Nevezetes felsorolók funkcionálisan

Visnovitz Márton, Horváth Győző

visnovitz.marton@inf.elte.hu
horvath.gyozo@inf.elte.hu
ELTE IK

Absztrakt. A programozás oktatásában klasszikusan a programozási tételekre alapozva az objektumelvű programozás irányába haladunk. Az objektumelvű programozás egyik megközelítése a felsorolók, felsorolható adatszerkezetek implementációja, a tanult programozási tételek alkalmazása ezen felsorolókra. Ez a cikk bemutatja, hogyan lehetséges az objektumelvű és funkcionális programozás paradigmáinak együttes használatával olyan felsorolókat létrehozni, melyekre alkalmazhatóak a hajtogatás elvén alapuló, funkcionálisan megvalósított programozási tételek.

Kulcsszavak: funkcionális programozás, objektumelvű programozás, felsorolók

1. Bevezetés

Korábbi cikkünkben [1] bemutatásra került, hogy hogyan lehetséges a magyarországi programozásoktatásban meghatározó programozási tételeket [2] a funkcionális programozásban jellemző módszerek segítségével megvalósítani. A kutatás célja azt vizsgálni, hogy a jelenleg széles körben oktatott ismeretanyag milyen mértékben valósítható meg különféle paradigmák (és azok kombinációjának) segítségével, ezzel szélesítve a bemutatott programozási paradigmák körét. Az Eötvös Loránd Tudományegyetemen a *Programozás* tárgy (korábbi nevén *Programozási alapismeretek)* foglalkozik a programozási tételek megismertetésével. Erre a tanegységre épül az *Objektumelvű programozás* (korábbi nevén *Programozás*), mely általánosítja a programozási tételeket felsoroló típusokra. Ebben a cikkben azt mutatjuk be, hogy hogyan lehet megvalósítani a felsorolókat (illetve néhány nevezetes felsorolható típust) a funkcionális programozás és az objektumelvű programozás mintáinak egyesítésével. Az így megvalósított típusokra alkalmazhatóak a korábbi cikkben [1] megvalósított, hajtogatáson (*folding*) alapuló programozási tételek.

2. Funkcionális felsorolók általánosan

Klasszikus értelemben *felsorolóknak* (vagy *felsoroló objektumoknak*) nevezzük azon típusokat, melyek képesek egy adatnak valamilyen értelemben vett első elemére ráállni, majd a soron következőre egészen addig, amíg van ily módon felsorolható elem [3]. Ez a definíció az imperatív, ciklusok segítségével történő felsorolást támogató meghatározás. A funkcionális programozás segítségével meghatározott programozási tételeket alapul véve más fajta definíciót kell adnunk, hogy felsorolás segítségével fel tudjunk dolgozni adatokat. A klasszikus meghatározáshoz hasonlóan szükségünk van arra, hogy el tudjuk kérni a felsorolónk valamilyen értelemben vett következő elemét, de mivel a további feldolgozás nem léptetéssel, hanem a maradék elemek rekurzív feldolgozásával történik (hajtogatás módszer), ezért a másik fő műveletünk a maradék elemek lekérdezése. Emellett szükség van a felsoroló ürességét (végét) jelző műveletre is. Ennek megfelelően mondhatjuk, hogy hogy a *funkcionális felsoroló* alatt olyan adatszerkezetet értünk, melyről *eldönthető, hogy üres-e, valamint felbontható első elemre és az azt követő elemeket felsoroló adatszerkezetre*. A klasszikus értelemben vett, és a funkcionális felsorolók műveleteit az **1. táblázat** mutatja.

„Klasszikus” felsorolók		**Funkcionális felsorolók**	
Első elemre állítás	*.First()*	Végére ért-e	*.vegeE()*
Végére ért-e	*.End()*	Következő elem	*.kovetkezo()*
Aktuális elem	*.Current()*	Maradék elemek felsorolója	*.maradek()*
Következő elemre állítás	*.Next()*		

1. táblázat: „Klasszikus” és funkcionális felsorolók műveletei

Fontos, hogy mivel funkcionális programozással dolgozunk ezért a felsorolókat úgy kell megvalósítani, hogy azoknak a belső állapota ne legyen megváltoztatható (*immuatable*), vagyis minden műveletet (metódus) úgy kell implementálni, hogy azok ne az eredeti adatszerkezetet módosítsák, hanem egy egyedi értéket, vagy egy új felsorolót adjanak eredményként.

