A C++ template metaprogramozás gyakorlati kérdései

Porkoláb Zoltán
gsd@elte.hu

Habilitációs tézisfüzet
2020

Köszönetnyilvánítás

Ezen dolgozat eredményeinek nagy részét hallgatóimmal, doktoranduszaimmal végzett közös kutatások során értük el. Ezennel szeretném megköszönni kollégáimnak, egykori diákjaimnak, szerzőtársaimnak a folyamatos kritikus megközelítést, együttgondolkodást, közös munkát.

Végül szeretnék köszönetet mondani családomnak a folyamatos támogatásukért.

A dolgozat elkészültét az „Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein" (EFOP-3.6.3-VEKOP-16-2017-00002) című projekt támogatta. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.

Tartalomjegyzék

1. fejezet

Bevezetés

A függvény- és osztálysablonok (template-ek) a C++ programozási nyelv alapvető fontosságú elemei, a parametrikus polimorfizmus alapkövei. Segítségükkel adatszerkezeteket és algoritmusokat tudunk típusokkal paraméterezni, ezáltal megragadhatjuk az absztrakciós hasonlóságokat anélkül, hogy futási időben extra költségeket kellene ezért fizetnünk. A template-ek intenzív használatára épülő Standard Template Library (STL) – a C++ szabványkönyvtár része – olyan újrafelhasználható eszközöket biztosít a fejlesztők számára, melyek a programozási hatékonyságot is nagyban segítik.

A C++-ban ahhoz, hogy a template-eket használjuk, ún. példányosítás (instntiation) szükséges. A példányosítás történhet implicit módon a fordítóprogram által, amikor egy template-re valamely új argumentummal hivatkozunk; vagy a programozó által explicit módon is. A példányosítás során a fordító új kódot hoz létre, lényegében behelyettesítve az aktuális argumentumokat a template paraméterek helyébe.

Ez a mechanizmus azzal a meglepő eredménnyel jár, hogy megfelelően definiált template-ek segítségével rákényszeríthetjük a szabványos C++ fordítót, hogy általunk definiált algoritmusokat hajtson végre fordítási időben. A jelenséget először Erwin Unruh demonstrálta 1994-ben, majd hamarosan belátták a C++ sablonok Turing-teljes voltát. Az ilyen módszerrel készített template metaprogramokat ma már széles körben használják a C++ programozási gyakorlatban.

Ugyanakkor a klasszikus programfejlesztési modellek és eszközök nehezen, vagy egyáltalán nem alkalmazhatóak a C++ template metaprogramokra. Kutatásaim kezdetén tisztázatlan volt, hogy milyen körben alkalmazhatóak hatékonyan a metaprogramok, melyek a gyakorlati korlátok, milyen programozási paradigmát érdemes követni, alig volt alkalmazói könyvtár és szinte teljesen hiányzott a template metaprogramok fejlesztését támogató eszközrendszer is.

Dolgozatomban összefoglalom a PhD fokozat megszerzését követően a fent leírt témakörben végzett kutatásaimat. A bemutatott eredmények többségét doktorandusz és MSc hallgatóimmal közös kutatásokban értük el, azok többszerzős cikkekben jelentek meg, így nem lehet azokat kizárólag egyetlen személyhez kötni. A bemutatás során azon eredményeket fogom részletezni, melyeket fontosnak és lényegi hozzájárulásnak tekintek a témához, és melyek létrejöttében jelentős szerepem volt.

A dolgozat szerkezete a következő. E bevezető fejezet után a 2. fejezetben ismertetem a **C++ template metaprogramozás elveit**, az eszközrendszerét és fejlődési irányait. A 3. fejezetben a **szakmaterület specifikus nyelvek** (domain specific languages, DSL) C++ gazdanyelvbe template metaprogramok segítségével történő **beágyazását** tárgyalom, annak szükséges eszközrendszerével. Az itt elért eredmények segítségével hatékonyabban és szabadabb szintaxissal lehet megadni a beágyazott nyelveket. A **Metaparse** könyvtár (lásd 3.1. alfejezet) önállóan is alkalmazható, fordítási időben futó parszer generátor bekerült a Boost könyvtárak közé. A 4. fejezetben a template metaprogramozás és a **funkcionális paradigma** kapcsolatát tekintem át az e téren elért eredményekkel együtt. Egyrészt megvalósítottunk számos, a funkcionális paradigmára jellemző nyelvi elemet, mint pl. monádokat és curryzést. Másrészt létrehoztunk egy **beágyazott nyelvet**, melyben **Haskell-szerű szintaxissal** lehet C++ template metaprogramokat fejleszteni. Az 5. fejezet a template metaprogramok nyomkövetésével és profilozásával kapcsolatos úttörő eredményeket ismerteti. A template metaprogramok esetében a potenciálisan hibás programok elemzése illetve **debuggálása** különösen nehéz feladat. Hasonlóan, a fordítási idejű algoritmusok hatékonysági elemzése is kihívás. A dolgozatban ismertetem azokat a kutatásaimat, melyek megoldást szolgáltatnak erre, és amelyek a C++ közösség által használt **Templight metaprogram debugger és profiler** kifejlesztéséhez vezettek, ami 2018 óta a **Clang C++ fordító beépített funkcionalitása**. A 6. fejezetben az előző három fő témakörön kívüli, a C++ template metaprogramozással kapcsolatos eredmények kaptak helyet. A dolgozat egy rövid összefoglalással fejeződik be.

Az esetek többségében a C++ fejlesztők nem közvetlenül használják a template metaprogramokat, hanem valamely mataprogramozási technikákon alapuló könyvtáron keresztül. A dolgozatban felsorolt eredmények ezért elsősorban (de nem kizárólagosan) a C++ template metaprogramok fejlesztői számára hasznosak.

2. fejezet

A C++ template metaprogramozás

Ebben a fejezetben röviden összefoglalom a dolgozat további részéhez szükséges alapvető ismereteket a C++ template metaprogramokról.

A sablon (template) a C++ programozási nyelv kulcsfontosságú eleme. Segítségével lehetségessé válik egyes adatszerkezetek vagy algoritmusok típussal való paraméterezése anélkül, hogy ez futási időben hatékonyságcsökkenéssel járna [73]. Ez az úgynevezett generikus absztrakció [15] elsődleges fontosságú, amikor olyan adatszerkezeteket definiálunk, mint pl. a lista vagy a halmaz és amikor olyan általános algoritmusokat szeretnénk megvalósítani, mint egy elem keresése valamely adatszerkezetben vagy a rendezés. Az ilyen generikus konstrukciók esetében az alapvető tulajdonságok, mint az adatszerkezet vagy az algoritmus működése megegyeznek, csak az aktuális típus(ok) különbözik/nek. A típusparaméteren keresztüli absztrakció – amit szokás parametrikus polimorfizmusnak is nevezni [13] – azt jelzi, hogy az eltéréseket a fordítási idejű típusparaméter behelyettesítésével fejezzük ki. A C++ legtöbb újrafelhasználható komponenseit, pl. a standard könyvtár konténereit és algoritmusait sablonokkal valósítják meg. A Standard Template Library (STL) a generikus programozás egyik legjellegzetesebb példája [8, 30, 37].

A C++ nyelvben a generikusokat a példányosítás (instantiation) segítségével valósítják meg. Más nyelvekben, mint pl. a Java, ahol típustörlés (type erasure) működik [82], a generikusak közvetlenül lefordíthatók. Ezzel ellentétben C++-ban a template-ek azonban közvetlenül nem lefordíthatóak. Lényegében kódsablonok, gyártási eljárások, melyekben egy vagy több szimbólum jelképezi a típusparamétert. C++-ban a template-eket nem fordítjuk le közvetlenül, hanem azokat példányosítani kell, azaz a sablon alapján, a típusparaméter helyébe valamely konkrét típust helyettesítve kapunk konkrét, lefordítható új kódrészletet, az ún. specializációt [17, 74, 75]. Így

minden egyes új template paraméter esetében új kód generálódik, ennek bizonyos esetben van is negatív hatása a generált kód méretére. Megjegyzendő, hogy nem csak típusokat alkalmazhatunk sablon paraméterként, hanem bizonyos feltételeknek megfelelő más, futási időben ismert értékeket is, pl. egész számokat, függvénymutatókat, stb. A lehetőségek folyamatosan bővülnek a C++11-től a C++20 verzióig.

A példányosítás történhet implicit módon, a függvényparaméterek típusából (C++17 óta az osztálykonstruktorok paramétereiből is [7]) következtetve, vagy explicit módon, egy vagy több típusparamétert a felhasználó által megadva. Az előző esetet nevezik típuskikövetkeztetésnek (type deduction). Amennyiben a típuskikövetkeztetés sikertelen, fordítási idejű hibaüzenetet kapunk, azaz a sablonok hibás alkalmazása nem okoz futási idejű hibát.

Egy fordítási egységben egy sablont egy adott argumentummal általában csak egyszer példányosítunk. Bár technikai okokból néha előfordul többszörös példányosítás is [85], ez nem garantált, hanem többnyire a már lepéldányosított specializációt fogjuk újra felhasználni. Ezt a jelenséget hívjuk *memoization*-nek.

Vannak helyzetek, amikor egy sablon valamely konkrét paraméterrel történő alkalmazásától eltérő viselkedést várunk el, mint a generikustól. Ilyen eset lehet például egy maximum függvény, mely általános esetben a paraméterek `operator<` függvénnyel történő összehasonlításán alapul, de ha két C jellegű stringet szeretnénk összehasonlítani, akkor az `strcmp` könyvtári függvényt kell alkalmazzuk. Ilyen helyzetekben megírhatjuk az adott paraméterre vonatkozó változatot, ún. teljes vagy explicit specializációt (explicit specialization). Az általános sablon változatot és az egyes specializációkat túlterhelhetjük, a fordítóprogram minden esetben a legspeciálisabb alkalmazható verziót fogja kiválasztani. Specializációt osztálysablonokra is írhatunk, maga a szabványos könyvtár is rendelkezik ilyennel (`std::vector<bool>`). Az osztálysablonok specializációjánál fontos megjegyezni, hogy a teljes (explicit) specializáció külön osztálynak számít, annak nemcsak implementációja, de a teljes publikus interfésze is eltérhet az általános változattól. E lehetőség kihasználásából az általános programozási gyakorlatban inkább problémák származnak, de ez teszi lehetővé template metaprogramok írását is.

