Template metaprogramok hatékony fejlesztése

Doktori értekezés

Sipos Ádám

Témavezető:
Porkoláb Zoltán

Az informatika alapjai és módszertana doktori program

Iskola-, és programvezető:
Demetrovics János

Eötvös Loránd Tudományegyetem Informatika Doktori Iskola
Budapest
2009

Köszönetnyilvánítás

Mindenek előtt szeretnék köszönetet mondani témavezetőmnek, Porkoláb Zoltánnak. Ő vezetett be a tudományos kutatások világába, amikor több, mint 5 éve egy TDK dolgozat megírására jelentkeztem nála. Ezt követően publikációink elkészítése során mindvégig támogatott, lelkiismeretesen tanított, és irányította munkámat. Az ő segítsége nélkülözhetetlen alapját képezte minden elért eredményemnek.

Köszönöm családomnak, barátaimnak és kollégáimnak, hogy mindvégig támogattak.

Ezúton szeretnék köszönetet mondani mindazoknak, akik segítették doktori disszertációm létrejöttét, részt vettek szakmai és nyelvi lektorálásában, illetve értékes hozzászólásaikkal segítették munkámat.

Köszönettel tartozom Zsók Viktóriának, Diviánszky Péternek és Kozsik Tamásnak, akik a funkcionális programozással kapcsolatos tudásukat osztották meg velem.

Köszönöm szerzőtársaimnak az elkészült publikációinkban való együttgondolkodást.

Tartalomjegyzék

Bevezetés

Ebben a dolgozatban a *C++ Template Metaprogramozás* (TMP) paradigmáját tárgyaljuk. Vizsgáljuk a paradigma felhasználásával írt programok (*metaprogramok*) szintaktikáját, szemantikáját, illetve az ezekből fakadó minőségi paramétereket. Megoldási javaslatokat adunk e paraméterek javítására.

A TMP a C++ programozási nyelv generikus elemére, a *template*-re épül. A TMP segítségével fordítási időben hajthatunk végre algoritmusokat, melyek segítségével hatékonyabb kódot (*expression templates*), könnyebben bővíthető könyvtárakat és fordítási idejű program-adaptációt hozhatunk létre. A TMP ma már önálló programozási paradigma, melyet már nem csak kutatási célokra, hanem ipari szoftvertermékek elkészítése során is felhasználnak, mivel felismerték a paradigma C++ fejlesztési hatékonyságra gyakorolt pozitív hatását.

Ugyanakkor a paradigma nem mondható elterjedtnek. Ennek okai elsősorban a kódolási sztenderdek és a fejlesztést segítő szoftvereszközök hiányában keresendőek. Az egyes megoldások gyakran ad hoc jellegűek, a hibajavítások heurisztikusak, ezáltal a projekt ráfordítások, fejlesztési költségek nehezen becsülhetőek. Általános probléma a TMP-vel kapcsolatban a körülményes szintaktika, melynek eredményeképpen nehezebben írható és karbantartható forráskódok születnek. Ezzel egyidejűleg a metaprogram kódok fordítási hatékonysága igen alacsony, hiszen a példányosítási láncok végrehajtása komoly plusz terheket ró a fordítóprogramokra, melyeket nem ilyen jellegű munkák elvégzésére terveztek. A bonyolult szintaktika rövid úton programozói hibákhoz vezet, ezért még nagy körültekintés és megfelelő metodika alkalmazása mellett is igen nehéz TMP-ben programozni, ugyanakkor a hibakeresés (debuggolás) is igen körülményes.

A dolgozatban e problémákra keresünk megoldási lehetőségeket. Bemutatunk már meglévő metaprogramozási könyvtárakat, megoldásokat, illetve saját kutatásaink alapján új módszerekkel, eszközökkel szolgálunk a hatékonyabb metaprogram-fejlesztéshez.

Az I. részben bemutatjuk a metaprogramozás fogalmát. Tárgyaljuk a template konstrukció nyelvi tulajdonságait, illetve az ezen tulajdonságok felhasználásával működő C++ template metaprogramozást. Ezután megvizsgáljuk a TMP-t, mint paradigmát.

A II. rész egy saját kutatási eredményünket mutatja be. E részben a metaprogramok írása és futása során esetlegesen bekövetkező hibákat vizsgáljuk. Az ISO/IEC 14882 (1998) C++ szabvány által meghatározott jólformáltsági kritériumokat kiterjesztetjük fordítási időben végrehajtott C++ template metaprogramokra.

A III. részben saját eredményeink alapján megoldási javaslatot adunk a metaprogramok bonyolult szintaktikájának leegyszerűsítésére. A template metaprogramok felett definiálunk egy helyettesítő funkcionális absztrakciós nyelvet, az EClean-t. Megadjuk az EClean átírási mechanizmusát az ISO/IEC 14882 (1998) szerinti C++ nyelvre. Az EClean a beágyazott kódot C++ template-ekké transzformálja, majd azt a C++ fordító értelmezi, csak úgy, mintha kézzel írt metaprogram kódról lenne szó. Ezáltal áttekinthető, könnyen karbantartható fordítási idejű kódot írhatunk. Bemutatunk egy kísérleti transzlátort a fenti átírásra és elemezzük a keletkeztett kód hatékonysági paramétereit.

A IV. részben hibaprevencióval és hibakereséssel foglalkozunk, valamint bemutatjuk e két témában elért eredményeinket. Meghatározzuk a C++ template metaprogramok hibakeresésének feltételeit. A *static_assert* fordítási idejű konstrukció segítségével módszereket adunk a metaprogramozás során elkövethető hibák megelőzésére. Ismertetjük a hibakeresési célokat szolgáló *g++* fordítóprogram-módosításunkat. Bemutatjuk *Templight* nevű TMP hibakereső rendszerünket, mellyel könnyebben nyomon követhető a template példányosítási lánc, vagyis a metaprogram futása. Módszereink erős eszközöket adnak a fejlesztők kezébe, és segítik a metaprogramozást.

Az V. részben áttekintjük a jelentősebb kapcsolódó munkákat. Bemutatjuk más nyelvek generikus elemeit, és összevetjük azokat a C++ template-ekkel. Elemezzük a sablon-szerződés modell és a C++ nyelv kapcsolatát, bemutatjuk a kapcsolódó nyelvi, és metaprogramozási elemeket. Végül a metaprogramozás néhány aktuális alkalmazási területét tekintjük át. Bemutatunk két nemsztenderd könyvtárat, melyek a szintaktikai részletkérdéseket elrejtve erős kiindulási alapot adnak a metaprogramozás során.

I. rész

C++ Template Metaprogramozás

A következőkben bevezetjük a dolgozatban használt alapfogalmakat. Áttekintjük általában a metaprogramozást, majd a template metaprogramozáshoz szükséges nyelvi eszközt, a C++ generikus nyelvi szerkezetét, a template-et mutatjuk be. Ismertetjük alapvető tulajdonságait, összehasonlítjuk más nyelvek generikus szerkezeteivel. Bemutatunk néhány olyan nyelvi elemet, melyek elengedhetetlenül fontosak a template-ek hatékony használatához. Tárgyaljuk, hogyan lehetséges template-ek segítségével irányítani a fordítóprogram működését, ezáltal algoritmusok végrehajtására kényszeríteni azt. Bemutatjuk e programozási stílus, a template metaprogramozás fontos módszereit. Megvizsgáljuk a template metaprogramozást mint paradigmát, illetve áttekintjük felhasználási területeit.

1. fejezet

Metaprogramozás

A *meta-* előtag jelentése: valami feletti, valamiről szóló. Ezen előtag egy új, magasabb absztrakciós szint bevezetését jelenti az adott világba. A szemlélet, hogy újabb absztrakciós szintekkel számunkra érthetőbben írjuk le a világot, a kezdetektől jelen van az informatikában és a számításelméletben. Gondoljunk például a *kétszintű nyelvtanokra*[51], ahol egy-egy meta-, illetve hiper-szintű környezetfüggetlen nyelvtant vezetünk be. A meta-szintű nyelvtan képes hiper-szintű szabályokat generálni, majd magukat a szavakat a hiper-szintű generált szabályokból vezetjük le. E technikával véges számú nyelvtani szabályból végtelen sok szabályt generálhatunk le. Sőt, környezetfüggő nyelvtant írhatunk le két környezetfüggetlen nyelvtannal. Tekintsük például a

$$\{a^n b^n a^n | n \geq 1\}$$

nyelvet, mely nem környezetfüggetlen. Ugyanakkor az alábbi kétszintű nyelvtan képes leírni a nyelvet úgy, hogy mindkét nyelvtanja környezetfüggetlen.

Meta-nyelvtan:

$$N \rightarrow 1|N1$$

$$X \rightarrow a|b$$

Hiper-nyelvtan:

$$S \rightarrow a^N b^N a^N$$

$$X^N 1 \rightarrow X^N X$$

$$X^1 \rightarrow X$$

Ilyen kétszintű nyelvtanok segítségével írták le például az Algol68 nyelv szintaxisát is, mivel ez sokat egyszerűsített a nyelv definícióján.

Hasonlóképpen képzelhető el a *metaprogram* fogalma. Itt olyan programról beszélünk, mely maga is egy programot manipulál. Tehát itt is kétszintű absztrakciót tekintünk, melynek alsó szintjén egy szokásos módon elkészített program, meta-szintjén pedig egy azt feldolgozó másik program, vagy kódrészlet helyezkedik el. A többszintű nyelvtant bemutató példánkkal analóg működést hajt végre egy *parser generátor program*, mint például az ANTLR[63] vagy a YACC[66]. Ezek az eszközök megkönnyítik egy új programozási nyelvet elemző *parser* megírását azáltal, hogy egy általunk megadott nyelvtanból (meta-szint) egy működő parser programot *generálnak* (hiper-szint).

Általában a meta-szint, illetve meta-szintek bevezetésétől azt várjuk, hogy *több* információt legyünk képesek *kifejezőbben* leírni, esetlegesen olyan információt, amely *magáról a hiper-szintről szól*, ezért a hiper-szint azt nem tárolhatja.

Tegyük fel például, hogy egy áruház raktárkészletét egy XML file-ban tároljuk[64], egy-egy árucikkről megadhatjuk, hogy hány darab van belőle, mennyi az egységára, ki volt a beszállító, stb. Ezen az absztrakciós szinten írhatjuk le a már megszokott módon az adatokat. Ám azt tárolni, *hogyan* írtuk le ezen adatokat, már a meta-szint feladata.

Adataink feletti *metainformációnak* nevezzük az olyan információkat, amelyek az adatainkról szolgáltatnak információt. Metainformáció ezen raktárkészlet-adatbázis szempontjából például az XML-hez tartozó XSD (vagy DTD, stb.) file, mely leírja, hogyan kell felépülnie egy jólformált adatbázis file-nak. Szintén metainformáció az XML file karakterkódolásának leírása (UTF-8, ANSI, stb.), hiszen ez sem tartozik hozzá a file tényleges adattartalmához, hanem annak minőségéről ír le információt.

Tekintsünk a programozás különböző területeiről ismert további metaprogramozási példákat:

- *Fordítóprogram* – Az általunk megírt forráskódot képes elolvasni és átformálni. Programunk jelen esetben nem is tud információt szerezni arról, hogy egy magasabb absztrakciós szintről tekintve csak egy adat, melyet a magasabb szinten manipulálnak.

- *Reflection* – Több programozási nyelv támogatja azt a funkciót, mely segítségével lehetséges futási, vagy fordítási időben magáról a programról információt szerezni (pl. egy osztály tagfüggvényeinek neve, száma, stb.), illetve magát a programot módosítani. Ilyen futási idejű önreflexiót támogató nyelv például a Java, mely ezt a funkcionalitást egy könyvtár és a nyelvi környezet együttesével valósítja meg.

  A reflection-t két további kategóriára szokás osztani: *introspection*-ről beszélünk, amennyiben csak olvasni tudjuk a programot, *intercession*-ről, ha módosítani is lehet.

- *Debugger rendszerek* – Egy adott futó program különféle adatait tudják nyomon követni, úgymint regiszterek, változók értéke, call stack állapota, vezérlés állapota, stb. Egy debuggerrel végigléphetünk a program futásán, azt felügyelhetjük, egyes esetekben akár a változókat kívülről módosítva befolyásolhatjuk a program állapotát.[69]

- *Generatív programozás* – E fogalom a programozás olyan formáját takarja, mely során a program egy részét automatikusan generáltatjuk valamilyen nyelvi eszközzel. Ilyen például az Aspektus-orientált programozás (AOP)[16]. A generatív programozással részletesen a 4.1. fejezetben foglalkoznk.

Dolgozatunkban a továbbiakban a metaprogramozás C++-specifikus részével, a *template metaprogramozással* foglalkozunk.

2. fejezet

Template – a C++ generikus szerkezete

2.1. Motiváció a generikusok bevezetéséhez

A *template*[42, 44, 55] a C++ nyelv nagy kifejezőerejének egyik legfontosabb összetevője, melyet viszont sokszor félreértenek, rosszul, vagy egyáltalán nem is használnak. 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 számok, illetve egy lebegőpontos számok maximumát meghatározó függvény. E két függvény a következőképpen valósítható meg:

```
int max (int a, int b)
{
    if (a > b) return a;
    else return b;
}

double max (double a, double b)
{
    if (a > b) return a;
    else return b;
}
```

Könnyen látható, hogy a két függvény közötti különbség csupán a paraméterek, illetve a visszatérési érték *típusa*. Programozás közben a karbantarthatóság, és könnyebb hibakeresés érdekében törekszünk a kódismétlés elkerülésére. Amennyiben lehetséges, szeretnénk e két függvényt összevonni, hiszen törzsük, és az általuk megvalósított algoritmus teljesen megegyezik. Ezen absztrakció megvalósítására lehetőséget nyújtanak nekünk az *előfordító (precompiler, preprocesszor) makrók*. Az előfordító egy egyszerű szövegműveleteket (keresés-csere, másolás, áthelyezés stb.) végrehajtó program, mely a C++ nyelvi fordító előtt fut le a forráskódon (bővebben ld. 2.4. fejezet). Tekintsük a következő C++ makrót:

```
#define MAX(a,b)    a > b ? a : b
```

Ezzel a megoldással több komoly probléma adódik. Egy `MAX(x,y)*2` hívástól például azt várnánk el, hogy a két szám maximumát megszorozza kettővel. Ehelyett a szöveghelyettesítés után a következő karaktersorozat lesz megtalálható az előfordított kódban: `x > y ? x : y*2`. Tehát amennyiben `y` nagyobb, valóban annak kétszeresét szolgáltatja a kifejezés, ellenkező esetben viszont `x` eredeti értékét. Ez a probléma kiküszöbölhető a következőképpen:

```
#define MAX(a,b)    ((a) > (b) ? (a) : (b))
```

Ezzel az új verzióval a `*2` karaktersorozat a külső zárójel mögé kerül, tehát a probléma megszűnik. Ám más probléma lép fel a `MAX(++x,y)` makróhívás esetén, mely a `++x > y ? ++x : y` sorozatot eredményezi. Látható, hogy `x`-et kétszer is megnöveljük, ám valószínűleg nem ez volt az eredeti programozói szándék.

Az igazi probléma viszont akkor jelentkezik, ha a két objektum értékének maximumával valamilyen műveletet is végezni akarunk. Ehhez ugyanis ismernünk kellene a paraméterként kapott objektumok típusát:

```
int max (int a, int b)
{
    int temp = a > b ? a : b;  // !! mi a paraméterek típusa?
```

```
    // müveletvégzés temp-pel
    return temp;
}
```

Sajnos azonban nincsen olyan eszköz a birtokunkban, amellyel kikövetkeztethetnénk a makró paramétereinek típusát, mivel a makró típustalan.

A szintaktikus hibák kiszűrésére, illetve a típusosság megtartására a nyelvben a template nyelvi szerkezet áll rendelkezésünkre. A template-ek segítségével hozhatunk létre egy olyan konstrukciót, amely e függvények közös részét egy sablonná formálja, és a hivatkozott típust mint argumentumot kezeli. Egy *függvény-template* definíciója a következő formájú:

```
template <typename T>
T max (T a, T b)
{
    T temp = a > b ? a : b;     // típusnév!!
    // müveletvégzés temp-pel
    return temp;
}
```

A `T` típusparaméter jelöli a majdan a függvénynek paraméterként átadott objektumok típusát. Egy template soha nem egy konkrét függvényt jelent, hanem egy "gyártási eljárást" melynek segítségével szükség esetén egy konkrét függvényt hozhat létre a fordító.

2.2. Osztály template-ek

A dolgozat hátralevő részében nagyrészt szintén a nyelv részét képező *osztály template*-ekkel foglalkozunk. Az osztály template-ek az előzőekhez hasonlóan egy típusokkal paraméterezhető struktúrát határoznak meg. C++–ban a következőképpen nézhet ki egy ilyen struktúra definíciója:

```
template <class T>
class list
{
public:
    list();
```

```
    void push_back(const T& x);
    bool empty() const;
    void sort();
    ...
private:
    T* v_;
};
```

Template paraméternek nevezzük a template kulcsszó után két "kisebb" és "nagyobb" jel (*angle bracket*) között a `class` vagy `typename` kulcsszavakkal prefixelt azonosítók 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*)[42] van szükség. Ez történhet a fordító által implicit, vagy a programozó által explicit módon is. A fordító akkor példányosít egy template-et, ha az abból létrejövő típusra szükség van. Például:

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

A `list` template mögötti angle bracket-ek 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). Az általunk definiált `MyType` típussal történő explicit példányosításra példa a következő részlet:

```
template class list<MyType>;
```

Ekkor a fordítót utasítjuk, hogy az adott ponton (mely csak a globális névtérben lehet) hozza létre a `list<MyType>` típust, ha addig nem tette meg. Ezzel

növelhetjük 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. 3.3. fejezet) is hasznos eszköze lehet.

Ugyanabból a template-ből legyártott két típus csak akkor egyezik meg, ha minden template argumentum megegyezik. Tehát `list<int>` típus soha nem lehet egyenlő `list<MyType>` típussal, annak ellenére, hogy ugyanabból a sablonból generálja őket a fordító. Ez megfordítva is igaz: amennyiben a `list<int>` típust már egyszer példányosította a fordító, abban a fordítási egységben már nem fogja ezt újra végrehajtani, és mindig az első példányosításkor létrejött kódhoz nyúl, amikor a későbbiekben a `list<int>` típusra történik hivatkozás. Ez a működés ellentétes pl. az Ada-éval, amely minden `new` utasítás esetén újrapéldányosít[72] (esetleges optimalizáció lehet a közös kód használata).

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

```
template<>
class list<bool>
{
public:
    list();
    void push_back(const bool& x);
    bool empty() const;
    void sort();
    ...
private:
    bitset v_;
    // itt teljesen más implementáció, reprezentáció lehet
};
```

Ne feledjük, hogy a template-ek csak sémák arra nézve, hogy majd a későbbiekben hogyan gyártson újabb, immár futtatható kódot belőlük a compiler.

Tehát mikor explicit módon definálunk a `class list<bool>`-et, akkor már nem egy sémát, hanem egy konkrét típust definiálunk. Így egyrészről szabad kezet kapunk, hogy mi kerüljön a specializáció törzsébe – tehát nem köt minket az eredeti template definíció –, másrészt, ezek után egy `list<bool>`... kifejezés kiértékelésekor a már meglévő, általunk definiált típust fogja használni, előnyben részesítve azt a sémával szemben. Példánkban kihasználjuk, hogy egy igazságértéket elég egy biten ábrázolni, igy egy bitek tárolására alkalmas adattipust használunk reprezentációként. Megjegyezzük, hogy hasonló reprezentációt használ a *Standard Template Library* std::vector<bool> típusa.

Részleges specializációnak nevezzük az olyan specializációt, melynek paraméterei között 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 push_back(const T*& x);
    bool empty() const;
    void sort();
    ...
private:
    T** v_;
    // újabb implementáció és reprezentáció
};
```

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

Fontos tulajdonsága a template definícióknak, hogy két menetben dolgozza fel 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.

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++ 8 (továbbiakban MSVC 8) és a GNU g++ 4.1 között nagy különbségek voltak már itt, az elemzés közben is. A paraméterezéstől függően a MSVC 8 (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. 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.

Újabb egyedülálló tulajdonsága a template-eknek, hogy nem csak típusokkal paraméterezhetők. Lehetnek egész értékek, függvénypointerek, sőt akár egy másik template is. Tekintsünk erre egy rövid példát:

```
template <class T>
class Q { ... };

template <class T>
class R { ... };

template<template <class T1> class Expr>
class A { ... };
```

```
template<template <class T1> class Expr>
class A<Q<T1> > { ... };

... A<Q<int> > ...
... A<R<double> > ...
```

A fenti szintaktikával írható le, hogy `A` egy olyan osztálytemplate-et (`Expr`) vár paraméterként, melynek egy típusparamétere van (`T1`). Látható, hogy a `Q<int>` típust argumentumként adjuk az `A` template-nek. Az általános esetben amennyiben az `A` használatára van szükség, az első verziót választja a fordító. Amennyiben viszont egy `Q`-ból generált típust adunk át `A`-nak argumentumként, a második részleges specializáció példányosul. Vegyük azt is észre, hogy ha két `>` jel kerülne egymás mellé, kötelező space-t tenni közéjük, ellenkező esetben `>>` operátorként értelmezi a nyelv parsere. Ez egy olyan inkonzisztencia a nyelvben, melyet a következő szabványban orvosolni fognak.

A példában látható tehát, hogy példányosításkor a C++ fordító *pattern matching*-et hajt végre, vagyis a megfelelő paraméterek helyére behelyettesítve az argumentumokat, azt vizsgálja, melyik template paramétermintája illeszkedik legjobban az elvárthoz. Ezután a kiválaszott template-et példányosítja. Az előzőekhez hasonlóan használhatóak a már említett egész számok is:

```
template <class T, int N>
class Array
{
   ...
   T t[N];
};

... Array<MyType,5> ...
```

A fenti példában a tömb méretét már fordítási időben használnunk kell, ezért nem lehet esetlegesen az `Array` típus konstruktorának paramétere. Ám template paraméterként megadhatjuk a tárolt `MyType` típus mellett a tömb nagyságát is, így azt felhasználhatjuk a `T t[N];` tömbdeklarációnál.

2.3. Álnevek és adattagok

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; sokszor a template-es kódok 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. Tekintsük a már példaként felhozott `list` adatszerkezet egy másik lehetséges kódrészletét:

