Hatékony típusparaméterek Java-ban

TDK-dolgozat

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

PROGRAMOZÁSI NYELVEK ÉS FORDÍTÓPROGRAMOK TANSZÉK

Szerző:

Soha Péter

programtervező informatikus MSc

2. évfolyam

Témavezető:

Pataki Norbert

egyetemi adjunktus

Budapest, 2020

Tartalomjegyzék

1. fejezet

Bevezetés

Ebben a dolgozatban a típusparaméterek és a generikus programozás témakörét tekintem át. Megnézem, hogy egyes programozási nyelvekben miként jelenik meg a többalakúság és az absztrakt szerkezetekre épülő programozás módszere. Azt a kérdést vizsgálom, hogyan lehet hatékonyabbá tenni a típusparaméterek kezelését, ezzel optimalizálva a kódot és hatékonyabbá téve a futást.

A programozási nyelvek különbözőek. Eltérnek a szintaxisban, szemantikában és az implementáció szintjén is. Ezt azt jelenti, hogy egy nyelvi elem, noha látszólag hasonlóan van megvalósítva két nyelvben, a működés még így is teljesen eltérő lehet. A típusparaméterezés egy sarkalatos pontja a modern programozási nyelveknek, melyek támogatják a típusrendszereket és a típusok absztrakcióját. Az egyes nyelvek a típusparaméterezést eltérő módon értelmezik. Példának okáért a C++ nyelvben bevezették a *template* fogalmát, amelyben placeholderekkel adhatjuk meg azokat a típusokat, amiket a compiler fordítási időben behelyettesít. Ezzel szemben a Java ugyanerre a problémára a generikusokat adja megoldásul, mely az előbbivel szemben fordítási időben típustörlés módszerének alkalmazásával több futási idejű konstrukcióra bízza a paraméterezhetőséget (hiszen az itt öröklés által van kifejezve).

A második fejezetben áttekintem, hogy különböző paradigmákat használó nyelvekben hogyan jelenik meg a generikusság és a polimorfizmus. Megnézem, miből fejlődött ki a *Java Generics*, milyen megoldásokat használtak előtte a programozók. Ezt összevetem a C++ nyújtotta *template*-ekkel, külön kitérve a *Template Metaprogramozás* paradigmájára,

mely számtalan módon segíti elő a kód fordításidejű optimalizálását és amely paradigma mind a mai napig hiányzik a Java eszköztárából. Rövid példákon keresztül megmutatom még az Ada és Clean nyelveket is, melyek a maguk módján egyedi megoldásokkal gazdagítják a generikus paradigmát. Előbbinek a legfigyelemreméltóbb tulajdonsága a generikusság szerződésalapú megközelítése, melynek hatása a későbbi nyelvekben is megjelent, utóbbi pedig a generikusságot és a mintaillesztést ötvözi, ezzel egy olyan egyedi szemléletmódot alkotva, amelynek elemeit átültetve más nyelvekbe, hatékony és könnyen alkalmazható konstrukciót kapnánk.

A harmadik fejezet a saját kutatási eredményeket foglalja össze. Bemutatásra kerül a *Java Template* konstrukciója, amellyel a C++ *template*-jeihez hasonló statikus kód írására nyílik lehetőség. Ezt a megoldást továbbgondolva áttekintem annak lehetőségét, hogy új elemek helyett közvetlenül a generikus osztályok példányosítása fordítási időben hogyan növeli a hatékonyságot, ha a feladat szempontjából ez előnyösebb, mint a típusok futás közbeni kikövetkeztetése.

A negyedik fejezetben a korábbi publikációim mérésein keresztül bemutatom, mekkora sebességnövekedést tudtam elérni a példányosított osztályok használatával a generikus verziókkal szemben. Kitérek arra is, miként hat fizikai eszköz a teljesítményre.

Az ötödik fejezet a fejlesztési lehetőségekről szól, kiemelve egy érdekes problémát, amely a kutatás során merült fel és amelynek megoldása elengedhetetlen a template konstrukció hatékony használatához a Java-ban.

A hatodik fejezetben összefoglalom a dolgozat során vizsgált témaköröket és a feltárt problémákat, valamint az eddig elért eredményeket.

2. fejezet

Típussal történő paraméterezés programozási nyelvekben

Maga a generikus paradigma azt a célt szolgálja, hogy egy problémára absztrakt, könnyen használható megoldásokat adhassunk, legyen az egy konténertípus implementálása, vagy programkönyvtárak fejlesztése. Egy ilyen probléma megoldásánál az egyik legfontosabb kikötés az, hogy ugyanaz a kód több, már korábban implementált (vagy beépített) típussal is kompatibilis legyen, esetleg típustól függően másképp hajtódjon végre. Hogy ezt elérjük, a generikus paradigmába bevezetésre került a *típussal történő paraméterezés*.
Ebben a fejezetben példákon keresztül mutatom meg, hogyan jelenik meg a generikusság a programozási nyelvekben.

2.1. Generikus programozás Javaban

A generikus elemek a Java 1.5-ös verziójában jelentek meg. Célja nem volt más, mint javítani a megírt kód minőségén, valamint elősegíteni az újrafelhasználást és a típusbiztonságot. Ugyanis az 1.5 előtt, ha általános (vagyis több típusra is működő, absztrakt) megoldásra volt szükség, az az alábbi módon volt definiálva:

```
1  //Non-Generic Stack implementation
2  class Stack {
3    private Object[] elements;
4    private int c = 0;
5    public Stack(int size){elements = new Object[size];}
```

```
6     public void push(Object item) {
7       if(c < elements.length)
8         elements[c++] = item;
9       else
10        throw new ArrayIndexOutOfBoundsException();
11    }
12    public Object pop() {
13      if(c > 0)
14        return elements[c--];
15      else
16        throw new ArrayIndexOutOfBoundsException();
17    }
18  }
```