A sablonok hivatkozhatnak egymásra, így ezek példányosítása esetenként újabb sablonok példányosítását indítja el. A példányosítások komplex láncolata és a specializációk kiválasztása teszi lehetővé a template metaprogramok létrehozását.

1994-ben Erwin Unruh, a C++ szabványbizottság egyik tagja egy kódrészletet mutatott be a bizottság tagjainak [83]. A program maga le sem fordult, ám a fordítóprogram által kiírt hibaüzenetekben sorrendben 2-től a prímszámok jelentek meg. Unruh ezzel a programmal azt demonstrálta,

hogy megfelelően elhelyezett template definíciókkal a szabványos C++ fordítóprogramot meghatározott algoritmusok végrehajtására lehet kényszeríteni. Az így készült programokat *C++ template metaprogramoknak* nevezzük.

Metaprogramoknak általában olyan programokat nevezünk, melyek (esetleg más) programokat elemeznek vagy módosítanak [15]. Ilyen értelemben a parszer generátorok, statikus elemzők, de maguk a fordítóprogramok is metaprogramok. A C++ template metaprogramok olyan értelemben speciálisak, hogy önhordók; azaz a program, ami végrehajtja őket, az a C++ fordító, és a kód, amit manipulálnak az maga a C++ forrás.

A C++ template metaprogramok bemutatására hagyományosan a faktoriális függvény értékének fordítási idejű kiszámítását mutatjuk be a 2.1 kódrészleten.

```
1  template <int N>
2  struct Factorial
3  {
4  public:
5    enum { value = N*Factorial<N-1>::value };
6  };
7  template<>
8  struct Factorial<1>
9  {
10   enum { value = 1 };
11 };
12 int main ()
13 {
14   const int r = Factorial<5>::value; // 120
15 }
```

2.1. példa. Faktoriális függvény C++03 template metaprogrammal

A 14. sorban hivatkozunk a `Factorial<5>::value` értékére, amit a fordítóprogramnak ki kell értékelnie. Ehhez elkezdi példányosítani az 1-6 sorok között megadott `Factorial<N>` sablont, ami rekurzívan példányosítja ugyanazt a sablont a 4, 3, 2 paraméterekkel is. Amikor `Factorial<2>` példányosításakor a `Factorial<1>::value`-ra hivatkozik, a fordító a 7-11 sorok között megadott specializációt használja fel. Végeredményben a fordítás eredményeként rendelkezésünkre áll a 120 érték `r` inicializálásához.

A rekurzió (és általa implementált ciklus) mellett elágazást is megvalósíthatunk, ahogy azt a 2.2 kódrészleten látjuk. Amennyiben az `if_` sablon első paramétere *igaz* értékű, akkor a sablonban definiált `type` típus megegyezik a `Then` paraméterrel, különben a specializációnak megfelelően a `type` az `Else` lesz. Az ilyen sablonokat szokás *metafüggvénynek* nevezni (lásd 4. fejezet).

```
template <bool Cond, class Then, class Else>
struct if_
{
  typedef Then type;
};
template <class Then, class Else>
struct if_<false, Then, Else>
{
  typedef Else type;
};
int main()
{
  if_< sizeof(int)<sizeof(long), long, int>::type i;
}
```

2.2. példa. Elágazás megvalósítása C++03 template metaprogrammal

Rekurzióval (ciklussal), elágazással és a sablonok példányoítási láncával lényegében egy komplett programozási nyelv áll rendelkezésünkre – melyben a C++ fordító fordítási időben hajt végre algoritmusokat. Az ilyen programokat nevezzük *C++ template metaprogramoknak*. Elméletileg bizonyítható, hogy a template metaprogramok Turing-teljes [86] nyelvet alkotnak.

A C++ fordítók rendkívül hatékonyan optimalizálják a kódot, hogy az gyorsabban fusson vagy kevesebb memóriát használjon. Azonban a fordító csak olyan optimalizációkat hajthat végre, melyek a kód működését nem változtatják meg. Sok esetben ennél hatékonyabb optimalizációkat is el lehet végezni azon alkalmazási területnek és annak szabályszerűségeinek ismeretében, mely számára a program készült. Ezeket a fordító nem ismeri, azonban ezek a optimalizációk template metaprogramok segítségével megvalósíthatók. Az ilyen optimalizációs célra készüt *expression template*-ek a C++ template metaprogramok első gyakorlati alkalmazásai voltak [87, 89, 90, 98].

A szakmaspecifikus nyelvek (DSL) egyre népszerűbbek. Ezek egyszerű nyelvek, melyeket egy adott terület számára fejlesztettek ki, itt hatékonyabbak, mint más nyelvek. Más területeken kevésbé használhatóak, ezért ott egyéb nyelvekkel együtt használják őket. Vannak széles körben használt szakmaspecifikus nyelvek, mint például az SQL vagy a reguláris kifejezések. Ezeket a nyelveket valamely általános célú programozási nyelvvel együtt szokták használni. A DSL által lefedett területre specifikus részeit a programnak a DSL-ben, a többi részt pedig az általános célú nyelven írják meg. Ha a DSL-ben írt kódrészleteket beágyazzák az általános célú nyelven írt kódba, *beágyazott* DSL-ekről (EDSL) beszélünk. Template metaprogramok segítségével DSL-ek hatékonyan beágyazhatók a C++ nyelvbe [68, 100].

Minden élő programozási nyelv fejlődik. Mielőtt új elemek kerülnének egy nyelvbe, érdemes annak hatásait, használati eseteit kipróbálni. A fordító módosítása gyakran túl költséges vagy nem is lehetséges. Ugyanakkor template metaprogramok segítségével számos nyelvi elemet a fordító módosítása nélkül is lehet szimulálni. Ilyen fordul elő pl. a move szemantika [101], a lambda kifejezések [103] vagy a C++20 nyelvi változatba bekerülő Concept-ek Boost könyvtárbeli megvalósításakor [99]. Korábbi kutatásaim során ilyen módszerrel hoztuk létre a strukturális altípusosság C++-beli megvalósítását [93].

Speciális alkalmazási területet (adatbázis elérés, reguláris kifejezések, stb.) támogató C++ könyvtárak számára hasznos ha az alkalmazási területtel kapcsolatos hibákat (adatbázis mezőinek típusaival kapcsolatos hibák, hibás reguláris kifejezések, stb.) már fordítási időben jelezni lehet. Template metaprogramok segítségével megvalósíthatóak ilyen ellenőrzések [19, 100]

A fejlesztők gyakran kénytelenek ismétlődő, csupán kevés részletben különböző kódot írni. Az esetek többségében ezeket a kódrészleteket meglévő kód másolásával és módosításával készítik el. Template metaprogramok használatával elérhető, hogy a C++ fordító generálja ezeket a kódrészleteket [1, 2].

A C++ programozási nyelv fejlődésével a template metaprogramok szerepe és technikája is változott. Kezdetben a numerikus számítások optimalizálására, *kifejezés sablonok* (expression template) implementálására alkalmazták [87, 89, 90]. Később, a C++14 verziótól [6, 71] a *constexpr* kifejezések egyszerűbb szintaxist adtak számos fordítási idejű számítás esetére. A constexpr azonban nem képes új típusokat létrehozni – ezért a template metaprogramokat jelen formájában nem fogja tudni teljesen helyettesíteni. Ugyanakkor a nyelv fejlődésével a C++ template metaprogramok létrehozása is egyszerűsödött.

Meg kell jegyezni, hogy az esetek többségében a C++ fejlesztők nem közvetlenül használják a template metaprogramokat, hanem valamely metaprogramozási technikákon alapuló könyvtáron keresztül. A `std::vector` puffer újraméretezése, a `std::shared_from_this` vagy a `std::common_type` osztály megvalósítása mind példák a fejlesztők által gyakran használt elemekre, melyek template metaprogramokra épülnek, de használatukhoz nincsen szükség metaprogramozási ismeretekre. A jelen dolgozatban felsorolt eredmények ezért elsősorban (de nem kizárólagosan) a C++ template metaprogramok fejlesztői számára hasznosak.

A C++ template metaprogramokról részletesebben a következő publikációkból lehet tájékozódni: [1, 2, 3, 14, 16, 35, 36, 47, 49, 54, 85, 86, 87, 89].

3. fejezet

Szakterület specifikus nyelvek és C++ template metaprogramok

Bár a szakterület specifikus nyelvek (Domain Specific Language, DSL) kiválóan teljesítenek az adott szakterületen, a programok végrehajtásának általában csak kisebb része esik ebbe a tartományba. Nagyobb részben a programok olyan általános tevékenységeket végeznek, mint az input és output, elágazások, ciklusok szervezése, szálak indítása, stb. ahol az általános célú programozási nyelvek hatékonyabbak. A szokásos megoldásként a szakterület specifikus nyelveket beágyazzák valamely általános célú programozási nyelvbe, ez utóbbit hívják ilyenkor gazdanyelvnek (host language). Ez a beágyazás lehet *külső*, ha valamely külső eszköz (pl. előfordító) segítségével történik; vagy *belső*, ha nincsen szükség más eszközre a gazdanyelv fordítóprogramján vagy interpreterén kívül. Bár számos jelenlegi DSL beágyazási technika a külső módszert alkalmazza [52, 91], a modern programozási nyelveket, mint a Haskell [25] és Scala [38], már a belső módszer támogatásának szempontjai szerint tervezték. A belső DSL integrációs módszer karbantartási szempontból előnyösebb, hiszen nincsen szükség a gazdanyelv és a külső eszközök közti sérülékeny kapcsolat fenntartására, másrészt lehetőség adódik a DSL és a gazdanyelv elemei közötti szemantikus kapcsolatok ellenőrzésére.