```
template <class T>
class list
{
public:
    typedef T value_type;
    ...
};

template <class U>
void print_container(const U& container)
{
public:
    ... typename U::value_type ...
    ...
};

list<int> l;
...
print_container(l);
```

Szükségünk lehet rá, hogy ismerjük például egy függvénynek átadott adatszerkezet által tárolt elemek típusát. Ezt csak úgy érhetjük el, ha maga az adatszerkezet gondoskodik arról, hogy ezt egy typedef segítségével a külvilág számára láthatóvá tegye, mint az a `print_container()` függvényben látható is.

Fontos megemlíteni a `typename U::value_type` kifejezésben a `typename` kulcsszó szerepét. Minden függő-nevet prefixálni kell ezzel a kulcsszóval, ugyanis ezzel írjuk le explicit módon a fordítónak, hogy a *scope operator* (`::`) mögött következő kifejezés egy típus, nem pedig egy statikus tagja a típusnak.

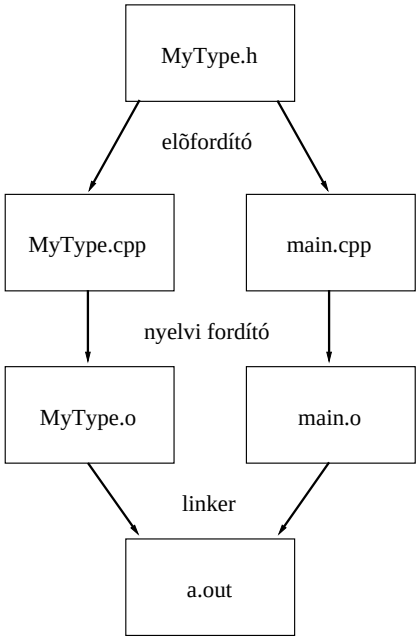
Adattárolásra sokszor használunk *enum*, illetve *static const* tagokat a template-es osztályokban a metaprogramozás segítésére. A már C-ből is jól ismert enum kulcsszóval egy új felsorolási típust hozhatunk létre, ennek értékei csak egészek lehetnek. A static const objektumokban minden fordítási időben használható értéket tárolhatunk a megfelelő típussal. Eltérő működésük miatt az enum használata hatékonyabb.

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. A class, struct és függvény template-k mellett léteznek *union* template-ek is, ám dolgozatunkban ezekre nem térünk ki részletesen. A metaprogramozás eszközei legnagyobbrészt a struct és a class template-ek, melyekkel viszont részletesen foglalkozunk.

2.4. A C++ fordítási modell

A C++ fordítási modelljét jórészt a C-től örökölte. Egy program *header* file-okból (`.h`, `.hpp`) és *cpp* file-okból (`.cpp`) épül fel. A headerek általában osztálydeklarációkat, osztálydefiníciókat és függvénydeklarációkat tartalmaznak, míg a cpp file-okban ezeknek az implementációját szokás elhelyezni. Minden cpp file külön *fordítási egység*, tehát önmagában értelmes egységet alkot, és a fordítóprogram *tárgykódot* (`.o`) tud létrehozni belőle. A *header* file-ok az előfordítás során kerülnek be a cpp-kbe az `#include` direktíva hatására. Ilyenkor egyszerű szövegmásolás történik, tehát minden fordítási egységbe bekerül a *header* tartalma, ahol hivatkozás történik rá. A tárgykódok összeszerkesztését a *linker* végzi, mely ugyanazokra az entitásokra való hivatkozásokat képes a kódban egy helyre irányítani, az entitás definíciójához. A szerkesztés eredménye a *bináris, futtatható kód* (`a.out`). A 2.1. ábra

ezt az általános esetet szemlélteti. A `main.cpp` file írja le a főprogramot, melyben egy felhasználói típust, a MyType-ot használjuk. A MyType típus deklarációja a `MyType.h` header file-ban, műveleteinek definíciója pedig a `MyType.cpp`-ben helyezkedik el.



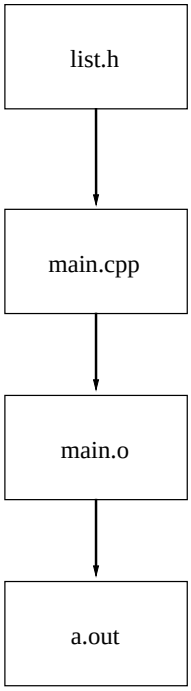
2.1. ábra. C++ fordítási modell I.

A több fordítási egységet használó modell előnye, hogy elkülöníthetők a fejlesztés egyes szakaszai. Például a MyType egyik tagfüggvényének implementációjában (`MyType.cpp`) történt változtatás nem érinti közvetlenül a többi egységet (`main.cpp`), tehát elég csak a módosított cpp-t újrafordítani. Más a helyzet, ha valamelyik *header* file (`MyType.h`) módosult, ekkor ugyanis az összes olyan cpp-t újra kell fordítani, amelyik használta azt.

A fordítási modell egyik különleges és érdekes része a template-ek, specializációik, illetve példányosításaik kezelése. A template-ek, mint említettük, nem önálló kódok, csak tervezési sémák arra nézve, hogyan kívánunk a jövőben konkrét típusokat létrehozni belőlük. Ám mivel nem konkrét kódok, nem generálható belőlük tárgykód, ezért a template definíciók nem írhatók külön cpp file-okba, nem képeznek külön fordítási egységet. Ezért minden osztály template tagjainak implementációja is a megfelelő header-be kell,

hogy kerüljön. Például ha a *list* osztály template-ünket a `list.h` file-ban definiáljuk, ugyanott ki is kell fejteni a tagfüggvényeket, azokat nem tehetjük egy esetleges `list.cpp` file-ba. Ez természetesen az előző modellt semmissé teszi: amennyiben a list valamelyik függvényének implementációja változik, muszáj újrafordítani az összes list-et használó fordítási egységet.

Mivel az előfordító a template kódját minden cpp file-ba bemásolja, többszörös definíciók és névütközések léphetnek fel, melyekre külön nyelvi szabályok vonatkoznak[55]. Ilyen például, hogy a template-ek ellenőrzése két lépcsőben történik: első olvasásakor csak szintaktikai és limitált szemantikai, míg a példányosításkor teljes szemantikai ellenőrzés. A template példányosítások a linker-re is terhet rónak: a linker feladata például két külön fordítási egységben példányosított azonos entitás közül az egyiket eldobnia, és minden hivatkozást a másikhoz kötnie[71], stb. Ugyanakkor a felesleges példányosítások és másolások lelassítják a fordítást.
A template-eket is tartalmazó fordítási modell látható a 2.2. ábrán.



2.2. ábra. C++ fordítási modell II.

E fenti problémákat megoldandó került a nyelvbe az `export` konstrukció. Ezzel szétválasztható egy osztály template és tagfüggvényeinek definíciója a szokásos módon header-, illetve cpp részre. Példánkban a list template 2.2. fejezetben bemutatott része kerülne a header-be, míg a tagfüggvények implementációja a következőképpen kerülne a `list.cpp`-be:

```
export template <class T>
void list<T>::push_back(const T& x)
{
    ...
}

export template <class T>
bool list<T>::empty() const
{
    ...
}
```

Az `export` kulcsszó használatával tehát újra az első fordítási modellt kapjuk vissza. Sajnos ezt a funkcionalitást ezen sorok írásakor csak a Comeau[74] fordítóprogram támogatja (emiatt az egyetlen valóban szabványos fordítónak mondható), és valószínűleg sok compiler gyártó soha nem is fogja implementálni.

Mint láttuk, a template-ek miatt a fordítás bonyolultabbá, lassabbá válik. Ezt felismerve dolgoztak ki a [47] cikkben egy alternatív fordítási modellt, mely lényege, hogy a C++ kódot köztes nyelvekre fordítja, és a műveleteket (mint pl. a példányosítás) itt végzi el. Meg kell még említenünk a [34] cikket, melyben a szerzők a template-ek fordítási mechanizmusának szemantikáját formalizálják, és a levezetések következményeként rámutatnak két anomáliára a nyelv sztenderdjében.

3. fejezet

Template metaprogramok

3.1. Az első template metaprogram

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 [58]. A program fordítása közben a compiler egy int paraméterű template példányosítását végezte el egy megadott felső határig 1-től indulva, majd egy típuskonverziós fordítási hibát adott. Ám ez a hiba csak azon létrejövő típusokat érintette, 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. A kód feldolgozása közben a fordító a következő hibaüzeneteket adta:

```
unruh.cpp 30: conversion from enum to D<2> requested in Pprint
unruh.cpp 30: conversion from enum to D<3> requested in Pprint
unruh.cpp 30: conversion from enum to D<5> requested in Pprint
unruh.cpp 30: conversion from enum to D<7> requested in Pprint
unruh.cpp 30: conversion from enum to D<11> requested in Pprint
unruh.cpp 30: conversion from enum to D<13> requested in Pprint
unruh.cpp 30: conversion from enum to D<17> requested in Pprint
```

Jól látható a hibaüzenetek alakja, melyek tartalmazzák a prímszámokat, és csak azokat. Látható, hogy a fordító számára teljességgel hibás (nem fordítható) program egy magasabb absztrakciós szinten értelmes működést hoz létre. E viselkedés oka a template példányosítások működésében keresendő.

3.2. Rekurzió

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:

```
int Factorial(int n)
{
    if (1 == n) 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, ezzel visszatér, 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;
}
```

Mint említettük, egy template-nek lehet egész szám paramétere, ezt a tulajdonságot használjuk ki a fenti kódban. 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. Az ebben elhelyezkedő `Factorial<N-1>::value` (most `Factorial<4>::value`) kiszámításához példányosítania kell a `Factorial<4>` típust és így tovább. Végül a specializáció miatt a már létező `Factorial<1>`-et fogja felhasználni, és nem próbálja meg azt az általános osztálytemplate-ből példányosítani.

E megoldás és előző, rekurzív függvényt használó programunk között a legszembetűnőbb különbség a jóval bonyolultabb és szokatlan szintaktika. Ennél fontosabb viszont, hogy a rekurzív template példányosítások "kikényszerítésével" fordítási időben rendelkezésre áll a `Factorial<5>` típus `value` értéke, Így a fordító már csak a `120` konstans értéket fogja beírni a generált kódba, nem `Factorial<5>::value`-t. Ez azt jelenti, hogy `r` értékadása lineáris idejű műveletből konstans-idejűvé szelídült. Tehát fordítási időben végeztünk el egy `O(n)` bonyolultságú műveletet, ami ugyan a fordítást (esetleg jelentősen[59]) lelassíthatja, viszont ha egyszer a fordító generálta a kódot, utána futási időben már nem lesz szükség műveletek elvégzésére (az értékadáson kívül, melyet valószínűleg kioptimalizál a fordító).

3.3. Terminológia

Nehéz meghúzni a határt egy ügyesebb template és egy ilyen összetett működést megvalósító *template metaprogram* (továbbiakban TMP) között. Nyilvánvalóan ez csak logikai tagolás lehet, hiszen amit mi template metaprogramnak nevezünk, valójában csak egy C++ kódrészlet. (A metaprogramok logikai tagolásával, és ennek gyakorlati problémáival foglalkozik az 5.1. fejezet.) Ugyanígy, annak érdekében, hogy a jövőben könnyebben különbséget tehessünk a template metaprogramok, illetve a "megszokott" C++ és egyéb nyelven írt programok között, az alábbi logikai tagolást, és terminológiát vezetjük be:

3.1. Definíció. *Template metaprogram*: Olyan template-ek, specializációik, és példányosításaik együttese, melyeket azzal a céllal építünk a programba, hogy azokkal művelet(ek)et végezzünk fordítási időben.

3.2. Definíció. *Futási idejű program*: Futási idejű rögzített típusrendszer keretében futási időben végrehajtott kód.

3.4. Elágazás

A fentiekben bemutattuk, hogyan lehetséges fordítási idejű rekurziót írni. A template-ek segítségével azonban egy ugyanilyen fontos programkonstrukció, az elágazás leírása is megoldható a következőképpen:

```
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 `Cond` fordítási idejű feltétel igazságértékétől függően fog a felső, illetve az alsó template definíció példányosulni. Ha a feltétel igaz, a fenti általános template definíciót, míg hamis esetben a lenti részleges specializációt használja fel a fordító. Így a létrejövő `IF` típus typedef-je az első, vagy a második paraméterként kapott típust kapja értékül.

A *Böhm-Jacopini–tétel*[4] szerint *Turing-teljes* az olyan nyelv, amelyben létezik elágazás és rekurzió. A fentiekben megmutattuk, hogyan lehetséges e két konstrukció létrehozása template-ek segítségével. Ebből következik, hogy a C++ fordítási idő szintén Turing-teljes[46], tehát elméletben kifejezőereje megegyezik a napjainkban használt programozási nyelvek nagyrészével (például a C++-éval is).

Egy fordítási idejű elágazás alkalmazásai lehetnek igen egyszerűek (pl. megvizsgálható, hogy az 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` elágazás aktuálisan használt példányának. A probléma az `enum` kiértékelésénél jelentkezik. 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. Ezen a ponton 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.

A fenti metaprogram tehát 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. Később ezt a kérdést részletesebben tárgyaljuk (ld. 5.1. fejezet).

4. fejezet

A template metaprogramozás mint paradigma

4.1. A paradigma kialakulása

A template metaprogramozás egyidős a template-ekkel. Unruh metaprogramja (ld. 3.1. fejezet), illetve az első expression template-ek (ld. 10.1. fejezet) után a következő fontos állomás az első Template Metaprogramming Workshop volt 2000-ben. A metaprogramozás egyre elterjedtebbé vált, egyre nagyobb számban jelentek meg a témát tárgyaló publikációk, és metaprogramozást felhasználó programozási könyvtárak.

E fejlődés eredményeként a TMP önálló *paradigmává* nőtte ki magát. Paradigmának nevezzük azt a gondolkodásmódot és eszköztárat, amelynek segítségével *absztrakciókat* hajtunk végre, és a feladatunkat kisebb részekre osztjuk fel a könnyebb kezelhetőség érdekében[32]. Az új paradigmák mindig az előző valamilyen gyengeségét próbálják orvosolni, vagy az egyre nagyobb és bonyolultabb szoftvertermékek kézbentartásához nyújtanak segítséget.

Ugyanakkor egy új paradigma nem "írja felül" az előzőeket, hanem azokra építve magasabb szintre emeli a kifejezőkészséget. Az objektum-orientált paradigma például az őt megelőző procedurális programozás eszköztárát felhasználva és kiegészítve definiálta magát. A globális függvények és adattagok tagfüggvényekké és tagokká váltak, létrehozva az *osztály* fogalmát.

Hasonlóan, a *generatív programozás paradigmája*[7] az objektum-orientáltságra és a procedurális programozásra épít, és az alábbi részparadigmákat, programozási stílusokat tartalmazza:

- *Generikus programozás (GP)* – E paradigma a típusok és algoritmusok *generikusságával* foglalkozik. Típusabsztrakciók segítségével kívánja általánosabbá tenni a megírt kódokat. Generikus programozás olyan nyelvekben elképzelhető, amelyek tartalmaznak generikus elemet. Ilyen elem Ada-ban a generic, C++-ban a template, stb., melyeket a 11. fejezetben tekintünk át. A C++ Standard Template Library alapvetően erre a paradigmára támaszkodik.

- *Aspektus-orientált programozás (AOP)* – Az AOP az objektum-orientáltság egy komoly hiányosságát, a *cross-cutting concern*-ek kezelését kívánja megoldani[16]. Ezek olyan, a programban előforduló koncepciók, melyek kódja a programban szétszórtan helyezkednek el, ezért nehéz őket egységesen kezelni és karbantartani. Ilyen *concern* például a naplózás. Amennyiben szeretnénk pl. minden hálózati kapcsolaton elvégzett művelet esetében naplózni az eseményeket, akkor egy nagy szoftver esetében ez sok helyen párszoros kódok elszórását fogja jelenteni. Az AOP megoldása erre az, hogy képes programkódunkat megváltoztatni, kódot *beszőni* adott szignatúrájú függvények elé vagy mögé, vagy befolyásolni egy osztályhierarchia tagjainak viszonyát, stb. Létezik fordítási, és futási idejű aspektus-szövő is. Előbbivel karbantarthatóbb, egyszerűbb kódot írhatunk, míg utóbbival akár futási időben változtathatjuk meg programunk viselkedését. Az AOP legnevezetesebb megvalósítása az AspectJ[17], a Java-ra írt aspektus-szövő.

- *Szándék-alapú programozás (Intentional programming, IP)* – A paradigma elvonatkoztat konkrét programozási nyelven történő implementációtól, és mindig magát a programot tartja szem előtt, azt valamilyen absztrakt formátumban tárolva[35]. Így elérhető a megszokottnál sokkal egyszerűbb *refactoring*, metaprogramozás, stb.

- *Template metaprogramozás*

Napjainkban sokszor nem egy-egy szigorúan vett paradigmával dolgozunk, hanem ún. *multiparadigmás* környezetben fejlesztünk. Ez azt jelenti, hogy programunk több paradigmában írt kódból áll össze. Ezen eljárást az adott programnyelvnek kell támogatnia, ilyen nyelv pl. a C++, melyben egyszerre dolgozhatunk procedurális, objektum-orientált, generatív, stb. stílusokban.

4.2. A metaprogramozás elemei

Egy paradigma esetében mindig meg kell vizsgálnunk, milyen elemekből áll össze, azok milyen funkcionalitást hordoznak. Ugyan a template metaprogramozás paradigmája szorosan kötődik a C++ nyelvhez, a futási idejű programok esetében megszokott entitások nagy része itt nem használható, illetve más elemeket használhatunk fel. A 4.1. táblázatban párhuzamot vonunk a metaprogramok és a futási idejű programok entitásai között.

Metaprogram	Futási idejű program
(template) class	alprogram (függvény, eljárás)
static const és enum osztály tagok	adat (konstansok, literálok)
szimbolikus nevek (típusnevek, typedef-ek)	változó
rekurzív template-ek, typelist	absztrakt adatstruktúrák
static const inicializáció enum definíció *(de nincs értékadás!)*	inicializáció

4.1. táblázat. Futási idejű és fordítási idejű programok közötti analógia

Egy megfelelő tulajdonságokkal felruházott osztálytemplate (önhivatkozás–rekurzió, részleges specializáció–elágazás) megfeleltethető egy alprogramnak, ekkor ugyanis egy különálló algoritmus végrehajtása történik a fordító segítségével. Mivel a template-ek egymásra is hivatkozhatnak, egyikük példányosítása okozhatja egy másik template példányosítását is. Ez egyfajta alprogram hívás, melynek következménye egy *call stack*-nek tekinthető. Mint

említettük, template-ek paraméterként kaphatnak egész számokat, típusokat, sőt más template-eket, stb. E tulajdonság a futási idejű függvények paraméterátadásának fordítási idejű párja.

Az adattárolást jellemzően enum és static const adattagok segítségével oldhatjuk meg. Ezek működése hasonlítható a futási idejű programok konstansaihoz (hiszen a metaprogram adatszerkezetek értéke nem változik). Egy típusnév, illetve typedef megfeleltethető a futási idejű változóknak. Minden művelet és számítás eredménye ezen adattárolók valamelyikébe kerül.

A futási időben megszokott alapvető adatstruktúrák (lista, vektor, stb.) szintén rendelkezésünkre állnak fordítási időben is (ld. 10.3.2. fejezet), általában valamilyen fastruktúra, vagy szekvencia formájában. A már említett enum és static const adattárolók inicializációja ugyanúgy történik mint a futási idejű entitásoké, ám fontos különbség, hogy a fordítási idejű adattárolók nem változtathatják meg értéküket. A nyelvben ugyanis nincsen értékadás, sem változók. Minden entitást (konstans, enum, típus) egyszer definiálunk a létrejöttekor, utána értékük és jelentésük nem megváltoztatható. Az értékadás hiánya miatt van szükség arra, hogy a ciklusokat rekurziókkal fejezzük ki, mivel nincs lehetőség egy esetleges ciklusváltozó értékének megváltoztatására. A metaprogramoknak ugyanakkor nincsen mellékhatásuk sem. A template metaprogramozás ezért *tiszta funkcionális nyelvnek* tekinthető. Ehhez a tényhez kapcsolódó kutatásainkról számol be a 7. fejezet.

Fontos látni, hogy bár a C++ erősen típusos nyelv, a template metaprogramozás nem az. Az általános `template <class T>` template definícióban lévő `T` template paraméter helyére bármilyen típusargumentum kerülhet. Hasonló konstrukció pl. a Java tetszőleges osztályra mutató referenciája, vagy pl. C-ben egy tetszőleges statikus típusú objektumra mutató `void*` pointer. Alapesetben nem tudunk tehát típusrendszert definiálni a template-ek felett. Az ezen egyszerű tényből fakadó problémákkal és megoldásaikkal foglalkozunk a 8., 9.2., 12.1., és 12.2. fejezetekben.

4.3. A paradigma elterjedtsége

A template metaprogramozásban még nem alakultak ki széles körben elfogadott programozási módszertanok, eszközök, könyvtárak, amelyek a metaprogramok fejlesztését támogatnák. Az egyes létező megoldások gyakran ad-hoc jellegűek, a hibajavítások heurisztikusak, ezáltal a fejlesztési költségek nehezen becsülhetőek. A kódolási sztenderdek, és a fejlesztést segítő szoftvereszközök hiánya szintén gátolja a metaprogramozás elterjedését. Sok esetben a fejlesztők már a template-ek használatától is idegenkednek, nem is beszélve az összetettebb template konstrukciókról. Ennek sokszor oka az is, hogy igen régi kódbázist tartanak karban (*legacy code*). E kódok sokszor C-s alapokból indultak ki, vagy olyan régiek, hogy a C++ nyelvnek még nem volt sztenderd része a template. Az ilyen kódokban nehézkes lehet olyan alapvető strukturális változtatásokat eszközölni, mint pl. a template-ek használatának bevezetése.

Kutatásaink során megvizsgáltuk, hogy a nyílt forráskódú (*open source*) szoftvereket gyűjtő SourceForge[73] elnevezésű oldalon hozzáférhető legnépszerűbb C++ projectek mekkora részében használnak *template*-eket, *összetett* template konstrukciókat (teljes specializáció, részleges specializáció, öröklődés template paraméterből) illetve template *metaprogramot*. A vizsgált projecteket, illetve a vizsgálat eredményeit a 4.2. táblázatban foglaltuk össze. Az értelemszerű ,,+" és ,,-" jelölések mellett ,,!"-lel jelöltünk egy speciális esetet, mellyel külön foglalkozunk.

A felsorolt projectek egyike sem tekinthető *legacy code*-nak, mind élő, napjainkban is folyó C++ fejlesztések, melyeket akadémai, ipari, és otthoni célokra egyaránt használnak most is. A táblázatból kiderül azonban, hogy bár az ilyen újnak mondható fejlesztések esetében a template már elfogadott, összetett konstrukciókat csak kevés esetben, metaprogramozást pedig egyáltalán nem használnak.

Az összetett template konstrukciók közül teljes specializációt használ a DC++ és a TortoiseCVS. A WinSCP template paraméterből származtatást, a 7-zip template paraméterrel példányosított típusból való származtatást használ.

Szoftver neve	Project indulása	Utolsó verzió	Leírás	Template	Összetett	TMP
7-zip 4.61	2004-Jun-12	2008-Nov-30	Tömörítőprogram	+	+	-
Audacity 1.2.6	2000-Maj-28	2006-Nov-16	Audio szerkesztő	+	-	-
DC++ 0.707	2001-Nov-17	2008-Nov-27	Filecserélő	+	+	!
FileZilla 3.1.6	2001-Feb-27	2008-Dec-02	FTP, FTPS, SFTP	+	-	-
Irrlicht 0.7.1	2003-Feb-18	2008-Nov-30	3D engine	+	-	-
MediaInfo 0.7.7.8	2003-Jul-29	2008-Dec-12	Médiafile információk	-	-	-
NotePad++ 5.1.1	2003-Nov-24	2008-Dec-14	Szövegszerkesztő	+	-	-
NSIS 2.41	2001-Mar-05	2008-Nov-20	Installer készítő	+	-	-
TortoiseCVS 1.10.9	2002-Mar-03	2008-Nov-23	CVS kliens	+	+	-
WinSCP 4.1.8	2003-Jul-13	2008-Dec-01	SCP kliens	+	+	-

4.2. táblázat. SourceForge projectek

Template metaprogramok tekintetében az egyetlen kivétel a DC++, mely metaprogramot ugyan szintén nem használ, de felhasználja a Boost könyvtár sok elemét. E könyvtár sok helyen alkalmaz metaprogramozási elemeket, egyben a metaprogramozás egyik sztenderd könyvtárának számít (ld. 10.3.2. fejezet).

A következő fejezetekben lehetséges megoldásokat adunk a metaprogramozással kapcsolatos problémákra, ezzel kívánva segíteni a paradigma használatát.

II. rész

Hibakategorizálás

A 3.1. fejezetben láttuk, hogy template metaprogramok esetében a *hibásság* fogalma nem egyértelmű. Az Unruh által írt kódban egy C++ nyelvi szabályokat tekintve helytelen, nem lefordítható kód szerepelt, mely azonban egy magasabb szinten értelmezhető működést hajtott végre. Amennyiben megoldásokat kívánunk adni bizonyos metaprogramokkal kapcsolatos problémákra – jelen esetben hibás metaprogramok javítására –, akkor pontosan definiálnunk kell, milyen eseményeket tekintünk hibának, illetve normális működésnek. Az alábbiakban ismertetjük a témában végzett kutatásunkat, mely arra irányult, hogy a C++ szabvány fogalomrendszerét metaprogramokra átültetve elkülönítsuk a hibás és nem hibás eseteket[28, 36].

5. fejezet

Hibakategorizálás

A fejezetben ismertetjük a C++ szabvány jólformáltságra vonatkozó definícióját. Ezután példákon keresztül mutatjuk be a terminológia kiterjesztését metaprogramokra. Végül a helyességen kívül más szempontokat is bemutatunk, melyek a fejlesztés során felmerülhetnek.

5.1. A C++ szabvány terminológiája

A C++ Nemzetközi Szabvány definiálja a *well-formed* (jólformált), és az *ill-formed* (rosszul formált) programok fogalmát[55]. Egy program *jólformált*, ha a szintaktikai és szemantikai szabályok, illetve a One Definition Rule betartásával készült. Egy program *rosszul formált*, ha nem jólformált.

Amennyiben bármilyen detektálható hiba történik, a fordítónak *diagnosztikai üzenetet* (*diagnostic message*, hibaüzenet) kell adnia. Ám nem az összes rosszul formált programra vonatkozik ez a követelmény. Sőt, amennyiben egy szabványban előírt követelmény nem teljesül, a program működése *nem definiált* (*"whenever this International Standard places a requirement on the execution of a program ... and the data encountered during execution do not meet that requirement, the behavior of the program is undefined and ... places no requirements at all on the behavior of the program"*).

Ugyanakkor még jólformált programok is okozhatnak hibákat amennyiben túllépik a rendelkezésükre álló erőforrások kereteit, gondoljunk például egy

futási idejű végtelen rekurzióra, amikor is a program felhasználja a teljes memóriaterületét.

Tegyük most fel, hogy 0-3-ig szeretnénk kiírni az egész számokat. A következő kódrészlet egy *rosszul formált* programot mutat be, mely *diagnosztikai üzenetet küld* ("ill-formed program with diagnostic message"), mivel az `i` változót nem definiáltuk. Ez fordítási idejű hibát eredményez, és a program futása meg sem kezdődik.

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

A következő példában lévő program lefordul és fut ugyan, de futása soha sem áll le.

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

E példa egy *rosszul formált* program amely *nem küld diagnosztikai üzenetet.* A fordító sikeresen bináris kódot generál a C++ forrásból, ám a program maga hibás algoritmuson alapul, ugyanis egy végtelen `for` ciklust tartalmaz. A hiba oka a hiányzó ciklusfeltétel, ez lesz az a programhiba (bug) amelyet esetleg egy hibakereső (*debugger*) eszköz segítségével tudunk megtalálni.