2.1. forráskód. Legacy verem példa

Látható itt, hogy az elemeket `Object`-ek tömbjeként ábrázoljuk. Ez egyfelől lehetővé teszi azt, hogy eltérő típusú objektumok tárolásához nem kell az adott típusra megírni a kódot, ugyanakkor ebben rejlik a konstrukció egyik legnagyobb sebezhetősége. Ehhez tekintsük az alábbi kódrészletet:

```
1     class Example{
2       public static void main(String[] args){
3         Stack st = new Stack(10);
4         st.push(new Integer(1024));
5         st.push(new String("Almafa"));
6         st.push(100);
7
8         String x = (String)st.pop();
9       }
10    }
```

2.2. forráskód. Legacy verem használata

Ebben a példában az elemek berakásakor semmilyen típusbeli megkötést nem tettünk, így bármilyen objektum berakható ugyanabba a verembe, hiszen az `Object` mindegyiknek az őse. Az elemek kivételekor expliciten kell kasztolnunk a veremtetőt a kívánt típusra, de mivel a jobb és a balérték típusát a fordító nem fogja előzetesen ellenőrizni, így (a példában) az alábbi futásidejű hibát fogjuk kapni (hiszen a kivett elem típusa `Integer`, míg a referenciáé `String`):

Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String at example.Example.main

2.1.1. Java generikusok

Ezen probléma kiküszöbölésére kerültek be a generikus elemek a Java nyelvtanába[1]. A 2.1-ben látott verem típus generikus változata a következő:

```
1  //Generic Stack implementation
2  class Stack<T> {
3    private Object[] elements;
4    private int c = 0;
5    public Stack(int size){elements = new Object[size];}
6    public void push(T item) {
7      if(c < elements.length)
8        elements[c++] = item;
9      else
10       throw new ArrayIndexOutOfBoundsException();
11   }
12   public T pop() {
13     if(c > 0)
14       return (T)elements[c--];
15     else
16        throw new ArrayIndexOutOfBoundsException();
17   }
18 }
```

2.3. forráskód. Generikus verem példa

Ez a megoldás szintén képes bármilyen a Java nyelvben szabványos elem tárolására anélkül, hogy a kódot meg kéne változtatni. Azonban egy fontos különbség megfigyelhető. A `push` művelet az eddigi `Object` helyett immár egy `T` típusú paramétert vár. Ez elsőre talán nem is tűnik akkora változásnak, ugyanakkor a különbség jelentős. Hogy ezt elérje, a generikusok fordítása a Java-ban a következőképpen történik:

A generikusok működése

- T minden előfordulását helyettesíti `Object`-tel (vagy a bounded class esetén a bound típusával)

- A típusbiztonság garantálása érdekében a fordító típuskasztolásokat illeszt a kódba
- A fordító bridge metóduskat generál, hogy biztosítsa a polimorfizmust

Ennek illusztrálására tekintsük a következő példát, amiben egy egyszerű generikus wrappert implementáltam, illetve egy nem-generikus leszármazottját:

```
1   class Box<T>{
2     public T item;
3
4     public Box(T item){this.item = item;}
5
6     public void setItem(T item){
7       this.item = item;
8     }
9   }
10
11  class BoolBox extends Box<Boolean>{
12    public BoolBox(Boolean item){super(item);}
13
14    public void setItem(Boolean item){
15      super.setItem(item);
16    }
17  }
```

2.4. forráskód. Bridge metódus példa 1

A típustörlés során a fordító a *Box* osztályból előállít egy olyan változatot, ahol a `T` minden előfordulását `Object`-re cseréli. Hogy biztosítsa a kompatibilitást és a polimorfizmust, a leszármazott osztályt az alábbi *bridge metódus*sal egészíti ki:

```
1  class BoolBox extends Box{
2    //...
3
4    //Bridge method
5    public void setItem(Object item){
6      setItem((Boolean)item);
7    }
8    public void setItem(Boolean item){
9      super.setItem(item);
10   }
```

```
11  }
```

2.5. forráskód. Bridge metódus példa 2

Itt a típustörlés után a függvényszignatúra eltérő lesz, mivel a `BoolBox`-ban egy `setItem(Boolean)`, míg a `Box`-ban egy `setItem(Object)` függvény van, így a leszármazott osztály függvénye túlterheli és nem felülírja az ősbelit. Ennek kiküszöbölésére a fordító egy *bridge metódus*t generál a kódba, ezzel biztosítva, hogy az altípusosság a megfelelő módon működjön.

Korlátos típusparaméterek

Bizonyos feladatokhoz szükség lehet arra, hogy a *tetszőleges* típus fogalmát (valamelyik irányba) leszűkítsük. Ilyenkor beszélhetünk a korlátos típus (*bounded type parameter*) fogalmáról, amely a Java-ban lehet felülről (*upper bounded type*), vagy (*wildcard*ok esetében) alulról is korlátos (*lower bounded type*).

Wildcardok

Tipikusan a kollekciókkal kapcsolatban léteznek olyan műveletek, amikor nem csak egy adott típust szeretnénk használni, hanem annak tetszőleges altípusait is. Ilyen helyzetekben hasznos eszköz a *wildcard* típusparaméter, amit egy kérdőjel karakterrel jelölünk (?). A wildcard előnye a generikus típusparaméterrel szemben, hogy alulról is korlátozható az elfogadott típusok halmaza a `super` kulcsszó segítségével. Tekintsük az alábbi példát, ami a `java.util.Collection<E>` interfész néhány függvényét tartalmazza:

```
1   interface Collection<E>{
2     ...
3     boolean removeAll(Collection<?> c);
4     boolean addAdd(Collection<? extends E> c);
5     boolean removeIf(Predicate<? super E> filter);
6     ...
7   }
```

2.6. forráskód. Korlátos típusparaméterek

Látható, hogy mindhárom függvény más kontextusban használja a *wildcard*-okat. Amíg a `removeAll` tetszőleges típusú elemek gyűjteményével tud dolgozni, addig az `addAll` felülről korlátozza a típusok halmazát (mivel olvassa a paraméter elemeit), a `removeIf`

pedig olyan predikátumot vár ami E-t, vagy annak valamelyik ősét tudja boolean-ná kiértékelni. Azt, hogy melyik változatot mikor használják az úgynevezett *get and put principle* foglalja össze:

Használjunk *felülről korlátos wildcard*-ot (extends), ha csak olvasunk elemeket egy gyűjteményből (hiszen így garantáljuk, hogy minden elem konvertálható T-re) és *alulról korlátosat* (super), ha csak beírunk elemeket (mivel így biztosítható, hogy minden elem típusa kompatibilis a kollekcióban tárolható típusokkal), de ugyanazon a konténeren sose használjuk egyszerre őket.

