorozza `n`-ig az értékeket. Mindez futási időben történik.

Tekintsük most a következő kódrészletet:

```
template <int N>
struct Factorial
{
    enum { value = N*Factorial <N-1>::value };
};

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main ()
{
    int r=Factorial<5>::value;
}
```

Mi történik ekkor? A két template definíció elemzése után a fordító megpróbálja kiértékelni a főprogramban lévő értékadó kifejezést. Ahhoz, hogy a `Factorial<5>::value` kifejezés értékét meghatározza, a számára egyelőre definiálatlan `Factorial<5>` típust kell létrehoznia. Megkeresi, hogy létezik-e ilyen nevű template, amiből a gyártást megkezdheti. A template létezik, tehát `N` helyébe `5`-öt írva lefordítja az adott kódrészletet. Ahhoz, hogy az ebben elhelyezkedő `Factorial<N-1>::value`-t (vagyis most

`Factorial<4>::value`-t) ki tudja számítani, példányosítania kell a `Factorial<4>` típust és így tovább. Végül, mint azt szintén a 1.1 fejezetben említettük, a már létező `Factorial<1>` típust fogja felhasználni és nem próbálja meg azt az általános template eljárásból példányosítani.

Miben más ez a második megoldás a függvényt használóhoz képest? A legszembetűnőbb különbség a jóval bonyolultabb és szokatlan szintaktika. Ami ennél fontosabb és már nem ennyire nyilvánvaló, az, hogy a rekurzív template példányosítások "kikényszerítésével" fordítási időben rendelkezésre áll a `Factorial<5>` típus `value` értéke, igy a fordító már csak a `120` konstans értéket fogja beírni a generált kódba, nem pedig `Factorial<5>::value`-t. Ez azt jelenti, hogy `r` értékadása lineáris idejű műveletből konstans-idejűvé szelídült. Ezek szerint fordítási időben sikerült elvégeznünk egy `O(n)` bonyolultságú műveletet, ami ugyan a fordítást (esetleg jelentősen, ld [11]) lelassíthatja, viszont ha egyszer a fordító generálta a kódot, utána futási időben már nem lesz szükség műveletek elvégzésére (az értékadáson kívül).

Nehéz meghúzni a határt egy ügyesebb template és egy *Template metaprogram* (továbbiakban TMP) között. Mi ezt a kifejezést olyan templateek, példányosításaik és specializációik együttesére használjuk, melyeket azzal a céllal építünk a programba, hogy azokkal művelet(ek)et végezzünk fordítási időben, vagyis melyek nem azt a célt szolgálják, hogy egy adott típusból (vagyis C++ classból vagy structból) egy sablont készítsünk. Ez nyilvánvalóan csak egy logikai tagolás, ennek gyakorlati problémájával foglalkozik 2.2.

Miután a metaprogramunk fordítási időben hajtódik végre, értelemszerűen más objektumokat kezel mint a futási idejű programok. Az alábbi táblázatban megpróbálunk párhuzamot vonni a metaprogramok és a futási idejű programok entitásai között.

Metaprogram	Futási idejű program
(template) osztály	függvény
static const és enum osztálytagok	adat
szimbolikus nevek (típusnevek, typedef-ek)	változó
static const értékadás enum definíció	értékadás

Az első metaprogram óta a TMP kifejezőereje nem várt új távlatokat nyitott. A mai napig a template-ekkel kapcsolatos problémák képezik a C++ nyelvvel kapcsolatos legaktuálisabb kutatási témákat, a bennük rejlő

lehetőségeket még csak most kezdjük megérteni. Ez olyannyira igaz, hogy a TMP Turing-teljes voltát is bizonyították: a Bohm-Jacopini–tétel szerint Turing-teljes az olyan nyelv, amelyben létezik elágazás és rekurzió. Próbáljunk fordítási idejű elágazást csinálni:

```
template <bool Cond, class TrueType, class FalseType>
struct IF
{
    typedef TrueType type;
};

template <class TrueType, class FalseType>
struct IF<false,TrueType,FalseType>
{
    typedef FalseType type;
}
```

A fordítási idejű elágazás alkalmazásai lehetnek igen egyszerűek (pl megvizsgálható, hogy a `int` és a `long` típusok ugyanakkora méretűek-e), vagy éppen igen bonyolultak, meglepőek (pl egy osztály fordítási időben dönti el, hogy két osztály közül melyikből öröklődjön).

A futási idejű elágazástól igen különböző ez az új, fordítási idejű `IF`. Megszoktuk, hogy a feltétel kiértékelése után legfeljebb az egyik ág fog lefutni. A template-es `IF`-nél viszont mindkettő példányosul!

Tekintsük a következő kódot:

```
struct Stop
{
  enum { value = 1 };
};

template <int N>
struct BadFactorial
{
  typedef IF<N==0,Stop,BadFactorial<N-1> >::value  PrevFact;
  enum { value=(N==0) ? PrevFact::value : PrevFact::value*N };
};
```

Tegyük fel, hogy a `BadFactorial<5>::value` értéket szeretnénk megkapni, amely reményeink szerint továbbra is `120`. `BadFactorial<5>` példányosítása közben először a `typedef`-fel új nevet adunk az `IF` elágazásnak,

mint típusnak. A gond az `enum` kiértékelésénél van. Ahhoz, hogy `PrevFact::value` értéket megkapjuk, példányosítani kell az `IF`-es kifejezést. Az `IF` mint template csak akkor példányosítható, ha az argumentumai konkrét típusok (nem számítva persze a nem-típus paramétereket). Tehát az `IF<5==0, Stop, BadFactorial<4> >` kifejezés kiértékeléséhez először szükség lesz `BadFactorial<4>` típusra is. Ennél a pontnál az történik amit szeretnénk, vagyis az elágazás második ága lesz aktív. A probléma az, hogy ennek semmi köze az `5==0` kifejezés igazságértékéhez, ugyanis az ennek eldöntésére képes `IF` típus még *létre sem jött*. A gond világosabbá válik, mikor rekurzívan eljutunk `BadFactorial<0>::value`-ig, `IF<0==0, Stop, BadFactorial<-1> >::value` példányosításához. Hiába válik igazzá a feltétel, az `IF` nem tudja megvizsgálni, ugyanis példányosításához szükség van `BadFactorial<-1>::value`-ra is. Látható, hogy egy végtelen rekurzió fordítása történik. A végtelen rekurzióval foglalkozunk még 2.2-ben.