5.2. A terminológia kiterjesztése

Tekintsük újra a 3.2. fejezetben már ismertetett faktoriális-számító metaprogramot. Tegyük fel, hogy a `Factorial<1>` specializáció tartalmaz egy szintaktikai hibát, az osztálydefiníció végéről lehagytuk a pontosvesszőt:

```
// metaprogram (A)
template <int N>
class Factorial
{
    public:
        enum { value = N*Factorial<N-1>::value };
};

template<>
class Factorial<1>
{
    public:
        enum { value = 1 };
} // hiányzó pontosvesszö
```

A példa egy *rosszul formált* template metaprogramot mutat be, mely *diagnosztikai üzenetet küld*. A metaprogram futása meg sem kezdődik, egyetlen template példányosítása sem ment végbe.

Az alábbiakban szintén egy *rosszul formált, diagnosztikai üzenetet küldő* metaprogramot mutatunk be. Ám ez a metaprogram "futni" kezd, vagyis a fordító megkezdi a `Factorial` metaprogram példányosítási láncának végrehajtását, ám a metaprogram (fordítási időben) *abortál*. A példa teljesebb megértésének érdekében tegyük fel, hogy a programban szintén definiált egy, az `N`-edik `Fibonacci` számot kalkuláló metaprogram.

```
// metaprogram (B)
template <int N>
class Factorial
{
    public:
        enum { value = N*Factorial<N-1>::value };
};
```

```
template<>
class Factorial<1>
{
    // hiányzó "public:" kulcsszó
    enum { value = 1 };
};

int main ()
{
    const int f = Fibonacci<4>::value;
    const int r = Factorial<5>::value;
}
```

A `Factorial<1>` teljes specializációt egy `class` formájában implementáltuk, mely konstrukció alapértelmezett láthatósági szabálya C++-ban `private`. Ennek következtében az `enum { value=1 }` tag is `private`. Ezért amikor a fordítónak a `Factorial<1>::value` értékre van szüksége `Factorial<2>` példányosítása közben, fordítási hibát kapunk, mivel utóbbi osztálynak nincs jogosultsága előbbi `private` tagjainak eléréséhez. Ugyanakkor a `Fibonacci` metaprogram futása zavartalan, függetlenül a `Factorial` hibájától. Ez ellentétben áll az előző esettel, mikor szintaktikai hibát vétettünk a pontosvessző lehagyásával. Most töröljük ki a `Factorial<1>` teljes specializációt:

```
// metaprogram (C)
template <int N>
struct Factorial
{
    enum { value = N*Factorial<N-1>::value };
};

// hiányzó specializáció az N==1 esetre

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

Mivel `Factorial`-nak nincs explicit módon megadott specializációja, ezért a `Factorial<N-1>` kifejezés fordítása el fog vezetni `Factorial<1>`, majd `Factorial<0>`, `Factorial<-1>` stb. példányosításához. Ez egy fordítási idejű végtelen rekurziót eredményez, mely addig tart, amíg a fordítóprogram maga is hibával terminál. A példa egy *rosszul formált* template metaprogram mely *nem küld diagnosztikai üzenetet.*

5.3. Más szempontok

Kutatásaink során tehát kiterjesztettük a metaprogramok világára a C++ nyelvi szabvány jólformáltság fogalmát. A fentieken túl azonban léteznek más szempontok is, melyeket figyelembe kell vennünk a fejlesztés során.

Erőforrások Egy fordító maga is program. Amennyiben a műveletvégzés (fordítás) során a rendelkezésére álló erőforrásokat (pl. memória) teljes egészében felhasználja, mega is futási idejű hibával leáll. Ez igen ritkán fordul elő a "megszokott" használati körülmények között. Ám metaprogramozás során könnyen előidézhető olyan speciális eset, melynek következtében a fordító belső hibával áll le. Ilyen lehet például egy nagyon bonyolult példányosítási lánc elindítása. Legutóbbi példánkban egy végtelen példányosítási láncot mutattunk be. Valójában különféle fordítók különbözőképpen reagálhatnak erre a kódra. A g++ 4.1 a C++ szabványban előírtaknak megfelelően leállítja a fordítást, amikor eléri a 17 szintű implicit példányosítást. A MSVC 6 fordító addig fut, amíg ki nem fogy a rendelkezésére álló erőforrásokból (tesztünkben ez `Factorial<-1308>` példányosítása közben történt). A MSVC 8 pedig `fatal error C1202: recursive type or function dependency context too complex` hibaüzenettel leállt, észlelve, hogy nem lesz elegendő erőforrása a számítás befejezéséhez.
Tegyük most fel, hogy `Factorial<125>::value`-t szeretnénk kiszámítani a programban, és mindezt az eredeti, jól működő `Factorial` metaprogrammal tesszük. Egy sztenderdet jól követő fordító hibát ad, mivel a szabvány 17 szintű implicit példányosítási láncot engedélyez. Ugyanakkor sok fordító, például a g++ is paraméterezhető több szintű példányosítás engedélyezésére.

Ebben az esetben a fordító folytatja a láncot, kockáztatva hogy kimeríti az erőforrásait, mely esetben maga is futási idejű hibával leáll. A mesterséges felső korlátot (pl. 17) a fordítóprogram természetesen nem azért alkalmazza, mert fizikailag nem tudna mélyebb példányosítási láncot kezelni. Épp ellenkezőleg, a korlátra azért van szükség, hogy a compiler minél hatékonyabban tudja a potenciális programozói hibákat kiszűrni, megelőzni.

Portabilitás A C++ nyelv megalkotása során fontos szempont volt, hogy számos architektúrán lefordítható és optimalizálható legyen ugyanaz a programkód. A Java-val ellentétben ezért a C++ nem tesz megkötést például a beépített egész típusok (`int`, `char`, stb.) méretére, csupán azt határozza meg, milyen nagyságsorrendben kell követniük egymást. Éppen ezért amennyiben például egy beépített típus méretétől függő döntést hozunk a programban, elveszítjük a kód *hordozhatóságát*, hiszen programunk különböző architektúrákon és fordítókon teljesen különbözőképpen fog működni. Gondoljuk most tovább eddigi példáinkat, tekintsük a következő kódrészleteket:

```
// futási idejü program
#include <iostream>
int main ()
{
    for (int i=0; i!=sizeof(int); ++i)
    {
        std::cout << i << std::endl;
    }
}
```

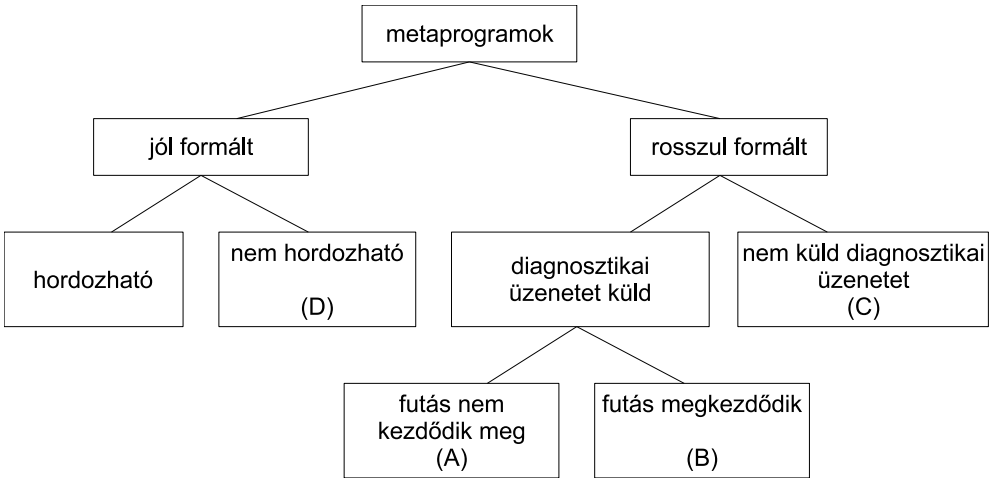
A programozó eredeti szándéka továbbra is az volt, hogy a 0-3 közötti számokat kiírja a képernyőre, a kódban feltételezve, hogy az `int` típus mérete 4 byte. Ez a kód azonban egyáltalán nem biztos hogy a kívánt algoritmust implementálja különböző rendszereken. A kód *jólformált, nem küld diagnosztikai üzenetet* ám nem eldönthető, hogy a kód hibás vagy hibátlan működést produkál-e. Ugyanezt a problémát a metaprogramok körében is értelmes vizsgálni:

```
// metaprogram (D)
template <int N>
class Factorial
{
    public:
        enum { value = N*Factorial<N-1>::value };
};

template<>
class Factorial<1>
{
    public:
        enum { value = sizeof(short)-1 };
};

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

Az `1` konstans helyett a kód a `sizeof(short)` kifejezést tartalmazza a teljes specializációban. Így most semmilyen garancia sincs arra, hogy a kód az eredeti programozó szándék szerint működik. Ugyanakkor ez a kód is *jólformált*, és *nem küld diagnosztikai üzenetet.*

5.4. Áttekintés

Az 5.1. ábrán összefoglaljuk az eddigiekben ismertetett kategorizálást.
Láttuk, hogy a template metaprogramok "futási" és "fordítási" ideje nehezen szétválasztható, a TMP gyakorlatilag interpreter nyelvnek is tekinthető, ezért szükséges volt bevezetnünk a jólformált, és a rosszul formált kategóriákat a metaprogramok világában. Az olyan könyvtárak, mint a boost::mpl e működésből fakadó hibákat szándékozzák orvosolni azáltal, hogy az implementációs részletkérdéseket, és a sokszor ismétlődő kódrészleteket elrejtik a felhasználó elől. Ezen sokak által használt így sok teszten átesett könyvtárak biztos támpontot nyújthatnak további fejlesztésekhez. Ám még e könyvtárak



5.1. ábra. Template metaprogram kategóriák

segítségével is a kódolás nagy része a felhasználóra marad, további hibalehetőségeket rejtve. E hibák megelőzésével, és a már programba került hibák kijavításával a 8., és 9.2. fejezetekben foglalkozunk.

III. rész

A nyelvi kifejezőerő növelése

Az előző fejezetekben ismertettük a template metaprogramozás alapjait, kategorizáltuk az elkövethető hibákat, illetve bemutattuk, milyen nehézséget okozhat a fejlesztőknek a szokatlan metaprogram szintaktika és szemantika. A következő fejezetekben saját kutatási eredményeinket ismertetve megoldásokat adunk a fenti problémákra, majd elemezzük azokat. Megmutatjuk, megoldásaink hogyan könnyítik meg a metaprogramok fejlesztését, és segítségükkel hogyan írhatunk robosztusabb, karbantarthatóbb metakódot.

E részben ismertetünk egy általunk kifejlesztett általános célú metaprogramozás interface-t, mely a TMP és a funkcionális programozás között már ismertetett kapcsolatra építve egy szintaktikailag egyszerűbb, a technikai részleteket elrejtő *programozási felületet* ad a fejlesztő kezébe[37, 39, 40].

6. fejezet

A prímszita algoritmus TMP-ben

Az alábbiakban bemutatunk egy Erathosztenészi szita algoritmust megvalósító metaprogramot, mely az `n`-edik prímszámot adja eredményül (a program Dezső Balázs munkájának felhasználásával készült). A program *handcrafted*, "kézzel írt", vagyis nem valamilyen kódgeneráló eszköz segítségével létrehozott kód. E példakódon keresztül bemutatjuk, hogy még egy ilyen egyszerűnek mondható algoritmus TMP megvalósítása is igen bonyolult.

```
#include <iostream>

template <int n>
struct Int
{
    static const int intValue = n;
    typedef Int<n-1> prior;
    typedef Int<n+1> next;
};

template <typename _Number>
struct InfiniteRange
{
    typedef _Number Head;
    typedef InfiniteRange<typename _Number::next> Tail;
};
```

```
template <typename _List, typename _Prime, bool mod>
struct Filter
{
    struct type
    {
        typedef typename _List::Head Head;
        typedef typename
        Filter<
            typename _List::Tail,
            _Prime,
            _List::Tail::Head::intValue % _Prime::intValue == 0
        >::type Tail;
    };
};

template <typename _List, typename _Prime>
struct Filter<_List, _Prime, true>
{
    typedef typename
    Filter<
        typename _List::Tail,
        _Prime,
        _List::Tail::Head::intValue % _Prime::intValue == 0
    >::type type;
};

template <typename _List>
struct PrimeFilter
{
    struct type
    {
        typedef typename _List::Head Head;
        typedef typename
        PrimeFilter<
           typename Filter<
              typename _List::Tail,
              Head,
              _List::Tail::Head::intValue % Head::intValue == 0
           >::type
        >::type Tail;
```

```
    };
};

typedef PrimeFilter<InfiniteRange<Int<2> > >::type Primes;

template <typename _List, typename _Number>
struct Get
{
    typedef typename
    Get<
        typename _List::Tail,
        typename _Number::prior
    >::type type;
};

template <typename _List>
struct Get<_List, Int<0> >
{
    typedef typename _List::Head type;
};

int main()
{
    std::cout << Get<Primes, Int<PRIME_NR> >::type::intValue
              << std::endl;
    return 0;
}
```

A metaprogram futása és a példányosítási lánc elindítása a `Get` metafüggvény meghívásával kezdődik. E metafüggvény feladata egy adott listából kiválasztani az `n`-edik elemet. Ehhez a fejelemet elhagyva a lista maradékára rekurzívan újrahívjuk `Get`-et, és az `n-1`. elemet választjuk ki, stb. Az eredetileg kívánt elemhez akkor érünk el, amikor `n=0` lesz a hívások során, ekkor az aktuális lista fejelemét kell visszaadnunk.

A listát a `Primes` típus reprezentálja, amely egy végtelen hosszú, egész számokból álló listát (`InfiniteRange`) szűr meg a `PrimeFilter` metafüggvény segítségével. A `PrimeFilter` a lista fejelemét megtartja (ezért kell a prímek listájának `2`-vel kezdődnie az első lépésben), a maradékon pedig

végrehajtja a `Filter` metafüggvényt. Ez argumentumként egy listát, egy prímszámot, és egy logikai kifejezést kap. A metafüggvény két részleges specializációjának rekurzív példányosításaival a lista maradékrészének minden eleméről eldönti, osztható-e az aktuális prímszámmal. Ha igen, akkor a szám biztosan nem prím, ezért elhagyjuk a listából. Ha nem, akkor bent hagyjuk a listában mint prímjelölt. A lista maradékrészét pedig rekurzívan vizsgáljuk tovább.

Az algoritmus igen gyors és hatékony, mivel csak a legszükségesebb elemeket tartalmazza. Jól látszik azonban, hogy maga a kód igen hosszú és bonyolult, nem könnyű fejben végigkövetni a példányosítási láncot, sőt, az eredeti algoritmus teljesen elveszik a szintaktikus elemek között. Továbbá ez egy viszonylag egyszerű algoritmust megvalósító metaprogram, mely demonstrációs célokat szolgál. Nagyobb programrendszerek, bonyolultabb könyvtárak sokkal terjedelmesebb metaprogram kódokat is tartalmaznak. E kódok nehezen olvashatóak, nehezen karbantarthatóak, nem intuitívek. A következő fejezetben megoldást adunk arra, hogyan lehetséges kiküszöbölni ezeket a problémákat.

7. fejezet

Az EClean rendszer

Mint azt a 4.3. fejezetben már tárgyaltuk, a metaprogramozás még nem tekinthető elterjedt programozási stílusnak. Ennek okai többek között a nehézkes szintaxis és a nehezen karbantartható metaprogram kód. Egyrészről ennek a problémának a megoldására hoztak létre különféle metaprogramozási könyvtárakat (ld. 10.3. fejezet), melyek az implementációs részleteket elfedik a könyvtárat használó programozó elől.

Ugyanakkor a 4. fejezetben bemutattuk, hogy a C++ template metaprogramozás a *funkcionális programozási paradigmához* tartozó nyelvekkel azonos tulajdonságokkal bír. Ugyanakkor jelenleg a metaprogramozási könyvtárak nem a funkcionális elvre alapulva építkeznek, nem használják ki a template-ek funkcionális jellegű tulajdonságait. Sőt, a metaprogramozás szintaktikus nehézségeit ezek a könyvtárak sem oldották meg teljes mértékben. A boost::mpl[56] könyvtár például felépítésében és megvalósításában leginkább a Standard Template Library-t veszi alapul, és figyelmen kívül hagyja azt a tényt, hogy az STL egy multiparadigmás környezetre épít. A könyvtárak használata ugyanakkor a makrók és a nyelvi korlátok miatt sokszor körülményes, a programozónak olyan nehézkes szintaktikával kell feleslegesen sokat gépelnie, mely ráadásul könyvtáranként is változik. Nagy segítséget jelenthetne a fejlesztésben egy letisztult, jól definiált *interface*, mely segítségével egyszerű szintaktikával írhatunk metaprogramokat.

Kutatásaink egyik fő célja egy áttekinthető és karbantartható funkcionális-stílusú template metaprogram interface biztosítása a programozó számára[37, 39, 40]. A programozó funkcionális stílusban írt C++-ba ágyazott kódját egy transzlátor klasszikus metaprogram kóddá képes alakítani, majd a fordító ezt a szokásos módon végrehajtani. Az eredményül kapott program szabványos C++ kód[55].

Az alábbiakban egy saját kutatási eredményünk alapján bemutatjuk, hogyan szimulálható template metaprogramok segítségével egy lusta kiértékelésű, tisztán funkcionális nyelv, a *Clean*[18] gráfátírásra alapuló kifejezéskiértékelő rendszerének működése. A Clean nyelv egy szintaktikai és szemantikai részhalmazát alapul véve definiáltuk az *EClean* funkcionális nyelvet. Ez a beágyazott nyelv segíti a karbantarthatóbb metaprogramok írását. Mivel az EClean is lusta kiértékelésű, kutatásunk középpontjában ez a kiértékelési stratégia és ennek a metaprogramokra való alkalmazhatósága áll. Módszerünkben felhasználjuk a Clean típusrendszerét, magasabb-szintű függvényeket és a konstruktor-alapú adatstruktúrákat. A Clean a lusta kiértékelésnek köszönhetően képes végtelen adatstruktúrák kezelésére is. Rendszerünk segítségével rövidebb, áttekinthetőbb, karbantarthatóbb metaprogram kód készíthető, mivel az implementációs részletkérdések a programozó elől rejtve maradnak.

Az alábbiakban először áttekintjük a funkcionális lusta kiértékelés alapvető működését és a TMP-ben használható tulajdonságait.

7.1. Lusta kiértékelés

A Clean nyelvű programokat a fordító egy kifejezésgráfban tárolja. Ezt a gráfot a futási idő automatikusan többszörösen *átírja*. A program futása a *Start* kifejezés jobb oldalán álló gráf átírásával kezdődik. Mi a gráfot *term*-ekre képezzük le, és rendszerünk is *term-átíró rendszer*.

A Clean adatszerkezetei közül először a listát mutatjuk be. A Clean listája láncolt listának tekinthető[18], ám az alábbiakban úgy tekintünk rá mint egy *head* (fejelem) és egy *tail* (a "többi"). E két részből a `Cons` nevű *konstruktor* képes listát létrehozni. Például a `[2,3,4]` listát írhatjuk

`Cons 2 (Cons 3 (Cons 4 Nil))` formában, ahol `Nil` reprezentálja a lista végét. A *lusta* kiértékelési stratégia alapja a *redex* fogalma, mely redukálható kifejezést jelent. A Clean kiértékelési stratégiájában egy *"redex csak akkor értékelődik ki, ha szükséges azt kiszámítani a végeredményhez"* (*"a redex is only evaluated when it is needed to compute the final result"*[27]). Ez a lusta kiértékelési stratégia lehetővé teszi, hogy listáink végtelen sok elemet tartalmazzanak, például a természetes számok halmaza az `[1..]` listával írható le. A végtelen listák használatára egy klasszikus példa az *Eratoszthenészi szita* algoritmus, mely az első tetszőlegesen sok prímet állítja elő.

Definiáljuk még az `enumFrom` konstruktort későbbi használat céljából. Ez a konstruktor egy végtelen listát generál egy adott számtól indulva. Tehát például a `[2..]` lista írható `enumFrom 2` formában. Természetesen ahhoz, hogy a fejelemet kiválaszthassuk a listából, egy átírási műveletre van szükség: `enumFrom n = [n : enumFrom (n+1)]`. Példánkban eredményként a `[2 : enumFrom (2+1)]`, vagyis `Cons 2 enumFrom (2+1)` listát kapjuk. Az összegzés végrehajtása után tehát `[2: EnumFrom 3]` adódik, a 2-ből és a `[3..]`-ból álló lista.

Az alábbiakban bemutatunk egy Erathosztenészi szita algoritmust megvalósító Clean programot, mely az első 10 prímszámot generálja. A példán keresztül bemutatjuk a Clean gráfátíró rendszerének működését, majd saját eredményünket, mely egy általános eljárást ad a Clean-ről metaprogramra való transzlációra. (Megj: az `R1`..`R5` szimbólumok a sorok számozására szolgálnak)

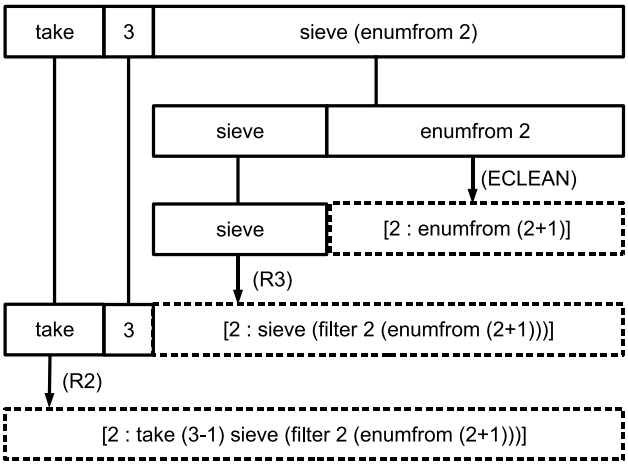
```
(R1)   take 0 xs = []
(R2)   take n [x:xs] = [x: take (n-1) xs]
(R3)   sieve [p:xs] = [p : sieve (filter p xs)]
(R4)   filter p [x:xs] | x rem p == 0  = filter p xs
                                       = [x : filter p xs]
(R5)   Start = take 3 (sieve [2..])
```

A Clean a *balról-jobbra, legkülső* (left-right, outermost) átírási stratégiát alkalmazza. Az első vizsgált kifejezés mindig a `Start` kifejezés. Egy függvény *alternatívái* között sorrendben haladva megpróbálja azokat a kifejezésre illeszteni. Amennyiben a minta tartalmaz konstruktort (például numerikus

konstanst), az adott pozíción lévő részkifejezést *redukálni* kezdi, vagyis rekurzívan kiértékeli addig, amíg a részkifejezés maga is konstruktorrá nem válik. Ekkor már elvégezhető a két konstruktor összehasonlítása. Amennyiben az illesztés sikertelen (a két konstruktor nem egyezik meg), az algoritmus a következő függvényalternatívára lép. Amennyiben az illesztés sikeres volt, és a konstruktornak nincs paramétere, a következő argumentum vizsgálata következik. Ha van paramétere, akkor ezek illesztése következik.

Amennyiben a bal oldal illesztése sikeres, az argumentumok behelyettesítésével a részkifejezést a függvényalternatíva jobb oldalára *írja át*. Az algoritmus akkor ér véget, ha a `Start` kifejezés maga is konstruktorrá válik. Ha egy függvény egyetlen alteratívája sem illeszkedik egy a függvényt tartalmazó kifejezésre, akkor az algoritmus hibával áll le.

Tekintsük példaként a fentiekben bemutatott `(R5) Start` kifejezést. A legkülső kifejezés egy `take` függvény, ezért `(R1)` illesztésével próbálkozik meg az algoritmus. Az `(R1)`-ben található `0` konstans és a `Start` kifejezésben lévő `3` konstans is konstruktorok, ezért összehasonlíthatóak, ám nem egyeznek. Ezért `(R1)` illesztése itt befejeződik, és az algoritmus `(R2)`-t kísérli meg illeszteni a `Start` kifejezésre.

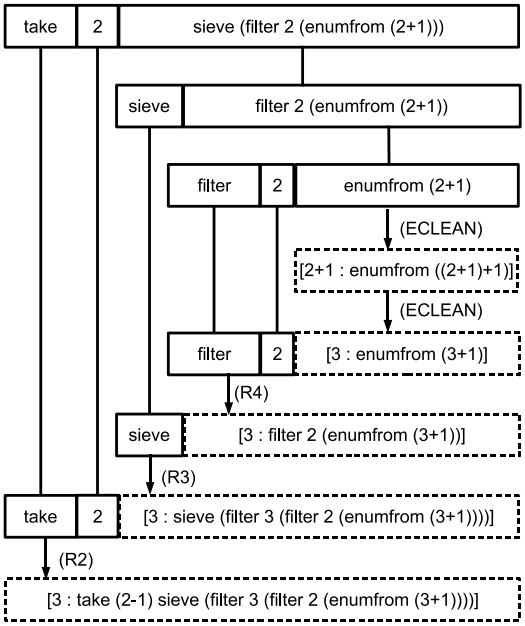


7.1. ábra. A `Start` kifejezés átírása I.