Ennek illusztrálására remek példa a java.util.Collections osztály copy metódusa (a példa kedvéért csak a szignatúráját írom le):

```
1    public static <T> void copy(
2                   List<? super T> dest,
3                   List<? extends T> src
4                    )
```

2.7. forráskód. Get and put principle példa

Mivel a forrás elemeit csak kivesszük, így itt minden olyan típus megengedett, ami *legalább* T, ezzel szemben a cél gyűjteménybe beleteszünk elemeket, így itt *legfeljebb* T támogatott.

2.2. C++ sablonok

A generikus paradigma támogatására a C++ nyelvben az osztály- és függvénysablonokat (template) használjuk. Segítségével lehetőség van típusok paraméterként való használatára[2]. A C++ a típusparamétereknél még azt is megengedi (szemben a Java-val), hogy a sablon csak a típusok felhasznált tulajdonságaitól függjön, illetve nem tesz kikötést arra se, hogy hierarchikus viszonyban kell állniuk egymással.

A template osztályokat hasonlóan kell deklarálni és definiálni, mint a hagyományos C++ osztályokat, ugyanakkor az osztályon kívül definiált tagoknál kötelező jelölni, hogy template-ről van szó:[3]

```
1    class Stack_is_full{};
2    class Empty_stack{};
3
4    template<class E, int max>
```

```
5     class Stack
6     {
7       E elements[max];
8       int top;
9
10      public:
11        Stack():top(0){}
12
13        void push(const E& element)
14        {
15          if(top < max){ elements[top++]=element; }
16          else{ throw Stack_is_full(); }
17        }
18
19        E pop(){...}
20        E top() const {...}
21        bool is_empty() const {...}
22    };
23    template<class E, int max>
24    bool Stack<class E, max>::is_full() const {...}
```

2.8. forráskód. Template verem példa

2.2.1. Template specializáció

A template-ek célja egy általános megoldás adása, amely független az aktuális típusoktól, ugyanakkor lehetnek olyan helyzetek, amikor szükség van, hogy bizonyos típusokra eltérő kódot generáljon a fordító. Erre jelent megoldást a *template specializáció*[4], amely azt fejezi ki, hogy egy template-hez többféle definíciót társítunk, így a fordító ennek segítségével tudja meghatározni a legjobban illeszkedő változatot.[3]

```
1     template<class E>
2     class Stack
3     {
4       E* elements;
5       unsigned int size;
6
7     public:
8       void push(const E& element){...}
```

```
9       E pop(){...}
10
11      Stack& operator=(const Stack& other){...}
12    };
13
14    template<>
15    class Stack<bool>
16    {
17      //...
18    };
19
20    Stack<bool>x;
21    Stack<string>y;
22    //...
```

2.9. forráskód. Template specializáció

Itt a fordító az `x` objektumnál a `bool` specializációt használja, míg az `y`-nál az általános sablont[5].

Teljes template specializáció

A sablonok esetén alap esetben lemásolásra kerül a template kódja, ami azt eredményezheti, hogy a lefordított kód fel fog duzzadni (noha egyébként a végrehajtás szempontjából kifejezetten előnyös). Ennek megelőzésére vezették be a *teljes template specializáció*t. A pointereket tartalmazó tárolók osztozhatnak egyetlen implementáción is, amit úgy érünk el, hogy készítünk egy specializációt `void*`-ra. Ezek a polimorfizmus megőrzése végett pointertípusokra fognak konvertálódni:[3]

```
1     template<>class Stack<void*>
2     {
3       public:
4         void** elements;
5         //...
6     };
```

2.10. forráskód. Teljes template specializáció

Ezt a változatot nevezzük *teljes template specializáció*nak, mivel ebben az esetben nincs paraméter, amit meg kéne adni, vagy le kéne vezetni.

Részleges template specializáció

Lehet olyan eset, amikor arra van szükségünk, hogy egy specializáció csak és kizárólag pointereket tároló `Stack` esetén legyen használva, akkor hasznos eszköz lehet a *részleges template specializáció*:[3]

```
1   template<class E>
2   class Stack<E*> : private Stack<void*>
3   {
4     public:
5       typedef Stack<void*>Base;
6
7       Stack():Base(){}
8       explicit Stack(int i):Base(i){}
9
10      E*& elem(int i) { return reinterpret_cast<E*&>(Base::
            elements(i)); }
11      E*& operator[ ](int i) { return reinterpret_cast<E*&>(Base
            ::operator[](i)); }
12      //...
13  };
```

2.11. forráskód. Részleges template specializáció

C++ Template Metaprogramozás

A template-ek példányosításakor lehetőségünk van algoritmusok végrehajtására is és az ilyen programokat nevezzük *template metaprogram*oknak. A konstrukció egyik előnye, hogy eszközt biztosít a fordításidejű kódmanipulációra úgy, mint kódtranszformálás, vagy optimálisabb kód generálása anélkül, hogy magához a fordítóhoz hozzá kelljen nyúlni[6].

Tekintsük az alábbi kódrészletet, amely egy faktoriálist kiszámító metaprogram. Az első template egy rekurzív példányosítás, amelynek értéke éppen N!. Ahogyan egy hagyományos módon megírt rekurziónál, itt is szükség van egy megállási feltételre, ami jelen esetben egy template specializáció az 1 értékre.

A végrehajtás során a fordító meghatározza a `main` függvénybeli kifejezést, ami a `Factorial<5>::value`. Ehhez meg kell határozni a `Factorial<5>` típust, amihez megnézi, van-e definiált template. Mivel létezik ilyen (a legelső a példában), így ennek használatával `N` helyébe beírva az 5 értéket, lefordítja a kódot. Ennek során el fog jutni a

`Factorial<N-1>::value` kifejezésig, aminek a típusa itt most `Factorial<4>`. Ezt hasonlóan kezeli az előzőhöz és a rekurzív kiértékelés során eléri a `Factorial<1>`-et is, amire már létezik specializáció, így a fordító ezt fogja használni.

```
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;
}
```

2.12. forráskód. Template metaprogram

A fordítás eredménye az egyes `Factorial` példányok `value` értékeinek szorzata, ami jelen esetben 5!, vagyis 120[7].