A C++ programozási nyelvben a template metaprogramozás a szakterület specifikus nyelvek belső integrációjának legígéretesebb módszere [1, 19, 48, 68]. Míg a klasszikus könyvtárak hibás alkalmazása csak futási időben jelezheti a problémákat, a template metaprogramok lehetővé teszik, hogy a DSL integráció fordítási időben történjen, és így fordítási időben szűrjük ki a DSL vagy annak gazdanyelvi alkalmazásának hibáit. A DSL integrációra példa Jossy Gil Ararat SQL integrációja [19], Eric Niebler Boost.Xpressive reguláris kifejezés- [100] és Boost.Proto kifejezés-template könyvtára [104]. Ezek a template metaprogram könyvtárak azonban hasonló problémával szembe-

sülnek: szintaktikájuk körülményes, nehezen követhető, ami ellentmond a szakterület specifikus nyelvek lényegének, az érthető, nagyobb kifejezőerejű világos megjelenésnek, ahogy azt a 3.1 kódpéldán is láthatjuk. Mindennek az az oka, hogy a DSL-t leíró sablonoknak teljesíteniük kell a C++ minimális lexikai és szintaktikus követelményeit: a gazdanyelv és a beágyazott DSL is szintaktikusan helyes C++ program kell legyen.

```
1 "(\\w+) (\\w+)!"                          // std::regex
2 (s1= +\_w)>>' '>>(s2= +\_w)>>'!'          // Boost.Xpressive
```

3.1. példa. Futási és fordítási idejű DSL integráció szintaxisa

3.1. Metaparse

Kutatásaink során Sinkovics Ábellel kidolgoztunk módszereket arra, hogy a C++ template metaprogram előnyeit összeegyeztessük a szakterület specifikus nyelvek kifejező szintaxisával. A megoldás a **Metaparse** könyvtár formájában öltött testet [48, 56, 57], ami egy fordítási idejű teljes elemző (parszer) infrastruktúra. A Metaparse inputja egy tetszőleges szintaxisú DSL, amelyet template metaprogramként fordítási időben egy C++ szintaxisnak megfelelő forráskóddá konvertál. A Metaparse egy egyetlen fejlécállományból álló (header-only) könyvtár, így rendkívül hordozható, és az egyetlen dolog, amit a felhasználónak tennie kell, hogy include-olja a megfelelő headert, és elindítson egy szabványos C++ fordítóprogramot.

A Metaparse segítségével átalakíthatóak a meglévő C++ template metaprogram alapú DSL-ek barátságtalan interfészei is. Kidolgoztuk pár ismert metaprogram alapú DSL javított interfészét, melyet Metaparse segítségével fordíthatunk le. A 3.2 kódrészletben a Boost.Xpressive reguláris kifejezéseit látjuk az eredeti és a Metaparse által módosított interfész formájában.

```
1 (s1= +_w) >> ' ' >> (s2= +_w) >> '!';
2 REGEX("(\\w+) (\\w+)!");                  // Metaparse
3 bos >> set[as_xpr('a')|'b'|'c'|'d'] >> range('3','8') \\
4     >> '.' >> 'f' >> 'o' >> 'o' >> eos;
5 REGEX("^[abcd][3-8]\\.foo$");             // Metaparse
6 bos >>  (s1=+range('0','9')
7     >> !(s2='.'>>*range('0','9')))
8     >>  (s3=set[as_xpr('C')|'F']) >> eos;
9 REGEX("^([0-9]+(\\.[0-9]*)?)([CF])$");    // Metaparse
```

3.2. példa. A Boost.Xpressive és a Metaparse-al javított interfész

A Metaparse implementációja Andersson Haskell alapú parszer generátorának felépítését követi [4]. Az elemi parszer kombinátorokból építkező konstrukció megőrzi a lefordítandó nyelvtan struktúráját. A Metaparse könyvtárat 2015 nyarán szigorú review után beszavazták a Boost könyvtárak [24] közé. A Boost.Metaparse [61] azóta is minden Boost verziónak automatikusan része, karbantartója Sinkovics Ábel. Jelenlegi tudásunk szerint a Metaparse az egyetlen fordítási időben működő, template metaprogramokat generáló parszer rendszer [49].

3.2. Metastring

Template metaprogramok szakterület specifikus nyelv integrációjaként való felhasználásakor az egyik legnehezebb feladat a tetszőleges, a C++ szintaxisnak nem feltétlenül megfelelő DSL forráskód reprezentálása C++ sablonok által felhasználható formátumként. Pár korábbi próbálkozás vagy külső eszközöket (pl. Python script) vett igénybe, vagy erősen korlátos kifejezőerejű és nem szabványos volt (pl. a Boost.MPL-ben a `wchar_t` használata) [1, 102]. Kidolgoztunk egy szabványos és kényelmes interfészt – a **Metastring** könyvtárat – arra, hogy karaktersorozatokat kényelmesen lehessen sablonok számára feldolgozható módon ábrázolni [66]. A módszert felhasználtuk a Metaparse könyvtárban, de hasznosságát demonstráltuk egy típusbiztos `safePrintf` implementációval is (lásd 3.3 kódrészlet), mely fordítási időben ellenőrzi a formátum string és az átadott paraméterek megfelelősségét[77].

```
1 safePrintf<"Hello %s! i = %d">("John",42);
```

3.3. példa. Típusbiztos printf és használata

A szakterület specifikus nyelvek C++ gazdanyelvi beágyazásával kapcsolatos publikációim az alábbiak: [47, 48, 49, 56, 57, 68, 77].

A 2012-es C++Now konferencián (Aspen, CO, US) előadott *Metaparse – Compile-time parsing with template metaprogramming* [56] című előadásunk a konferencia *Best presentation* díját nyerte el.

4. fejezet

A C++ template metaprogramozás és a funkcionális paradigma kapcsolata

A C++ template metaprogramozás és a funkcionális programozási paradigma kapcsolata jól ismert [12, 20, 21, 34, 36, 47, 69, 92]. A template metaprogramok fordítási időben értékelődnek ki, ezáltal természetszerűen minden programelemükre teljesül a *hivatkozási helyfüggetlenség* (referential transparency), ami a funkcionális paradigma egyik fő jellemzője. A vezérlési szerkezetek például ennek megfelelően nem tartalmaznak ciklust, ehelyett az klasszikus funkcionális módon rekurzió és mintaillesztés segítségével valósul meg. Hasonlóan nem találunk értékadást sem, csak inicializálást.

Annak ellenére, hogy a futási idejű programok és a template metaprogramok szerkezete eltérő, találunk bizonyos párhuzamokat. Komplex feladatok esetén mindkét paradigmában szükségszerű a feladatok részekre bontása. Abrahams és Gurtovoy definiálja [1] a *metafüggvény* (metafunction) fogalmát: olyan template osztályt értve ezalatt, ami példányosulásakor egy előre meghatározott beágyazott típust definiál, mint a függvény „értéke”. E beágyazott típus neve gyakran `type`. Ezek a típusok argumentumként ugyancsak átadhatóak template-eknek, így teljes függvényhívási láncokat tudunk létrehozni.

Futási időben inputról beolvasott értékeket értelemszerűen nem használhatunk, de konstansokat és fordítási időben végzett műveletekkel (pl. a `sizeof` operátorral) meghatározott értékeket igen. Ezeket is beágyazhatjuk template osztályokba (mint skalár paramétert), és az aritmetikai műveleteket metafüggvényként megírva teljes mértékben felhasználhatjuk őket.

A metafüggvények – a funkcionális paradigmának megfelelően – teljes értékű polgárok a metaprogramok világában, őket átadhatjuk más metaprogramoknak paraméterként, vagy visszatérő értékként is szolgáltathatjuk. A

4.1 kódrészletben egy ilyen példát láthatunk.

```
1  // Accumulate(n,f) := f(0) + f(1) + ... + f(n)
2  template <int n, template<int> class F>
3  struct Accumulate
4  {
5    enum { value=Accumulate<n-1,F>::value + F<n>::value };
6  };
7  template <template<int> class F>
8  struct Accumulate<0,F>
9  {
10   enum { value = F<0>::value };
11 };
12 template <int n>
13 struct Square
14 {
15   enum { value = n*n };
16 };
17 int main()
18 {
19   cout << Accumulate<3,Square>::value << endl;
20   return 0;
21 }
```

4.1. példa. Metafüggvény, mint metafüggvény paramétere

Ahogy korábban említettem, a C++ programozási nyelv eredeti szempontjai között nem szerepelt a template metaprogramok támogatása. Ennek eredményeképpen a metaprogramok szintaxisa bonyolult, azokat nehéz írni, megérteni és karbantartani. Ehhez adódik, hogy a korábbi template metaprogram könyvtárak, mint pl. a Bost.MPL [102] maguk is gyakran az imperatív programozás eszközrendszerét próbálták leképezni a metaprogramok funkcionális világára. Értelemszerűen merül fel a lehetőség, hogy a C++ template metaprogramok világát azok természetes paradigmájának megfelelően funkcionális nyelvi elemekkel bővítsük.