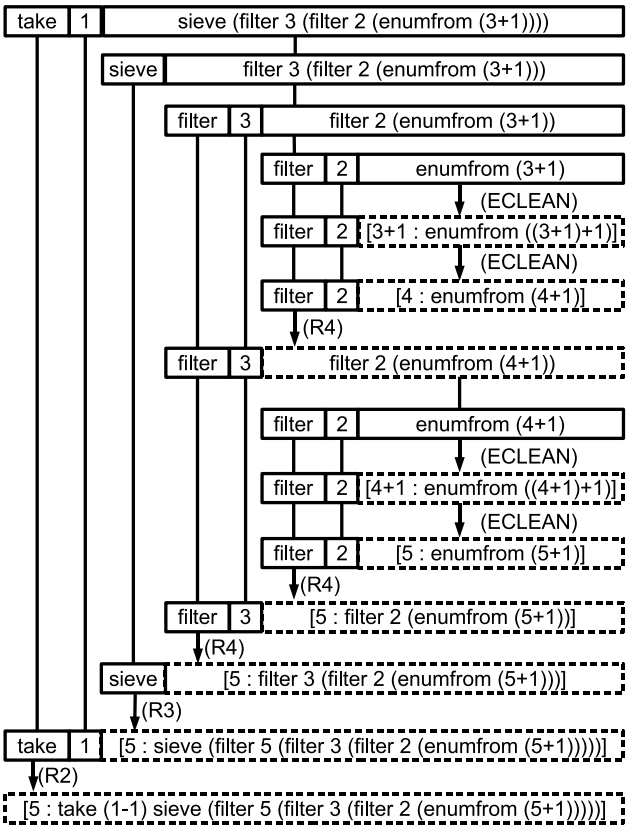
Tekintsük a 7.1. ábrát. Az `n` paraméterbe `3` helyettesítődik, majd az illesztés folytatásához `sieve enumfrom 2` kiértékelésére van szükség, hiszen ki kell deríteni, a kifejezés eredménye valóban lista-e, és ha igen, meg kell

határozni a fejelemét. Ekkor `sieve (enumfrom 2)` vizsgálata következik. Az egyetlen lehetséges alternatíva `R3`, melynek illesztése során ki kell értékelni az `enumfrom 2` kifejezést. Mivel ez a nyelvi szabályok szerint a `[2 : enumfrom (2+1)]` formába írható, ezért itt egy átírás és a részkifejezés cseréje történik meg. Ekkor `sieve [2: enumfrom (2+1)]` már írható `(R3)` szerint `[2: sieve (filter 2 (enumfrom (2+1)))]` formában. Ekkor térünk vissza a részkifejezés kiértékelését elindító `(R2)`-re, mely bal oldalának második argumentuma immár lista formájú `x=2, xs=sieve filter 2 enumfrom 2+1` értékekkel. Ekkor elvégezhető az `(R2)` szabály szerinti átírás, melynek eredményeképpen egy konstruktort kapunk a `Start` kifejezésben. Ekkor az algoritmus leállna, de a háttérben egy nyelvi szabály a lista fejelemét leveszi, kiírja a képernyőre, majd újrahívja a `Start` kifejezést az előző redukció eredményével.

A 7.2. és 7.3. ábrákon a további átírási lépések láthatóak. Minden ábrán található kezdő kifejezés az azt megelőző lépés következménye. Az egyszerűség kedvéért csak `(R2)` illesztésétől kezdődően mutatjuk be a redukciót, ekkorra a `take` első argumentumában található kivonás művelet már kiértékelődött.



7.2. ábra. A `Start` kifejezés átírása II.



7.3. ábra. A `Start` kifejezés átírása III.

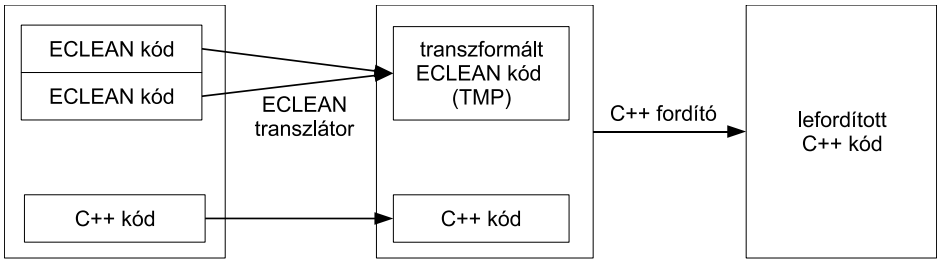
A fentiekben ismertetett algoritmust implementáltuk template metaprogramozás segítségével, így szimulálva a lusta kiértékelési stratégia működését.

7.2. Az EClean rendszer megvalósítása

Az alábbiakban példákon keresztül mutatjuk be egy EClean metaprogram transzformálását C++ metaprogram kóddá. Egy EClean kódrészleteket tartalmazó C++ program fordítása az alábbi lépésekből áll (ld. 7.4. ábra):

- A C++ preprocesszor indítása, mely bemásolja a szükséges header file-okat, és végrehajtja a szükséges makróhelyettesítéseket. A kiegészítő metaprogramokat tartalmazó EClean könyvtárat is ezen a ponton importáljuk.

- A forráskódot felosztjuk C++ és EClean részekre.
- Az EClean részeket a transzlátor C++ template-ekké transzformálja.
- Ez a transzformált forráskód kerül a C++ fordítóhoz.
- A C++ fordító megkezdi a példányosítási láncot azokon a helyeken, ahol a *Start* kifejezés előfordul a kódban.
- A transzlátor által generált kód szimulálja a Clean gráfátíró rendszerét, és így végrehajtja az EClean programot.



7.4. ábra. Az EClean transzformálási és fordítási folyamata

Az alábbiakban a prímszita példáján keresztül ismertetjük az EClean–TMP átírás algoritmusát. Az átírást végző transzlátor Java-ban íródott az ANTLR generátor[63] felhasználásával. A parser rész a Clean nyelv egy részhalmazát ismeri fel, mivel célunk nem egy teljes Clean compiler megvalósítása, hanem egy metaprogramozást segítő beágyazott nyelv készítése volt (l. A. függelék).

A transzlátor TMP formára írja át a függvény- és függvényalternatíva definíciókat. A 4.2. fejezetben kifejtettük, hogy metaprogramok írásához milyen nyelvi konstrukciók állnak rendelkezésünkre. Most ezek közül a typedef-eket és template-ekből készített típusokat használunk fel EClean kifejezések reprezentálására. Sok segítő template-et (mint pl. a fordítási idejű elágazást megvalósító boost::mpl::if_c-t) a boost::mpl könyvtárból használtunk fel. A könyvtárra kompatibilitási okokból esett a választás. Természetesen lehetséges lenne ezeket a segéd-template-eket magunktól megírni, ám amennyiben beépíthetünk rendszerünkbe egy magas színvonalú és alaposan letesztelt könyvtárat, jó választás azt felhasználni.

Tekintsük most az alábbi kódrészletet, melyben a `take` függvény megvalósításának első része látható:

```
template <class __p1, class xs>
struct take : function
{
    typedef typename RedexLoop<__p1, __p1::ctor>::result
        __p1_redex;
    typedef typename take_1_1<__p1_redex, xs,
        __p1_redex::intValue == 0>::redex redex;
};
```

Minden függvényhez tartozik egy általános template, mely pontosan annyi paramétert fogad, mint a függvény. A paramétereket a class kulcsszó előzi meg, tehát a template-eket elméletileg bármilyen típussal paraméterezhetnénk, ezzel típushibát okozva, hiszen a függvény implementációja számít rá, hogy adott pozíción pl. Int vagy Int-ekből álló lista helyezkedik el. A típusbiztonságról ugyanakkor maga a transzlátor gondoskodik, az átírási algoritmus és a transzlátor ellenőrzései garantálják, hogy nem készül hibás kód, hanem az EClean fordítás hibaüzenettel leáll.

A függvény- és függvényalternatíva definíciók (ld. `take`, 2. sor) publikusan származnak a `function`, míg a konstruktorok a `constructor` típusból, melyek implementációja a következő:

```
struct function
{
    static const bool ctor = false;
};

struct constructor
{
    static const bool ctor = true;
};
```

Ezek a *tag*-ek arra szolgálnak, hogy a redukció során el tudjuk dönteni egy adott részkifejezésről, hogy az konstruktor-e. Ezt az információt `ctor` tartalmazza, amely a származtatás során kerül be a template-ekbe. Az in-

formációra a `RedexLoop` rekurzióban van szükségünk (ld. pl. `take`, 4. sor). Ennek megvalósítása a következő:

```
template <class x, bool ctor>
struct RedexLoop
{
    typedef typename x::redex redex;
    typedef typename
        RedexLoop<redex,redex::ctor>::result result;
};

template <class x>
struct RedexLoop<x,true>
{
    typedef x result;
};
```

A rekurziót a szokásos TMP módon valósítottuk meg. Amíg a második paraméter false, vagyis a részkifejezés nem konstruktor, addig az első, általános template implementáció példányosul. Ez a kifejezés `redex` tagtípusát példányosítva elvégez egy lépést a redukcióból, majd rekurzívan hívja magát a redukált kifejezéssel. A rekurzió akkor ér véget, mikor a második paraméter true lesz, vagyis a részkifejezés konstruktor lett. Ezen a ponton kell megemlítenünk egy, a template-es term-átírási és a Clean-es gráfátírási módszer közötti különbséget. Amikor Clean-ben egy részkifejezés átírása megtörtént, akkor a részkifejezést ábrázoló részgráfot lecseréljük a redukáltra, és az összes rámutató pointert átirányítjuk, vagyis az adott kifejezés mindvégig redukált formában lesz jelen. Erre nincs lehetőségünk TMP-ben, hiszen nem gráf formában tároljuk a kifejezést, az adott kifejezésekre nem pointerek állnak rendelkezésre. A template-ekben, típusokban történő kifejezésleírás egyfajta *érték szerinti* szemantikának tekinthető, vagyis ha legközelebb redukálni kell az adott kifejezést, akkor a rekurziót újra végre kell hajtani. Szerencsére a C++ fordító az egyszer már lepéldányosított típusokat (`RedexLoop` bármilyen argumentumokkal) megjegyzi, és legközelebb minimális többletköltség árán újra fel tudja használni. Ezt a tulajdonságot nevezzük *memoization*-nek[1]. Szintén megjegyezzük, hogy a term-ek használata

gráfok helyett általában teljesítménycsökkenést okozna. Ugyanakkor jelen template-es megvalósításunk esetében éppen a gráf reprezentáció jelentene többletköltséget, másfelől a memoization is segítségünkre van a term-ek esetében.

Tekintsük most újra a fenti `take` kódrészletet. A paraméterek a `__p1` és `xs` neveket viselik, melyeket a transzlátor a következőképpen állapított meg. Az EClean–TMP átírás mindig egy adott függvény első alternatívájával kezdődik, és prioritási sorrendben halad lefelé (ezzel egyben garantáljuk az alternatívák prioritását, ahogyan azt Clean-ben megszokhattuk). Az első alternatíva bal oldala most `take 0 xs`, tehát ezt az alternatívát kívánjuk illeszteni az aktuális részkifejezésre. A már ismertett template fejléc megírása után a transzlátor a paramétereket kezdi vizsgálni. Az első paraméter egy konstans, ezért nincs a programozó által adott neve. Ugyanakkor a metaprogram hivatkozik az értékre, ezért a transzlátor automatikusan generál nevet a paraméternek a "`__p`–*paraméter pozíció*" minta alapján, ez példánkban `__p1` lesz. Megjegyezzük, hogy az átírás során többször kell mesterségesen neveket generálnunk, ez adott esetben névütközéshez vezethet a felhasználó neveivel. Az ütközések azonban enyhe megszorításokkal (pl. `_` karakter tiltása) feloldhatóak. Második paraméterünk `xs` neve a kódban is előfordul, így azt felhasználhatjuk a névütközés veszélye nélkül.

A `take` implementációja tükrözi a Clean logikáját. Mivel első kifejezésünk konstans (ezt a transzlátor dönti el), ezért az első argumentumot redukálnunk kell, hogy elvégezhessük rajta a mintaillesztéshez szükséges összehasonlítást. Ezt a műveletet a már említett `RedexLoop` végzi el. A redukálás eredményét a "*változónév*–`_redex`" típusban tároljuk. Az összehasonlítást egy segéd template, `take_1_1` végzi, melynek eredménye a `redex` típus lesz, mely a részkifejezés átírt változatát reprezentálja.

Tekintsük most `take_1_1` definícióját:

```
template <class __p1, class xs>
struct take_1_1<__p1, xs, true> : function
{
    typedef Nil redex;
};
```

```
template <class n, class __p2>
struct take_1_1<n, __p2, false> : function
{
    typedef typename RedexLoop<__p2, __p2::ctor>::result
            __p2_redex;
    typedef typename __p2_redex::head x;
    typedef typename __p2_redex::tail xs;
    typedef IntList<x, IntList<
                take<clminus<n, Int<1> >, xs>, Nil> > redex;
};
```

A transzlátor átíró logikája szerint egy adott változó értékéről mindig egy hozzá tartozó template dönt, melynek neve a következőképpen áll elő: "*függvénynév–_–alternatíva pozíciója–_–paraméter pozíciója*". Példánkban `take_1_1` az első függvényalternatíva első paraméteréről hoz döntést, hogy az megegyezik-e `0`-val.

Amennyiben a `__p1_redex::intValue == 0` igazságértéke `true`, a felső részleges specializáció példányosul, a kifejezés redukciója pedig az üres listát reprezentáló `Nil` típust eredményezi (definícióját ld. később). Ha az összehasonlítás hamis eredményt ad, a második specializáció lép érvénybe. Ekkor tudjuk, hogy az első alternatíva illesztése már nem lehetséges, hiszen az egyik bal oldalon álló konstruktor nem egyezik meg a részkifejezésben találttal. Ekkor a következő alternatívára lépünk, és azt kíséreljük meg illeszteni.

Második szabályunk a `take n [x:xs]` bal oldallal rendelkezik (és második paraméterünk egy lista, melynek nincs saját neve, ezért `__p2`-ként hivatkozunk rá). Mivel a transzlátor által biztosított típushelyességet feltételezve `n` és `__p2` tetszőleges értékkel rendelkezhetnek (egyikük sem konstruktor), ezért az illesztés triviálisan teljesül teszőleges argumentumokra. A második alternatíva illesztésekor tehát nem lesz szükségünk `take_2_n` alakú segéd template-ekre. Mivel a transzlátor minden alternatíva átírásakor megvizsgálja, hány konstruktor van a bal oldalon, ezért a jelenlegi helyzetet is detektálja, és már a második alternatíva jobb oldalát írja be `take_1_1`-be (így lehetséges, hogy `sieve` és `filter` is egyetlen template-ből áll, hiszen tetszőleges argumentumokkal használhatóak, ld. később).

Ahhoz, hogy a második argumentumként átadott listából `x` és `xs` értékét megszerezzük, redukálnunk kell a kifejezést (hiszen lehet pl. függvényhívás az adott pozíción). Ezt a szokásos `RedexLoop` konstrukció végzi, majd a *head* és *tail* részt a megfelelő nevű típusokba mentjük el. Ekkor már átírható a bal oldal a megfelelő lista formátumra.

A jobb oldal felhasznál több olyan EClean konstrukciót, melyeket eddig nem definiáltunk. A következőkben ezeket ismertetjük:

```
template <int n>
struct Int : constructor
{
    typedef Int<n> redex;
    static const int intValue = n;
};

struct Nil : constructor
{
    typedef Nil redex;
};

template <class x, class xs>
struct IntList : constructor
{
    typedef x head;
    typedef xs tail;
    typedef IntList<head,tail> redex;
};

template <class x, class y>
struct clplus : function
{
    typedef typename RedexLoop<x, x::ctor>::result op1;
    typedef typename RedexLoop<y, y::ctor>::result op2;
    typedef Int<op1::intValue + op2::intValue> redex;
};

template <class x, class y>
struct clminus : function
{
```

```
    typedef typename RedexLoop<x, x::ctor>::result op1;
    typedef typename RedexLoop<y, y::ctor>::result op2;
    typedef Int<op1::intValue - op2::intValue> redex;
};

template <class x>
struct enumfrom : function
{
    typedef IntList<x, enumfrom<clplus<x, Int<1> > > > redex;
};
```

Az `Int` konstruktor `redex` kifejezése pedig a kapott érték maga (ez a csomagolási eljárás megegyezik az Int2Type[2] eljárással). Szükséges még az `intValue` mező, mely C++ int formában tárolja az értéket.

Vegyük észre, hogy az Int fenti megvalósítása egy EClean fejlesztési döntés volt. Többféleképpen vezethetnénk be ugyanis új típusokat a nyelvbe. A Clean például az ún. *Nat* típust axiomatikusan definiálja egy a nulla értéket reprezentáló Zero és a rákövetkezési relációt leíró Succ segítségével, úgy, hogy minden értéket ezekből a Peano-axiómákból vezet le. Ugyanakkor ez a módszer template-ekre alkalmazva komoly erőforrásigényt támasztana a C++ fordítóval szemben, mivel sok template példányosításra lenne szükség egyetlen értékhez is. Másik lehetőségünk, hogy az Int típust számos konstruktorral vezetjük be, mely gyakorlatilag a használni kívánt egész számok halmazát definiálná elemenként. Ez utóbbi eljárás alkalmazható esetünkben is. Rendelkezésünkre állnak a C++ beépített típusok, jelen esetben az int. Megvalósításunkban minden Int egyszerűen egy int érték egy template-be *csomagolva*. Egyrészről ez a választás gyorsabb, hatékonyabb és memóriatakarékosabb is, mint az axiomatikus megoldás template-ekre lefordítva. Mivel célunk továbbra is a C++ programozók segítése, számukra elfogadható a C++ beépített típusok használata azok korlátaival együtt.

A már felhasznált `Nil` az üres listát reprezentáló konstruktor, melynek `redex` kifejezése szintén saját maga.

Az általános Int-ekből álló listát az `IntList` konstruktor írja le, mely definiálja a fejelemet tartalmazó `head` és a többi elemet tartalmazó `tail` típusokat, illetve a `redex` kifejezést, mely a lista maga.

A `clplus` és `clminus` konstrukciók már függvények, ezért a `function` *tag*-et kapják meg, feladatuk redukció esetén a két argumentum redukciója, majd a kapott `Int` értékkel történő műveletvégzés.

Végül az `enumfrom` a megfelelő Clean-es végtelen listát implementáló konstrukció (`enumFrom`) template-es párja.
Tekintsük most a `sieve` függvény definícióját:

```
template <class __p1>
struct sieve : function
{
    typedef typename RedexLoop<__p1,__p1::ctor>::result
            __p1_redex;
    typedef typename __p1_redex::head prime;
    typedef typename __p1_redex::tail rest;
    typedef IntList<prime, sieve<filter<prime,rest> > > redex;
};
```

A paraméterként kapott listának nincs neve, ezért a `__p1` néven hivatkozunk rá. Ahhoz, hogy a benne lévő `prime` fejelemet és a lista maradékát, `rest`-et felhasználhassuk, `__p1`-et redukálnunk kell `RedexLoop` segítségével. Ekkor az eddigiek felhasználásával definiálhatjuk az átírási szabály jobb oldalát. Vegyük észre, hogy mivel `sieve` egyetlen alternatívájának bal oldalán nem szerepel konstruktor, nem volt szükséges összehasonlítást végeznünk. A transzlátor ezért olyan kódot generált, mely egyetlen `sieve` példányosítás alapján segéd template-ek nélkül képes kiszámítani az eredményt.
Az alábbiakban ismertetjük `filter` implementációját:

```
template <class p, class __p2>
struct filter : function
{
    typedef typename RedexLoop<__p2,__p2::ctor>::result
            __p2_redex;

    typedef typename _p_redex::head x;
    typedef typename _p_redex::tail xs;

    typedef typename RedexLoop<x,x::ctor>::result x_redex;
    typedef typename RedexLoop<p,p::ctor>::result p_redex;
```

```
    typedef typename
        if_c<x_redex::intValue % p_redex::intValue == 0,
    filter<p_redex, xs >,
            IntList<x_redex, filter<p_redex, xs > >
        >::type redex;
};
```

Egyetlen alternatívánk bal oldalán itt sem szerepel konstruktor, ezért – mint `sieve` esetében – itt sincs szükség döntéshozatalra. A már bemutatott módon redukáljuk a listát tartalmazó kifejezést, majd kiszámítjuk `x` és `xs` értékét. Mivel a jobb oldalon látható oszthatósági vizsgálathoz szükségünk van `x` és `p` értékére, ezért a transzlátor az elágazás konstrukciót generálva a paraméterek redukciójához is kódot ír.

A `filter` elágazásánál transzlátor-szintű optimalizációt hajtunk végre, mivel a Clean az `(R4)` kódot a következő formára írná át:

```
filter p [x:xs] = filter_2 p [x:xs] (x rem p == 0)
filter_2 p [x:xs] True = filter p xs
filter_2 p [x:xs] _    = [x : filter p xs]
```

Ehelyett a megoldás helyett mi a már említett boost::mpl::if_c konstrukció segítségével implementáljuk az elágazást, mely az oszthatóságtól függően a két típus valamelyikét kiválasztva állapítja meg `redex`-et.

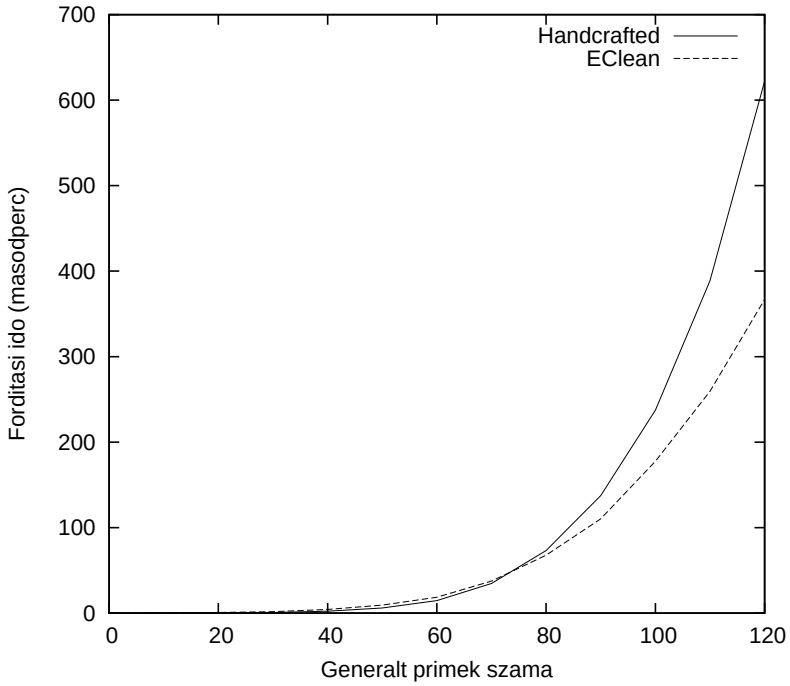
7.3. Az elért eredmények

A kód kifejezőképessége Mint bemutattuk, a transzlátor segítségével definiálhatunk függvényeket, használhatunk elágazást, használhatunk egy beépített típust (Int) és listákat, melyekből template kód generálódik. Lássuk, ezen konstrukciók segítségével hogyan írhatjuk le a 6. fejezetben bemutatott programot! (A példa kedvéért definiáltnak tekintjük a `sum` metaprogramot, mely összegzi az egy Int-eket tartalmazó típuslistában lévő elemeket.)

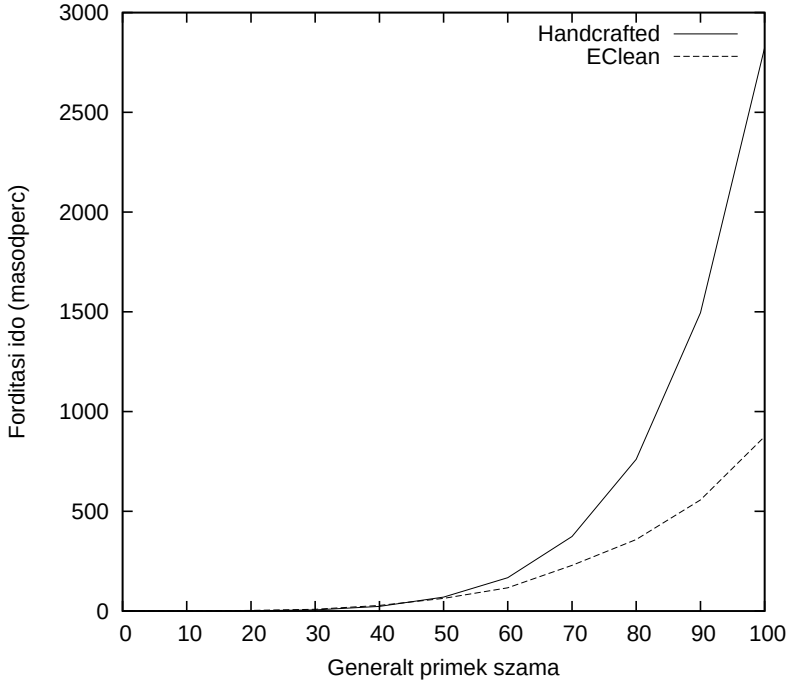
```
BEGIN_ECLEAN(sieve_eclean)

take :: Int [Int] -> [Int]
take 0 xs = []
take n [x:xs] = [x,take (n-1) xs]

filter :: Int [Int] -> [Int]
filter p [x:xs] | x rem p == 0 = filter p xs
                               = [x:filter p xs]

sieve :: [Int] -> [Int]
sieve [prime:rest] = [prime: sieve (filter prime rest)]

Start = take 3 (sieve [2..])

END_ECLEAN

int main()
{
    const int q = sum(ecleanstart(sieve_eclean));
    return 0;
}
```

A fenti, immár EClean-C++ kódon látható az egyszerű, kifejező szintaxis. A programozónak nem kell a megvalósítási részletekkel foglalkoznia illetve a template metaprogramok szintaktikai és szemantikai nehézségeit leküzdenie. Funkcionális nyelvek használatában gyakorlott programozók számára tehát lehetőség nyílt könnyen írható és érthető metaprogram kód írására.