Megszorítások a template paramétereken

A C++ 2020-as szabványába bekerülő egyik lehetőség a *constraint*-ek és *concept*-ek használata, amelyek eszközöket biztosítanak arra, hogy feltételek mentén szűkítsük a template által elfogadott típusok halmazát. A mögöttes elv nagyban hasonlít a Java nyelvben is látott *bounded type parameter*-re, ahol az `extends` kulcsszó után felsorolt típusokkal lehetőség van meghatározni, milyen típusok kompatibilisek a sajátunkkal. Ugyanakkor megemlítendő, hogy bár a logika a Java megoldásához hasonló, a megvalósítás, miszerint feltételeket (*szerződéseket*) fogalmazunk meg, inkább az Ada nyelv *with function* szerkezetével mutat rokonságot.

A *concept* egy nevesített, `bool` predikátum, amellyel megkötéseket (*constraints*) fogalmazhatunk meg egy template argumentumra[8].

```
#include <string>
#include <cstddef>
#include <concepts>
using namespace std::literals;

template<typename T>
concept Hashable = requires(T a) {
  { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

struct meow {};

template<Hashable T>
void f(T);

int main() {
  f("abc"s); // OK
  f(meow{}); // Error
}
```

2.13. forráskód. Concept példa

A *constraint* egy logikai feltétel, amely fordítási időben kiértékelődik egy típusra (az értéke egy `bool`, azaz a típus megfelel-e a megfogalmazott feltételnek, vagy sem). Egy *constraint* szerepelhet egy *concept* törzsében, de önmagában is használható[9].

```
template<Incrementable T>
void f(T) requires Decrementable<T>;

template<Incrementable T>
void g(T) requires Decrementable<T>;
```

2.14. forráskód. Constraint példa

2.3. Generikus típusok Ada-ban

Az Ada nyelv generikus megoldása több szempontból is eltérő a C++-ban látott template-ektől. A 2.15 példa egy verem típus leírása, amit a `generic` kulcsszó vezet be. Az ezt követő `type Element is private` deklaráció jelzi, hogy a típusparamé-

ter az `Element` lesz, amire a nyelv megköveteli, hogy legyen rajta egyenlőségvizsgálat és értékadás művelet értelmezve. Ezen felül a következő sorba látható `with function "<" (X,Y: in Element) return Boolean is <>;` miatt csak rendezhető típusok jöhetnek szóba. Ez a rendezés lehet egy sablonparaméter, vagy az `Element` kisebb művelete.

Ez egyben azt jelenti (szemben a C++-szal, ahol a template argumentumairól semmilyen információval nem rendelkezünk előre), hogy az `Element`-re kikötéseket fogalmazunk meg, amivel biztosítjuk, hogy az aktuális típus megfeleljen az implicit követelményeknek. Ilyen követelmény az is (a példánál maradva), hogy a típuson legyen megvalósítva valamilyen rendezés.

Ezt a megoldást nevezzük az Ada-ban *sablon-szerződés modell*-nek[10]. Ezzel előzetes megkötések tehetők a template argumentumokra, amelyek nem teljesülése esetén a fordítóprogram nem engedi a kód lefordítását (akkor is, ha a szerződést be nem tartó függvényt meg se hívtuk).[7]

```
1   generic
2     type Element is private;
3     with function "<" (X,Y: in Element) return Boolean is <>;
4   package Stacks is
5
6     type Stack is limited private;
7
8     procedure Push (V : in out Stack; X : in Element);
9     procedure Pop (V : in out Stack; E : Element);
10
11    function Top (V : Stack) return Element;
12    function Is_Empty(V : Stack) return Boolean;
13    function Is_Full(V : Stack) return Boolean;
14
15    Empty, Full : exception;
16    ...
17  end Stacks;
```

2.15. forráskód. Ada generikus interfész

2.4. Clean

A generikusság nem csak a multiparadigmás és OOP nyelvek sajátja, a Clean-ben ugyanúgy lehetőség van a használatára, bár az előbbiektől kissé eltérő módon. Legyen a példa az alábbi két típus.

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

2.16. forráskód. Clean példa

A `List` elemei az üres lista, illetőleg a `Cons` által generált *head-tail* pár, amik 'a' típusú elemeket tartalmaznak. A `Tree`-ben (szintén 'a' típusú) leveleket (`Leaf`), valamint részfákat (`Node`) tárolunk. Megfigyelhető, hogy a generikusság a mintaillesztés egy következménye, hiszen az 'a' paraméter helyén tetszőleges elemi típus lehetne.
Definiáljunk erre a két struktúrára egy egyváltozós-egyértékű map függvényt az elemenkénti feldolgozáshoz.

```
1    fmaplist :: (a -> b) (List a) -> (List b)
2    fmaplist f Nil = Nil
3    fmaplist f (Cons x xs) = Cons (f x) : Cons (fmaplist f xs)
4
5    fmaptree :: (a -> b) (Tree a) -> (Tree b)
6    fmaptree f (Leaf a) = Leaf (f x)
7    fmaptree f (Node l r) = Node (fmaptree f l) (fmaptree f r)
```

2.17. forráskód. Clean map

A tény, hogy a két típus felépítése differens, maga után vonja az implementációk eltérését is. Azonban ugyanazzal a céllal lettek létrehozva (elemenkénti feldolgozás), így a kód méretét csökkentendő, érdemes volna egy függvénnyé összevonni. Ehhez használhatóak a Clean generikus alapelemei, a *prelude*-ok, melyek az alábbiak:

```
1    :: UNIT = UNIT
2    :: PAIR a b = PAIR a b
3    :: EITHER a b = LEFT a | RIGHT b
```

2.18. forráskód. Clean prelude

Ezeket használva tetszőleges szerkezet definiálható, így a példában használtak is:

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

2.19. forráskód. Clean prelude adatszerkezetek

A típustörlés után már egységesen tudjuk kezelni a generikus adattípusokat, így egyetlen függvény is el tudja látni azt a feladatot, amit korábban kettővel oldottunk meg. Tekintsük tehát a generikus `map` függvényt:

```
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)
```