Futási időben végrehajtott C++ kód készítéséhez vannak nyelvi elemek és könyvtárak, melyek funkcionális programozási elemeket támogatnak. Beépített nyelvi elemeknek tekinthetjük a lambda kifejezéseket [5], vagy az `operator()` felhasználását függvényszerű objektumok (functorok) írásához [40]. Könyvtárként megvalósítható a currying, lusta kiértékelésű listák, generátorok és hasonló funkcionális elemek [27, 28]. A 4.1 táblázat összehasonlítja a futási idejű funkcionális programok és a template metaprogramozás eszközrendszerét. Ugyanakkor a template metaprogramok támogatása funkcionális

	Futási idejű funkcionális program	C++ template metaprogram
értékek	futási idejű adat (konstansok, literálok)	`static const` és `enum` osztálytagok
változók	hagyományos változók	szimbolikus nevek (típusnevek, aliasok)
inicializálás	konstansok generátorok	`static const` inicializálás `enum` értékek
értékadás	*nincs*	*nincs*
input/output	monádokkal	nincs interaktív input warning-ok, hibaüzenetek
elágazás	mintaillesztés függvény specializáció	mintaillesztés template specializáció
ciklus	függvény rekurzió	rekurzív template-ek
alprogram	függvény	(template) osztály
adattípus	absztrakt adatszerkezetek	typelist, `boost::mpl::vector`
típusok	type class (Haskell)	concepts

4.1. táblázat. Funkcionális és template metaprogramok összehasonlítása

nyelvi eszközökkel nyitott kutatási terület volt. Két kutatási irányban indultunk el: az egyikben a jelenlegi template metaprogramozási eszközrendszert bővítettük a funkcionális nyelvekben megszokott elemekkel, a másikban egy szakterület specifikus beágyazott nyelvet alkalmaztunk (lásd 3. fejezet), melyben funkcionális módon Haskell-szerűen lehet leírni a template metaprogramokat.

4.1. Template metaprogramozás bővítése funkcionális elemekkel

Ezen kutatási irányon belül megvizsgáltuk a C++ template metaprogramozás és a funkcionális programozási nyelvek kapcsolatát [69]. A tapasztalatok [63, 64] alapján kidolgoztunk olyan template metaprogram technikákat, melyek a funkcionális paradigma egyes ismert elemeinek felelnek meg, így natív módon támogatják a template metaprogramok funkcionális stílusban történő fejlesztését. Az alábbi eredményeket értük el:

- Kidolgoztuk a metaprogramok **lusta kiértékelésének** lehetőségét, amelyet tetszőleges metaprogramra lehet alkalmazni.

- A funkcionális nyelvekben ismert **curryzés** megvalósítása.
- Az **algebrai adattípusok** template metaprogrambeli megvalósítása.
- Haskell-jellegű **típusosztályok** (typeclass) kidolgozása.
- Template metaprogram kifejezések tárolása, paraméterként átadása, függvényértékként történő visszatérése a funkcionális **let kifejezések** megvalósításához. Ezen alapulva a Boost.MPL könyvtárénál hatékonyabb megoldás **lambda kifejezések** implementációjára.
- Módszer a **mintaillesztések** létrehozására, mellyel megvalósíthatóak a **case kifejezések**.
- Módszer a **monádok** megvalósítására C++ template metaprogramozásban. Számos Haskellben elérhető monád implementálása C++ template metaprogramként.
- Módszer a Haskellben elérhető **do jelölés** (do notation) megvalósítására template metaprogramozásban.
- Egy monádokon alapuló módszer kidolgozása a **kivételkezelés** szimulálására C++ template metaprogramozásban.

Az alábbi példán az mutatjuk be, hogyan lehet az eredményeket felhasználni a funkcionális programozási paradigmának megfelelő C++ template metaprogramozás során.

```
1 [(i,j) | i←[1..100],j←[1..100],is_relative_prime i j]
```

4.2. példa. List comprehension Haskellben

A 4.2 kódrészleten egy Haskell funkcionális nyelvi elemet, az ún. *list comprehension*-t látunk, ami az 1 és 100 közötti relatív prím párokat generálja le. Ugyanezt a kódot a do jelöléssel (még mindig Haskellben) a 4.3 kódrészleten láthatjuk.

```
1 do
2   i ← [1..100]
3   j ← [1..100]
4   guard $ is_relative_prime i j
5   (i, j)
```

4.3. példa. Do jelölés Haskellben

Mindezt a kidologozott C++ metaprogramozási technikákat alkalmazva, szinte egy az egyben írhatjuk át C++ kóddá, ahogy azt a 4.4 kódrészleten látjuk.

```
do_c<list_tag,
  set<i, mpl::range_c<int, 1, 101>>,
  set<j, mpl::range_c<int, 1, 101>>,
  guard<is_relative_prime<i, j>>,
  mpl::pair<i, j>
>
```

4.4. példa. Do jelölés átírása C++ template metaprogram szintaxisra

4.2. Template metaprogramozás szakterület specifikus nyelv segítségével

E kutatásban olyan interpretert valósítottunk meg, mely fordítási időben egy Haskell szintaxisú DSL-t hajt végre metaprogram segítségével. A beágyazott kódrészlet értelmezése és végrehajtása ugyanabban a fordítási lépésben történik meg, így lehetővé válik, hogy C++ template metaprogramokat egy natív, funkcionális nyelvű szintaxissal írjunk le. A megoldás során felhasználtuk a 3. fejezetben leírt módszereket, így többek közt a Metastring könyvtárat.

```
typedef meta_hs::define<_S(
  "fact n =                "
  "  if n == 0             "
  "    then 1              "
  "    else n * fact (n-1)"
)>::type
  ::get<_S("fact")>::type fact;

typedef fact::apply<mpl::int_<3>>::type factorial3;
```

4.5. példa. A factorial függvény DSL-beli definíciója és meghívása

Mint a 4.5 példán látható, a beágyazott DSL segítségével lényegében a Haskell nyelvnek megfelelő szintaxisban adhatjuk meg a `fact` template metafüggvény kódját, ami fordítási időben kerül interpretálásra. Mindez nagyban egyszerűsíti a C++ metaprogram fejlesztők és karbantartók feladatait.

A template metaprogramozás és a funkcionális programozás kapcsolatát tárgyaló tézisemhez tartozó publikációim: [47, 48, 49, 56, 62, 65, 67, 68, 69, 77].

5. fejezet

C++ template metaprogramok nyomkövetése

A programfejlesztés (jelenleg) elsősorban emberi tevékenység, mely során elemezzük a megoldandó problémát, tervezési döntéseket hozunk és szándékainkat a számítógép számára érthető formára képezzük le. Ez az utolsó lépés többnyire valamely programozási nyelven megírt kód létrehozását jelenti. A fordítóprogram ezután megkísérli a forráskódot értelmezni lexikális, szintaktikus és szemantikus elemzés segítségével. Amennyiben ez sikeres, további lépések következnek a kód generálása, optimalizálása céljából.

Ugyanakkor számos esetben előfordul, hogy a program, melyet a fordítóprogram elfogadott nem az elvárt módon működik. Mindezt többféle dolog okozhatja egy egyszerű (a szintaxist szerencsétlen módon nem befolyásoló) elgépeléstől az implementáció során elkövetett tévedésen át a komoly tervezési hibáig. Bár számos módszer ismert arra, hogy csökkentsük a specifikációtól eltérő program létrehozásának esélyeit, azt teljesen sohasem fogjuk tudni kizárni. A maradék esetekben nekünk kell megtalálni a konkrét hiba forrását és kijavítani azt. Ezt a tevékenységet hívjuk nyomkövetésnek (debugging).

A debuggerek olyan szoftverek, melyek a támogatják a nyomkövetési tevékenységet. Fő feladatuk, hogy segítségükkel feltárjuk és megértsük a rejtett események azon láncolatát, melyek a hiba bekövetkeztéhez vezettek. Ez legtöbbször a program vezérlési folyamatának követésével, a memória területek (változók, tömbök, rekordok) tartalmának kinyerésével és a végrehajtási környezet (pl. a függvényhívási láncok) felderítésével lehetséges. A debuggerek gyakran további lehetőségeket is adnak a hibakeresési folyamat hatékonyabbá tételéhez, mint pl. *megállási pontok* (breakpoint) elhelyezése a kódban vagy a lépésenkénti végrehajtás, melyek során az utasításokat egyenként hajthatjuk végre és függvényhívásoknál eldönthetjük, hogy belépünk-e a hívott függvénybe vagy átlépjük annak végrehajtási lépéseit. Bár a debuggerek fo-

lyamatosan fejlődnek, a hibakeresés továbbra is a programozói munka egyik legtöbb tudást igénylő, de egyben legfrusztrálóbb tevékenysége.

A template metaprogramozás egyik közismert problémája a költséges karbantarthatóság, melynek egyik összetevője a nyomkövetés nehézsége. A template metaprogramokat a fordítóprogram „hajtja végre", de könnyen belátható, hogy a C++ fordítóprogram debuggálása nem vezetne eredményre, hiszen a fordítóprogram tevékenységének (előfordító végrehajtása, lexikai egységekre bontás, az absztrakt szintaxisfa felépítése, típusellenőrzés, optimalizálás, kódgenerálás, stb.) elhanyagolható része a sablonok példányosítása, és az ilyenkor kinyerhető adatok is csak nagyon áttételesen adnának információt a metaprogram végrehajtásáról. Külön problémát okoz, hogy a C++ template metaprogramok esetében nagyon korlátosan állnak rendelkezésünkre a hagyományos nyomkövetési eszközök is, így például nem tudunk log üzeneteket formázni és kiírni, az összes output tevékenységünk erősen fordító specifikus figyelmeztetésekre (warningokra) korlátozódik. A template metaprogramok nyomkövetésére speciális eszközöket kell készítenünk. A 2000-es évek közepéig egyáltalán nem állt rendelkezésre ilyen eszköz.

Ebben a fejezetben a C++ template metaprogramok nyomkövetésével kapcsolatos kutatásaimat és eredményeimet tárgyalom. Elsőként áttekintem a lehetséges hibafajtákat, majd két megközelítést ismertetek a nyomkövetésre: a forráskód-instrumentáció alapút és a fordítóprogramba integrált megvalósítást. Ezek után ismertetem az LLVM/Clang C++ fordító részeként megvalósult metaprogram debuggert és felhasználási tapasztalatait. A fejezet végén a további lehetséges kutatási irányokat taglalom.