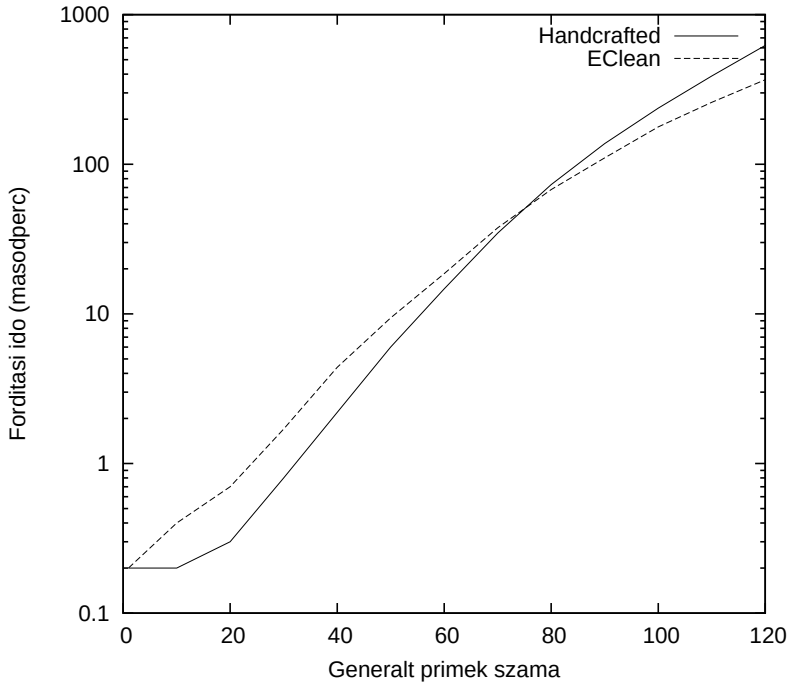
Fordítási sebesség A 7.5., 7.6., 7.7., és 7.8. összehasonlító grafikonok a kézzel írt kód, és az EClean rendszer által generált template metaprogram fordítási ideje közötti különbséget mutatják. A 7.5. és 7.7. ábrák egy 2x4x2.8G CPU-8G RAM-Linux-g++4.1.0 konfiguráción kapott eredmények rendre lineáris, majd logaritmikus skálán ábrázolva. A 7.6. és 7.8. ábrák szintén kétféle skálán mutatják be egy 1x2x1.8G CPU-2G RAM-WinXP2-Cygwin-g++3.4.4 gépen kapott fordítási időket.



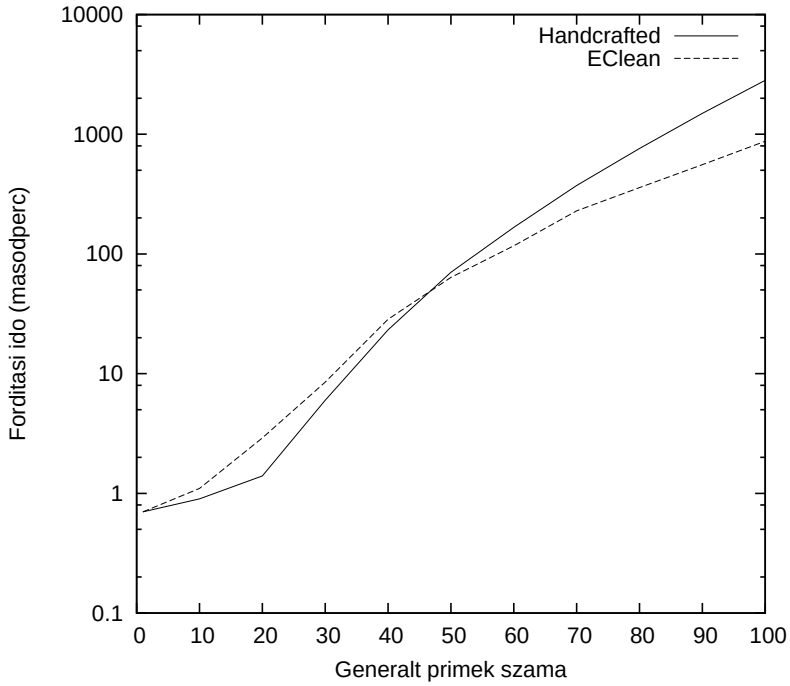
7.5. ábra. A kézzel írt kód és az EClean megoldás fordítási ideje I.



7.6. ábra. A kézzel írt kód és az EClean megoldás fordítási ideje II.



7.7. ábra. A kézzel írt kód és az EClean megoldás fordítási ideje III.



7.8. ábra. A kézzel írt kód és az EClean megoldás fordítási ideje IV.

Látható, hogy az "általános" EClean megoldás egy architektúra-függő alsó korláttól kezdődően gyorsabbá válik mint a 6. fejezetben bemutatott nemgenerált metaprogram. Ez az *ELOC* (Effective Lines of Code) szoftvermetrika használatával könnyen belátható. Tehát a bonyolult szintaktikai és szemantikai elemek elrejtése mellett hatékonyabb kódot is kaptunk. Egyben demonstráltuk, hogy a megfelelő paradigma megválasztása kulcsfontosságú a hatékony programozáshoz illetve jól karbantartható, hatékony kód előállításához. Esetünkben a template metaprogramozás funkcionális programozási jellegét kihasználva adtunk jobb minőségű megoldást a nem lusta funkcionális paradigmában készítettnél.

Továbbfejlesztési lehetőségek és limitációk

- Az egyik legérdekesebb kérdés e hibrid eljárás kapcsán az, hogy lehetséges lenne-e EClean előfordítási időben eldönteni, hogy mely számítások elvégzéséhez van valóban szükség a template metaprogramozás használatára, és melyek azok az egyszerű kifejezések, melyeket transzlátorunk már feldolgozás közben automatikusan kiértékelhetne. Például amennyiben felismeri, hogy egyszerű faktoriális számításról van szó (csak rekurzió és szorzás van a műveletben), elvégezhetné maga a lusta kiértékelést anélkül, hogy template-es formára írná át a függvényeket és fordítási időben a C++ fordítóra hagyná annak hagyományos értelemben vett kiértékelését. Ugyanakkor ha megengedjük, hogy az EClean kifejezések belsejében C++ konstansok és típusok legyenek írhatók, egyfajta *callback* mechanizmust hozunk létre. Mivel ezek a kifejezések bekerülnének a generált template kódba is, ez a lehetőség jól kihasználható kapcsolatot teremtene a két réteg között. Egyben meg is magyarázza, miért van mindenképpen szükség arra, hogy template metaprogram alapokra építsük fel rendszerünket, ahelyett hogy egy egyszerű előfordítót használnánk minden művelet elvégzésére.

- A transzlátor megköveteli, hogy minden függvény az első használat előtt deklarált legyen. Ennek az az oka, hogy nem állt szándékunkban egy teljes funkcionalitást biztosító Clean fordítót implementálni, mely

képes a függvény első használata alapján kikövetkeztetni annak szignatúráját. Ugyanakkor a C++ programozóknak teljesen megszokott az a megkötés, hogy minden függvényt deklarálniuk kell annak első használata előtt.

- Jelenleg egyszerűsítési megfontolásokból a listák nem biztosítanak tetszőleges szintű mintaillesztést (pl. listák listája, függvényhívások listákban). Ez a hiányosság ugyanakkor transzlátor- és metaprogram-szinten is kezelhető lenne.

- Érdekes lehetőség lenne különleges EClean típusok bevezetése, mint pl. Type, mely egy C++ típust reprezentálna (mint egy típusváltozó), vagy a Func, mely egy függvényt írhatna le (hasonlóan egy függvénypointerhez). Ezekkel az eszközökkel EClean előfordítási időben végezhetnénk műveleteket C++ típusokkal, függvényekkel.

IV. rész

Hibaprevenció, hibakeresés

Az előző részben bemutatott EClean rendszer használata csak egy lehetőség, mellyel bonyolultabb template metaprogram kódok jobban karbantarthatóvá, érthetőbbé válnak. Ám sok esetben nem kötjük meg azt a kompromisszumot, hogy a karbantarthatóság a fordítási idő rovására menjen, és továbbra is natív template-es kódot kívánunk írni. A következőkben egyéb alternatív megoldási javaslatokat adunk e kódok fejlesztésének megkönnyítéséhez. A következő részben olyan programozási konstrukciókat ismertetünk, melyek segítségével *biztonságosabban* írhatunk metaprogramokat, *invariánsokat*, illetve *elő-, és utófeltételeket* fogalmazhatunk meg rájuk[26, 36]. Még a legnagyobb körültekintés mellett sem zárható ki azonban, hogy a fejlesztés során hibát vétünk. A hibák könnyebb kiszűréséhez egy a futási és fordítási idő analógiáját (ld. 4.2. fejezet) felhasználó hibakereső (*debugger*)[28, 29, 30] rendszert fejlesztettünk ki, mely segítségével a template példányosítási lánc végigkövethető, és a hiba könnyebben detektálható. E rendszert összevetjük egy *compiler módosítást*[36] használó másik hibakereső megoldásunkkal.

8. fejezet

Hibaprevenció

Az alábbiakban azokat a módszereket ismertetjük, melyek segítségével már a programírás folyamán csökkenthetjük a hibák felbukkanásának valószínűségét. Általában a megszokott futási idejű programok környezetében az alábbi hibaprevenciós eszközöket használhatjuk:

- A programba olyan kódrészletet tehetünk, mely egy informatív hibaüzenet kiírásával leállítja a program futását, ha sérül egy a programra jellemző invariáns, elő-, vagy utófeltétel (*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

Próbáljuk meg ezeket az irányelveket lefordítani a template-ek és a TMP nyelvére.

8.1. Concept checking, conceptek

Mint azt a 12. fejezetben látni fogjuk, a jelenleg szabványos C++ nyelvben nincs lehetőség explicit kikötéseket tenni a template argumentumok különféle tulajdonságaira. Emiatt a TMP típustalan (ld. 4. fejezet), és ez

komoly nehézségeket okozhat a programfejlesztés során. Nekünk magunknak kell gondoskodnunk olyan ellenőrzésekről, mint pl. megfelelő típusú paramétert kap-e a metaprogramunk, vagy létezik-e benne olyan typedef amilyenre számítunk. Ezekhez az ellenőrzésekhez, illetve a template-ek feletti típusrendszer kialakításához nyújtanak segítséget a 12.1. fejezetben említett Boost Concept Checking és Loki könyvtárak, illetve a 12.2. fejezetben tárgyalt concept-ek. Ezeket a segédeszközöket a megfelelő fejezetekben részletezzük.

8.2. Static assert

Következő vizsgált 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. Így elkerülhető, hogy például 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 mit tegyünk abban az esetben, amikor bizonyos elvárásokat nem teljesít egy érték, típus. 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`-té 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))` a következő kimenetet adja 3 vizsgált fordító alatt:

g++ 4.1: `N/A`

MSVC 6: `error C2466: cannot allocate an array of constant size 0`

MSVC 8: `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 (kivétel a g++, mely egy belső fordítóhiba miatt mégis elfogadta a nulla méretű tömböt). A probléma viszont most az, hogy a kapott hibaüzenetből sehogyan sem lehet a hiba valódi okára következtetni. Kissé bonyolultabb, de éppen e 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. A `sizeof` operátor azonban nem alkalmazható definiálatlan típusra, ez fordítási hibát okoz. Megjegyezzük, 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. [56].

Tekintsük újra a `STATIC_ASSERT_2(sizeof(int)==sizeof(double))` utasítás kimeneteit!

g++ 4.1:
```
error:  invalid application of sizeof' to
incomplete type STATIC_ASSERTION_FAILURE<false>'
```

MSVC 6:
```
error C2027:  use of undefined type
'STATIC_ASSERTION_FAILURE<0>'
```

MSVC 8:
```
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 [2]-ben is ismertetett megoldás egy másik formája. Látható, hogy egy második makróparaméter is megjelent. Ez az üzenetátadás céljára szolgál. Definiáljunk egy üres structot, mely 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++ 4.1:
```
error:  invalid application of sizeof'
incomplete type STATIC_ASSERTION_FAILURE<false,
SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE>'
```

MSVC 6:
```
N/A
```

MSVC 8:
```
error C2027:  use of undefined type
'STATIC_ASSERTION_FAILURE<__formal,msg>'
with
[
 __formal=false
 msg=SIZEOF_INT_NOT_EQUAL_TO_SIZEOF_DOUBLE
]
```

Megj: A MSVC 6 nem tudta lefordítani az assertet, mivel a compilerben még nincs részleges template specializáció.

A módszer további finomítását a [28] cikkben közöltük. Definiáljunk olyan `struct`-ot, 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 assert-ezés 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 például 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>)
```

Tekintsük a kimeneteket:

g++ 4.1:
```
error:  invalid application of sizeof' to
incomplete type STATIC_ASSERTION_FAILURE<false,
SIZEOF<int>::NOT_EQUAL_TO::SIZEOF<double> >'
```

MSVC 6:
```
N/A
```

MSVC 8:
```
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.

Megjegyezzük, hogy a következő C++ szabványnak már várhatóan részét fogja képezni egy `static_assert` utasítás, mellyel a fent bemutatott működést válthatjuk ki.

A *static assert* segítségével modellezhetünk *design by contract*-ot TMP-ben[28]. Ezen konstrukciókat megfelelően elhelyezve elő-, utófeltételeket, és invariánsokat ellenőrizhetünk. A témát részletesen tárgyalja [26].

9. fejezet

Hibakeresés

Sajnos a fent ismertetett módszerek ellenére sem kizárt, hogy hibát vétünk a programban. Ilyenkor valamilyen hibakereső (*debugging*) módszerhez fordulhatunk. Tekintsük át, milyen lehetőségeink vannak a futási idejű programok esetében.

- *Naplózhatjuk* a változók értékeit, a program által végrehajtott műveleteket.
- Igénybe vehetjük egy *debugger* program szolgáltatásait, melyek legfontosabb funkciói a következők:
  - A változók értékét figyelemmel kísérhetjük. Változón itt olyan entitást értünk, mely adatot tárol.
  - Utasításról-utasításra lépve (*step*) megfigyelhetjük a folyamatot
  - Elhelyezhetünk töréspontokat (*breakpoint*) a programban, ahol a futás leáll, így átugorhatunk nagyobb blokkokat
- *Intuitív módon*, "ránézésre", vagy a kód végigondolásával, végigkövetésével megpróbálhatjuk kitalálni a hiba okát
- *Slicing* használata. E terület 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[3]. E témával dolgozatunkban nem foglalkozunk, de megjegyezzük, hogy mivel a metaprogramok sok, a slicing szempontjából előnyös tulajdonsággal rendelkeznek, későbbi kutatás tárgyát képezhetné template metaprogramok *slicing*-ja.

Kutatásaink során a naplózás, és a debugger rendszerek témáira koncentráltunk. Fordítsuk most le e két módszerrel kapcsolatos elvárásainkat a TMP világ nyelvére! A "változók" tehát számunkra a példányosuló template-ek argumentumait és a template-ek törzsében található fordítási idejű konstansokat jelentik.

- Iratassuk ki fordítás közben automatikusan a "változók" értékeit
- A debugger
  - tartsa nyilván a "változókat", és azok legyenek lekérdezhetőek
  - kezelje a step- és breakpointokat: ez az adott feltételeknek megfelelő argumentumú template-ek példányosulásakor a fordítás átmeneti leállítását jelenti

A futási idejű programozási paradigmánkban naponta használt eszközök 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. Ennek kérdéseivel jelen dolgozatban nem foglalkozunk.
- A fordítót próbáljuk hibakeresésre használni:
  - Magának a fordítónak erre a célre történő módosításával, ld. 9.1. fejezet
  - A fordítóval irattatjuk ki a kívánt üzeneteket, majd ezeket feldolgozzuk, ld. 9.2. fejezet

9.1. A 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 4.2. fejezetben található összevetés 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 a témában elért eredményeinket ismertetjük[28, 36].

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. Sőt, még ha helyes programról is van szó, a fordítás menete compiler-enként különbözhet, mely alapvetően befolyásolhatja a metaprogramokkal történő műveletvégzés eredményét. Ha az adott fordító viselkedését kívánjuk megérteni, a konkrét fordítót kell módosítanunk.

A C nyelven írt forráskód hozzáférhetősége, valamint viszonylagos egyszerűsége miatt a g++ 3.4 compilert vizsgáltuk. A 3.4-es verziójú fordító ugyan réginek mondható, ám sok helyen a mai napig is használják, tartva az újabb, esetlegesen kompatibilitási problémákat okozó verzióktól. Az itt ismertetett eljárás szinte változtatás nélkül működik az újabb g++ verziókon is.

A compiler módosítása szempontjából számunkra fontos file-ok és függvények a következők.

- `pt.c`, `instantiate_class_template(tree)`: Ez a függvény tartalmazza az algoritmust, amely a tényleges template példányosítást végzi
- `typeck.c`, `complete_type(tree)`: Ez a függvény felelős a típusok "befejezéséért", mint például egy `class` legyártása egy template-ből

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` nevű adatstruktúrákban tárolja a fordításnál megszerzett információkat. Ez a struktúra tulajdonképpen `union`-ok összessége. 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. tárolja el.

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)))`: E 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` (egész szám 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 `tree`-ben kezünkbe adja a template összes ismert információját, lehetőségünk nyílik arra, hogy kövessük a példányosítási sorozatot. Igen egyszerű és mégis nagyon jól használható megoldás, ha kiiratjuk a példányosult `class` template nevét az argumentumaival együtt. A már 5.1. fejezetben ismertetett végtelen rekurziót implementáló hibás metaprogramra a megoldásunk a következő kimenetet adja, melyből jól látszik a végtelen példányosítási lánc:

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

A megoldás előnye, hogy a forráskódot nem kell módosítani (azaz az eljárás *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.

A concept checking módszerek segítésére elképzelhető lenne a fenti módosítás továbbfejlesztése oly módon, hogy a class fordítótól lekérdezett tulajdonságait egy class template-be "csomagolva" fordítási időben rendelkezésre bocsájtanánk. 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-et is példányosítaná az adott típussal mint argumentum. A létrejövő típus mezőit a megfelelő értékekre beállítva, azt a programozó mint egy speciális *reflection* segédeszközt használhatná.

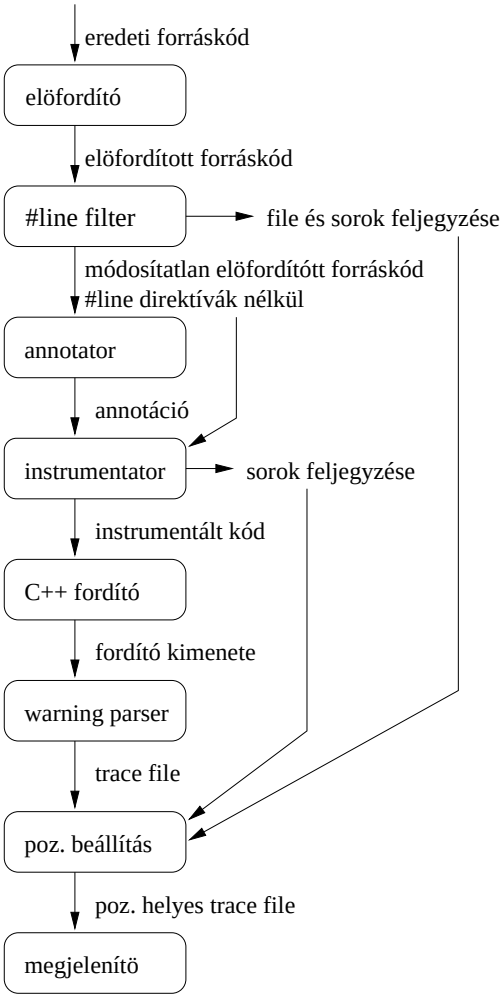
9.2. A Templight rendszer

A hibakeresés témakörében végzett kutatásaink további eredménye a következőkben ismertetett Templight[28, 29, 30] template metaprogram debugger rendszer. A fejezetben áttekintést adunk a rendszerről, annak elvéről, és működéséről, az implementációs részletek megtalálhatóak a [28] cikkben.

Az előző fejezetben egy compiler módosításon alapuló megoldást mutattunk be. Ennek hiányossága, hogy fordítófüggő, csak a g++ fordítóra alkalmazható, hiszen annak belső felépítését, implementációját módosítottuk. Ugyanakkor a compiler kimenetét legfeljebb egy szöveges file-ba rögzíthetjük, majd abban végigelemezhetjük. Ám semmilyen segédeszköz nem áll rendelkezésünkre amely megkönnyítené ezt a hibakeresési folyamatot. Ahhoz, hogy egy fordítóprogramba épített általános, megbízható metaprogram debugger rendszert kaphassunk, minden fordító gyártójának implementálnia kellene azokat az elvárásokat melyeket egy ilyen rendszerrel szemben támasztunk.

Ám ennél egyszerűbb úton is elérhetünk egy kényelmesebb debugger rendszerhez. Az előző ötlettel analóg módon elérhetjük, hogy a template példányosításokat, típusargumentumokat, belső typedef-eket, stb. a fordítási folyamat közben egy *trace file*-ba írhassuk, majd ezt a megfelelő formátumú trace file-t elemzési célokra felhasználhassuk. Ezt fordítóprogram-módosítás nélkül csak úgy érhetjük el, ha informatív *warning* üzeneteket generáltatunk a fordítóval, ezek tartalmazzák azokat az információkat, melyeket ki szeretnénk nyerni [1]. A feladat tehát az, hogy a kódot *instrumentáljuk*, vagyis eredeti funkcionálitását megőrizve olyan kódokkal egészítsük ki, melyek képesek ezeket a warning-okat kibocsájtani. Ez az eljárás szokásosnak mondható a debuggerek, profilerek, és program slicer-ek világában. Itt emlékeztetünk az eljárás alapötletét adó futási és fordítási idejű programok közötti párhuzamra melyet a 4.2. táblázatban mutattunk be.

Valahányszor a fordító példányosít egy template-et, definiál egy belső típust, stb., a beillesztett kódrészlet részletes információt szolgáltat az eseményről. A kívánt információt a megfelelő warning-okból kell összegyűjtenünk, és egy trace file-ban elhelyeznünk. A front-endünk ezt a trace file-t használja a különféle hibakeresési műveletek végrehajtására. A 9.1. folyamatábra áttekintést nyújt a rendszer működéséről.



9.1. ábra. A Templight keretrendszer működése

Bemenetként a rendszer egy C++ forrásfile-t kap, kimenetként pedig egy trace file-t generál, melyben olyan események szerepelnek, mint pl. "*X template példányosítása kezdődik*", vagy "*belső typedef-et találtunk*". A

folyamat a preprocesszor futtatásával kezdődik, mely eredményeképpen a szükséges header-ek és makródefiníciók a forrásfile-ba kerülnek. Az előfordító egyben jegyzeteket is készít arról, hogy melyik kódrészlet honnan származik. Ez az információ egy későbbi vizualizációhoz, vagy bármilyen IDE-szintű felhasználáshoz szükséges.

Az ezután következő *annotálás* során gyakorlatilag feltérképezzük a kódot, megkeressük benne a template-eket, és egy XML file-ba exportáljuk nevüket, soruk számát, stb. Ezt az annotációs file-t használja fel az *instrumentálás* szakasza. Ennek során olyan kódrészletet szúrunk be az annotátor által megjelölt helyekre melyek a fordítás során warning-okat fognak generálni. Ezek a kódrészletek szándékosan olyan formájúak, hogy előre lehessen tudni róluk, pontosan milyen formátumú warning-ot fognak generálni. Jelenleg egy double-ról int-re történő konverziót végez a kód, de ez a konkrét megvalósítás lecserélhető annak érdekében, hogy a compilerek egy szélesebb körét támogathassuk.

A fordítás outputját egy text file-ba mentjük, majd ebből a szövegből a példányosítási adatokat kinyerve egy XML file-t generálunk, kihasználva, hogy jól ismerjük a warning-ok formáját a megfelelő compiler-en. Végül ez az a file, mely a tényleges példányosítási láncot és annak háttérinformációit tartalmazza. Ezen az adatfile-on léphetünk végig, állíthatunk breakpoint-okat. Ezt a technikát *post-mortem debugging*-nak nevezzük, hiszen a futási idejű programokkal ellentétben itt nem a tényleges végrehajtás közben kísérjük nyomon az eseményeket, hanem azok után. Természetesen ettől függetlenül ugyanúgy rendelkezésünkre áll a programon történő változtatás és hibajavítás lehetősége.

A Templight rendszerhez a felhasználó munkáját megkönnyítendő készült egy Visual Studio plugin is melynek segítségével grafikus felületen is végezhető debuggolás. A plugin beépül az IDE menüjébe és a beépített debuggerhez nagyon hasonlóan használható[29].

A fentiekben bemutatott módszereket (compiler módosítás és a Templight rendszer) kutatásaink során *profiling*-ra, a kód hatékonyságának és futási idejének vizsgálatára is felhasználtuk. Ezen eredményeinket tárgyaljuk a [31] publikációban.

V. rész

Kapcsolódó munkák

Az első metaprogram óta a TMP kifejezőereje nem várt új távlatokat nyitott. Ezen új programozási paradigma a generatív programozás[7] egyik önálló programozási stílusává nőtte ki magát. A mai napig a template-ekkel kapcsolatos problémák képezik a C++ nyelv legaktuálisabb kutatási témáit, a metaprogramozásban rejlő lehetőségeket még csak most kezdjük megérteni. A számításigényes műveletek fordítási időben való elvégzése (mint azt a 3.2. fejezetben bemutattuk a `Factorial` példán keresztül) csak egy a TMP felhasználási lehetőségei közül. A következő részben áttekintjük a template metaprogramozás terén eddig elért főbb kutatási eredményeket, számunkra fontos publikációkat illetve a legfontosabb aktuális alkalmazási területeket.

10. fejezet

Metaprogramok alkalmazása

10.1. Expression templates

A C++ *Expression Template*-ek [49] olyan technikai megoldásokat takarnak, melyek segítségével optimálisabb, gyorsabb kódot generáltathatunk a fordítóval. Tegyük fel, hogy létezik egy `Array` típusunk, mely egy tömböt reprezentál, és értelmezett rajta az összeadás művelete:

```
Array A, B, C, D;
D = A + B + C;
```

A fordító ebből olyan kódot generál, melyben az `A + B` művelet eredménye egy temporális lokális változó lesz, melyhez ezután hozzáadjuk `C`-t. Ezután a végeredmény `D`-be másolódik. Ez összesen 3 ciklus, melyben a megfelelő tömbök elemein végzünk műveletet. A tárigény is nagy, mivel 2 temporális objektum jön létre, melyek a részeredményeket tárolják, majd megsemmisülnek. Az objektum-orientált paradigmában megírt kódunk tehát jóval kevésbé lesz hatékony, mintha például Fortran-ban írtuk volna.

Ugyanakkor látható, hogy egyetlen ciklussal megoldható lenne a feladat, hiszen minden iterációban csupán az `i`-edik tagok összegét kell a megfelelő `D`-beli elembe másolni. Így a temporális objektumokat sem foglaljuk le feleslegesen. Egy megfelelő kifejezésfa felépítésével ez a probléma fordítási időben kiküszöbölhető és optimalizálható.

Vezessük be az `X` template-et, mely egy művelet elvégzését reprezentálja, első és harmadik paramétere a két operandus, második paramétere pedig a művelet. Ugyanakkor legyen `plus` az összeadást reprezentáló osztály, melynek `apply()` műveletét hívhatja meg `X`, amikor a művelet elvégzésére ténylegesen szükség van:

```
template <typename Left, typename Op, typename Right>
class X { };

struct plus
{
    static double apply(double a, double b)
    {
        return a+b;
    }
};
```

Az `A + B` kifejezés tehát `X<Array, plus, Array>` formába írható, és mivel a C++ jobbról balra zárójelez, ezért a teljes kifejezés `X<Array, plus, X<Array, plus, Array> >` formájú. A megfelelő értékadó és összeadó operátorok definiálása után fordítási időben a következő átalakítások történnek:

```
D = A + B + C;
  = X<Array,plus,Array>(A,B) + C;
  = X< X<Array,plus,Array>, plus, Array>
     ( X<Array,plus,Array>(A,B), C);
```

Így az összeadást fordítási időben egyetlen kifejezéssé transzformáltuk. A további technikai részletek a [49] cikkben találhatóak.

Ez a kifejezésfa az optimalizálás mellett más előnnyel is jár: a kifejezés hatékonyan adható át paraméterként függvényeknek. Ezek a kifejezés-paraméterek a függvény törzsében *inline*-osíthatóak, így hatékonyabb, gyorsabb lesz a generált kód, mintha a klasszikus C-stílusú ún. callback függvényeket használnánk.

Az expression template-ek a metaprogramozás egyik legrégebbi felhasználási területe: már 1994-ben megjelentek az első ilyen próbálkozások, ezek ve-

zettek el a későbbi könyvtárak, pl. a Blitz++ [60], a PETE, és az uBLAS[56] kifejlesztéséhez.

10.2. Aktív könyvtárak

A programozási nyelvek fejlődésével párhuzamosan jelennek meg az egyre fejlettebb felhasználói könyvtárak. Már a Fortran programok erősen támaszkodtak a gyakran ismétlődő feladatokat megvalósító programkönyvtárakra.

Az objektum-orientált programozási nyelvek elterjedésével a könyvtárak is átalakultak; függvények halmaza helyett állapottal rendelkező osztályok, öröklődési hierarchiák jelentek meg. Mindazonáltal ezek a könyvtárak 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.

Tekintsük újra a 2. fejezetben bemutatott `max` függvényt, mely két elem maximumát határozza meg. Ám tegyük fel azt is, hogy – ellentétben az eddigi példával – a két elem nem azonos típusú.

```
template <class T, class S>
! T vagy S ! max(T a, S b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Világos, hogy erősen típusos nyelvekben fordítási időben nem tudjuk kiválasztani előre, melyik típusú érték lesz a nagyobb, vagyis melyik típust kell a visszatéréshez megadnunk (ezt a döntési problémát a felkiáltójelek közé írt visszatérési típusokkal jelöltük). Másrészről amint megismerjük `a` és `b` típusát, gyakran a változók konkrét értékeinek ismerete nélkül is meg tudjuk állapítani, melyik lenne az alkalmasabb típus a visszatéréshez. Ha például `a` típusa int, és `b` típusa double, akkor az alkalmasabb visszatérő típus a double, függetlenül attól, hogy futási időben `a` vagy `b` értéke lesz-e nagyobb.

Egy általános könyvtár írója azonban ezt a döntést nem hozhatja meg, hiszen `max` függvénye megírásakor még nem ismeri `a` és `b` típusát. Tipikusan ez a helyzet a generikus könyvtárak esetében, így a C++ template-ek, a Java és C# generikusok és az Ada és Eiffel típusparaméterek esetében is[25].

Egy template metaprogram segítségével azonban megoldhatjuk a fenti problémát. Feltesszük, hogy a visszatérési érték tárolására az a típus alkalmasabb, melynek byte-ban számított mérete a nagyobb (ez természetesen csak leegyszerűsítés). Ezt az értéket a `sizeof` operátor segítségével ismerhetjük meg. Vezessük be a harmadik típusparamétert, `R`-et, mely fordítási időben az `IF` elágazástól függően kap értéket:

```
template <class T, class S,
          class R = IF<sizeof(T)<sizeof(S),S,T>::type >
R max(T a, S b)
{
    if (a > b)
        return a;
    else
    return b;
}

z = max(i,x)
z == 3.0;       // rendben
```

Ez a függvény tehát fordítási időben képes döntést hozni, és a döntéstől függően más-más viselkedést produkálni. Az ilyen, fordítási idejű döntéseket hozó, illetve helyzetspecifikus optimalizációkra képes algoritmusokból álló könyvtárat *active library*-nek, azaz *aktív könyvtárnak*[8] nevezzük. Az aktív könyvtárak képesek optimalizálni a generált kódot illetve olyan ellenőrzéseket végezni, amelyek túlmutatnak a fordító absztrakciós szintjén stb. Az aktív könyvtárakat alapvetően 3 csoportra oszthatjuk:

- *A fordító kiterjesztése.* Az aktív könyvtárak kiterjeszthetik a fordítóprogramot információkkal. Elkészíthetnek vagy specializálhatnak algoritmusokat, optimalizálhatják a kódot. Gyakori alkalmazásuk matematikai célú könyvtárak optimalizációja (pl. Blitz++) a sebesség növelésére vagy a memória használat csökkentésére[15].

- *Domain-specifikus eszközök támogatása.* Az aktív könyvtárak kiterjeszthetik a programozási környezetet, hogy támogatást nyújtsanak domain-specifikus eszközök számára[12].

- *Metaprogramozás.* Az aktív könyvtárak tartalmazhatnak *metakódot*, melyet végre lehet hajtani fordítási időben. A metakód felismeri programozási környezetét, például fordítás során a fenti `max` függvény a paramétereinek típusát, és ennek megfelelően önálló algoritmusokat képes végrehajtani. Ugyanaz a metakód eltérő fordítási környezetekre képes eltérően reagálni, így rendkívül hatékonyan lehet a könyvtár utólagos konfigurálását, optimalizálását telepítési vagy alkalmazási időben elvégezni.

10.3. Template metaprogram könyvtárak

10.3.1. Loki

A Loki a [2]-ben leírtakat implementáló általános célú könyvtár. A könyv nagy része néhány *tervminta*[10] (pl. *factory*, *visitor*, *singleton*, stb.) C++ nyelven történő megvalósítását tárgyalja. A tervminták implementálásához sok helyen template metaprogram-betéteket használ a szerző, és bevezeti a *típuslisták* (*typelist*) fogalmát. Mint a neve is mutatja, ez az adatszerkezet egy típusokat tároló lista. A `Typelist` két paraméterű template, melynek első paramétere a *Head*, második pedig a *Tail* típus. A megvalósítás a következő:

```
template <class T, class U>
struct Typelist
{
    typedef T Head;
    typedef U Tail;
};

struct NullType {};
```

A `Typelist` template-nek nincsenek metódusai, sem tagjai, ugyanis maga a típus és annak neve hordozza a tárolt információt. A `NullType` típus egy

egyszerű üres struct, mely a lista végét jelző *tag*-ként funkcionál. Egy char, short és int típusokat tartalmazó lista a következőképpen írható le ennek segítségével:

```
typedef
Typelist<char, Typelist<short, Typelist<int, NullType> > > q;
```

Logikailag tehát mindig egy "fejelem-lista többi eleméből álló lista" párosról beszélünk. Rendelkezésünkre állnak a listák definiálást megkönnyítő makrók is, melyek segítségével egyszerűbb szintaktikával írhatunk le típuslistákat. Vegyük észre, hogy a lista hosszától függően mindig különböző makrót kell használnunk, hiszen a C++ nem kezel változó paraméterszámú makrókat. Lássunk egy példát, melyben az előzővel ekvivalens típuslistát hozunk létre:

```
typedef TYPELIST_2(short,int) q;
```

A Loki könyvtár számos, a `Typelist`-hez kapcsolódó algoritmust definiál. Ezek közül most az `IndexOf` metafüggvényt mutatjuk be, mely megállapítja, hogy egy listában (példánkban a fenti `q` listában) hányadik pozíciót foglalja el egy adott típus (példánkban int):

```
template <class TList, class T> struct IndexOf;

// (1)
template <class T>
struct IndexOf<NullType, T>
{
    enum { value = -1 };
};

// (2)
template <class T, class Tail>
struct IndexOf<Typelist<T, Tail>, T>
{
    enum { value = 0 };
};
```

```
// (3)
template <class Head, class Tail, class T>
struct IndexOf<Typelist<Head, Tail>, T>
{
    private:
        enum { temp = IndexOf<Tail, T>::value };
    public:
        enum { value = (temp == -1 ? -1 : 1 + temp) };
};
```

Az `IndexOf<q,int>::value` hívás hatására a (3) specializáció példányosul `Head=char`, `Tail=typelist<short, typelist<int, NullType> >` és `T=int` argumentumokkal. A `temp` tag kiértékeléséhez rekurzívan meghívjuk az algoritmust a lista `Tail` részére, egészen addig, amíg vagy a `NullType` típus lesz a lista (vagyis a végére értünk), vagy maga a `T` (a keresett típus). Ha nem találtuk meg a típust, akkor az (1) részleges specializáció `value` értékét, `-1`-et kapja értékül `temp`. Ellenkező esetben visszafelé lépve azt számoljuk meg, hogy milyen mélységben vagyunk a rekurzióban, vagyis `T` milyen távolságban van a rekurzió elejétől, a fejelemtől.

A listák head-tail alakú leírását a későbbiekben a 7. fejezetben használjuk még fel. A típuslisták természetesen nem az egyedüli fordítási idejű adatszerkezetek, az alábbiakban a Boost könyvtár által nyújtott lehetőségekkel foglalkozunk.

10.3.2. Boost::MPL

A Boost Libraries[56] az egyik legnevezetesebb és legismertebb C++ könyvtárgyűjtemény. E gyűjteményből jelen dolgozatban három könyvtárral foglalkozunk. E fejezetben áttekintjük a Boost Metaprogramming Library-t (boost::mpl[56]) és a Boost Type Traits (boost::type_traits) könyvtárat, míg a 12.1. fejezet foglalkozik a Boost Concept Checking-el.

A C++ szabványos könyvtára, a Standard Template Library[42] olyan a mindennapi programozást nagyban segítő segédeszközöket tartalmaz, mint generikus *konténerek* (adatszerkezetek, mint pl. std::list, std::vector), *algoritmusok* (a rendezőalgoritmus std::sort(), a valamilyen tulajdonságú elemet

megkereső std::find(), stb.), az ezeket összekötő *iterátorok* illetve egyéb kiegészítő elemek. A metaprogramok elterjedése okán felmerült az igény az alkalmazott meta-algoritmusok, meta-tárolók, meta-iterátorok, meta-függvények stb. egységbe foglalására és egy, az STL-hez hasonló koherens formában elérhetővé tételére. Ezen elemeket a boost::mpl könyvtárban implementálták. A könyvtár tartalma a következő fő elemekből áll össze:

Konténerek Az STL-hez hasonló logikát és funkcionalitást biztosító adatszerkezetek. Mint metaprogramozás esetében sokszor, most is *típusokkal* dolgozunk, és ezeket használjuk fel adattárolásra. Ez a 10.3.1. fejezetben bemutatott Loki könyvtár eljáráshoz nagyon hasonlóan történik. Az alábbiakban a talán legtöbbet használt STL konténer, az std::vector-nak a boost::mpl-beli párját, a boost::mpl::vector használatát mutatjuk be.

```
typedef vector<char,short,int> v1;
```

Mint látható, maguk a típusnevek tartalmazzák a tárolt információt. Két különböző tartalmú *vector* soha nem lesz egyenlő, hiszen eltérő típust definiálnak. Mivel típusról van szó, a typedef kulcsszóval történő átnevezéssel definiálhatjuk a `v1` azonosítót, mellyel később az adott konténerre hivatkozhatunk. A műveletek az STL-ben megszokottaktól eltérően nem tagfüggvényei a konténereknek, hanem globális metafüggvények. Minden a boost::mpl-ben szereplő, műveletet végző metafüggvény kimenete egy `type` nevű típusnév, ez tartalmazza az eredményt, melyre később hivatkozni tudunk.

```
typedef push_back<v1,long>::type v2;
```

Algoritmusok Rendelkezésre áll az STL-es algoritmusok nagy része, használatuk logikailag megegyezik az ott megszokottakkal. Az alábbiakban a `v2` vector-ban keressük az int típust, és egy arra mutató iterátort kapunk vissza:

```
typedef find<v2,int>::type iter;
```

Iterátorok A megszokott koncepciót megvalósító iterátorok és iterátorkategóriák is megtalálhatóak a könyvtárban. A példában az előbbiekben definiált `iter` iterátoron alkalmazzuk a `distance` metafüggvényt, mely megállapítja, hány elem távolságra van iterátorunk `v2` elejétől. Ezt pedig a `begin` metafüggvénnyel számítjuk ki.

```
const int r = distance<begin<v2>::type,iter>::value;
```

Adattípusok Fordítási időben egy magasabb absztrakciós szinten dolgozunk, mint azt futási idejű programok esetében megszoktuk. Típusaink értékeket tároló entitásokká válnak, és a concept-ek ezeknek metatípusai. A futási időben megszokott entitások, úgymint számok, igazságértékek nem használhatóak közvetlenül olyan helyeken, ahol típusokra van szükségünk, ezért ezen értékeket valamilyen típus formájában kell eltárolnunk. A boost::mpl erre is lehetőséget ad adattárolói segítségével. Az `int_` egy egész számot vár paraméterként, például a 4-et. Így a létrejövő típus az `int_<4>` nevet fogja viselni, és ezt a konstrukciót már közvetlenül felhasználhatjuk azokon a helyeken, ahol csak típusok szerepelhetnek. Hasonlóan a `bool_` template true vagy false értéket kap argumentumként. Ezen típusok `value` mezője pedig visszaadja a tartott értéket.

```
bool r = empty<v2>::type::value;
```

Általános metafüggvények A metaprogramozás során sokszor használt általános funkcionalitást megvalósító metafüggvények találhatóak meg itt. Ilyenek a fordítási idejű elágazást megvalósító `if_`, aritmetikai műveletek (plus, minus, stb.), összehasonlítások (less, equals, stb.), logikai műveletek, stb. Az alábbi példában a fordítási idejű elágazás egy felhasználását mutatjuk be: megvizsgáljuk, hogy `v2` vektorunk üres-e, és ettől függően az int vagy a char típusokkal térünk vissza.

```
typedef if_<empty<v2>::type, int, char>::type r;
```

A Boost egyik alapkövetelménye a tartalmazott könyvtárak interoperabilitása. A boost::mpl könyvtár például szoros kapcsolatban áll mind a boost::concept_check-kel, mind a boost::type_traits-el, mivel közös bennük a metaprogramozás elősegítésére való törekvés. Ezen utóbbi könyvtár a típusok különféle tulajdonságainak számontartására képes. Az alábbiakban röviden áttekintjük e könyvtárat.

A *traits* fogalma általában azt jelenti, hogy egy típus valamilyen tulajdonságát (*type_trait*) vagy egy viselkedésmintát (*policy*) template specializációk segítségével írunk le, ezáltal flexibilisebb, bővíthetőbb kódot kapunk. Tegyük fel, hogy implementáltunk egy `T` típusparamétert fogadó mátrix osztály template-et. Amennyiben a `T`-ben kapott argumentum típus byte-onként másolható (*POD* típus), akkor a mátrix másoló konstruktora fordítási időben döntést hozhat, és a tartalma lemásolásához a memcpy() függvényt hívhatja meg, ezzel optimálisabb, gyorsabb kódot eredményez. Amennyiben pedig a típus nem POD-típus, annak értékadó operátorát hívhatja végig minden elemre.

```
template <typename T>
struct is_pod
{
    enum { value = false };
};

template <>
struct is_pod<long>
{
    enum { value = true };
};

template <typename T, bool B>
struct copy_trait
{
    static void copy(T* to, const T* from, int n)
    {
        ... // operator=
    }
};
```

```
template <typename T>
struct copy_trait<T, true>
{
    static void copy( T* to, const T* from, int n)
    {
        ... // memcpy()
    }
};

template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix
{
    ...
};
```

Definiáljuk az `is_pod` template-et, melynek `value` mezője alapértelmezés szerint false, majd e template-et specializáljuk a long típusra, ahol value értéke true lesz. Ugyanakkor a `copy_trait` template-nek is két megvalósítása van: általános esetben operator= hívásokat végez, míg egy POD típus esetében meghívja a memcpy() függvényt. Látható a példában, hogy egyrészről a long típusról leírtunk egy bizonyos tulajdonságot (POD típus), ugyanakkor a mátrix osztály ezen tulajdonsághoz tartozó kétfajta viselkedés közül tudott választani a másoláskor.

A template specializációkon alapuló *policy-based design* a Loki könyvtár (ld. 10.3.1. fejezet) egyik alappillére, míg a típusok tulajdonságaival a boost::type_traits foglalkozik. E könyvtárban típusinformációkat lekérdező, és típusokat transzformáló eljárások találhatóak, ám fontos kiemelni, hogy e könyvtár is nagyban támaszkodik a Loki-ban elért eredményekre. Röviden tekintsük át a boost::type_traits tartalmát.

Típuskategorizálás Itt olyan trait-eket találunk, melyekkel egy típust kategorizálhatunk. Megtudhatjuk, hogy pointer-e, beépített típus-e, stb. Az alábbiakban bemutatjuk az `is_pointer` metafüggvényt:

```
typedef int* const intp;
const bool r1 = is_pointer<intp>::value;
```

Általános típustulajdonságok Ide tartoznak azon trait-ek, melyek eldöntik, egy adott típus konstans-e, POD-típus-e, van-e destruktora, stb. A példában megvizsgáljuk, hogy a fent definiált `intp` típus const típus-e:

```
const bool r2 = is_const<intp>::value;
```

Típusok közötti kapcsolatok Ezen trait-ek segítségével megállapítható, hogy két típus szülő-gyerek viszonyban áll-e egymással, megegyeznek-e, illetve hogy konvertálható-e egyik típus a másikra. Az alábbiakban eldöntjük, hogy a két típus megegyezik-e:

```
const bool r3 = is_same<intp,int*>::value;
```

Típustranszformációk A típusokról nem csak tulajdonságokat tudhatunk meg, hanem újakkal is felruházhatjuk őket, amivel típustranszformációt hajtunk végre. Például hozzáadhatunk vagy törölhetünk egy pointert egy típusból, befolyásolhatjuk, hogy const típus-e vagy nem, stb.

```
typedef add_pointer<intp>::type intpp;
```

11. fejezet

Generikusok más nyelvekben

Tisztáznunk kell a C++ template mechanizmusa és más objektum-orientált nyelvek un. *generic* megoldása közötti különbségeket. A fejezetben áttekintjük az Ada, Java, C#, Eiffel, D és Clean nyelvek generikus megoldásait.

11.1. Ada

Tekintsük tehát az Ada-t. E nyelvben egy generikus[24] interfész a következő formájú:

```
generic
    type T is private;
    with function "<" (x,y: in T) return boolean is <>;
package list_package is
    type list is private;
    procedure push_back (l: in out list; x: in T);
    function empty (l:in list) return boolean;
    procedure sort(l:in out list);
    ...
end list_package;
```

A konstrukció a `generic` kulcsszóval kerül bevezetésre. A `type element is private` kifejezés azt jelenti, hogy `element` egy típusparaméter lesz, mellyel szemben az Ada megkötéseket tesz: kell, hogy legyen értékadás és egyenlőségvizsgálat művelet értelmezve rajta.

Ugyanakkor a `with function "<" (x,y:in T) return boolean is <>;` sor azt is kiköti, hogy a típusnak rendezhetőnek kell lennie, és ez a rendezőfüggvény sablonparaméter lesz, vagy a `T` alapértelmezett kisebb művelete (ez utóbbit a `<>` szintaktikával írtuk le).

Ez a felfogás szöges ellentétben áll a C++-éval, általában ugyanis egy C++ 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. Így csak jóval később derülhet ki, hogy adott típus megfelel-e a vele szemben implicit módon támasztott követelményeknek. Ilyen követelmény példánkban, hogy definiált legyen a típuson egy rendezés, mely a `sort()` függvényhez valószínűleg szükséges lesz.

Egyes programozási nyelvek, pl. az Ada is a fenti probléma megoldására az ún *sablon-szerződés modellt*[25] 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. A C++ és a sablon-szerződés modell kapcsolatát a 12. fejezetben bővebben tárgyaljuk.

Az Ada a C++-hoz és az Eiffel-hez hasonlóan a generic-eket egyfajta gyártási eljárásnak (sablonnak) tekinti. Amikor a paraméterek helyén egy konkrét típus jelenik meg, a fordító a sablon alapján legyárt egy új kódrészletet (eljárást vagy osztályt). Ez a folyamat a példányosítás. Ám a C++-al ellentétben az Ada generic-jeit explicit módon példányosítani kell az első használat előtt. Erre szolgál a `new` kulcsszó:

```
package IntList is new list_package(Integer);
```

Az egész számokat tartalmazó lista típus csak ettől a ponttól válik használhatóvá. Ez a működés is ellentétes a C++ automatikus példányosítási mechanizmusával. E szigorú explicit példányosítási eljárás oka az Ada biztonságra törekvő filozófiájából is ered.