2.20. forráskód. Clean generic map

Ennek felhasználásával (a szükségszerűen definiált generikusra át- és visszaalakító függvények segítségével) már sokkal egyszerűbben használhatjuk az adatszerkezeteket.[7]

3. fejezet

Sablonok Javában

A generikusok és maga a Java felépítése miatt a paraméteres típusok megvalósítása nem minden esetben optimális. Ezen okok egyike a dinamikus típusosság, vagyis, hogy egy paraméter aktuális típusa függhet futási idejű adatoktól. Ha pontosan meg tudjuk határozni a típusok azon halmazát, amivel egy adott esetben dolgozni fogunk érdemesebb lehet előre legenerálni az ezzel (statikusan) típusozott kódot. Erre megoldásul két lehetőséget is definiáltam. Az első a *Java Template*, amely nagyban merít a C++-os sablonokból, de ezzel együtt több nem Java-szabványos elemet használ, így az integrációja egy valós környezetbe több munkát kíván. Erre definiáltam egy megoldást, amivel közvetlenül a generikus osztályok példányosíthatóak és nincs szükség semmilyen új elemre a kódban.
Ugyanakkor az a probléma továbbra is fennállt, hogy a Java "bőbeszédűsége" miatt a kézzel írt kód elég sok boilerplate-et tartalmaz[11]. Ennek kiküszöbölésére több könyvtár is létezik, mint pl. a Project Lombok[12], amely úgy próbál javítani a kódminőségen, hogy annotációk használatával generálja le a metódusokat, viszont ez a megoldás nem képes template-ek alapján hasonló struktúrájú osztályokat generálni. Nézzük az alábbi példakódot:

```
1     class Car{
2       private String manufacturer;
3       private int age;
4
5       public String getManufacturer(){
6         return this.manufacturer;
7       }
8
```

```
9       public int getAge(){
10        return this.age;
11      }
12
13      public void setManufacturer(String manufacturer){
14        this.manufacturer = manufacturer;
15      }
16
17      public void setAge(int age){
18        this.age = age;
19      }
20
21    }
```

3.1. forráskód. Java kód Lombok nélkül

```
1     @Getter
2     @Setter
3     class Car{
4       private String manufacturer;
5       private int age;
6     }
```

3.2. forráskód. Lombok-annotációkkal kiegészített osztály

Látható, hogy a második osztály jóval rövidebb és ennek okán átláthatóbb lett. Ezt úgy éri el a Lombok, hogy a Getter/Setter annotációk feldolgozásakor automatikusan belegenerálja a getter/setter metódusokat az adattagokhoz (final tagok esetén a settereket kihagyja), melyek viselkedése ugyanolyan lesz, mintha csak kézzel írtuk volna.

3.1. A Java Template

A *Java Template* megalkotásához a fő motivációt az jelentette, hogy a generikus típusok, bár sokkal egyszerűbbé teszik az újrafelhasználható kód írását bizonyos esetekben túl nagy plusz költséget követelnek minimális nyereség eléréséhez.

```
1     //Stack template
2     template(T)
3     class Stack {
4       private T[] elements;
```

```
5       //...
6       public void push(T item) { elements[c++] = item; }
7       public T pop() { return elements[c--]; }
8     }
```

3.3. forráskód. Java template

A `template` kidolgozásakor a fő szempontom az volt, hogy a C++-os sablonhoz hasonló konstrukciót tegyek elérhetővé. Ehhez deklaráltam a `template` kulcsszót, amely után (hasonlóan a C++-hoz) megadhatók az adott scope-ban használható típushelyettesítők vesszővel elválasztott sorozata. Ezek a placeholderek a fordítás előtt az aktuális típussal lesznek helyettesítve (részletesen lásd később). Az esetleges névütközések elkerülése végett az eszköz ezen típusok azonosítóját használja, hogy egy egyedi csomagnevet generálhasson az osztálynak. [13]

3.1.1. Template paraméterek és csomagnevek

Hogy biztosíthassam az elérési út egyediségét, az alábbi megkötéseket tettem a típusokra (legyen T egy tetszőleges java-szabványos típusnév):

- Ha T egy primitív típus, vagy literál, akkor nincs megkötés
- Ha T egy objektumtípus, akkor elvárt a minősített név használata, mivel a Java Template és a példányosító program ezen verziója még nem képes az importokat kezelni, ez pedig fordítási hibához vezethet

Hogy ne sérüljenek a Java szabvány csomagnevekre vonatkozó megkötései, az alábbi szabályokat definiáltam:

- Ha T típus primitív, vagy literál, akkor az elérési útban egy "a" prefixszel szerepel, vagyis az alakja: "aT"
  - Kivételt képez ez alól, ha a literál a "bstract" vagy "ssert" szavakkal egyenlő, hiszen ezek szuffixei az `abstract`, illetve `assert` kulcsszavaknak, így a nem megengedett helyen történő használatuk fordítási idejű hibához vezet
- Ha T egy objektumtípus, akkor a minősített név a része lesz az elérési útnak

Ugyanakkor az egyes OS disztribúciók sajátosságai gondot okozhatnak a generálás során (Windows alatt egy út hossza legfeljebb 260 karakter lehet).[13]

3.1.2. Példányosítás

A template-ből generált osztály törzse után (hasonlóan a C++ makrókhoz) deklarálásra kerül a template egy példánya az aktuális paraméterek listájával. Hogy elkerülhetőek legyenek a hozzárendelésből adódó hibák, az alábbi szabályokat definiáltam:[13]

- Az aktuális paraméterek száma nem lehet kevesebb, mint a formális paramétereké
- Objektumtípusok esetén a minősített nevek használata elvárt
- Egy T paraméter akkor és csak akkor helyettesíthető egy literállal, ha ez a paraméter kizárólag a kifejezések jobb oldalán található és a baloldallal típushelyes, vagy ahol a Java szabvány engedi a literálok használatát

3.1.3. Az implementációról

A template-ek példányosításához az eszköz a C++ *preprocesszor*t használja, aminek az aktuális típusok listáját argumentumként lehet átadni. A példányosítás menete a következő:[13]