A fenti metaprogram fordítási időben végtelen rekurziót hajt végre, melynek befejezése akkor történik, amikor a fordító eléri implementációs határait, vagy rosszabb esetben feléli a rendszer összes erőforrását. A fenti program tehát *hibás* metaprogram, mely demonstrálja, hogy az eltérő megközelítés mennyire könnyen vezethet hibás megoldáshoz. A következő fejezetben ezt a kérdést részletesebben tárgyaljuk.

A TMP-nek, ennek az új programozási paradigmának a felhasználási lehetőségei közül csak egyik a számításigényes műveletek fordítási időben való elvégzése. A TMP segítségével programjaink hatékonyabbak, könnyebben bővíthetőek lesznek, sok eljárást automatizálhatunk. A TMP másik nagy felhasználási területe a fordítási idejű *program-adaptáció*. Ennek során az általánosan és platformfüggetlenül megírt program képes a kurrens alkalmazási környezethez fordítási időben alkalmazkodni. Ilyen adaptációra lehet példa a másoló konstruktor megírása. Egy általános T template paraméter típusú listát másoló konstruktor kénytelen az ismeretlen T típus másoló műveleteit (pl. `operator=`) használni. Amennyiben tudnánk, hogy a T típus egyszerűen másolható (azaz POD típus), akkor a másolást elvégezhetnénk hatékonyabban is, (pl. a `memcpy` függvény hívásával). Írhatunk olyan metaprogramot, amely a fordítási időben megismert T típus alapján kiválasztja a hatékony ill. helyes eljárást.

E folyamatok implementálásában segít például a `Loki` *typelist*-je. Ez egy olyan speciális, rekurzív template, mely tulajdonképpen egy típusokból álló lista. Használatával elérhető például, hogy bizonyos műveleteket minden egyes felsorolt típusra elvégezzen a fordító (pl példányosítson velük templateeket, vagy vagy futtasson metaprogramot, melyben argumentumként használja fel őket), így elég egyszer felsorolni őket, majd később már csak erre

a listára kell hivatkozni. Alexandrescu több alapvető algoritmust is implementál a könyvtárban [7] (ezek legnagyobbrészt rekurzióval megoldott metaprogramok), melyek hagyományos értelemben vett listaműveleteket végeznek (pl append, length, find stb) típuslistákon fordítási időben; így egy igen erős eszközt ad kezünkbe fordítási idejű komplex algoritmusok végrehajtásához. A Loki bizonyos értelemben a C++ template metaprogramozás ma elért csúcsteljesítménye. Dolgozatunkban azonban nem a kivételes tehetségű profik hibátlan megoldásaival, hanem a mindennapi – gyarló – programozó által elkövethető és minden bizonnyal el is követett hibák megelőzésével és felderítésével szeretnénk foglalkozni.

2. Template metaprogramok és a debuggolás

Az előző példán láttuk, hogy a metaprogramok írása közben könnyen véthetünk olyan hibát, amely a (meta)program fordítási idejű végrehajtása során hibás működést okoz. Ugyanakkor a hibás működés specifikációja metaprogramok esetén nemtriviális. Gondoljunk Unruh programjára, amely egy "hibás" program volt, mégis pont ezáltal a szerző szándékai szerint működött. A következőkben ezért elsőként a hibás program meghatározását próbáljuk megadni.

2.1. Hibák és javításuk

A programozás során valamilyen programozási nyelven fogalmazzuk meg elgondolásainkat, a begépelt forráskódot a fordító megpróbálja értelmezni, lexikális, szintaktikai és szemantikai elemzésnek veti alá. Amennyiben a forrásprogram helyes, abból futtatható kódot generál. Az esetek egy részében viszont ez a kód egyáltalán nem a szándékunk szerinti hatást váltja ki. Ennek okai lehetnek egyszerű elírások, rosszul szervezett algoritmusok, vagy akár súlyos elvi hibák is.

Két lehetőségünk van. Megpróbálhatunk olyan eszközöket felhasználni már *a programírás folyamán*, amelyek ezen hibák felbukkanásának valószínűségét csökkentik. Ilyenek például:

- A programba olyan kódrészletet tehetünk, mely egy informatív hibaüzenet kiírásával futási, vagy már fordítási időben leállítja a programot, illetve annak fordítását, ha sérül egy a programra jellemző invariáns (*assert, static assert*)
- Elméleti módszerek, mint például a helyességbizonyítás. Több évtizedes múltra tekint vissza e tudományterület, egyben a programozáselmélet központi témája. Számos irányzat foglalkozik a kérdéssel, de közös bennük a programok valamilyen formális nyelven történő leírása, majd matematikai alapú helyességbizonyítása.

Másrészt ha már hibás a program, valamilyen hibakereső (*debug*) módszerhez fordulhatunk. Melyek ezek?

- intuitív módon, "ránézésre", vagy a kód végigondolásával, végigkövetésével megpróbálhatjuk kitalálni a hiba okát
- *Slicing.* E tudományterület azzal a kérdéssel foglalkozik, hogy egy adott változót a program mely területei módosíthatják, így a programot dekomponálva jelentősen lecsökkenhet a tüzetesebb vizsgálatot

igénylő programrészek száma. Megkülönböztetünk statikus és dinamikus slicing-ot. A statikus slicing fordítási időben próbálja elvégezni a kód szűkítését, míg a dinamikus módszer esetében az egy adott lefutás során érintett utasításokat vizsgáljuk. A terület szintén kiterjedt irodalommal rendelkezik (pl [16]).

- a program egy adott pontján kiírattathatjuk a képernyőre a változók értékeit
- igénybe vehetjük egy debugger program szolgáltatásait, melyek legfontosabb funkciói a következők:
  - a változók értékét figyelemmel kísérhetjük
  - utasításról-utasításra lépve (*step*) megfigyelhetjük a folyamatot
  - elhelyezhetünk töréspontokat (*breakpoint*) a programban, ahol a futás leáll, így átugorhatunk nagyobb blokkokat

Más problémák merülnek fel pl imperatív, dekleratív, interpretált, vagy többszálú programok hibáinak keresésekor, így ezek debuggolása más gondolkodásmódot, technikákat igényel.