11.2. Java

A Java generic módszere más elveken alapul[5, 20]. 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();
}
```

A Java esetében a típusinformációk elvesznek, és az összes generic argumentum a közös bázisosztályra, Object típusúra konvertálódik. Ezt a módszert nevezzük *type erasure*-nek, típustörlésnek. E módszer a szó szoros értelmében eltörli a behelyezett objektumok típusát, tehát minden típusinformációt elveszítünk. Mivel a generic konstrukció csak a Java 1.5 verziójában került be a nyelvbe, a type erasure lehetővé teszi, hogy a bináris kód szintjén az 1.4-es verziójú kódokkal kompatibilis kódot fordíthasson a compiler. Ugyan e művelet miatt a konténerek maguk lényegében típustalanok, a Java generic mégis garantálja a típusbiztonságot: nem helyezhetünk el Integer objektumokat egy `LinkedList<String>`-ben. Tehát e típustalan viselkedést a generic el tudja fedni előlünk, ezzel csökkentve a hibák elkövetésének kockázatát. Ugyanakkor az 1.5-ös Java megjelenése előtt megírt kódokban a konténerek e típustalanságát kézzel kellett a programozónak

kezelnie, minden esetben a megfelelő típusra konvertálni a konténerből kivett objektumot. Ez kényelmetlen megoldás, és sok hibalehetőséget hordoz magában, hiszen ugyanazt a koncepciót (a tárolt típus neve) többször kell leírnunk a kódban, egymástól független helyeken. A generic elem ezt a terhet leveszi a vállunkról, és a konténer ezt a műveletet maga végzi el.

C++-ban minden példányosítás egy új, teljesen különálló osztályt vagy függvényt hoz létre a megfelelő template-ből. A Java ezzel szemben minden generikusból egyetlen példányt tárol a kódban, és minden példányosítást erre a kódrészletre vezet vissza. Ez a megoldás helytakarékosabb, ám a polimorfikus viselkedés miatt kissé lassabb. Viszont miután a generic nem példányosít, nem áll módunkban specializációt sem írni, így a C++-ban szokásos módon nem is metaprogramozhatunk Javaban. Mindamellett a C++ template metaprogramozás egyes rész-technikáit sikeresen implementálták Java-ban is [48].

Java-ban lehetőségünk van az ún. *wildcard*-ok ("joker" karakterek) használatára[62]:

```
void printCollection(Collection<?> c)
{
    for (Object e : c)
    {
        System.out.println(e);
    }
}
```

A `?` wildcard-dal jelölhetünk egy tetszőleges típust, melyből példánkban egy `Collection`-t hozunk létre és ezt kapja a `printContainer()` függvény paraméterként. Ez hasonló ahhoz a megoldáshoz, amellyel a C++ példányosításkor pattern matching-et hajt végre. Emlékeztetünk a 2.3. fejezetben bemutatott `print_container()` C++ függvényre, melynek paramétere egy tetszőleges típus volt, melyről feltételeztük, hogy egy konténer. Tegyük fel, hogy a `Collection` template definiált, és valamilyen konténert takar. Ekkor Java-s `printCollection()` függvényünk a következő formában írható C++ nyelven:

```
template <class T>
void print_collection(const Collection<T>& c)
{
    ...
}
```

A C++ fordító is tetszőleges típust behelyettesíthet a `T` helyébe és végrehajthatja a példányosítást.

Erőssége a Java generikusoknak, hogy már nyelvi szinten is támogatják a típusparaméterekre történő megkötéseket. Például megkövetelhetjük, hogy a típusargumentum valósítson meg egy interface-t, származzon egy adott típusból, stb.[5]:

```
public class List<T extends Comparable<T>>
{
    public void Sort()
    {
        if (x.compareTo(y) == 0) ...
    }
}
```

A kódban a `generic` fejlécében kikötöttük a `T` típusparaméterről, hogy rendezett kell legyen. Ennek megfelelően a `Sort()` függvényben használhatjuk a `compareTo()` műveletet a két objektum összehasonlítására.

11.3. C#

A C++ és a Java mellett napjaink egyre szélesebb körben használt objektumorientált nyelve a C#. A C# nem használja a type erasure-t, hanem a generic-ek futási idejű példányosítását hajtja végre. Csak úgy, mint a Java esetében, a C# fordító is szintaktikai és szemantikai ellenőrzéseket hajt végre a generic-en, majd azt a kódba helyezi. Amikor futási időben a Common Language Runtime (a futási időt felügyelő rendszer) egy olyan kódrészletet dolgoz fel, amelyben a generikust használják adott típusargumentummal, akkor megtörténik a példányosítás.

Ennek a megoldásnak előnye, hogy a futási időben történő konvertálást elkerülve optimálisabb lesz a futás. Emellett az értékszemantikás beépített típusok (int, bool, stb.) konténerbe helyezésekor nem történik meg a Javaban elengedhetetlen *boxing*, vagyis az objektum referenciaszemantikássá konvertálása. C#-ban is van lehetőségünk explicit kikötéseket megadni a típusokról. A generic szintaktikája igen hasonló a Java-éhoz, azzal a különbséggel, hogy a megkötéseket a `where` kulcsszó előzi meg:

```
public class List<T> where T : IComparable
{
    public void Sort()
    {
        if (x.CompareTo(y) == 0) ...
    }
}
```

11.4. Eiffel

Az objektum-orientált nyelvek között meg kell említenünk az Eiffel nyelvet. Az Eiffel szintén egy kissé eltérő működést mutat mint az előbb említett nyelvek[21]. Itt a `LIST` osztályt a következőképpen tehetjük generikussá:

```
class LIST [T -> COMPARABLE]

feature
    push_back (x: T) is ...
    empty : BOOLEAN is ...
    sort is ...
    ...
end
```

`T` a típusparaméter, mely tetszőleges Eiffel típus lehet. Mivel a nyelvben minden típus implicit módon származik az `any` típusból, a kódban olyan tulajdonságokat használhatunk ki, melyek már ezen ősosztályban rendelkezésre álltak, úgymint értékadás, egyenlőségvizsgálat, `clone, equal, copy`. Mivel az Eiffel a *design by contract* elvein alapul, természetesen van lehetőség egyéb explicit kikötésekre a típusargumentummal szemben, itt, mint a Java

példában is, megköveteljük a típus rendezését. A generikust neve és argumentumai leírásával példányosíthatjuk.

11.5. D

A D nyelv[67] a C++ egyik utódjának tekinthető. A D nyelvi szinten támogatja a fordítási idejű programozást a `static if` (fordítási idejű elágazás), `static assert` (fordítási idejű assert) konstrukciókkal. Ezen konstrukciók már a C++ template metaprogramozási tapasztalatok felhasználásával kerültek be a nyelvbe.

E nyelvben is a `template` a generikusokra való hivatkozás kulcsszava. Az eddig bemutatott nyelvekkel szemben nem korlátozza, milyen nyelvi konstrukció (függvény, osztály, stb.) lehet sablon, gyakorlatilag tetszőleges kódrészletet paraméterezhetünk típussal. Minden ilyen sablonná formált kódrészletet el kell azonban nevezni. Speciálisan, ha a sablonnak egyetlen tagja van, amely egy osztály, és kettejük neve megegyezik, akkor a szintaxis rövidíthető, és csak a típusazonosító nevét kell kiírni.

```
class list(T)
{
    void push_back(T x);
    bool empty();
    void sort();
    ...
}
```

Hasonlóan a C++ eljárásához, az adott template deklarációnak szintaktikusan helyesnek kell lennie, szemantikus ellenőrzés csak példányosításkor történik. A példányosítás a template mögé írt felkiáltójellel és a típusargumentumok felsorolásával kérhető: `list!(int)`

11.6. Clean

Természetesen nem csak objektum-orientált és multiparadigmás nyelvekben létezik generic. Az alábbiakban a 7. fejezetben már bemutatott Clean nyelv generikus szerkezeteit tekintjük át. A Clean az eddig bemutatott nyelvektől eltérő módon vezeti be a generic-fogalmat. Definiáljunk két egyszerű algebrai típust:

```
:: List a = Nil | Cons a (List a)
:: Tree a = Leaf a | Node (Tree a) (Tree a)
```

A `List` típus elemei tehát az üres lista, illetve a `Cons` konstruktorral előállított *head-tail* páros. A lista elemei "`a`" típusúak. A `Tree` elemei falevelek (`Leaf`), illetve elágazások, melyek részfákat tartalmaznak (`Node`). Az "`a`" típusú elemeket a falevelek tartalmazzák.

Látható, hogy a generikusság már a mintaillesztés következménye, hiszen az "`a`" paraméter helyén bármilyen elemi típus állhatna. Tegyük fel azonban, hogy szeretnénk e két adatstruktúrára definiálni a `map` függvényt, mely egyváltozós-egyértékű elemenkénti feldolgozást végez.

```
fmaplist :: (a -> b) (List a) -> (List b)
fmaplist f Nil         = Nil
fmaplist f (Cons x xs) = Cons (f x) : Cons (fmaplist f xs)

fmaptree :: (a -> b) (Tree a) -> (Tree b)
fmaptree f (Leaf a)   = Leaf (f x)
fmaptree f (Node l r) = Node (fmaptree f l) (fmaptree f r)
```

Mivel a két típus struktúrája teljesen eltér, ezért a két függvény implementációja is különböző lesz. Ugyanakkor céljuk – elemenkénti feldolgozás – megegyezik, ezért érdemes lenne a két függvényt egyesíteni. A Clean generikus programozásának alapköveit *prelude*-nak nevezzük, ezek a következőek:

```
:: UNIT       = UNIT
:: PAIR a b   = PAIR a b
:: EITHER a b = LEFT a | RIGHT b
```

E nyelvi konstrukciók segítségével bármilyen struktúra leírható, tekintsük fenti példáinkat ebben az új alakban:

```
:: ListG a :== EITHER UNIT (PAIR a (List a))
:: TreeG a b :== EITHER a (PAIR b (PAIR (Tree a b) (Tree a b)))
```

Mindkét típushoz definiálható egy az adott struktúrát a fenti generikus formára hozó, illetve eredeti formára visszaíró függvény. Ezek esetünkben a `fromList`, `toList`, `fromTree`, `toTree`. A `from..` nevű függvények bizonyos értelemben véve típustörlést hajtanak végre (hasonlóan a Java eljárásához), míg a `to..` alakúak a lefelé történő konverziót hajtják végre.

A generikus elemekkel felírt adatstruktúrák a típustörlés után egységesen kezelhetőek, tehát elég a struktúrák bejárásához egyetlen függvényt írni, generikus `map`-et. A függvény definícióját a `generic` kulcsszó vezeti be:

```
generic map a b :: a -> b
map{|Int|} x                  = x
map{|UNIT|} UNIT              = UNIT
map{|PAIR|} fx fy (PAIR x y)  = PAIR (fx x) (fy y)
map{|EITHER|} fl fr (LEFT x)  = LEFT (fl x)
map{|EITHER|} fl fr (RIGHT x) = RIGHT (fr x)
```

Ezen új, generikus `map` felhasználásával eredeti függvényeink az alábbi formát öltik:

```
fmaplist f x = toList (map f (fromList x))
fmaptree f x = toTree (map f (fromTree x))
```

12. fejezet

A C++ és a sablon-szerződés modell kapcsolata

A 11.1. fejezetben kifejtett sablon-szerződés modell azt a tényt takarja, hogy egy adott nyelv végez-e ellenőrzéseket generikusainak paramétereire nézve, vagyis lehetséges-e explicit kikötéseket tenni az argumentumok tulajdonságait illetően. Ilyen lehet például, hogy létezik-e adott szignatúrájú és nevű tagfüggvénye, vagy definiált-e benne egy bizonyos `typedef`. Mint említettük, a C++ nem alkalmazza a sablon-szerződés modellt. Ez azt jelenti, hogy sok esetben már csak a példányosítás közben derül ki, hogy az argumentum típus valamely tulajdonságára szükségünk lenne, ám a típus azzal nem rendelkezik.

Tegyük fel, hogy a 2.2. fejezetben bemutatott `list` template-ünk `sort()` műveletével kívánjuk a konténer tartalmát nagyság szerinti sorrendbe tenni. Tegyük fel, hogy felhasználói `MyType` típusunkon nincsen értelme a rendezésnek, tehát a kisebb operátor (`operator<`) definiálása nem történik meg:

```
template <class T>
class list
{
    void sort()
    {
        if (x<y) ...
    }
```

```
    bool empty() const
    {
        ...
    }
};

// nem létezik operator<(MyType, MyType)
```

Nyilvánvalóan a `sort()` függvény implementációjában valahol szerepel két elem összehasonlítása a kisebb operátor segítségével. Mivel a template argumentumok tulajdonságaira nézve semmilyen ellenőrzés nem történik, nyugodtan példányosíthatjuk `list`-et a beépített `int`, és saját `MyType` típusunkkal is:

```
(1) list<int> v;
(2) list<MyType> c;
```

Amennyiben ezzel párhuzamosan ugyanezt a kódot Adában is fejlesztenénk, a (2) sorban fordítási hibát kapnánk, hiszen `list` interfészéhez hozzátartozna az az információ, hogy az argumentumtípuson értelmezett rendezésre szükségünk lesz. A C++ azonban eltekint ettől az ellenőrzéstől. A probléma egészen addig nem jelentkezik, amíg nem próbáljuk meg kihasználni a "rendezhetőség" tulajdonságot:

```
if (v.empty()) { ... }
v.sort();

if (c.empty()) { ... }
// c.sort(); // !! nem müködne
```

Mint látjuk, még a `c` konténerre is tetszőleges olyan függvényt (jelen példában `empty()`) meghívhatunk, mely nem használja a rendezést. Akkor kapunk csak fordítási hibát, ha magát a `sort()` függvényt próbálnánk meghívni. Ekkor az `operator<`-et hívó, összehasonlítást végző soron jelezne hibát a fordító.

Hasonló jellegű hiba lép fel, hogyha a Standard Template Library (STL) algoritmusait nem megfelelően használjuk:

```
#include <list>
#include <algorithm>
int main()
{
    std::list<int> v;
    std::sort(v.begin(), v.end());
}
```

Ez az ártalmatlannak tűnő program egy listán próbál rendezést végrehajtani, ám az eredmény egy igen hosszú, és bonyolult fordítási hiba. Sokszor még gyakorlott C++ programozóknak is komoly fejtörést okoz a compiler hibaüzenete, melynek csak egy részletét mutatjuk be:

```
...
stl_algo.h:2360: error: no match for 'operator+' in
'__first + 16'
stl_algo.h: In function 'void std::__insertion_sort
(_RandomAccessIterator, _RandomAccessIterator)
[with _RandomAccessIterator = std::_List_iterator<int>]':
stl_algo.h:2363:   instantiated from
'void std::__final_insertion_sort(_RandomAccessIterator,
_RandomAccessIterator) [with _RandomAccessIterator =
std::_List_iterator<int>]'
stl_algo.h:2714:   instantiated from
'void std::sort(_RandomAccessIterator, _RandomAccessIterator)
[with _RandomAccessIterator = std::_List_iterator<int>]'
sort_error.cpp:6:   instantiated from here
stl_algo.h:2273: error: no match for 'operator+' in
'__first + 1'
stl_algo.h:2363:   instantiated from
'void std::__final_insertion_sort
(_RandomAccessIterator, _RandomAccessIterator)
[with _RandomAccessIterator = std::_List_iterator<int>]'
stl_algo.h:2714:   instantiated from 'void std::sort
(_RandomAccessIterator, _RandomAccessIterator)
[with _RandomAccessIterator = std::_List_iterator<int>]'
sort_error.cpp:6:   instantiated from here
stl_algo.h:2279: error: no match for 'operator+' in '__i + 1'
```

A hibát az okozza, hogy az `std::sort()` függvény elvárja, hogy az argumentumként kapott iterátorok Random Access Iterator-ok (közvetlen elérésű bejárók) legyenek, vagyis tetszőleges elemhez közvetlen hozzáférést biztosítsanak. Az `std::list` iterátora azonban Bidirectional Iterator (kétirányú bejáró), mely nem teljesíti ezt a feltételt, így a művelet fordítási hibát okoz.

Az Ada egy sokkal érthetőbb, világosabb hibaüzenetet fogalmazna meg már a példányosítás kezdetén, mégpedig azt, hogy a generic-nek átadott argumentum valamely tulajdonsága nem felel meg a generic elvárásainak. Ezen a ponton látszik a sablon-szerződés modell vitathatatlan előnye. Vegyük azonban észre, hogy a modell egyben erős megszorítással is jár. Kizárja ugyanis annak lehetőségét, hogy a sablonnak csak azt a részét használjuk, amelynek előfeltételeit az argumentum teljesíti, ilyen volt például `list<MyType>` példányosítása után `empty()` meghívása. Mivel az Ada már magát a példányosítást sem engedélyezte volna, a konténerben még a hibát nem okozó függvényt sem használhattuk volna. A C++ nyelv létrehozásakor Stroustrup szándékosan vetette el a sablon-szerződés modellt[41].

Ugyanakkor a fenti példa is bemutatja, milyen könnyű a template-ek paraméterezésével kapcsolatos hibát véteni. A tapasztalat azt mutatta, hogy C++-ban is sokszor egyszerűbb programozást, jobb karbantarthatóságot, egyszerűbb hibakeresést tenne lehetővé egy, a sablon-szerződés modellt támogató, vagy annak legalább modellezésére törekvő eszköz. A nyelvi szintű megoldás sokáig nem volt lehetséges, hiszen a C++ szabványosított nyelv, a fordítógyártóknak nem ajánlatos saját "nyelvjárásokat" bevezetniük, melyben esetleg egy ilyen nyelvi szintű eszközt valósítottak volna meg. Sokáig tehát csak a könyvtár-szintű megoldás volt az egyetlen lehetséges út.

12.1. Concept checking

A Boost Concept Checking (boost::concept_check[56]) és a Boost Type Traits (ld. 10.3.2. fejezet). programkönyvtárakban olyan C++ konstrukciókat definiáltak a szerzők, melyek a C++ template-ek eszközrendszerét felhasználva, futási idejű költségek nélkül képesek bizonyos limitált ellenőrzésre az argumentumok tulajdonságait illetően, hibás argumentumot észlelve fordítá-

si idejű hibával jelezni azt. Ezt az ellenőrzést *concept checking*-nek nevezzük. A boost::concept_check-ben olyan eszközök állnak rendelkezésre, melyek képesek eldönteni, hogy két osztály szülő-gyerek viszonyban áll-e egymással, egy adott típus pointer-e, stb. Ugyanakkor sok funkcionalitás hiányzik a könyvtárból, például máig sem ismert olyan konstrukció, melynek segítségével eldönthető lenne, hogy egy adott osztálynak létezik-e default (paraméter nélküli) konstruktora. A `list` template-et az alábbiak szerint kiegészítve ki lehetne kötni, hogy a `T` típus rendezhető legyen, ezzel modellezve a sablon-szerződés modellt használó nyelvek konstrukcióit:

```
template <class T>
class list
{
    BOOST_CLASS_REQUIRE(T, boost, LessThanComparableConcept);
    // ...
};
```

A `BOOST_CLASS_REQUIRE` *tag* osztály template-ek törzsében alkalmazható, és jelen esetben a `LessThanComparableConcept` osztálytemplate segítségével képes ellenőrizni, hogy a `T` paraméterben kapott típusnak létezik-e kisebb operátora. Amennyiben igen, a fordítás továbbhalad e sor fölött, ellenkező esetben egy jól értelmezhető mesterséges hibaüzenettel leáll a fordítás (a fordítás leállításáról ld. még 8.2. fejezet).

A boost::concept_check könyvtár bevezetése is hozzájárult ahhoz, hogy a sablon-szerződés modellre, erre a fontos koncepcióra több figyelem irányuljon a C++-os világban. Ennek is köszönhető, hogy a következő C++ szabványban bevezetik a nyelvbe a *concept* fogalmát. Ez a konstrukció képes lesz nyelvi szinten elvégezni a típusok tulajdonságainak ellenőrzését.

12.2. Concept

Mivel a concept checking egyre fontosabb tényezővé vált a template-ek használatának terjedése miatt, a szabványosító bizottság több éves munkába kezdett egy új nyelvi eszköz, a *concept* bevezetése érdekében. Ezen nyelvi elem a következő C++ szabványban bevezetésre kerül. A concept-ek segítségével

a C++ megteremti a lehetőségét, hogy ha kívánjuk (nem kötelezően), definiáljunk egy elvárt interface-t a template-hez. Egy concept-ben leírhatjuk, milyen tulajdonságok (függvények, typedef-ek) meglétét várjuk el egy adott típustól, melyet template argumentumként kívánunk átadni.

Az első (és ezidáig egyetlen) concept-eket implementáló C++ fordító a ConceptGCC[68] fejlesztése 2003-ban kezdődött. A fordító nagy segítséget nyújtott az elgondolások teszteléséhez, finomításához. Sokáig két külön concept javaslat (Dos Reis és Stroustrup[9], illetve Järvi és mások[14]) létezett párhuzamosan, majd ezeket egyesítve született meg a közös elképzelés, mely a mostani szabványjavaslat alapját is adja[13].

A concept rendszer három alapelemből áll össze: a concept-ben írhatjuk le a típussal szemben támasztott követelményeket, a `requires` kulcsszó segítségével egy template megszabhatja, milyen concept-et kielégítő típusokat vár el, míg a *concept_map*-ek segítségével leírhatjuk, egy adott típus hogyan valósít meg egy concept-et (ha ez nem triviális). Azt mondjuk hogy egy típus megvalósít (*model*) egy concept-et, ha követelményeinek eleget tesz. Egy concept finomíthat egy másikat (*refine*) ha minden olyan tulajdonságot elvár amelyet az eredeti.

Maga a concept függvényszignatúrák (*function signature*) és típusnevek (*associated type*) listája, melyeket a concept-nek megfelelni akaró típusnak tartalmaznia kell. Az alábbi concept azt követeli meg, hogy az egyenlőség operátor (`operator==`) definiált legyen `T` és `U` típusok fölött:

```
concept EqualityComparable<typename T, typename U = T>
{
    bool operator==(T a, U b);
};
```

Minden olyan típus melynek létezik egyenlőség operátora (akár globális-, akár tagfüggvényként) megvalósítja ezt a concept-et. Az előző concept-et finomíthatjuk, új tulajdonságokat adhatunk hozzá. Például elvárhatjuk, hogy az egyenlőség operátor mellett rendezés is definiált legyen a típuson:

```
concept Comparable<typename T, typename U = T>
    : EqualityComparable<T,U>
{
    bool operator<(T a, U b);
};
```

Az `auto` kulcsszó használatával kissé módosíthatunk a concept viselkedésén:

```
auto concept EqualityComparable<typename T, typename U = T>
{
    bool operator==(T a, U b);
};
```

Az előző két esetben explicit módon deklarálnunk kell, mi módon valósítja meg típusunk a követelményeket, míg az `auto` kulcsszóval ellátott concept-ek esetében nem. E kulcsszó ugyanis azt írja le, hogy a concept próbáljon meg *szintaktikus* egyezést keresni a függvényszignatúra és a típus között. Tehát amennyiben típusunknak valóban létezik egyenlőség és kisebb operátora is, a concept ezeket automatikusan alkalmazza, míg az első két példa esetében ez további deklarációkat igényel. Az `auto` kulcsszót nélkülöző concept-ek esetében ugyanis a concept_map nyelvi konstrukció segítségével *szemantikus* kapcsolatot kell meghatároznunk a concept és a típus között. Tehát ha típusunknak nem létezik egyenlőség operátora, de mi mégis definiálni tudunk valamilyen egyenlőségfogalmat, akkor ezt a concept_map segítségével leírhatjuk, és típusunk meg fog felelni az `EqualityComparable` concept-nek. Tekintsük az alábbi példát:

```
struct Person
{
    int id;
    string name;
    typedef int ImportantType;
};
```

A `Person` típusnak nincs egyenlőség operátora, de két ilyen objektumot tekinthetünk egyenlőnek, ha `id`-jük megegyezik. Az alábbi concept_map segítségével írhatjuk le ezt a szemantikus egyenlőséget:

```
concept_map EqualityComparable<Person>
{
    bool operator==(Person a, Person b)
    {
        return a.id == b.id;
    }
};
```

Ha egy típus szintaktikusan kielégít egy concept-et (pl. az int típusnak biztosan van egyenlőség operátora), és nem használtuk az auto kulcsszót, akkor is leírhatjuk a szintaktikus kapcsolatot:

```
concept_map EqualityComparable<int>
{
}
```

Mint említettük, lehetséges követelményeket tenni azzal kapcsolatban, hogy a típus milyen belső típusokkal, illetve typedef-ekkel rendelkezzen. Az alábbi concept-ben meghatározzuk, hogy az annak megfelelni kívánó típusnak legyen egy `ImportantType` nevű belső típusneve.