- A template forrásfájl beolvasása és tokenizálása
- A csomagnév generálása a megadott típusok alapján
- A template kiegészítése szabványos C++ makróvá
- A makró példányosítása a generált állományban
- A makró preprocesszálása a C++ *preprocesszor* használatával és a Java forráskód generálása

3.1.4. Template-ek a gyakorlatban

Hogy bemutassam a megoldás előnyeit a generikusokkal szemben, tekintsük példaként a 2.1-es és 3.3-as forráskódokat. Feltesszük, hogy nincs alul- és túlindexelés, illetve a kurzor minden művelet után helyes állapotban van. Példányosítsuk mindkét megoldást az alapértelmezett integrális típussal (ami a `template` esetén `int`, míg a generikusnál `Integer`). A lényeges különbségeket az alábbi pontokban foglalom össze:[13]

- A `template`-nél a T formális paraméter minden előfordulása helyettesítve lesz `int`-tel, így a bytecode-ba már ez fordul, míg a generikus esetén először a Java fordítónak végre kell hajtani a típustörlést és be kell helyettesíteni a típuskorlátot (bounded type), vagy az `Object-et`, ha nincs ilyen. Ez azt is jelenti, hogy a háttérben objektumokkal kell dolgoznunk a futás során primitívek helyett

- A `template`-nél a `push` művelet egy `int`-et vár paraméterként és ezt egy `int[]` típusban tárolja. Generikus esetben a függvény szintén `int` paramétert vár, de ezt impliciten be kell csomagolnia egy `Integer` objektumba, míg a tároláshoz `Object[]`-t használ

- A `pop` művelet a `template` esetében a legfelső `int` értéket adja vissza (feltételezzük, hogy a hívó oldalon is `int`-et várunk). A generikus osztály esetén a `pop` művelet először leveszi a verem tetején lévő `Object`-et, amit expliciten `Integer`-ré kell konvertálnia, majd ezt adja vissza. Ebben az esetben ráadásul a hívási oldalon szükség lesz egy implicit unboxing-ra is, hiszen `int` értéket várunk.

3.2. A generikusok példányosítása

A Java Template egy sok lehetőséget magában rejtő konstrukció, de megvannak a maga hátrányai. Komplex logikájú osztályoknál nem a legjobb választás, így szükség volt egy olyan megoldásra, ami hasonló elvek mentén, de külső eszközök (mint a *g++* fordító a `template`-eknél) és új nyelvi elemek nélkül is használható. Ezen elvek mentén fejlesztettem tovább a példányosító eszközömet, amivel már korábban[13] is foglalkoztam. Ebbe a verzióba belekerült egy *típustábla*, ami memóriaként működve egy asszociatív konténerben tárolja a formális és aktuális paramétereket. Ezt úgy éri el, hogy a forrásfájl feldolgozásakor azonosítja a deklarált formális paraméterek neveit (amik a generikusokra vonatkozó szabályok miatt egyediek) és ezeket kulcsként használja egy map típusú konténerben. Az egyes kulcsokhoz az argumentumokként megadott aktuális paraméterek neveit rendeli, így ennél a lépésnél fontos, hogy az eszköz a megfelelő sorrendben kapja meg a paramétereket. Az itt begyűjtött információ jelentősen egyszerűsíti a típusok helyettesítését, hiszen a formális nevek ismeretében célzottan lehet keresni és cserélni a szükséges tokeneket, valamint a típusinformációk a folyamat során bármikor könnyen hozzáférhe-

tőek.[14]

Az eszköz a feldolgozás során az alábbi lépéseket hajtja végre:

- Betölti a forrásfájlt és lokalizálja a generikus paramétereket
- Feldolgozza a paraméterként kapott típusokat és ezzel az információval feltölti a *típustáblát*. Annak érdekében, hogy a kódgenerálás során elkerüljem a hiányzó típusokból adódó hibákat, az aktuális paraméterek számának legalább annyinak kell lennie, amennyi formális paraméter volt a forrásban. Ebben a lépésben az, hogy egy típus korlátos-e, nem lényeges, mivel mindegyik egy konkrét típusra lesz cserélve, amely egyben egy felső korlátot is definiál a kompatibilis típusokra
- Összeszerkeszti a csomagnevet. Ebben a lépésben fog először megmutatkozni a *típustábla* haszna, hiszen az egyedi csomagnév generálásához elegendő a tábla értékhalmazán végigiterálni és összefűzni a neveket. A Java standardnak való megfelelés érdekében a minősített nevek elemeit határoló pontokat backslash karakterekre cseréli. Ehhez a lépéshez módosítottam a példányosító eszközt, hogy támogasson generikus típusokat, mint argumentumok. Ezt úgy értem el, hogy a generikusoknál használt generikus paramétereket is beleveszem a csomagnévbe, valamint az ennek megfelelő útvonalra generálom az osztály forráskódját.

3.2.1. Az eszköz használata

Egy két paraméteres, generikus `Pair` típuson keresztül bemutatom a működését a példányosító eszköz ezen verziójának.[14]

```
1   public class Pair<K,V>{
2     private K first;
3     private V second;
4
5     public K getFirst(){return first;}
6     public V getSecond(){return second;}
7
8     public void setFirst(K _first){first = _first;}
9     public void setSecond(V _second){second = _second;}
10  }
```

3.4. forráskód. Generikus Pair példa

Ebben a példában a `K` és `V` paraméterek az első, illetve második argumentumnak feleltethetőek meg. Ebben a példában a példányosítás végrehajtásához az eszközömnek az alábbi paraméterekre van szüksége (kötött sorrendben):

- A `Pair` osztály forráskódját tartalmazó fájl (megkötés, hogy a forrásfájl ebben a verzióban csak egy osztályt tartalmazhat)
- A `K` aktuális típusát, ami objektumtípus esetén minősített név kell legyen
- A `V` aktuális típusát, hasonlóan megszorítva, mint `K` esetében

Legyenek most a konkrét típusok `int` és `boolean`, ebből adódóan a példányosításhoz tartozó parancs:

```
$java Tool Pair.java int boolean
```