5.1. Template metaprogramozással kapcsolatos hibák

A C++ fordítás első ütemében az előfordító lépéseinek végrehajtása után a nyelvi elemek lexikális azonosítása, majd a szintaktikus elemzés következik, melynek eredményeképpen a fordítóprogram megpróbálja létrehozni a forráskód belső reprezentációját egy absztrakt szintaxis fa (Abstract Syntax Tree, AST) formájában [88]. A C++ sablonok példányosítás előtt szintén az AST egy részfájaként jelennek meg [85]. A legszerencsésebb hibaeset akkor fordul elő, ha szintaktikus hibát ejtünk a forráskódban és az AST előállítása sikertelen. Az ilyen hibás formátumú (ill-formed) programokra példa a 5.1 kódrészlet. Ilyenkor a template metaprogramok végrehajtása el sem kezdődik, és a fordító általában a hiba helyét és okát pontosan meghatározó hibaüzenetet ad.

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

5.1. példa. Hibás formátumú program, példányosítás nem kezdődik el

Kellemetlenebb helyzet áll elő a 5.2 kódrészlet esetében. A program szintaktikusan helyes, de a `Factorial<1>` specializációban a `value` helyett `ivalue` adattag szerepel. Mindez nem okoz problémát a szintaktikus elemzés során, hiszen önmagában mind az általános template, mint a specializáció helyes, és amíg nem példányosítjuk a sablonokat, addig nem történik meg a sablonban szereplő nevek feloldása sem. Amikor viszont hivatkozás történik `Factorial<5>::value` értékére, elkezdődik a példányosítási lánc, a `Factorial` sablon sorra példányosul az 5, 4, 3, 2 értékekkel. Végül `Factorial<2>` megpróbál hivatkozni a `Factorial<1>::value`-ra, de ez sikertelen lesz. Itt fogunk hibaüzenetet kapni, amely tartalmazni fogja a teljes példányosítási láncot, ami kellően bonyolult template metaprogram esetben valószínűleg nehezen elemezhető lesz a fejlesztő számára.

```
template <int N>
struct Factorial
{
public:
  enum { value = N*Factorial<N-1>::value };
};
template<>
struct Factorial<1>
{
  enum { ivalue = 1 };  // ivalue instead of value
};
int main ()
{
  const int r = Factorial<5>::value;
}
```

5.2. példa. A példányosítás elkezdődik, névfeloldási hiba

Ismét más jellegű hibával találkozunk a 5.3 kódrészletben. Itt szintaktikailag helyes Factorial sablont látunk, a specializáció sincs elrontva. Ugyanakkor a főprogramban negatív értéket adunk a sablon paramétereként, ezért a példányosítás sohasem fog hivatkozni a specializációra. Az eredmény elvileg egy végtelen példányosítási lánc lenne, ami az erőforrások elfogytával a fordítóprogram összeomlásával járna. Ezt megelőzendő, a legtöbb C++ fordítóprogram limitálja a példányosítási mélységet, hogy ilyen esetekben is képes legyen értelmes hibaüzenetet adni. A C++ szabvány szerint a végtelen rekurzív példányosítás nem definiált viselkedés, de a C++17 szabvány ajánlást ad a maximális példányosítási mélységre, melynek értéke 1024 [7]. Ettől az egyes fordítóprogramok eltérhetnek, és ezt az értéket akár a fordítóprogram paramétereként is megadhatjuk (pl. a `g++` és a `clang++` fordítóprogramok esetében a `-ftemplate-depth=N`). Az `N` limit elérésekor hibaüzenetet kapunk, de közben az összes példányosítás is kiíródik. A gyakorlati esetekben `N` nagy értéke miatt ez az információ nehezen értelmezhető.

```
1  template <int N>
2  struct Factorial
3  {
4  public:
5    enum { value = N*Factorial<N-1>::value };
6  };
7  template<>
8  struct Factorial<1>
9  {
10   enum { value = 1 };
11 };
12 int main ()
13 {
14   const int r = Factorial<-5>::value; // negative arg
15 }
```

5.3. példa. Végtelen rekurzív példányosítás

Végül előfordulhat, hogy a maximális rekurzív template példányosítási mélység olyan magas értékre lett állítva, hogy a fordítóprogram erőforrásai hamarabb elfogynak, mintsem elérnénk a példányosítási limitet. Ilyenkor a fordítóprogram összeomlik és lényegében semmilyen információt nem tudunk kinyerni a hibás C++ template metaprogramról.

A C++ template metaprogramok potenciális hibafajtái ontológiáját részletesen tárgyalja [47, 70].

5.2. Nyomkövetés instrumentálással

Ha a futási időben működő debuggerekkel keresünk analógiát, akkor felvázolhatjuk a C++ template metaprogram debuggerrel szembeni követelményeinket. A futási idejű nyomkövetésnél elsődleges a függvényhívási lánc (call stack) megállapítása és az adott hívási szinten (stack frame) átadott függvényparaméterek és definiált változók értéke [72, 84]. Az ennek megfelelő metaprogram információ a template példányosítási lánc és az adott sablonparaméterek értéke.

Az egyik kézenfekvő megoldás a template példányosítások nyomkövetésére, ha arra kényszerítjük a fordítóprogramot, hogy minden példányosítás valamilyen outputot generáljon. Ahhoz, hogy a rekurzív példányosítások egymáshoz való kölcsönös viszonyát pontosan fel tudjuk térképezni, jeleznünk kell minden példányosítás kezdetét és befejezését. Mivel a fordítóprogram diagnosztikai üzenetekkel kommunikál a külvilággal, észszerű, hogy olyan kódrészletet szúrjunk be a template definíciókba, amelyek figyelmeztető üzeneteket (warning) generálnak, azon belül pedig pontosan közlik a forráskódon belüli pozíciójukat és a template paraméter(eke)t. Az ilyen beszúrások természetesen megváltoztatják a forráskód sorokat, ezért szükséges az üzenetek pozícióját valamilyen módon visszaképezni az eredeti forráskód sorokra.

A nyomkövetés menete ezek alapján a következő. Elsőként a forráskódon végrehajtjuk az előfordító lépéseit, hogy pusztán C++ tokenekből álló forrást állítsunk elő. Ezek után instrumentáljuk a forrást, minden template definíció elejére és végére fordítófüggő, specifikus warningot kibocsájtó kódrészletet szúrunk be. A kibocsájtott warning azonosítja a helyet, a sablont, annak template paramétereit és valami olyan extra információt, ami segít megkülönböztetni az általunk előidézett warningokat a fordító által amúgy is generálandó figyelmeztetésektől. Ezek után lefordítjuk a programot, elkapjuk az általunk generált warningokat és a kinyert információt visszaképezzük az eredeti forráskódra. Az instumentációs módszer pontos leírása megtalálható Mihalicza József PhD dolgozatában [33].

A módszert bizonyos korlátokkal a C++ sablonok példányosítási idejének mérésére is lehet használni. Amennyiben a fordító kimenetén megjelenő warningokra időpecsétet rakunk, úgy az egyes sablonok példányosításának kezdő és végidőpontjait összevetve következtethetünk az adott template példányosulási idejére. Az eredmény nem pontos, hiszen a warning generálások és a processzek közötti kommunikáció is megnöveli a mért értékeket, de az egyes sablonok példányosulási idejénk arányai kellően stabilak maradtak [45].

Az instrumentáláson alapuló C++ template metaprogram debugger prototípusát, a *Templight 1.0* rendszert 2006-ban mutattuk be a GPCE konferencián [43], és később több nemzetközi fórumon [44, 45] is ismertettük.

5.3. A fordítóprogram módosítása

Az instrumentáláson alapuló módszernek nagy előnye, hogy tetszőleges szabványos fordítóprogrammal működik. Sajnos azonban fel lehet sorolni hátrányos tulajdonságokat is. (1) Az egyes warning generáló kódrészletek fordítófüggőek, sőt, azon túl verziófüggőek lehetnek. (2) A kódinstrumentálás és a warning-ok visszaképezése az eredeti kódra külső eszközöket vesz igénybe, ami az egész módszer alkalmazhatóságát nehezíti. (3) A figyelmeztetést generáló sablonok fordítása és maga a figyelmeztetések keletkezése is többletmunkával jár, ami jelentősen növeli a fordítási időt. Ez egyben torzítja a profilozást is (bár az egyes template-ek példányosítási költségeinek aránya azért stabil marad).

Legfőképpen azonban az instrumentációs módszernek van egy súlyos funkcionalitásbeli hiányossága is. Mivel minden C++ template csak (fordítási egységenként) egyetlen példányban példányosul, a debugger csak a sablon első példányosítását fogja jelezni, a *memoization* (lásd 2. fejezet) hivatkozásokat nem. Olyan rekurzív template metaprogramoknál, ahol többször hivatkozunk ugyanarra a template paraméterű sablonra[1], ez igen jelentős információvesztést okoz. Instrumentálással azonban a memoization-ök nem kimutathatóak.

Miután rájöttünk, hogy a memoization problémáját csak a fordítóprogram szintjén lehet megoldani, elkezdtünk dolgozni a fordítóprogram módosításán. Jelenleg az LLVM/Clang tekinthető a legmodernebb C++ fordítóprogramnak, rendszerint itt jelennek meg elsőként az új nyelvi elemek. Ennek részben az az oka, hogy a fordító egy objektumelvű könyvtár fölé épült, így kényelmesen módosítható, újrafelhasználható. A Templight 2.0 projektben Borók-Nagy Zoltán, majd Mikael Persson dolgozott a Clang fordító számára készült patch-en [41]. A patch-et a letöltött Clang forráskódra kellett alkalmazni, és így a fordító egy módosított verziójához jutottunk, amely minden template akció (beleértve a memoization-öket is) elején és végén a megfelelő üzeneteket bocsájtotta ki több lehetséges formátumban (XML, YAML) is. Mivel az üzenetek kibocsájtása nem járt új sablonok példányosításával, mint az instrumentációra alapuló módszer esetében, ez a megoldás alkalmas profilozásra is [32, 33, 46].