A szekvenciális programokkal szemben egy többszálú programnál például szemponttá válik egy változó védettsége. Meg kell állapítani, hogy párhuzamosan futó programrészek nem érik-e el egyszerre ugyanazt a memóriaterületet, mely versenyhelyzet majdnem biztosan hibás működéshez fog vezetni. Igy történhet meg, hogy egy program önmagában tökéletesen fut, ám ha két szálon indítjuk el, azonnal lefagy. Ezt az alapvető hibát könnyen ki lehet védeni, néha mégis igen nehéz rájönni, melyik is az a változó, amely elérését nem védjük *guard*-okkal és egyszerre próbálja két processz manipulálni. A párhuzamos programok debuggolás szempontjából egyébként is igen problémásak, hiszen könnyen elképzelhető, hogy akár évekig hiba nélkül elfutnak és ezután egyetlen-egyszer "összakad" két szál és ez végzetes hibát okoz. Még az esetleges programleállásnál is sokkal komolyabb probléma, hogy az ilyen hibákat gyakorlatilag lehetetlen reprodukálni, igy a hibát kiküszöbölni sem lehet.

A 2.3 fejezetben foglalkozunk a C++ Template metaprogramokra specifikus hibakeresési illetve hibamegelőzési eszközökkel.

2.2. Különbség a fordítási és a futási idejű programozás között

Vizsgáljuk meg közelebbről, milyen típusú hibákba ütközhetünk általában programozás közben! Nevezzük *futási idejű program*nak az olyan futtatható kódokat, amelyek forrásfileból történő fordítása közben a compiler metaprogramot nem futtat, vagyis a programban szereplő template-ek csak azt a célt szolgálják, hogy adott típusokat és függvényeket típusokkal tudjunk paraméterezni.[6]

Ez az a paradigma, amelyben szokásosan programozunk C++ nyelv alatt. Ezesetben a *hibás program* és a *hibásan futó program* fogalmak között látszik az éles határ. Ha lefordul a program, akkor a fordító szerint (lexikálisan, szintaktikailag és szemantikusan is) helyes, és abból futtatható kódot tud generálni. Ha pedig le sem fordul, akkor hibás. Egy teljesen más kérdés, hogy a program által leírt algoritmus helyesen működik-e vagy nem. Tegyük fel, hogy 0-4-ig levő egész számokat szeretnénk kiírni a képernyőre.

```
#include <iostream>
int main ()
{
    for (int i=0; ; ++i) std::cout << i << std::endl;
}
```

Ez egy *hibásan futó program.* A program helyes, lefordul, de egy végtelen `for` ciklust implementál. A hiba oka a kifelejtett ciklusfeltétel, amely hibát valamilyen módszerrel majd meg kell találnunk. Vagyis: a forrásfile *nyelvileg helyes*, de *algoritmikusan* hibás[7].

```
#include <iostream>
int main ()
{
    for (i=0; i!=4;++i) std::cout << i << std::endl;
}
```

Ez pedig egy hibás program, mivel az `i` változó nem definiált. Ez fordítási hibát fog okozni. Mindettől függetlenül maga az algoritmus helyes, és ha az

[6]Egy másik fordítási idejű programozási lehetőség, melyet most nem ismertetünk, a preprocesszor metaprogramming. Feltételezzük, hogy a program nem használ ilyen eszközöket. A témával bővebben [6] foglalkozik.

[7]Talán természetesebben hangzana az *szintaktikai* és *szemantikai* helyességről beszélni, ám e fogalmakat a fordítóprogramok elméletében használjuk, teljesen más értelemben. Mivel a dolgozatban fordítóprogramokkal is foglalkozunk, ezért félreértésekhez vezetne, ha nem vezetnénk be új megjelöléseket.

`i` változó létezne, valóban azt csinálná amit szeretnénk. Vagyis: a forrásfile nyelvileg hibás, de algoritmikusan helyes.

Tehát az éles határt egy program nyelvi helyessége, vagy nem-helyessége között a fordíthatósága jelenti. Mi viszont pontosan a fordítási időben manipulálunk, ezért TMP-világban már nem ennyire egyértelmű a nyelvi és algoritmikus helyesség közötti különbség.

Tekintsük újra a 1.2-ben ismeretetett Factorial metaprogramunkat, és tegyük fel, hogy a faktoriális-számítás rekurziójának feloldására szolgáló `Factorial<1>` template specializációt hibásan implementáljuk, a következőképpen:

```
template <int N>
struct Factorial
{
    enum { value = N*Factorial<N-1>::value };
};

template<>
class Factorial<1>
{
    enum { value = 1 };
};

int main ()
{
    int r=Factorial<5>::value;
}
```

Az eredeti programhoz képest látszik a különbség: a teljes specializációt most struct helyett egy class-ként implementáltuk, ez lehetett pl egyszerű elírás. A probléma most az, hogy C++ban a class default láthatósági szabálya a `private`. Igy most akaratlanul private taggá tettük az `enum { value=1 };` definíciót. Ezért fordítási hibát fogunk kapni mikor a compiler `Factorial<1>::value`-t próbálja elérni a nem friend `Factorial<2>` típus példányosítása közben. Az algoritmusunk helyes, hiszen `Factorial<1>::value` értéke `1`, csak éppen ez az érték nem elérhető. Vagyis gyakorlatilag egy deklarálatlan változónak próbálunk értéket adni, mint fenti futási idejű példánkban.

Most töröljük ki teljesen a template specializációt:

```
template <int N>
struct Factorial
{
    enum { value = N*Factorial<N-1>::value };
};

int main ()
{
    int r=Factorial<5>::value;
}
```

Mivel a struct template-nek nincs explicit specializációja, a `Factorial<N-1>` példányosíttatás miatt `Factorial<1>` után `Factorial<0>`, majd `Factorial<-1>` stb. következik. Vagyis logikailag egy fordítási idejű végtelen rekurziót írtunk.

Valójában a különböző fordítók különbözőféleképpen reagálhatnak erre a kódra. A g++ 3.4-es a szabvány által előírt 17 szintű implicit példányosítás után leáll [8]. A MSVC 6 fordítója addig fut, amig a rendelkezésére álló erőforrásokat ki nem használta (tesztünkben `Factorial<-1308>` példányosításáig jutott el). Az MSVC 7.1 pedig *fatal error C1202: recursive type or function dependency context too complex* hibaüzenettel azonnal leállt, észlelve, hogy nem lesz elegendő erőforrása a számítás befejezéséhez.