A példányosítás befejeztével rendelkezésünkre áll a megfelelően típusozott `Pair` osztály egy egyedi csomagban, amely biztosítja a kétértelmű nevek okozta hibák elkerülését. Maga a generált kód a következő:

```
1   package aint.aboolean;
2   public class Pair{
3     private int first;
4     private boolean second;
5
6     public int getFirst(){return first;}
7     public boolean getSecond(){return second;}
8
9     public void setFirst(int _first){first = _first;}
10    public void setSecond(boolean _second){second = _second;}
11  }
```

3.5. forráskód. Generált forráskód

4. fejezet

Kiértékelés

Ebben a fejezetben bemutatom a Java Template hatékonyságát. Ennek eléréséhez a futásidejű teljesítményre fókuszáltam, tekintve, hogy a program szempontjából ez a rész sokkal fontosabb a fordítás hosszánál. A példányosító eszköz, amelyet a `template`-ek támogatására készítettem, nagy számú I/O taszkot futtat, így a hatékonysága függ a háttértártól[15].

Azt vizsgáltam, hogyan hat a megközelítésem a futásidőre. Ehhez egy felhőalapú virtuális gépet használtam, amin Ubuntu 16.04 LTS operációs rendszer volt Java 8 JVM-mel. Két tesztet futtattam rajta nagy számú elemmel, hogy jobban kiemeljem az egyes esetek közti különbségeket. Az első tesztesetnél egy vermet használtam. A generikus verzió `Integer`, míg a példányosított `int` típusú elemeket tárolt és a saját eszközömmel került példányosításra. Ezzel a megoldással elkerültem az `int` és `Integer` közti autoboxingot, amivel jelentős mennyiségű memóriát spórolt meg.
A második esetben a vermet `Integer` típussal példányosítottuk, így a generikus és a template paraméter pontosan ugyanaz. Ugyanakkor a példányosított verzióban a verem tudja magáról, hogy `Integer` elemeket tárol `Object`-ek helyett, így a futásidejű validációk száma limitálható.

A megoldásom mindkét esetben jobb futási eredményeket tudott felmutatni, főképp az első tesztnél. Az átlagos futási idő a hosszútávú tesztelések esetén azt mutatta, hogy a template esetében a futási idő a generikus változatnak mindössze `2.63%`-a. Ezeket a méréseket 800000 `push` és `pop` művelet elvégzésével validáltam és megállapítottam, hogy nagy mértékben csökkenthe-

tő a dinamikus memória allokálás és az autoboxing mértéke ebben az esetben. A második tesztesetnél, amikor a template és a generikus paraméter egyezett, azt figyeltem meg, hogy az eredmények eléggé egyenlőnek mutatják a két konstrukció hatékonyságát, hiszen a saját megoldással a futásidőt a generikus verzió 82.645%-ára tudtam leszorítani, mely még így is 20%-os sebességnövekedést jelent.

5. fejezet

Fejlesztés és jövőbeli kutatási irányok

Habár az új metódusok valóban előnyösnek bizonyultak, új problémák vetődtek fel a kutatások során, melyek közül az egyik a kontextusfüggő adattagok esete. Tekintsünk példaként egy tetszőleges logikájú generikus konténert, amely az elemeket egy `Object[]` adattagban tárolja. Egészítsük ki az osztály deklarációját egy eltérő azonosítójú, de azonos típusú adattaggal (a példa kedvéért legyen az azonosító `otherArray`). Ezt a Java nyelvtana természetesen engedélyezni fogja, hiszen a mezők azonosítása a nevük alapján történik.
Példányosítsuk most ezt az osztályt egy tetszőleges konkrét típussal (legyen ez most `int`), ekkor az `Object` minden előfordulását `int`-re cseréli az eszköz. De ez problémát jelent, hiszen `otherArray` típusa is `Object[]` és ennek megváltoztatása hibához vezet. Ebben a helyzetben az alábbiakat tehetjük:

- Az `Object` minden előfordulását (függetlenül annak környezetétől) `int`-re cseréljük: Ebben az esetben az `otherArray` mező funkcionalitása sérül, mivel az `Object[]` szerepe kontextusfüggő
- A példányosító eszköz ezen verziójának bővítése, hogy osztályozni tudja a mezőket: Ez a megoldás jelentős komplexitás-növekedéssel járna a kontextusfüggő nyelvtani szabályok miatt, ráadásul számolni kéne a téves azonosítás lehetőségével is, így ez a megoldás sose lehetne száz százalékig pontos
- A cserélendő azonosítók listáját argumentumként átadni az eszköznek: Minden mezőről el tudnánk dönteni, hogy szükséges-e cserélni a típust, vagy sem. Habár ez

tűnhetne a legjobb megoldásnak, az extra argumentumok miatt a eszköz használata sokkal nehezebbé válna és az integrálhatósága is csökkenne

- Annotációk bevezetése: A Java nyelvben lehetőség van úgynevezett *annotációk*at írni a kódba, amelyek a fordítónak, vagy egyéb felhasznált könyvtárnak, illetve eszköznek szóló információkat tartalmaznak. Ezen módszerrel megjelölhetőek lennének azok a mezők, amelyeknél a eszköz cserélni fogja a deklarált típust.

6. fejezet

Összefoglalás és eredmények

6.1. Bevezetés

A programozási nyelvek változatosak. Különböző a szintaxisuk, a szemantikájuk és megvalósításuk is. Azaz egy nyelvi elem, amely két nyelvben ugyanarra a feladatra való, lényegesen eltérő működéssel bír. A modern nyelvek egyik alappillére a típusok paraméterezése, amivel megvalósítható a típusok absztrakciója és a típusrendszerek támogatása. Ezeket a paramétereket, azonban az egyes nyelvek eléggé eltérő módon értelmezik. Legjobb példa erre a C++ és a Java nyelvek. Előbbi a template fogalmának bevezetésével támogatja a típusparaméterezést, amelyet fordítási időben értékel ki és generálja le a kívánt kódot, míg utóbbi az `1.5`-ös verziótól tartalmazza a Java Generics-et, amely egy dinamikusan típusos, futási időben kiértékelt alternatívája a template-nek.