A patch alkalmazása bizonyos kényelmetlenséggel jár. Amennyiben valaki valamely specifikus platformot használ, alkalmaznia kell a patchet a fordítóprogram forrására, majd le kell fordítani a Clang-ot, ami idő és erőforrás igényes. Amennyiben a fordító kódja jelentősen megváltozik (ami a Clang fejlődése miatt rendszeres) a patch karbantartójának felelőssége az illeszkedéshez

[1] Gondoljunk akár csak egy egyszerű Fibonacci sorozat kiszámolására.

szükséges módosításokat elvégezni. Ezért mind a Templight fejlesztők, mind a felhasználók elsődleges érdeke volt, hogy a kód bekerüljön az LLVM/Clang kódbázisába [50, 51]. Ez végül Umann Kristóf munkája nyomán 2018-ban történt meg, a Clang 7.0 verzió óta a Templight a fordítóprogram egy beépített funkciója [105].

5.4. A Templight felhasználásai

A C++ template metaprogramok nyomkövetésére több fontos alkalmazás épül. A Templight outputja szövegalapú vagy strukturált (XML vagy YAML) formátumú. Ezt értelmezni semmivel sem könnyebb, mint bármely karakteres debuggert (mint pl. gdb) használni. Ahhoz, hogy a metaprogramok működését könnyebb legyen megérteni, kifejlesztettük a **Templar** grafikus megjelenítő szoftvert, amely az output felhasználásával interaktívan képes megjeleníteni a metaprogram struktúráját [11]. A Templar alkalmas a Templight kimenetének interaktív visszajátszására és annak grafikai megjelenítésére is. Eközben a lejátszást megállíthatjuk megállási pontok lerakásával, reguláris kifejezésekkel pedig kiszűrhetjük az érdektelen template akciókat.

Egy másik irány a Templight outputjának felhasználására és értelmezésére a Sinkovics Ábel és Kucsma András által létrehozott **Metashell** rendszer [60]. A Metashell a modern programozási nyelvek interaktív REPL rendszereihez (Read-Execure-Print-Loop) hasonlít: interaktívan hajt végre C++ template metaprogramokat. A Metashell rendelkezik a szokásos debuggerek funkcionalitásával is; pl. megállási pontot rakhatunk egy kódsorra. Természetesen itt sem a fordítás valódi megállítása történik ilyenkor, hanem a Templight által szolgáltatott nyomkövető kimeneten történő előrehaladást állítjuk meg [58, 59].

A C++ template metaprogramok gyakran alkalmazzák a függvények túlterhelésén illetve a megfelelő specializáció kiválasztásán alapuló technikákat. Számos észrevétel érkezett a Templight felhasználóitól, hogy érdemes lenne a túlterhelési folyamatot is nyomkövetni. Jelenleg ebben az irányban folytatunk kutatásokat.

A C++ template metaprogramok nyomkövetésével és profilozásával foglalkozó tézishez kapcsolódó publikációim: [11, 32, 43, 44, 45, 46, 47, 48, 50, 51, 68].

Az LLVM/Clang C++ fordító 7.0 verziójától kezdve támogatja a Templight template nyomkövetési funkcionalitást.

6. fejezet

További kutatási témák

Ebben a fejezetben összefoglalom azokat a C++ template metaprogramozással kapcsolatos jelentősebb kutatásaimat, melyek nem kapcsolódnak közvetlenül az előző tézisekhez.

Ahogy azt a 2. fejezetben is említettük, a C++ template metaprogramozás alkalmas arra, hogy segítségével új nyelvi konstrukciókat vizsgáljunk meg anélkül, hogy a fordítóprogramot módosítani kellene. Ilyen módon kísérleteztek a Boost könyvtárak készítői a később a C++11 nyelvi verzióban megvalósult move szemantikával, az okos mutatókkal (smart pointer) és az inicializálás új lehetőségeivel is [31]. Kutatásaink során mi is alkalmaztuk a template metaprogramozás lehetőségeit a új nyelvi elemek bevezetésére. Az ún. **feature-oriented programozás** lehetőségeit vizsgálva egy prototípus könyvtárat implementáltunk, amely a **strukturális altípusosság** megvalósítására szolgál [42, 93].

A C++ nyelv jelenleg csak minimálisan rendelkezik **önelemző** (introspection és reflection) képességekkel. Ugyanakkor régóta folynak kísérletek C++ template metaprogram könyvtár alapú önelemző megoldásokkal. A [94] cikkünkben ismertetünk egy kísérleti könyvtárat, amelyre azóta számos cikkben hivatkoztak. A könyvtár metaprogramozási eszközökkel lehetővé teszi, hogy fordítási időben információkat nyerjünk az egyes globális- vagy tagfüggvényekről, adattagokról. A kinyert információk felhasználhatók további metaprogramok számára a további kódgeneráláshoz. A szabványos megoldás (tőlünk függetlenül és részben a fordítóprogram támogatással) a C++11 `type_traits` könyvtárába került bele. A (statikus) reflection végleges nyelvi megoldása a hosszú távú (C++23, C++26) tervek közé tartozik, ehhez ad javaslatokat [26] cikkünk.

Ahogy arról a 2. fejezetben is írtam, a C++ nyelv sablonjai eredetileg megszorítás nélküliek [18]. A 2000-es évektől számos ötlet született a sablonszerződés modell (**concepts**) megoldására akár könyvtár [24, 29, 53], akár

nyelvi [22, 76] szinten. Ehhez a törekvéshez csatlakozva bizonyos elő- és utófeltételek teljesülése ellenőrzésére dolgoztunk ki megoldást [39] cikkünkben. A [78, 80] cikkeinkben pedig az akkor tárgyalt sablon-szerződés modell architektúra skálázhatóbbá tételére tettünk javaslatot. Sajnos a C++11 verzió tervezetéből kivették a sablon-szerződés modellt és végül a C++20 verzióig kellett várni, hogy csökkent funkcionalitással **Concepts** néven bekerüljön a szabványba.

Bizonyos más nyelvekben (Java, Eiffel) megtalálható, de a C++-ból egyelőre **hiányzó nyelvi elemeket** (`final` metódusok, metódus átnevezés) valósítottunk meg template metaprogramok segítsévével [79] cikkünkben. Az automatikus konverziók sokszor jelentenek potenciális veszélyt C++ programokban. Olyan burkoló osztálysablonokat hoztunk létre, amelyek megőrzik a paraméterosztály funkcionalitását, de a konverziók tiltásával biztonságosabbá teszik a nyelvet [9, 10]. Ez utóbbi cikk egyben bevezető a biztonságosabb típuskonstrukciók (**safe types**) irányába történő jelenlegi kutatásainknak is.

Sablonkönyvtárak segítségével **elosztott rendszerek szimulációjára** is van lehetőség. A [96, 97] cikkekben ismertetjük azon munkákat, melyekkel a D-Clean elosztott koordinációs nyelv [95] működési szemantikáját valósítottuk meg.

Az **aktív könyvtárak** olyan C++ template metakönyvtár alapú kódok, melyek fordítási időben képesek adaptációra, pl. a példányosítási környezettől függő optimalizációkra. Ilyen aktív könyvtárként hoztunk létre egy **véges automata** könyvtárat [23], amely az automata formális nyelvben megadott definíciója alapján fordítási időben hozza létre az automatát, optimalizációkat (pl. redukciót) és ellenőrzéseket (pl. izolált csomópontok detektálása) végez rajta.

Mivel a C++ template metaprogramok a forráskódon kívül nem használnak más input adatot, ezért azokat determinisztikusnak szokás tekinteni. Ezt az általános felfogást kérdőjeleztük meg [81] cikkünkben, amelyben egy **véletlenszám generátort** implementálunk template metaprogramként és bemutatjuk pár alkalmazását is.

Kapcsolódó publikációim: [9, 10, 23, 26, 39, 42, 78, 79, 80, 81, 93, 94, 96, 97].

7. fejezet

Összefoglalás

Jelen dolgozatban bemutattam a C++ template metaprogramozás fontosabb kutatási területeit és az azokkal kapcsolatban elért eredményeimet.

A szakmaterület specifikus nyelvek C++ gazdanyelvbe való beágyazására különösen alkalmasak a C++ template metaprogramok, hiszen így külső eszköz nélkül, de a gazdanyelv és a beágyazott nyelv közötti szemantikus kapcsolatokat ellenőrizve történik meg a beágyazás. Ahhoz, hogy hatékonyabban és szabadabb szintaxissal lehessen megadni a beágyazott nyelveket, kifejlesztettük a Metastring és Metaparse könyvtárakat. A metaparse könyvtár önállóan is alkalmazható fordítási idejű és fordítási időben futó parszer generátor, ami bekerült a hivatalos Boost könyvtárak közé is.

A C++ template metaprogramozás a funkcionális programozási paradigma egy megjelenési formája, mégis eszközrendszerében sokáig hiányoztak a funkcionális nyelvekre jellemző elemek. Ezt pótlandó, egyrészt számos elemet megvalósítottunk, mint pl. monádokat, curryzést, algebrai adattípusokat, lusta kiértékelést, a let és do szerkezeteket, case kifejezéseket és másokat. Másrészt létrehoztunk egy beágyazott nyelvet, melyben Haskell-szerű szintaxissal lehet C++ template metaprogramokat fejleszteni.

A template metaprogramok esetében a potenciálisan hibás programok elemzése illetve debuggálása valamint profilozása nehéz feladat. Ennek megoldását kezdetben a kód instrumentálásában láttuk, majd ennek elvi korlátait belátva az LLVM/Clang fordító módosításán kezdtünk dolgozni. Az így létrehozott Templight metaprogram debugger és profiler a Clang 7.0 verzió óta a C++ fordító beépített funkcionalitása.

A dolgozat végén az előző három fő témakörön kívül több más, a C++ template metaprogramozással kapcsolatos eredményt is ismertettem.

Irodalomjegyzék

[1] Abrahams, D., Gurtovoy, A., *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond.*, Addison-Wesley, Boston, ISBN-13: 978-0321227256, 2004.

[2] Alexandrescu, A., *Modern C++ Design: Generic Programming and Design Patterns Applied.*, Addison-Wesley, Boston, ISBN-13: 978-0201704310, 2001.

[3] Allain, A., *Constexpr – Generalized Constant Expressions in C++11*, `http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html`.