Visszatérve programunkhoz, láthatjuk tehát, hogy nyelvileg helyes de algoritmikusan hibás. Most viszont, ellentétben a futási idejű példával, valamilyen *fordítási hibát kapunk!*

Az ok egyszerű: a template metaprogramok "futási" és "fordítási" ideje egybeesik, a TMP gyakorlatilag egy interpreter nyelvnek is tekinthető. Ezért is nehéz meghatározni a hibás és a hibásan futó kategóriák közötti különbséget a metaprogramok világában.

Egy harmadik hibalehetőség ami szempontként merül fel a fordító által felhasználható erőforrások kérdése. Tegyük fel, hogy `Factorial<25>::value`-t szeretnénk felhasználni valahol a programban, valamint, hogy az eredeti, teljesen helyes `Factorial` metaprogramot használjuk. A 25 konstanssal való példányosíttatás egy szabványt követő compiler esetében (pl g++ 3.4)

[8] A fordítót fel lehet úgy paraméterezni, hogy akár 99 implicit példányosítást is elvégezzen.

továbbra is fordítási hibához fog vezetni, hiszen a compiler túl fogja lépni az előírt 17 mélységű implicit template példányosítást.

A dolgozatban a továbbiakban feltesszük, hogy a vizsgálni kívánt metaprogram nyelvileg helyes, algoritmikusan hibás és a compiler nem fogy ki az erőforrásokból.

2.3. Segédeszközök helyes programok írásához

Az 2.1 fejezetben felsoroltuk, milyen lehetőségeink vannak általában a hibák megelőzésére, illetve megkeresésére. Próbáljuk meg ezeket az irányelveket lefordítani a template-ek és a TMP nyelvére.

2.3.1. Concept checking

Említettük 1.1-ben, hogy a C++ template-ek (jó és egyben rossz) tulajdonsága, hogy nem teszünk explicite kikötéseket a template argumentumokra. A *concept checking* (vagy *static interface checking*) kutatási terület ezt a problémát járja körül, olyan megoldások után kutatva, amelyekkel valamilyen módon mégis meg tudunk fogalmazni bizonyos elvárásokat nem csak template argumentumokkal, hanem bármilyen fordítási időben ismert értékkel, típussal kapcsolatban. A témával számtalan cikk foglalkozik, és az eredményeket különféle könyvtárakban implementálták ld pl a boost könyvtárat ([6]), vagy Alexandrescu Loki könyvtárát ([7]). Ilyen template introspection könyvtárra tesz javaslatot [18].

2.3.2. Static assert

Következő eszközünk a futási idejű programoknál alkalmazott technika, az *assert*. Az `assert` egy olyan függvény, mely egy logikai kifejezést vár bemenetként (tehát valamilyen feltételt vizsgáltatunk meg vele), és ha az hamis, egy *assertion failed* hibaüzenettel leállítja a programot. Igy elkerülhető, hogy pl adott pillanatban sérült egy invariáns, de azt egy újabb hiba rögtön ki is javítja, vagyis nem siklunk át egy logikai hibán sem.

Egy szintet feljebb lépve, a *static assert* az egyik lehetséges válasz arra a kérdésre, hogy ha a már 2.3.1-ben említett elvárásokat nem teljesíti egy érték, típus, akkor mit tegyünk? A *static assert* a detektálás helyén leállítja a fordítást, igy elkerülve egy logikailag hibás program létrejöttét. Törekszünk arra is, hogy lehetőleg az assert valamilyen hibaüzenetet is tartalmazzon, így könnyítve meg a programozónak a hiba megtalálását.

A legegyszerűbb kivitelezési mód egy makró segítségével történik, a következőképpen:

```
#define STATIC_ASSERT_1(C) char static_assert_unnamed[int(C)];
```

Tehát amennyiben az adott `C` feltétel igaz, az értéket `int`-é konvertálva `1`-et, ha pedig hamis, `0`-át kapunk. Az assert a C++ nyelv azon tulajdonságára alapszik, hogy 0 hosszúságú tömböt nem hozhatunk létre, ez fordítási hibához fog vezetni. A `STATIC_ASSERT_1(sizeof(int)==sizeof(double))` assert a következő kimenetet adja 3 vizsgált fordító alatt:

g++ 3.4:	*N/A*[9]
MSVC 6.0:	*error C2466: cannot allocate an array of constant size 0*
MSVC 7.1:	*error C2466: cannot allocate an array of constant size 0*

Vagyis a makró úgy működik, ahogy vártuk, a fordítás hibaüzenettel leáll (és ami legalább ilyen fontos: mikor igaz a feltétel, nem áll le). A probléma viszont most az, hogy a kapott hibaüzenetből sehogyan sem lehet a hiba valódi okára következtetni.

Valamivel bonyolultabb, de éppen a fenti problémára egy jobb megoldást nyújt a boost könyvtár `BOOST_STATIC_ASSERT`-je, mely váza a következőképpen néz ki:

```
template <bool> struct STATIC_ASSERTION_FAILURE;
template <> struct STATIC_ASSERTION_FAILURE<true>{};

template<int x> struct static_assert_test{};

#define STATIC_ASSERT_2( B ) \
   typedef static_assert_test< \
      sizeof(STATIC_ASSERTION_FAILURE< (bool)( B ) >)> \
         static_assert_typedef_;
```

A módszer hasonló: a kiértékelendő kifejezés értékét `bool` típusúvá alakítjuk, majd megpróbáljuk a `sizeof` operátort alkalmazni a `STATIC_ASSERTION_FAILURE` template-ből `true`-val, vagy `false`-al példányosított típusra. Látható a template specializációból, hogy ha a kifejezés igaz, akkor az üres, de létező `STATIC_ASSERTION_FAILURE<true>` típusról van szó, de ha nem, akkor a *definiálatlan* `STATIC_ASSERTION_FAILURE<false>`-ról. Ilyen típusra pedig nyelvi hiba `sizeof` operátort hívni.[10]

[9] A g++ egy fordítóhiba miatt mégis elfogadta a nulla méretű tömböt...

[10] Látható, hogy az assert egy új `typedef`-et hoz létre, melyből egy névtérben csak egy lehet. A `typedef`-ek végére a `__LINE__` preprocesszor makró segítségével egy sorszám ragasztásával egy fordítási egységen belül egyértelmű neveket lehet létrehozni. Részletesen ld [6].