```
auto concept ImportantTypeConcept<typename T>
{
    typename ImportantType = T::ImportantType;
};
```

Következő példánkban bemutatjuk, hogyan írhatunk concept-eket felhasználó, bizonyos típustulajdonságokat megkövetelő, ún. megszorított *constrained* függvény template-et. Ezt a `requires` kulcsszó segítségével tehetjük meg:

```
template<typename T>
requires EqualityComparable<T>
void f(const T& t1, const T& t2)
{
    ... if(t1 == t2) ...
}
```

Tehát az `f()` függvény csak olyan típusokkal példányosítható, melyek kielégítik az `EqualityComparable` concept követelményeit. Ugyanez a viselkedés elérhető egy egyszerűbb, rövidebb szintaktikával is:

```
template<EqualityComparable T>
void f(const T& t1, const T& t2)
{
    ... if(t1 == t2) ...
}
```

Összefoglalva tehát a concept-ek haszna a fejezetben ismertetett példák esetén mutatkozik meg, amikor pl. egy nem rendezhető típust adunk át a `sort()` függvény template-nek. Ugyanakkor a concept-ek felhasználhatóak a metaprogramok biztonságosabbá tételéhez is, mivel segítségükkel a metaprogramok fölött típusrendszert definiálhatunk. A concept-nélküli C++-ban a template metaprogramok túlnyomó többsége típustalan, hiszen pl. egy `template <class T>` fejlécű template esetében `T` helyébe tetszőleges C++ típust írhatunk, esetleg tévedésből. Mint láttuk, a concept fordítási időben pontosan olyan tulajdonság-ellenőrzéseket végez, mint egy absztrakciós szinttel lejjebb a C++ fordító a futási idejű programok entitásain. Egy bonyolult metaprogram esetében egy egyszerű elírás is igen bonyolult fordítási hibákhoz vezethet, míg egy ugyanilyen egyszerű elírás futási idejű programok esetén érthető fordítási hibaüzenethez vezetne. Amennyiben maguk a template paraméterek is valamilyen C++ típussal leírhatóak, mint például a `Factorial` metaprogram esetében, ahol egy int paramétert adtunk át, addig nyilvánvalóan nem történhet probléma, hiszen amennyiben nem egy egész számot adunk át a template-nek, egyszerű fordítási hibát kapunk. A probléma akkor következik be, ha például a 10.3.1. fejezetben bemutatott `IndexOf` *typelist* algoritmust helytelenül használjuk. Tekintsük a következő példát:

```
template <class TList, class T> struct IndexOf;

template <class TList>
struct P
{
    enum { value = IndexOf<TList, int>::value };
};
```

```
typedef TYPELIST_3(bool,int,double) TL;
typedef int UL;

const int r = P<UL>::value;
```

Tegyük fel, hogy a `P` metaprogram egy egyszerű elírás miatt `TL` helyett az `UL` típust kapja paraméterként. Mivel `TList` tetszőleges típusparaméter lehet, ezen a ponton nem kapunk semmilyen hibát, ám az argumentum az elvárt típuslista helyett egy int lesz. Gondot okoz, hogy a probléma nem is derül ki még az `IndexOf` hívás helyén sem, hiszen ez a metaprogram sem végez ellenőrzéseket argumentumaira vonatkozóan. Így az eredmény egy hosszú és bonyolult hibaüzenet lesz. Ugyanakkor concept-ek használatával a következőképpen nézhetne ki a kód:

```
auto concept TypeListConcept<typename T>
{
    typename Head = T::Head;
    typename Tail = T::Tail;
};

template <TypeListConcept TL, class T> struct IndexOf;

template <TypeListConcept TList>
struct P
{
    enum { value = IndexOf<TList, int>::value };
};

typedef TYPELIST_3(bool,int,double) TL;
typedef int UL;

const int r = P<UL>::value;
```

Ebben az esetben már `P` példányosításának kezdetén egy egyszerű hibaüzenetet kapunk, ha nem megfelelő paramétert adunk át.

A concept jelentős újítást fog jelenteni a nyelv logikájában, és egy igen erős eszközt ad a programozó kezébe. A fent leírtakon kívül hasznos lehet az ún. *axiom*-ok használata:

```
concept LessThanComparable<typename T>
{
    bool operator<(T, T);
    axiom Irreflexivity(T x) { !(x < x); }
    axiom Transitivity(T x, T y, T z)
    {
        if (x < y && y < z) x < z;
    }
}
```

Mint látható, az `axiom` segítségével a típusargumentum szemantikájára is megfogalmazhatnánk különféle állításokat, melyek nemteljesülése esetén szintén hibát jelezne a fordító. Sajnos ez a funkció jelen sorok írásakor nem működőképes a tesztfordítóban. A *szemantikus concept-ek* ugyanakkor érdekes és fontos kutatási terület lehet a jövőben, hiszen segítségükkel pl. fordítási időben megbecsülhetnénk egy program algoritmikus bonyolultságát.

Kutatásaink során elemeztük a concept-ek és concept_map-ek modularizációs problémáit[43]. Tegyük fel, hogy szeretnénk naplózni egy template argumentum `f()` függvényhívásai előtt és után (a feladat hasonló az aspektus-orientált programozás egyik klasszikus példájához). A fentebb definiált `Person` típus pedig egy concept_map segítségével írja le a konkrét naplózási tevékenységet, mely során az objektum `name` mezőjét egy file-ba írja:

```
concept LogConcept<typename C>
{
   void f(C&);
}

concept_map LogConcept<Person>
{
   void f(MyType& c)
   {
      // file nyitása, név kiírása, file bezárása, hibakezelés
      c.f();
      // file nyitása, név kiírása, file bezárása, hibakezelés
   }
};
```

A programkód több tervezési problémát rejt. Egyfelől a sok naplózáshoz kapcsolódó utasítás között elveszik az eredeti `f()` függvény hívása, holott az központi szerepet tölt be a koncepciónk megfogalmazásakor. Másfelől a kód igen redundáns, hiszen pontosan ugyanazokat a lépéseket hajtjuk végre a két naplózási fázisban. E két probléma nehezen karbantartható kódot eredményez, és e jelenségekre az aktuális concept tervezet nem is ad kellő megoldásokat. Cikkünkben javaslatot tettünk a class nyelvi elem működésének lemásolására, és a private-protected-public láthatósági köröket leíró hármas bevezetésére. A naplózási kódrészletet egy külön private függvénybe téve sokkal jobb minőségű kódot kapunk:

```
concept LogConcept<typename C>
{
   void f(C&);
}

concept_map LogConcept<Person>
{
public:
   void f(MyType& c)
   {
      log();
      c.f();
      log();
   }

private:
   void log()
   {
      // file nyitása, név kiírása, file bezárása, hibakezelés
   }
};
```

E megoldás a gyakorló C++ programozók számára könnyen érthető, hiszen e logikával nap mint nap találkoznak az osztályok kapcsán. Az ötlet megvalósíthatóságának vizsgálatához implementáltunk egy előfordítót, mely a segédfüggvényt egy egyedi névtérbe helyezi át, majd a hívási helyeket e függvény hívásával helyettesíti. A részletek a [43] cikkben találhatóak.

Összefoglalás

Bevezetés

A *C++ Template Metaprogramozás* (TMP) a C++ programozási nyelv generikus elemére, a *template*-re épül. A TMP felhasználási lehetőségei közé tartozik többek között: számításigényes műveletek fordítási időben való elvégzése, fordítási idejű döntések, optimálisabb, gyorsabb kód generáltatása a fordítóval, fordítási idejű kódtranszformáció, reflection, stb.

Az első metaprogram óta a TMP kifejezőereje nem várt új távlatokat nyitott. A mai napig a template-ekkel kapcsolatos problémák képezik a C++ nyelv legaktuálisabb kutatási témáit, a metaprogramozásban rejlő lehetőségeket még csak most kezdjük megérteni. A TMP ma már önálló programozási paradigma, melyet már nem csak kutatási célokra, hanem ipari szoftvertermékek elkészítése során is felhasználnak.

Ugyanakkor a template metaprogramozás jelenleg minden előnye ellenére sem elterjedt paradigma. Ennek okai elsősorban a kódolási sztenderdek és a fejlesztést segítő szoftvereszközök hiányában keresendőek. A template metaprogramozásban még nem alakultak ki széles körben elfogadott programozási módszertanok, eszközök, könyvtárak, amelyek a metaprogramok fejlesztését támogatnák. Az egyes megoldások gyakran ad hoc jellegűek, a hibajavítások heurisztikusak, ezáltal a projekt ráfordítások, fejlesztési költségek nehezen becsülhetőek. Általános probléma a TMP-vel kapcsolatban a körülményes szintaktika és a hosszú kódok, melyek eredményeképpen nehezebben írható és karbantartható forráskódok születnek. Ezzel egyidejűleg a metaprogram kódok fordítási hatékonysága igen alacsony, hiszen a példányosítási láncok

végrehajtása komoly plusz terheket ró a fordítóprogramokra, melyeket nem ilyen jellegű munkák elvégzésére terveztek. A bonyolult szintaktika rövid úton programozói hibákhoz vezet, ezért még nagy körültekintés és megfelelő metodika alkalmazása mellett is igen nehéz TMP-ben programozni, ugyanakkor a hibakeresés (debuggolás) is igen körülményes.

A dolgozat célkitűzése

A dolgozatban a template metaprogramozás paradigmáját tárgyaltam. Vizsgáltam a paradigma elterjedtségét, a paradigma felhasználásával írt programok szintaktikáját, szemantikáját, illetve az ezekből fakadó minőségi paramétereket. Megoldási javaslatokat adtam ezen paraméterek javítására. Bemutattam már meglévő metaprogramozási könyvtárakat, megoldásokat, illetve saját kutatásaim alapján új módszereket, eszközöket javasoltam a hatékonyabb metaprogram-fejlesztéshez.

Tézisek

I. Tézis – Az ISO/IEC 14882 (1998) C++ szabvány által meghatározott jólformáltsági kritériumokat kiterjesztettem fordítási időben végrehajtott C++ template metaprogramokra.
Amennyiben megoldásokat kívánunk adni bizonyos template metaprogramokkal kapcsolatos problémákra – jelen esetben hibás metaprogramok javítására –, akkor pontosan definiálnunk kell, milyen eseményeket tekintünk hibának, illetve normális működésnek. A témában végzett kutatásom arra irányult, hogy a C++ szabvány fogalomrendszerét metaprogramokra átültetve elkülönítsük a hibás és nem hibás eseteket. A dolgozatban definíciót adtam a *jólformált* és a *rosszul formált* metaprogramok fogalmára, párhuzamba állítottam a *futási idejű programok* és a *metaprogramok* lehetséges hibatípusait. A helyességen kívül a fejlesztés során fontos egyéb szempontok szerint is elemeztem a metaprogramokat: áttekintettem, milyen hibákat okozhat, ha a

metaprogram futása közben a fordítóprogram felhasználja összes erőforrását, illetve ha nem hordozható kódot írunk. Megvizsgáltam, hogy a különféle template metaprogram hibák milyen következményekkel járhatnak, illetve ezen hibatípusok hogyan függnek össze a template metaprogramozás azon tulajdonságaival, melyek következtében funkcionális és interpreter nyelvként tekinthetünk rá.

A témával foglalkozó publikációim: [28, 38]

II. Tézis – A template metaprogramok felett definiáltam egy helyettesítő funkcionális absztrakciós nyelvet, az ECLEAN-t. Megadtam az ECLEAN átírási mechanizmusát az ISO/IEC 14882 (1998) szerinti C++ nyelvre. Létrehoztam egy kísérleti transzlátort a fenti átírásra és elemeztem a keletkeztett kód hatékonysági paramétereit.

Az egyik legkomolyabb hibalehetőség abban rejlik, hogy a TMP szintaxisa igen bonyolult. Még egyszerűbb algoritmusokat, koncepciókat is csak nehezen fejezhetünk ki benne, hosszú kóddal és szintaktikai buktatókkal. Ugyanakkor a jelenlegi könyvtárak és egyéb kódok kevéssé veszik figyelembe azt a tényt, hogy a TMP tiszta funkcionális nyelvnek tekinthető, és ezért érdemes funkcionális logikával megközelíteni a programozását. A probléma megoldása érdekében definiáltam az EClean nyelvet, mely a *Clean* tisztán funkcionális, lusta kiértékelésű nyelv egy szintaktikai és szemantikai részhalmaza. Kidolgoztam egy előfordítót, mely képes egy EClean programot speciális template kóddá alakítani, mely C++ fordítási időben már hagyományos metaprogramként fut. Az általam javasolt rendszer egy letisztult, funkcionális jellegű beágyazott interface-t ad a programozó kezébe, elrejtve a szintaktikai részleteket. Az Erathosztenészi szita algoritmus egy lusta kiértékelést használó változatán bemutattam, hogy a javasolt rendszer által generált kód hatékonysága megegyezik egy, a hagyományos módon készített metaprograméval.

A témával foglalkozó publikációim: [37, 39, 40]

III. Tézis – Meghatároztam a C++ template metaprogramok hibakeresésének feltételeit. Hibakeresési célokra módosítottam a nyílt forráskódú g++ fordítóprogramot. Részt vettem a Templight template metaprogram debugger rendszer kidolgozásában és tesztelésében.

A metaprogramozás során komoly problémát jelent, hogy egyrészről a lehetséges hibák ellen nehezen tudunk védekezni, másrészről a már meglévő hibákat nagyon nehézkes felderítenünk. Ez a két tényező negatív hatással van a fejlesztés hatékonyságára. A hibaprevenció kapcsán bemutattam a metaprogramozásban sokszor használt *static assert* konstrukciót, mellyel különféle feltételek teljesülését ellenőrizhetjük le fordítási időben, és értelmes üzenetet generálhatunk hiba esetén. Javaslatot tettem a konstrukció egy lehetséges módosítására is, mely segítségével informatívabb hibaüzeneteket generálhatunk. A hibakeresés megkönnyítése érdekében módosítottam a g++ compilert, hogy az fordítási időben információt szolgáltasson a példányosuló template-ekről, így könnyebben nyomon kísérhetjük a példányosítási láncot vagyis a metaprogram futását. Az előző ötlettel analóg módon elérhető, hogy a template példányosításokat, típusargumentumokat, belső typedef-eket, stb. a fordítási folyamat közben egy *trace file*-ba írhassuk, majd ezt a megfelelő formátumú trace file-t elemzési célokra felhasználhassuk. Javaslatot tettem az ezen feladatot megoldó *Templight* nevű template metaprogram debugger rendszer használatára.

A témával foglalkozó publikációim: [28]

További kutatási lehetőségek

A Templight debugger jelenleg csak *post-mortem* debugging-ra képes, tehát a már lefutott metaprogram példányosítási láncát tudjuk nyomon követni. Nagy segítséget nyújtana, ha a hagyományos debuggerekhez hasonlóan már (TMP) futási időben be tudnánk avatkozni, változókat módosítani, stb.

A hibaprevenció témájában fontos kutatási terület lenne a template metaprogramok szerződés alapú (contract-based) fejlesztése, melynek elősegítésé-

hez a dolgozat eszközöket szolgáltat. Hasonlóan érdekes kutatási irány lenne az altípusjeles típusok metaprogramokra alkalmazhatóságának vizsgálata.

A hibakeresés slicing alapra helyezése szintén érdekesnek tűnik, hiszen 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. Mivel a metaprogramok futási és fordítási ideje egybeesik, a statikus slicing megegyezne a dinamikus slicing-gal.

Fontos lenne a metaprogramok kódbonyolultságának és pszichológiai bonyolultságának vizsgálata is, hogy a fejlesztésre fordítandó erőforrások jobban becsülhetőek legyenek.

Érdemes lenne az EClean rendszer interface-ét további funkcionális nyelvekre kiterjeszteni, elsősorban Haskell-re. Fontos eredmény lenne az EClean és a template rendszer közötti callback mechanizmus, melynek segítségével már EClean előfordítási időben tudnánk manipulálni C++ típusokat, függvényeket.

A. függelék

Az EClean nyelvtana

```
compilation_unit ::= (meta_part)+ EOF
meta_part ::= meta_begin (EOL)* (main_expr)* meta_end (EOL)*
main_expr ::=
  (start_expression| function_declaration |
  function_path_definition) EOL
start_expression ::= "Start" ASSIGN expression
function_declaration ::= name SCOPE signature ARROW type
signature ::= (type)+
name ::= ID
type ::= list | "Int"
list ::= LBRACKET type RBRACKET
function_path_definition ::=
  name parameter_list
  (
    IF expression ASSIGN expression EOL* ASSIGN expression |
    ASSIGN expression
  )
parameter_list ::= (atomic | list_pattern)+
list_pattern ::=
  LBRACKET
  (
    RBRACKET |
    expression ((COMMA | COLON) expression)* RBRACKET |
    expression THRU RBRACKET |
    expression THRU expression RBRACKET
  )
```

```
argument_list ::= (argument)+
atomic ::= constant | variable
argument ::= atomic | LPAREN expression RPAREN | list_pattern
constant ::= numeric_constant
variable ::= ID
numeric_constant ::= INT_CONSTANT
expression ::= equalityExpr (EQUALS equalityExpr)?
equalityExpr ::= orExpr (OR orExpr)*
orExpr ::= andExpr (AND andExpr)*
andExpr ::= plusMinusExpr ((PLUS | MINUS) plusMinusExpr)*
plusMinusExpr ::= mulDivExpr ((ASTERISK | PER) mulDivExpr)*
mulDivExpr ::= remDivExpr ((REM | DIV) remDivExpr)?
remDivExpr ::= expression_element | LPAREN equalityExpr RPAREN
expression_element ::= function_call_or_atomic | list_pattern
function_call_or_atomic ::= function_call | atomic
function_call ::= name argument_list
meta_begin ::= BACKAPOS
meta_end ::= BACKAPOS

COLON: ':'          COMMA: ','
SCOPE: "::"         LT: '<'
BACKAPOS:"`"        GT: '>'
APOS:"'"            ID: ('a'..'z' | 'A'..'Z' | '_')
LPAREN: '('             ('a'..'z' | 'A'..'Z' | '0'..'9'| '_')*
RPAREN: ')'         INT_CONSTANT: '0' | ('1'..'9')('0'..'9')*
LCURLY: '{'         WS : (' ' | '\t' | '\f')+
RCURLY: '}'         EOL :(: "\r\n" | '\r' | '\n' ))+
LBRACKET: '['       DOT: '.';
RBRACKET: ']'
ARROW: "->"
EQUALS: "=="
ASSIGN : '='
OR: "||"
AND: "&&"
SEMICOLON: ';'
ASTERISK: '*'
PLUS: '+'
THRU: ".."
MINUS: '-'
PER: '/'
IF: '|'
```

B. függelék

Summary

In the present thesis work the template metaprogramming paradigm is discussed. The goal of the thesis was to provide new methods and tools for more effective development of metaprograms. To fulfill the desired goal I have done research and provided results in the following main areas.

I have generalized the notion of a well-formed C++ program defined by the ISO/IEC 14882 (1998) C++ standard, extending this definition to template metaprograms executed at compilation time.

I have defined EClean, an abstract functional language over template metaprograms. I have defined the transcription mechanism of EClean for the ISO/IEC 14882 (1998) C++ language, implemented an experimental translator that carries out the transcription, and I have analyzed the efficiency of the generated code.

I have defined the conditions for debugging C++ template metaprograms. For debugging purposes I have modified the open source g++ compiler, and have taken part in the research, implementation, and testing of the Templight template metaprogram debugger.

To summarize, I have provided tools for preventing errors during development of metaprograms, creating maintainable code, and debugging metaprograms in order to assist C++ programmers in TMP development.

Irodalomjegyzék

[1] D. Abrahams, A. Gurtovoy: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, 2004

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

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

[4] C. Böhm, G. Jacopini: *Flow diagrams, Turing Machines and Languages with only Two Formation Rules*, Comm. of the ACM, 9(5): pp.366-371, 1966

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

[6] J. O. Coplien: *Multi-Paradigm Design for C++*, Addison-Wesley, 1998

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

[8] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. Veldhuizen: *Generative Programming and Active Libraries*, Springer-Verlag, 2000

[9] G. Dos Reis, B. Stroustrup: *Specifying C++ concepts*, POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on

Principles of Programming Languages, pp.295-308, ACM Press, USA, 2006

[10] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[11] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, J. Willcock: *A Comparative Study of Language Support for Generic Programming*, In Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications pp.115-134, USA, 2003

[12] Y. Gil, K Lenz: *Eliminating impedance mismatch in C++*, In Proceedings of the 33rd international conference on Very large data bases table of contents, pp.1386-1389, Austria, 2007

[13] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine: *Concepts: linguistic support for generic programming in C++*, Proc. of the 2006 OOPSLA Conference, pp.291-310, 2006

[14] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek: *Algorithm specialization in generic programming: Challenges of constrained generics in C++*, PLDI '06: Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation, pp.272-282, ACM Press, USA, 2006

[15] Z. Juhász, Á. Sipos, Z. Porkoláb: *Implementation of a Finite State Machine with Active Libraries in C++*, In Lammel et al ed.: GTTSE 2007, LNCS 5235, pp.474-488, Generative and Transformational Techniques in Software Engineering (GTTSE), Braga, 2008

[16] G. Kiczales: *Aspect-Oriented Programming*, AOP Computing surveys 28(es), 154-p, 1996

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: *An overview of AspectJ*, LNCS Vol.2072, pp.327-353, Springer-Verlag, ECOOP, Budapest, 2001

[18] P. Koopman, R. Plasmeijer, M. van Eekelen, S. Smetsers: *Functional programming in Clean*, 2002

[19] S. Kothari, M. Sulzmann: *C++ templates/traits versus haskell type classes*, Technical Report TRB2/05, The National University of Singapore, 2005

[20] D. Marshall: *Programming Microsoft Visual C# 2005: The Language*, Microsoft Press, 2006

[21] B. Meyer: *Eiffel: The Language*, Prentice Hall, 1991

[22] S. Meyers: *Effective STL*, Addison-Wesley, 2001

[23] D. Musser, A. Stepanov: *Algorithm-oriented Generic Libraries*, Software-practice and experience, 27(7), pp.623-642, 1994

[24] Nyékyné Gaizler J. (szerk). et al:
Az Ada 95 programozási nyelv Egyetemi tankönyv, Budapest, ELTE Eötvös Kiadó, 1999

[25] Nyékyné Gaizler J. (szerk) et al: *Programozási nyelvek*, Kiskapu Kft, 2003

[26] N. Pataki, T. Kozsik, Z. Porkoláb: *Properties of C++ Template Metaprograms*, Proceedings of the 7th International Conference on Applied Informatics (ICAI), Eger, 2007

[27] R. Plasmeijer, M. van Eekelen: *Clean Language Report*, 2001

[28] Z. Porkoláb, J. Mihalicza, Á. Sipos: *Debugging C++ Template Metaprograms*, In The ACM Digital Library, pp. 255-264, GPCE, Portland, USA, 2006

[29] Z. Porkoláb, J. Mihalicza, Á. Sipos, N. Pataki: *Templight, a Template Metaprogram Debugger*, demo, ECOOP, Berlin, Germany, 2007

[30] Z. Porkoláb, J. Mihalicza, Á. Sipos, N. Pataki: *Templight, a Template Metaprogram Debugger*, poszter, ECOOP, Berlin, Germany, 2007

[31] Z. Porkoláb, J. Mihalicza, N. Pataki, Á. Sipos: *Towards Profiling C++ Template Metaprograms*, In Horváth et al ed.: Proceedings of SPLST 2007, pp. 96-111, Symposium on Programming Languages and Software Tools, Dobogókő, 2007

[32] Z. Porkoláb, V. Zsók: *Teaching Multiparadigm Programming Based on Object-Oriented Programming*, 10th Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts, ECOOP, Nantes, 2006

[33] J. Siek: *A language for Generic Programming*, doktori értekezés, Indiana University, 2005

[34] J. Siek, W. Taha: *A Semantic Analysis of C++ Templates*, In ECOOP 2006: European Conference on Object-Oriented Programming, Nantes, France, July 2006.

[35] C. Simonyi, M. Christerson, S. Clifford: *Intentional software*, In Proc. of the 21st annual ACM SIGPLAN OOPSLA, pp.451-464, 2006

[36] Sipos Á.: *Hatékony metaprogramozás*, TDK dolgozat, 2004

[37] Á. Sipos, N. Pataki, Z. Porkoláb: *Lazy Data Types in C++ Template Metaprograms*, ECOOP-MPOOL Workshop, Berlin, 2007

[38] Á. Sipos, Z. Porkoláb, I. Zólyomi: *On the Correctness of Template Metaprograms*, In Kovács et al ed.: Proceedings of 7th International Conference on Applied Informatics, pp. 301-308, The 7th International Conference on Applied Informatics (ICAI), Eger, 2007

[39] Á. Sipos, Z. Porkoláb, V. Zsók: *Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms*, Studia Universitatis Babes-Bolyai Informatica Vol LIII/2, pp.55-66, 2008

[40] Á. Sipos, V. Zsók: *EClean - An Embedded Functional Language*, In Porkoláb et al ed.: Workshop on Generative Technologies Draft Proceedings, pp. 1-12, Workshop on Generative Technologies (WGT), Budapest, 2008

[41] B. Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, 1994

[42] B. Stroustrup: *The C++ Programming Language*, 3rd ed., Addison-Wesley, 1997

[43] Z. Szűgyi, Á. Sipos, Z. Porkoláb: *Towards the Modularization of C++ Concept Maps*, In Porkoláb et al ed.: Workshop on Generative Technologies Draft Proceedings, pp. 33-43, Workshop on Generative Technologies (WGT), Budapest, 2008

[44] D. Vandevoorde, N. M. Josuttis: *C++ Templates: The Complete Guide*, Addison-Wesley, 2002

[45] T. Veldhuizen: *Template Metaprograms*, `http://osl.iu.edu/~tveldhui/papers/Template-Metaprograms`

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

[47] T. Veldhuizen: *Five compilation models for C++ templates*, First Workshop on C++ Template Programming, Erfurt, 2000

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

[49] T. Veldhuizen: *Expression Templates*, Lippman ed: C++ Gems (SIGS Books and Multimedia, 1996), pp. 475-488

[50] Wettl F., Mayer Gy., Sudár Cs.: *Latex kezdőknek és haladóknak*, Panem Kft, 1998

[51] A. van Wijngaarden: *The Generative Power of Two-Level Grammars*, LNCS Vol.14, pp.9-16, Springer-Verlag, 1974

[52] Z. Yao, Q. Zheng, G. Chen: *AOP++: A Generic Aspect-Oriented Programming Framework in C++*, LNCS Vol.3676, pp.94-108, Springer-Verlag, 2005

[53] Zólyomi I.: *Az objektumcsaládok polimorfizmusa*, diploma dolgozat, 2003

[54] I. Zólyomi, Z. Porkoláb: *Towards a template introspection library*, LNCS Vol.3286 pp.266-282, Springer-Verlag, 2004

[55] ISO/IEC 14882 International Standard. Programming languages: C++. American National Standards Institute, September 1998.

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

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

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

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

[60] Blitz++ könyvtár, `http://www.blitz.org`

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

[62] Java generikusok, `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

[63] ANTLR parser generátor, `http://www.antlr.org/`

[64] XML, `http://www.w3.org/XML/`

[65] OpenC++, `http://opencpp.com/`

[66] YACC parser generátor, `http://dinosaur.compilertools.net/`

[67] D nyelv, `http://www.digitalmars.com/d/`

[68] ConceptGCC, `http://www.generic-programming.org/software/ConceptGCC/`

[69] GNU debugger, `http://www.gnu.org/software/gdb/`

[70] AspectC++, `http://www.aspectc.org/`

[71] http://aszt.inf.elte.hu/~gsd/halado_cpp/

[72] Ada Reference Manual,
http://www.adaic.org/standards/05rm/RM-Final.pdf

[73] SourceForge, http://sourceforge.net

[74] Comeau fordító, http://comeaucomputing.com/