[4] Andersson, L., *Parsing with Haskell*, `https://fileadmin.cs.lth.se/cs/Education/EDAN40/assignment3/parser.pdf`.

[5] ANSI/ISO C++ Committee, *The C++11 Standard, ISO International Standard, ISO/IEC 14882:2011(E)*, – Information technology – Programming languages – C++, 2011.

[6] ANSI/ISO C++ Committee, *The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014(E)*, – Programming Language C++, 2014.

[7] ANSI/ISO C++ Committee, *The C++17 Standard, ISO International Standard, ISO/IEC 14882:2017(E)*, – Programming Language C++, 2017.

[8] Austern, M. H., *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, ISBN-13: 978-0201309560, 1998.

[9] Baráth, Á., Porkoláb, Z., *Towards Safer Programming Language Constructs*, Studia Universitatis Babeş-Bolyai Informatica **LX:1**, pp. 19–34. (2015).

[10] Baráth, Á., Porkoláb, Z., *Life without implicit casts: safe type system in C++*, In: ACM (eds.) Proceedings of the 7th Balkan Conference on Informatics Conference, BCI'15, New York, Amerikai Egyesült Államok : ACM Press, (2015) Paper: 6, Pages: 4, 2015. `https://doi.org/10.1145/2801081.2801114`,

[11] Borók-Nagy, Z., Májer, V., Mihalicza, J., Pataki, N., Porkoláb, Z., *Visualization of C++ Template Metaprograms*, In: Vinju, J; Marinescu, C (eds.) 10th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010 Piscataway (NJ), USA, IEEE Computer Society, (2010) pp. 167-176., 10 p.

[12] Caro, M., *Haskell to c++ template metaprogramming translator*, 2010. `http://code.google.com/p/phaskell/w/list`.

[13] Coplien, J. O., *Multi-Paradigm Design for C++*, Addison-Wesley Longman Publishing Co., Boston, MA, USA, ISBN-13: 978-0201824674 1998.

[14] Czarnecki, K., Eisenecker, U. W., Glück, R., Vandevoorde, D., Veldhuizen, T. L., *Generative Programming and Active Libraries*, In: Jazayeri M., Loos R.G.K., Musser D.R. (eds) Generic Programming. Lecture Notes in Computer Science, vol 1766. Springer, Berlin, Heidelberg Springer-Verlag, 2000.

[15] Czarnecki, K., Eisenecker, U. W., *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, ISBN-13: 978-0201309775, 2000.

[16] Eisenecker, U. W., Blinn, F., Czarnecki, K., *A Solution to the Constructor-Problem of Mixin-Based Programming in C++*, In: First C++ Template Programming Workshop, October 2000, Erfurt.

[17] Ellis, M., Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, ISBN-13: 978-0201514599, 1990.

[18] Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J., *A Comparative Study of Language Support for Generic Programming*, Proceedings of the 18th ACM SIGPLAN OOPSLA 2003, pp. 115-134.

[19] Gil, Y., Lenz, K., *Simple and Safe SQL queries with C++ templates*, In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.

[20] Golodetz, S., *Functional programming using c++ templates (part 1)*, Overload, 81 (October 2007), `http://www.accu.org/var/uploads/journals/overload81.pdf`.

[21] Golodetz, S., *Functional programming using c++ templates (part 2)*, Overload, 82 (December 2007), `http://www.accu.org/var/uploads/journals/Overload82.pdf`.

[22] Gregor, D., Järvi, J., Siek, J. G., Dos Reis, G., Stroustrup, B., Lumsdaine, A., *Concepts: Linguistic Support for Generic Programming in C++*, In: Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.

[23] Juhász, Z., Sipos, Á., Porkoláb, Z., *Implementation of a Finite State Machine with Active Libraries in C++*, In: Ralf Lammel, Joost Visser, Joao Saraiva (Eds.): Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Lecture Notes in Computer Science 5235 Springer 2008, ISBN 978-3-540-88642-6., pp. 474-488.

[24] Karlsson, B., *Beyond the C++ Standard Library, A Introduction to Boost*, Addison-Wesley, ISBN-13: 978-0321133540, 2005.

[25] Marlow, S. (editor), *Haskell 2010 Language Report*, `https://wiki.haskell.org/Haskell`, 2010.

[26] Márton, G., Porkoláb, Z., *C++ Compile-time Reflection and Mock Objects*, Studia Universitatis Babeş-Bolyai Informatica **LIX**, pp. 5–20. (2014).

[27] McNamara, B., Smaragdakis, Y., *Functional programming in C++*, Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming, pp.118-129, 2000.

[28] McNamara, B., Smaragdakis, Y., *Functional Programming in C++ Using the fc++ Library*, SIGPLAN Notices 36, 4 (2001), 25–30, 2001.

[29] McNamara, B., Smaragdakis, Y., *Static interfaces in C++*, In: First C++ Template Programming Workshop, October 2000, Erfurt.

[30] Meyers, S., *Effective STL – 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, ISBN-13: 978-0201749625, 2001.

[31] Meyers, S., *Effective modern C++*, O'Reilly Media, ISBN 978-1-4919-0399-5, 2014.

[32] Mihalicza, J., Pataki, N., Porkoláb, Z., *Compiler support for profiling C++ template metaprograms*, In: Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11), Oct 2011, pp. 32–43, 2011.

[33] Mihalicza, J., *Analysis and Methods for Supporting Generative Metaprogramming in Large Scale C++ Projects*, PhD Disszertáció, 2014.

[34] Milewski, B., *What does haskell have to do with c++?*, 2009, `http://bartoszmilewski.wordpress.com/2009/10/21/what-does-haskell-have-to-do-with-c/`.

[35] Milewski, B., *Functional Data Structures in C++* C++Now, Aspen, CO, US, 2015, `https://www.youtube.com/watch?v=0sB09djvfl4`.

[36] Muñoz, J. M. L., *Monads in c++ template metaprogramming*, 2008, `http://bannalia.blogspot.com/2008/06/monads-in-c-template-metaprogramming.html`.

[37] Musser, D. R., Stepanov, A. A., *Algorithm-oriented Generic Libraries*, Software – Practice and Experience, 27(7) July 1994, pp. 623-642.

[38] Odersky, M., Spoon, L., Venners, B., *Programming in Scala: Updated for Scala 2.12 3rd Edition*, Artima, ISBN-13: 978-0981531687, 2016.

[39] Pataki, N., Kozsik, T., Porkoláb, Z., *Properties of C++ template metaprograms*, In: Olajos, P., Tómács, T., Kovács, E. (eds.): Proceedings of the 7th International Conference on Applied Informatics : (ICAI 2007, January 28 – 31, 2007, Eger, Magyarország : Eszterházy Károly College, (2009) pp. 265-270., 6 p.

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

[41] Persson, M. S., *Templight 2.0 - Template Instantiation Profiler and Debugger*, `https://github.com/mikael-s-persson/templight`

[42] Porkoláb, Z., Zólyomi, I., *A feature composition problem and a solution baseed on C++ template metaprogramming*, Lecture Notes in Artifical Intelligence 4143 LNCS pp. 459-470., 12p. (2006).

[43] Porkoláb, Z., Mihalicza, J., Sipos, Á., *Debugging C++ template metaprograms*, In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings, ACM 2006, ISBN 1-59593-237-2, pp. 255-264.

[44] Porkoláb, Z., Mihalicza, J., Sipos, Á., Pataki, N., *Templight, a Template Metaprogram Debugger*, Poster, European Conference on Object-Oriented Programming (ECOOP), Berlin, Németország, 2007.

[45] Porkoláb, Z., Mihalicza, J., Sipos, Á., Pataki, N., *Towards Profiling C++ Template Metaprograms*, In Ed. Zoltán Horváth. Proceedings of the 10th Symposium on Programming Languages and Software Tools: SPLST 2007. (2007) ISBN:9789634639251, pp.96–111.

[46] Porkoláb, Z., Mihalicza, J., Pataki, N., Sipos, Á., *Analysis of profiling techniques for C++ template metaprograms*, Annales Universitates Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica, 30. pp. 97-115., 19p. 2009.

[47] Porkoláb, Z., *Functional Programming with C++ Template Metaprograms*, In: Zoltán Horváth, Rinus Plasmeijer, Viktória Zsók (eds.) Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Lecture Notes in Computer Science, LNCS Vol 6299, ISBN 978-3-642-17684-5, 2010. pp.306-353.

[48] Porkoláb, Z., Sinkovics, Á., *Domain-specific language integration with compile-time parser generator library*, In: E. Visser and J. Järvi (eds): Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010.

[49] Porkoláb, Z., Sinkovics, Á, Siroki, I., *DSL in C++ Template Metaprogram*, In: Zsók V., Horváth Z., Csató L. (eds) Central European Functional Programming School. CEFP 2013, Lecture Notes in Computer Science, vol 8606. Springer, Cham, `https://doi.org/10.1007/978-3-319-15940-9_3`.

[50] Porkoláb, Z., Borók-Nagy, Z., Mihalicza, J., *Debugging and Profiling C++ Template Metaprograms*, Presentation, C++Now 2013, Aspen, Co, USA, 2013, `http://www.youtube.com/watch?v=i0bzwHBiNc4`.

[51] Porkoláb, Z., *Debugging and Profiling C++ Template Metaprograms*, presentation, MeetingC++ 2013, Düsseldorf, Germany, http://meetingcpp.com/tl_files/2013/talks/Debugging%20TMP%20-%20Porkolab.pdf.

[52] Seaton, C., *Katahdin programming language homepage*, http://www.chrisseaton.com/katahdin

[53] Siek, J., Lumsdaine, A., *Concept Checking: Binding Parametric Polymorphism in C++*, In: First C++ Template Programming Workshop, October 2000, Erfurt.

[54] Siek, J., Lumsdaine, A., *Essential Language Support for Generic Programming*, In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, New York, NY, USA, pp 73-84.