Nézzük újra a kimeneteket a
`STATIC_ASSERT_2(sizeof(int)==sizeof(double))`
utasításra!

g++ 3.4:	*error: invalid application of sizeof' to incomplete type STATIC_ASSERTION_FAILURE<false>'*
MSVC 6.0:	*error C2027: use of undefined type 'STATIC_ASSERTION_FAILURE<0>'*
MSVC 7.1:	*error C2027: use of undefined type 'STATIC_ASSERTION_FAILURE<__formal>' with [__formal=false]*

A csupa nagybetűvel írt "STATIC_ASSERTION_FAILURE" üzenet már jóval egyértelműbben utal arra, hogy egy mesterségesen előidézett hibáról van szó, nem pedig valamilyen más hibáról.

Ezt a koncepciót gondolja tovább a következő programrészlet:

```
template <bool, class msg> struct STATIC_ASSERTION_FAILURE;
template <class msg> struct STATIC_ASSERTION_FAILURE<true,T>{};

template<int x> struct static_assert_test{};

#define STATIC_ASSERT_3( B , error)  \
   typedef static_assert_test< \
      sizeof(STATIC_ASSERTION_FAILURE< (bool)(B),error >)> \
         static_assert_typedef_;
```

Ez a [7, 26. oldal]-ban is ismertetett megoldás egy másik formája. Látható, hogy egy második makróparaméter is megjelent. Ez az üzenetátadás célját szolgálja. Definiáljunk egy üres structot, amely neve jól leírja a hibát. Eddigi példánk most így módosul:

```
struct SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE {};
STATIC_ASSERT_3(sizeof(int)==sizeof(double),
                SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE)
```

Tekintsük a kimeneteket:

g++ 3.4:	*error: invalid application of sizeof' to incomplete type STATIC_ASSERTION_FAILURE<false, SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE>'*
MSVC 6.0:	*N/A*[11]
MSVC 7.1:	*error C2027: use of undefined type 'STATIC_ASSERTION_FAILURE<__formal,msg>' with [__formal=false msg=SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE]*

A módszer tovább finomítható. Alkossunk olyan structot, melyben több szinten beágyazott template structok találhatóak, így a hibaüzenet egy-egy ilyen rész-struct teljes nevéből áll össze, a megfelelő template paraméterek helyére a kívánt típusokat behelyettesítve. Ez főként akkor hasznos, ha pl az assertezés idején még nem ismerjük a vizsgált template argumentumokat, vagy például ha ugyanazt a hibaüzenetet szeretnénk újrafelhasználni. A hibaüzenet paraméterként természetesen ugyanolyan értékeket tartalmazhat, mint amiket egy template-nek argumentumként lehet adni, típusok mellett akár pl egész számokat.

```
template <class T> struct SIZEOF
{
    struct NOT_EQUAL_TO
    {
        template <class U>
        struct SIZEOF
        {
        };
    };
};

STATIC_ASSERT_3(sizeof(int)==sizeof(double), \\
typename SIZEOF<int>::NOT_EQUAL_TO::template SIZEOF<double>)
```

[11] A MSVC 6 nem tudta lefordítani az assertet, mivel a compilerben még nincs részleges template specializáció.

Tekintsük a kimeneteket:

g++ 3.4:	*error: invalid application of sizeof' to incomplete type STATIC_ASSERTION_FAILURE<false, SIZEOF<int>::NOT_EQUAL_TO::SIZEOF<double> >'*
MSVC 6.0:	*N/A*[12]
MSVC 7.1:	*error C2027: use of undefined type 'STATIC_ASSERTION_FAILURE<__formal,msg>' with [__formal=false msg=SIZEOF<int>::NOT_EQUAL_TO::SIZEOF<double>]*

Ez a makró egy jól megkonstruált hibaüzenettel már jól használható.

A TMP világ egyik jellegzetessége, hogy bár a C++ erősen típusos nyelv, a template metaprogramok nem azok! Mivel bármilyen típus lehet template argumentum, érdemes lehet egy-egy bonyolultabb metaprogram/template elejét egy asserttel kezdeni, amely megállapítja, hogy a bejövő típus valóban megengedhető-e ("bonyolultabbon" azt értve, hogy az okozott hiba áttételesen is megjelenhetne, teljesen váratlanul, valahol a class törzsében)

2.4. Az ismert módszerek, és miért nem működnek most

A 2.1 fejezetben felsoroltunk néhány módszert futási idejű programok debuggolására. Mit jelentenek ezek az elvek a TMP világban?

- valahogyan a programozó tudtára adni a "változók" értékét (vagyis a példányosuló template-ek argumentumait és a template-ek törzsében található fordítási idejű konstansokéit) - ehhez modellezni kellene a két említett módszert:
  - a debugger tartsa nyilván a változókat, és azok legyenek lekérdezhetőek
  - írassuk ki automatikusan a változók értékeit
- static assertekkel feltételek ellenőrzése

[12] ld mint előbb

- step és breakpointok megvalósítása - ez az adott paraméterű templateek példányosulásakor a fordítás átmeneti leállítását jelentheti

Az első pontban felsorolt eszközök, amiket futási idejű programozási paradigmánkban naponta használunk, nem állnak rendelkezésre a szokott módon, ha metaprogramot írunk. Nincs képernyőre író utasításunk (sőt, gyakorlatilag semmilyen utasításunk sincs), nincs semmilyen keretrendszerünk. Ám még így is több lehetőség közül választhatunk:

- nyelvi szinten próbáljuk meg kezelni a problémát
- a fordítót próbáljuk hibakeresésre használni:
  - a fordítóval irattatjuk ki a kívánt üzeneteket- ld 2.6
  - magának a fordítónak erre a célre való módosításával ld 2.5

2.5. Fordítóprogram módosítása

Mivel mi meta-szinten programozunk, meta-szintű hibakeresési lehetőségekre van szükségünk. Programunk fordítási időben hajtódik végre, ebből következően az általa kezelt objektumok sem egyeznek meg a futási időben kezelt objektumokkal. A 1.2 alatti összehasonlítás ezt a különbséget igyekszik megvilágítani. Látható, hogy azok az entitások, melyek vizsgálata metaprogramok futása közben érdekes lehet elsősorban a fordítóprogram számára információt hordozó objektumok (pl típusok, konstansok). Ebből adódik az az újszerű megoldási lehetőség, hogy magát a compilert változtatjuk debuggolási eszközzé. Ebben a fejezetben ezzel foglalkozunk.