6.2. A dolgozat célja

A dolgozatban a típusparaméterezés és a generikus programozási paradigma témakörét vizsgáltam. Számos elterjedt és népszerű nyelv megoldásait elemezve megnéztem, hogyan jelenik meg bennük a polimorfizmus és az absztrakt adatszerkezetekre alapuló fejlesztés. Bemutattam olyan lehetőségeket is, amelyek ma még nem részei egyes nyelvek szabványainak, de jelentősen megkönnyítik a programozók munkáját és amint a szabvány részévé válnak, lényegesen csökkentik a hibák előfordulási esélyét. Bemutattam az általam alkotott, a Java generikusokra épülő elemet, a *Java Template-*

et, amellyel a célom egy fordítás és futás szempontjából hatékony eszköz létrehozása. Ezt továbbfejlesztve azt vizsgáltam, hogyan lehet közvetlenül a generikus osztálysablonokból a Java szabályrendszerébe illeszkedő, statikusan típusos kódot előállítani.

6.3. Kutatási eredmények

6.3.1. Java Template

Megalkottam egy, a C++ template-jein alapuló nyelvi konstrukciót Java-ban, amivel (bizonyos bonyolultság alatt maradva) lehetővé vált statikusan típusos adatszerkezetek implementálása.
A generikus osztályok előnye, hogy dinamikusan típusosak, így a kód újrafordítása nélkül is képesek több típus támogatására. Ugyanakkor ez hátrányt is jelenthet olyan esetekben, amikor pontosan tudjuk, hogy milyen típusokkal fogjuk használni az adott adatszerkezetet. Az ezen problémát vizsgáló kutatásaim során bevezettem a *Java Template*-et, amely lehetővé teszi, hogy a típusparaméterek aktuális típusai már fordítási időben ismertek legyenek. Definiáltam a szabályokat, amelyek garantálják, hogy a generált kód megfelel a Java nyelvtani megkötéseinek, illetve meghatároztam egy módszert arra, hogyan tudom garantálni a típusok egyediségét és elkerülni a névütközéseket.
A témával foglalkozó publikációim: [13]

6.3.2. Generikusok példányosítása

Továbbfejlesztettem a Java Template konstrukcióját és definiáltam, hogyan lehet külső elemek nélkül magukat a generikus osztályokat példányosítani. Létrehoztam egy példányosító eszközt, amivel lehetőség volt a korábbi elvek mentén, de pusztán a szabványos Java kódból statikusan típusos példányokat előállítani.
Növeltem a számítási erejét a korábban megalkotott eszközömnek azzal, hogy kibővítettem egy asszociatív konténerrel, ami a fordítás során tartalmazza a formális és aktuális típusparaméterek értékét, így lehetővé téve a hatékony kódgenerálást és minimalizálva a kódban szükséges nyelvidegen elemek használatát. Egy egyszerű *Pair* típuson bemutattam a fordítás lépéseit és azt, milyen megkötések mellett használható hatékonyan egy

fejlesztési folyamat lépéseként.

A témával foglalkozó publikációim: [14]

Irodalomjegyzék

[1] Ken Arnold, James Gosling és David Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005. ISBN: 0321349806.

[2] Gábor Horváth, Norbert Pataki és Márton Balassi. "Code Generation in Serializers and Comparators of Apache Flink". *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS'17. Barcelona, Spain: ACM, 2017, 5:1–5:6. ISBN: 978-1-4503-5088-4. DOI: 10.1145/3098572.3098579. URL: http://doi.acm.org/10.1145/3098572.3098579.

[3] Bjarne Stroustrup. *The C++ Programming Language (special edition)*. Addison-Wesley, 2000. ISBN: 0-201-70073-5.

[4] Scott Meyers. *Effective STL*. Addison-Wesley, 2001. ISBN: 0-201-74962-9.

[5] Norbert Pataki és Zoltán Porkoláb. "Extension of Iterator Traits in the C++ Standard Template Library". *Proceedings of the Federated Conference on Computer Science and Information Systems*. Szczecin, Poland: IEEE Computer Society Press, 2011, 911–914. old.

[6] Ábel Sinkovics. "The connection between C++ template metaprogramming and functional programming". Dissz. Eötvös Loránd University, 2013.

[7] Adam Sipos. "Template metaprogramok hatékony fejlesztése". Dissz. Budapest: Eötvös Loránd Tudományegyetem, 2009.

[8] Douglas P. Gregor és tsai. "Concepts: linguistic support for generic programming in C++". *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. Szerk. Peri L. Tarr és William R.

Cook. ACM, 2006, 291–310. old. DOI: 10.1145/1167473.1167499. URL: https://doi.org/10.1145/1167473.1167499.

[9] Andrew Sutton és Bjarne Stroustrup. “Design of Concept Libraries for C++”. *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*. Szerk. Anthony M. Sloane és Uwe Aßmann. 6940. köt. Lecture Notes in Computer Science. Springer, 2011, 97–118. old. DOI: 10.1007/978-3-642-28830-2_6. URL: https://doi.org/10.1007/978-3-642-28830-2_6.

[10] Judit Nyékyné Gaizler, szerk. *Programozási nyelvek*. Budapest, Hungary: Kiskapu Kft., 2003. ISBN: 9-639-30147-7.

[11] D. Nam és tsai. “MARBLE: Mining for Boilerplate Code to Identify API Usability Problems”. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, 615–627. old. DOI: 10.1109/ASE.2019.00063.

[12] *Project Lombok*. https://projectlombok.org/.

[13] Péter Soha és Norbert Pataki. “Effective type parametrization in Java”. *AIP Conference Proceedings* 2116.1 (2019), 350007. old. DOI: 10.1063/1.5114360. eprint: https://aip.scitation.org/doi/pdf/10.1063/1.5114360. URL: https://aip.scitation.org/doi/abs/10.1063/1.5114360.

[14] Péter Soha és Norbert Pataki. “Instantiation of Java Generics”. Acta Cybernetica (to appear). 2020.

[15] Bence Babati, Norbert Pataki és Zoltán Porkoláb. “C/C++ Preprocessing with modern data storage devices”. *2015 IEEE 13th International Scientific Conference on Informatics*. 2015, 36–40. old. DOI: 10.1109/Informatics.2015.7377804.

Forráskódjegyzék