[55] Sinkovics, Á.: *Interactive Metaprogramming Shell based on Clang*, Lecture at C++Now conference, 2015 Aspen, Co. US, https://www.youtube.com/watch?v=oCbeXpJKzlM

[56] Sinkovics, Á., Porkoláb, Z., *Metaparse – Compile-time parsing with template metaprogramming*, Presentation on C++Nw conference, Aspen, Co, USA, 2012. http://2012.cppnow.org/session/metaparse-compile-time-parsing-with-template-metaprogramming.

[57] Sinkovics, Á., *The User Manual of mpllibs metaparse*, http://abel.web.elte.hu/mpllibs/metaparse/manual.htm.

[58] Sinkovics, Á., *Interactive Metaprogrmaming Shell based on Clang*, Presentation at C++Now, Aspen, Colorado, 15th May, 2014, https://github.com/boostcon/cppnow_presentations_2014/blob/master/files/2014_cppnow_metashell.pdf?raw=true.

[59] Sinkovics, Á., *Compile-time tools supporting generic programming in C++*, Presntation at CppCon, Bellevue, Washington, 24th Sept, 2015, https://github.com/CppCon/CppCon2015/raw/master/Presentations/Compile-time%20tools%20for%20generic%20programming%20in%20C%2B%2B/Compile-time%20tools%20for%20generic%20programming%20in%20C%2B%2B%20-%20Abel%20Sinkovics%20-%20CppCon%202015.pdf.

[60] Sinkovics, Á., *The homepage of Metashell*, http://metashell.org/.

[61] Sinkovics, Á., *Boost Metaparse library homepage*, `https://www.boost.org/doc/libs/1_72_0/doc/html/metaparse.html`.

[62] Sinkovics, Á., Porkoláb, Z., *Expressing C++ Template Metaprograms as Lambda Expressions*, In: Zoltán Horváth, Viktória Zsók, Peter Achten, Pieter Koopman (eds): Proceedings of Tenth Symposium on Trends in Functional Programming, Komárno, Slovakia, 2-4 June 2009, pp. 97-111.

[63] Sinkovics, Á., *Functional extensions to the boost metaprogram library*, Electr. Notes Theor. Comput. Sci. 264, 5 (2010), 85–101.

[64] Sinkovics, Á., *Nested lamda expressions with let expressions in C++ template metaprorgams*, In: Porkoláb, Z., Pataki, N., (eds.) WGT'11 (2011), vol. III of WGT Proceedings, pp. 63–76., 2011.

[65] Sinkovics, Á., Sajó, E., Porkoláb, Z., *Towards more reliable C++ template metaprograms*, In: Jaan Penjam, 12th Symposium on Programming Languages and Software Tools (SPLST'11), Tallinn, Estonia, 5-7 October, 2011., pp. 260-271.

[66] Sinkovics, Á., Abrahams, D., *Using strings in C++ template metaprograms, 2012*, `http://cpp-next.com/archive/2012/10/using-strings-in-c-template-metaprograms/`.

[67] Sinkovics, Á., Porkoláb, Z., *Implementing monads for C++ template metaprograms*, In Science of Computer Programming, Available online 23 January 2013, ISSN 0167-6423, 10.1016/j.scico.2013.01.002, `http://www.sciencedirect.com/science/article/pii/S0167642313000051`.

[68] Sinkovics, Á, Porkoláb, Z., *Domain-Specific Language Integration with C++ Template Metaprogramming*, In: Marjan Mernik (Ed), Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013. 33-56. Web. 30 Apr. 2014. ISBN 978-1-4666-2092-6 (hardcover) – ISBN 978-1-4666-2093-3 (ebook), doi:10.4018/978-1-4666-2092-6.ch002.

[69] Sipos, Á., Porkoláb, Z., Zsók, V., *Meta<fun> – Towards a functional-style interface for C++ template metaprograms*, In Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII, 2008/2, Cluj-Napoca, 2008, pp. 55-66.

[70] Sipos, Á., *Effective development of C++ Template Metaprograms*, PhD thesis. Eötvös Loránd University, Budapest, Hungary, 2009.

[71] Sommerlad, P., *C++14 Compile-time computation*, (ACCU 2015), `http://wiki.hsr.ch/PeterSommerlad/files/ACCU2015VariadicVariableTemplates.pdf`.

[72] Stallman, R. M., Pesch, R. H., *Using GDB: A Guide to the GNU Source-level Debugger: GDB Version 4.0*, July 1991. Free software foundation, 1991.

[73] Stepanov, A., *From Mathematics to Generic Programming; 1st edition*, Addison-Wesley, ISBN-13: 978-0321942043, 2014.

[74] Stroustrup, B., *The C++ Programming Language, 4th Edition*, Addison-Wesley, ISBN-13: 978-0321563842, 2013.

[75] Bjarne Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, ISBN-13: 978-0201543308. 1994.

[76] Dos Reis, G., Stroustrup, B., *Specifying C++ concepts*, In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006: pp. 295-308.

[77] Szűgyi, Z., Sinkovics, Á., Porkoláb, Z., *C++ metastring library and its applications*, In GTTSE (2009), J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, (eds.): vol. 6491 of Lecture Notes in Computer Science, Springer, pp. 461–480. 2009.

[78] Szűgyi, Z., Sipos, Á., Porkoláb, Z., *Towards the Modularization of C++ Concept Maps*, Electronic Notes In Theoretical Computer Science 238: 2 pp. 71-82., 12p. 2009. `https://doi.org/10.1016/j.entcs.2009.05.008`.

[79] Szűgyi, Z., Pataki, N., Mihalicza, J., Porkoláb, Z., *C++ Method Utilities*, In Kollar, J.; Novitzka, V. (eds.): Informatics 2009: Proceedings of the 10th International Conference on Informatics (2009) pp. 112-117., 6 p.

[80] Szűgyi, Z., Pataki, N., Porkoláb, Z., *Towards More Scalable C++ Concept Maps*, In Penjam, Jaan (eds.): Proceedings of 12th Symposium on Programming Languages and Software Tools, SPLST'11, Tallinn, Estonia, Tallinn University of Technology, (2011) pp. 8-19. 2011.

[81] Szűgyi, Z., Cséri, T., Porkoláb, Z., *Random number generator for C++ template metaprograms*, In Kiss, Ákos (ed.): Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13, Szeged, Hungary, University of Szeged, (2013) pp. 156-169., 2013.

[82] Torgersen, M., Hansen, C. P., Ernst, E., Ahe, P., Bracha, G., Gafter, N., *Adding Wildcards to the Java Programming Language*, In: Proceedings of the 2004 ACM Symposium on Applied Computing (SAC) 2004, pp. 1289-1296.

[83] Unruh, E., *Prime number computation*, ANSI X3J16-94-0075/ISO WG21-462.

[84] Ushey, K., *Debugging with LLDB*, `http://kevinushey.github.io/blog/2015/04/13/debugging-with-lldb/`.

[85] Vandevoorde, D., Josuttis, N. M., Gregor, D., *C++ Templates: The Complete Guide, 2nd edition*, Addison-Wesley, ISBN-13: 978-0-321-71412-1, 2017.

[86] Veldhuizen, T., *C++ Templates are Turing Complete*, Technical Report, Indiana University Computer Science, 2003, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670`.

[87] Veldhuizen, T., L., Dennis Gannon, D., *Active libraries: Rethinking the roles of compilers and libraries*, In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23.

[88] Veldhuizen, T., *Five compilation models for C++ templates*, In First Workshop on C++ Template Metaprogramming, October 2000.

[89] Veldhuizen, T., *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[90] Veldhuizen, T., *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[91] Visser, E., *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9.*, In C. Lengauer (eds): Domain-Specific Program Generation, vol. 3016 of Lecture Notes in Computer Science, 2004, pp. 216–238. Spinger-Verlag, June 2004.

[92] Zalewski, M., Priesnitz, A. P., Ionescu, C., Botta, N., Schupp, S., *Multi-language library development: From Haskell type classes to C++ concepts*. In: MPOOL 2007 Ecoop workshop reader, 2007.

[93] Zólyomi, I., Porkoláb, Z., Kozsik, T., *An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming*, In

Pfenning F., Smaragdakis Y. (eds.): Generative Programming and Component Engineering. GPCE 2003. Lecture Notes in Computer Science, vol 2830. Springer, Berlin, Heidelberg, pp. 209-227, 2003.

[94] Zólyomi, I., Zoltán Porkoláb, Z., *Towards a General Template Introspection Library*, In Karsai G., Visser E. (eds): Generative Programming and Component Engineering. GPCE 2004. Lecture Notes in Computer Science, vol 3286. Springer, Berlin, Heidelberg, pp.266-282, 2004. https://doi.org/10.1007/978-3-540-30175-2_14.

[95] Zsók, V., Koopman, P., Plasmeijer, R., *Generic executable semantics for D-clean*, Electronic Notes in Theoretical Computer Science, 279(3), pp.85-95., 2011.

[96] Zsók, V., Porkoláb, Z., *Rapid Prototyping for Distributed D-Clean using C++ Templates*, Annales Universitates Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica, 37. pp. 19-46., 28p. 2012.

[97] Zsók, V., Porkoláb, Z., *The Distributed D-Clean Model Revisited by Templates*, AIP Conference Proceedings 1389 : Vol A pp. 877-880. , 4 p. (2011) https://doi.org/10.1063/1.3636873

[98] *The Blitz++ library*, http://www.oonumerics.org/blitz.

[99] *The Boost Concept checking library*, http://www.boost.org/libs/concept_check/concept_check.htm.

[100] *The Boost Xpressive regular library*, http://www.boost.org/doc/libs/1_40_0/doc/html/xpressive.html.

[101] *The Boost move library*, https://www.boost.org/doc/libs/1_63_0/doc/html/move.html.

[102] *The Boost Metaprogramming library*, http://www.boost.org/libs/mpl/doc/index.html.

[103] *The Boost Lambda library*, http://www.boost.org/doc/libs/1_40_0/doc/html/lambda.html.

[104] *The Boost Proto library*, http://www.boost.org/doc/libs/1_37_0/doc/html/proto.html.

[105] *The Clang C++ Compiler*, https://clang.llvm.org/