Felmerül a kérdés, miért a fordítót módosítjuk, és miért nem készítünk egy külön kódelemző programot, mely szintén tudná a forráskódot "felülről" kezelni. Figyelembe kell azonban venni azt a tényt, hogy az egyes fordítóprogramok egymástól gyökeresen eltérő módszereket alkalmaznak és a fordítás kivitelezésére nincsen nyelvi szabvány, csak a fordítás eredményére. Ebből következően egy külső eszköz helyesen megállapíthatja egy metaprogramról, hogy az a nyelvi szabványnak megfelel-e, de egy hibás programot minden fordító más és más módon értelmezhet, ezért egy külső eszköz nem tudna adekvát információt nyújtani erről a működésről. Ha az adott fordító viselkedését kívánjuk megérteni, a konkrét fordítót kell módosítanunk.

2.5.1. A g++ compiler háttere

A C nyelven írt forráskód hozzáférhetősége, valamint viszonylagos egyszerűsége miatt a g++ 2.95 compilert vizsgáltuk. A tesztelés folyamán a következő, számunkra fontos fileokat és függvényeket találtuk:

- `pt.c`, `instantiate_class_template(tree)`: itt találtuk meg az algoritmust, amelyik a tényleges template példányosítást végzi

- `typeck.c`, `complete_type(tree)`: ez a függvény felelős a típusok "befejezéséért", mint például egy class legyártása egy templateből

A `g++` compiler két fő részből áll. Egyik a *back-end*, amely a GNU minden fordítójának közös felülete, ez végzi az adatok tárolását és a számításokat. Ezen kívül minden támogatott nyelvhez (pl Pascal, Ada, Java, stb) létezik egy külön ún. *front-end*, ami már a nyelvspecifikus tulajdonságokat kezeli és tárolja (pl. kulcsszavak).

A back-end a `tree.h` fileban definiált `tree` adatstruktúrákban tárolja a fordításnál megszerzett információkat. Ez a struktúra tulajdonképpen úniók összessége, és a szintaxisfa minden csúcsához egy ilyen adatcsomagot rendel hozzá a fordító, majd az elemzések folyamán ezeket tölti fel a jellemző információkkal. Például ha egy osztálydefiníciót talál, akkor a benne lévő tagfüggvények neveire mutató pointereket, a szülő osztályok node-jaira mutató pointereket, vagy éppen az osztály méretét, stb.

A fent említett függvények is paraméterként ilyen tree struktúrákkal dolgoznak. A többszintű dereferenciák miatt hamar bonyolulttá válik az egy csúcshoz tartozó információk elérése, ezért ezt makrók segítik. Számunkra a következők fontosak:

- `CLASSTYPE_TI_ARGS(tree)`: pointer a `type` által leírt template argumentumait tartalmazó vektorra, ezt iteráljuk végig

- `TREE_VEC_LENGTH(tree)`: az előző makró által visszaadott lista hossza

- `TREE_VEC_ELT(tree,int)`: a vektor `i`. eleme

- `IDENTIFIER_POINTER(DECL_NAME(TYPE_NAME(tree)))`: ezekkel a makróhívásokkal kapjuk meg a példányosuló típus nevét, a következő formában: `templatenév<arg1,arg2,...>`

- `TREE_CODE(tree)`: az említett vektor egy elemének fordítókódja, pl `INTEGER_CST` (egy egész szám mint template argumentum), `INTEGER_TYPE` (egész szám típus) stb

Azzal, hogy az `instantiate_class_template` függvény elején "elkapunk" minden példányosítást, valamint a fordító az argumentumként kapott treeben kezünkbe adja a template összes ismert információját, lehetőségünk nyílik arra, hogy kövessük a példányosítási sorozatot.

Igen egyszerű és mégis nagyon jól használható megoldás, ha kiíratjuk a példányosult class template nevét az argumentumaival együtt.

A már 2.2-ben ismertetett végtelen rekurziót implementáló hibás metaprogramra a megoldásunk a következő kimenetet adja, melyből jól látszik a végtelen példányosítási lánc:

```
MetaDebug: Factorial<5>, arg1type: integer_type, arg1val: 5
MetaDebug: Factorial<4>, arg1type: integer_type, arg1val: 4
MetaDebug: Factorial<3>, arg1type: integer_type, arg1val: 3
MetaDebug: Factorial<2>, arg1type: integer_type, arg1val: 2
MetaDebug: Factorial<1>, arg1type: integer_type, arg1val: 1
MetaDebug: Factorial<0>, arg1type: integer_type, arg1val: 0
MetaDebug: Factorial<-1>, arg1type: integer_type, arg1val: -1
MetaDebug: Factorial<-2>, arg1type: integer_type, arg1val: -2
...
```

A megoldás előnye, hogy a forráshoz nem kell hozzányúlni (azaz non-intrusive). A fenti példa is mutatja, hogy önmagában az az információ, hogy az egyes sablonok milyen sorrendben és milyen paraméterekkel példányosulnak komoly segítség a metaprogramunk vizsgálatában. Mindezt a futási idejű programok változóinak kiíratásához hasonlíthatjuk.

2.5.2. Továbbfejlesztési lehetőségek

A concept checking módszerek segítésére elképzelhető lenne a fenti compiler hack továbbfejlesztése oly módon, hogy a class fordítótól lekérdezett tulajdonságait egy class template-be "csomagolva" fordítási időben rendelkezésre bocsátanánk. Ez a működés hasonló lenne a C++ *typeid* operátorához, melyet bármely kifejezésre meg lehet hívni és egy objektumot ad vissza.

```
template <class CheckType>
struct Concepts
{
    enum { hasDeriveds = 0 };
    enum { countMemberFunctions = 0 };
    ....
}
```

Egy adott típus példányosításakor a fordító a Concepts class template-t is példányosítaná az adott típussal mint argumentum, és beállítva a mezőket a megfelelő értékekre, azt a programozó mint egy speciális reflection segédeszköt használhatná.

Kissé hasonló módszert akalmaz a boost `type_traits` könyvtára. Ugyanakkor a type_traits elsősorban felhasználói metaprogramokkal nyeri ki az információt a kódból, melyek kifejezőereje korlátos. Igy például nyitott kérdés,

hogyan lehet megállapítani, hogy egy osztály rendelkezik-e default (paraméter nélkül hívható) konstruktorral. Lehetséges, hogy ennek fordítási idejű `bool` konstanst eredményező eldöntése elméletileg nem lehetséges felhasználói könyvtárból. Az ilyen jellegű típusinformációk kigyűjtése egyértelműen a fordítóprogram feladata.

2.6. Üzenet generálása template példányosításkor

Az előző fejezetben ismertetett megoldás a fordítóprogramot módosítja. Ez a lehetőség nem mindíg áll rendelkezésünkre. Az alábbiakban egy olyan módszert ismertetünk, amely külső eszközök igénybevétele nélkül, pusztán a nyelv szabályainak kihasználásával valósít meg hasonló funkcionalitást.

2.6.1. Osztályszintű változók

Igen nagy problémát jelenthet az olyan "változók" módosulásának felderítése, melyek nem egy függvény lokális változói, hanem osztályszintűek. Használjuk ki a fordító azon tulajdonságát, hogy általában ha egy class template példányosítás sorozat közben valahol hiba történik, akkor az adott létrejövő típus neve mellett kiírásra kerül az is, hogy annak példányosítását melyik class kérte. Tekintsük újra a már többször említett faktoriális-számító metaprogramunkat, és "gépeljünk el" benne egy karaktert, írjunk `N-1` helyett `N-2`-őt:

```
template <int N>
struct Factorial
{
    enum { value = N*Factorial <N-2>::value };
};

template<>
struct Factorial<1>
{
    enum { value = 1 };
};

int main ()
{
    int r=Factorial<9>::value;
}
```

A program eredménye nyilván a páratlan számok szorzata lesz 9-ig.

Szúrjuk be a következő kódrészletet a programba:

```
#define DEBUG int r__[-1];
```

majd helyezzük el a DEBUG makró egy hívását a class template törzsében az enum alatt:

```
template <int N>
struct Factorial
{
    enum { value = N*Factorial <N-2>::value };
    DEBUG
};
```

Ez azt fogja eredményezni, hogy minden egyes `N` argumentummal történő példányosításkor a létrejövő típusban egy hibát fog generálni a fordító, és a hibaüzenet tartalmazni fogja visszafelé a példányosítási sorrendet. A `g++` compiler (igen hosszú hibaüzenetének) eleje például a következő:

warning.cpp: In instantiation of 'fact<3>':
warning.cpp:16: instantiated from 'fact<5>':
warning.cpp:16: instantiated from 'fact<7>':
warning.cpp:16: instantiated from 'fact<9>':
....

Nyilvánvaló hátrány, hogy a rekurzió feloldását, a `Factorial<1>` típus felhasználását nem írja ki a compiler, ui ez konkrét típus, tehát nem szerepel a példányosítási láncban.

Figyeljük meg, hogy módszerünk akkor is működik, ha a faktoriális értékét páros számmal indítjuk. Ebben az esetben a metaprogram szintén súlyosan hibás lenne: ha egy páros szám faktoriálisát akarnánk kiszámolni, a lánc a `N==2` után `Factorial<0>` példányosításával folytatódik, így átugrottuk `Factorial<1>`-et, mely feloldhatta volna a rekurziót. Következmény: végtelen rekurzió. E probléma újra a TMP interpreter nyelv voltára vezethető vissza.

Nem használható jól a módszer abban az esetben, ha a példányosítási lánc nem lineáris, értve ezalatt, hogy egy adott példányosított class többször előfordul a hivatkozások között. Amennyiben ugyanis egy már korábban példányosult osztályhoz érünk, nem kapunk üzenetet annak hibás voltáról. Ilyen eset fordul elő például a *BubbleSort* algoritmus Veldhuizen-féle ([1]) megvalósításakor.

Úgy tűnhet, szinte semmire nem jó ez a kis trükk, hiszen "látszik" mi a hiba. De gondoljunk bele, milyen sokszor történik meg, hogy akár igen rövid kódrészletben sem veszünk észre egy teljesen triviális hibát. Másrészt ha például egy sinus értékeket összegekkel közelítő metaprogramról van szó (ld [1], mely igen bonyolult képleteket használ (pl `return 1-(I*2*M_PI/N)* (I*2*M_PI/N)/(2*K+2)/(2*K+3) * SineSeries<N*go,I*go,J*go, (K+1)*go>::accumulate();`) belátható, hogy könnyű azokat elírni, és igen nehezen találjuk meg a hibát később. (Ezesetben azért működőképes ez a debug eljárás, mert a `K` futó változó miatt minden egyes `SineSeries` hivásnál új típus fog létrejönni, hacsak nem épp a legutolsó `K+1` paramétert írjuk el...)

A módszer az ilyen apróbb (ám sokszor igen bosszantó) elírások felderítése mellett alkalmas lehet például typelist algoritmusok debuggolására, mivel azok általában szintén olyan rekurziók, ahol minden típus egyszer példányosul.

2.6.2. Függvények lokális változói

Fent említettük, hogy Veldhuizen buborékrendezését, mely egy `int` tömb tartalmát helyben rendezi, nem tudjuk az előző módszerrel debuggolni. Tekintsük most ezt a metaprogramot.

```
template<int N>
class IntBubbleSort {
public:
    static inline void sort(int* data)
    {
        IntBubbleSortLoop<N-1,0>::loop(data);
        IntBubbleSort<N-1>::sort(data);
    }
};

class IntBubbleSort<1> {
public:
    static inline void sort(int* data)
    { }
};

template<int X, int Y>
class IntBubbleSortLoop {
```

```
private:
    enum { go = (Y <= X-2) };

public:
    static inline void loop(int* data)
    {
        IntSwap<Y,Y+1>::compareAndSwap(data);
        IntBubbleSortLoop<go?X:0,go?(Y+1):0>::loop(data);
    }
};

class IntBubbleSortLoop<0,0> {

public:
    static inline void loop(int*)
    { }
};

template<int I, int J>
class IntSwap {
public:
    static inline void compareAndSwap(int* data)
    {
        if (data[I] > data[J])
            swap(data[I], data[J]);
    }
};

inline void swap(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int v[5]={4, 2, 6, 1, 7};
    IntBubbleSort<5>::sort(v);
}
```

Hogyan működik ez a program? `IntBubbleSort sort` függvénye szolgáltatja a külső ciklust, mely `N`-től `1`-ig csökken és minden iterációban meghívja a belső ciklust, melyet `IntBubbleSortLoop loop` implementál. Ebben a belső ciklusban pedig az `Y` változó fog növekedni 0-tól `X-2`-ig, mely esetben még lefut ez a hátultesztelő ciklus, majd ezután a `IntBubbleSortLoop<0,0>` specializáció leállítja azt. Fordítási időben fogjuk így megkapni a rendezett tömböt.

Bárhol is vétünk hibát (legyen ez most egy hibás ciklusfeltétel: `go = (Y <= X-2)`), meglepő módon ezzel az igen bonyolult metaprogrammal egyszerűbb a dolgunk mint a faktoriális-számítóval. Itt ugyanis futási időben végrehajtandó kód is létrejön, ezek az `inline` függvények lesznek. Ekkor viszont újra használható a már jól megszokott módszer: irassuk ki a változókat a `compareAndSwap` függvény elején. De melyeket? Természetesen a template argumentumként megkapott `I`-t és `J`-t, hiszen ezek határozzák meg a megcserélendő számok pozícióját. Nem meglepő a futási időben kapott kimenet:

```
I=0 J=1
I=1 J=2
I=2 J=3
I=3 J=4
I=4 J=5
....
```

Látható, hogy `J` felveszi az `5` értéket, mely egy 5 elemű tömb esetén C++ban nem szerencsés... Visszafelé következtetve rájöhetünk, hogy `J` értékét IntSwap "állítja be", mikor meghívja `IntSwap<4,5>::compareAndSwap(data)`-t. Látszik, hogy `Y` egyel túlfut a szükségesnél.

3. Összefoglalás

Dolgozatunkban kísérletet tettünk arra, hogy módszereket adjunk template metaprogramok helyes megírásának támogatására. A template metaprogramok használata új programozási paradigma, melyek a futási idejű algoritmusok egy részét fordítási idejű tevékenységgel helyettesítik. Ezáltal azonban az esetleges programozói hibák új fajtái keletkezhetnek, melyek kivédésére, vagy utólagos felderítésére a jelenlegi (futási idejű) eszközök alkalmatlanok. Jelen dolgozat legjobb tudomásunk szerint az első, mely megpróbál elméleti és gyakorlati tanácsokat adni ezen hibák megelőzésére és felderítésére. Dolgozatunknak ugyancsak célja volt, hogy felhívja a figyelmet erre az aktuális, de eddig nem kutatott problémakörre.

Tárgyaltuk a metaprogramok nyelvi és algoritmikus hibásságának különbségét, példákat hoztunk fel olyan esetekre, melyekben a specifikációtól eltérő metaprogramok nyelvileg hibásak, algoritmikusan hibásak ill. kimerítik a rendelkezésre álló erőforrásokat. Áttekintettük azokat a módszereket, amelyek helyes metaprogramok írásában támogatnak. A Concept checking könyvtárak mellett javaslatot tettünk a static assert hibaüzeneteinek egy új formájára, melyek a jelenleginél informatívabbak.

Abban az esetben, ha a metaprogram nem a specifikációnak megfelelően működik, szükségünk van e viselkedés vizsgálatára, "debuggolására". Áttekintettük, miért nem működnek a futási idejű programvizsgálati eszközök. Helyettük két javaslattal élünk, melyeket demonstratív céllal implementáltunk is. A fordítóprogram módosításával, melyet a `g++` fordítón mutattunk be, lehetővé válik a példányosítások nyomkövetése non-intrusive módon. A fordítóprogram módosítása nélkül dolgozik másik módszerünk, mely segítségével üzeneteket generálunk példányosításkor.

Későbbi kutatás tárgyát képezhetné template metaprogramok slice-olása. A metaprogramok sok, a slicing szempontjából előnyös tulajdonsággal rendelkeznek, melyek megkönnyítenék az ilyen módszerek alkalmazását. Elsősorban a statikus slicing-ot segíti elő, hogy a futási idejű programokhoz képest nem kell számolnunk ismeretlen értékű adatokkal (pl. user input), nincsenek virtuális függvények, stb. Metaprogram viszont lehet egy metaprogram argumentuma, ezért a hívási környezet (calling context) vizsgálata nem megkerülhető. További kutatási irány a már 2.5.2-ben említett concept checking támogatása a fordító kiterjesztésével. Meg kell jegyeznünk, hogy a nemrégen elfogadott C++ nyelvi kiterjesztési javaslat (Technical Report 1, TR1) javaslatot tesz a szabványos `<type_traits>` könyvtár bevezetésére, amelyet egyes fordítók már támogatnak.

Hivatkozások

[1] Todd Veldhuizen,
`http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms`

[2] Todd Veldhuizen, *Just when you thought your little language was safe: "Expression Templates" in Java*, LNCS Vol.2177. pp.188-206., 2001

[3] David Vandevoorde, Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, Addison Wesley, 2002

[4] Bjarne Stroustrup, *A C++ programozási nyelv*

[5] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994

[6] Boost dokumentáció, `http://www.boost.org/libs/libraries.htm`

[7] Andrei Alexandrescu, *Modern C++ design*, Addison-Wesley, 2001, `http://www.moderncppdesign.com/`

[8] GCC dokumentáció, `http://gcc.gnu.org/onlinedocs/`

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

[10] ANSI/ISO C++ szabvány

[11] `http://users.rcn.com/abrahams/instantiation_speed/`

[12] `http://aszt.inf.elte.hu/~gsd/`

[13] `http://www.blitz.org`

[14] `http://www.erwin-unruh.de/Prim.html`

[15] `http://home.earthlink.net/~patricia_shanahan/debug/`

[16] Beszédes, Á., Gergely, T., Szabó, Zs. M., Csirik, J., and Gyimóthy T., *Dynamic Slicing Method for Maintenance of Large C Programs*, In Proceedings of the 5th CSMR 2001, pages 105-113. Lisbon, Portugal, 2001.

[17] `http://www.cs.ucl.ac.uk/staff/w.emmerich/~lectures/`
`/3C05-03-04/ProgramSlicing.pdf`

[18] Zólyomi István, Porkoláb Zoltán, Towards a template introspection library, LNCS Vol.3286 pp.266-282 2004

[19] Wettl Ferenc, Mayer Gyula, Sudár Csaba, *Latex kezdőknek és haladóknak*, Panem Kft, 1998

[20] Zólyomi István, *Az objektumcsaládok polimorfizmusa*, 2003

[21] Scott Meyers, *Effective STL*, Addison-Wesley

[22] Gilad Bracha, Generics in the Java Programming Language, `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

[23] James O. Coplien, *Multi-Paradigm Design for C++* Addison-Wesley, 1998