2.1. Gyűjtemények

A felsorolók egyik speciális esetei a *gyűjtemények* (továbbiakban általános típusként *G*-vel jelöljük a gyűjtemények általános típusát). Ezen felsorolók valamilyen értékeket tárolnak, ezeket lehet felsorolni. Gyűjtemények esetében lehetséges a $G \rightarrow G$ típusú, gyűjteményről gyűjteményre leképező programozási tételeket alkalmazni, mint amilyen a *másolás* vagy a *kiválogatás*. Ahhoz, hogy ezeket a tételeket meg tudjuk valósítani szükség van egy további műveletre, a gyűjteményhez új elemet hozzáadó *.beszur()* metódusra.

3. Funkcionális felsorolók megvalósítása

A funkcionális felsorolók megvalósításában ötvözzük az objektumorientált és a funkcionális programozás módszereit. Ezáltal megvalósul az egységbe zárás, illetve a típusmegvalósítás osztályok segítségével, de ezeket az osztályokat és példányaikat úgy kell használnunk, hogy azok ne sértsék a funkcionális programozás elveit. Az implementációban az objektum-példányok nem megváltoztathatók és a függvények tiszta függvényként működnek, vagyis adott bemenetre mindig ugyanazon kimenetet produkálják. Ez utóbbi megkötést úgy értelmezzük, hogy egy osztály példányának esetében egy metódusnak magát az objektumot (*this*) is bemenetének tekintjük. Az alább adott megvalósítás olyan, hogy egy egyszer létrejött objektumnak nem tud megváltozni a belső állapota, ezért ez a tiszta függvény tulajdonság teljesül a metódusaira is. A megvalósításhoz a TypeScript [4] programozási nyelvet választottuk, mivel ez támogatja az objektumorientált és a funkcionális paradigmát is, így nem okoz gondot a programozási koncepciók együttes használata sem.

Az objektumorientált programozás lehetőséget nyújt arra, hogy olyan implementációt készítsünk a felsorolóinkhoz, mely egyszerre szolgál *interfészként* a konkrét felsorló-típusokhoz, illetve tartalmazza metódusként a programozási tételek megvalósítását is a megfelelő interfészre. Ennek megfelelően a *felsorolók* és a *gyűjtemények* általános osztályát *absztrakt osztályok* segítségével valósítottuk meg. Az absztrakt osztályokon belül a szükséges műveletek metódusai megvalósítás nélkül, kizárólag típusszignatúrával szerepelnek. Az absztrakt osztályokat sablonosztályokként, tetszőleges *H* típusú felsorolt elemekre implementáltuk:

```
abstract class Felsorolo<H> {
  abstract vegeE:     () => boolean
  abstract kovetkezo: () => H
  abstract maradek:   () => Felsorolo<H>
}
```

1. ábra: A felsorolók absztrakt osztályának megvalósítása

```
abstract class Gyujtemeny<H> extends Felsorolo<H> {
  abstract maradek: () => Gyujtemeny<H>
  abstract beszur:  () => Gyujtemeny<H>
}
```

2. ábra: A gyűjtemények absztrakt osztályának megvalósítása

A gyűjtemény osztályban felüldefiniáljuk a maradék művelet szignatúráját (**2. ábra**). Erre azért van szükség, hogy a gyűjteményekre definiált műveletek láncolva típushelyesek maradjanak. A példában (**3. ábra**) látható kifejezés típushelytelen lenne, ha *maradek* metódus *Felsorolo* típusú értékkel térne vissza, hiszen arra nincs definiálva a *beszur* művelet.

```
gyujtemeny.maradek().beszur(ertek)
```

3. ábra: A gyűjtemények absztrakt osztályának megvalósítása

4. Programozási tételek funkcionális felsorolókra

A programozási tételek korábbi cikkünkben [1] bemutatott funkcionális megvalósításai általánosíthatók az előbbiekben bevezetett *Felsoroló* és *Gyűjtemény* absztrakt típusokra. A tömbökre megvalósított tételek egyes műveletei párhuzamba állíthatóak a fent meghatározott metódusokkal. A megfeleltetést a **2. táblázat** mutatja.

Művelet	**TypeScript tömbre**	**Funkcionális felsorolókra**
Üresség vizsgálata	`x0 === undefined` *a következő elem definiálatlan*	`x.vegeE()`
Következő elem	`[x0, ...xs]` *a tömb felbontása első és további elemekre, ezek közül az előbbi*	`x.kovetkezo()`
További elemek	`[x0, ...xs]` *a tömb felbontása első és további elemekre, ezek közül az utóbbi*	`x.maradek()`
Elem beszúrása	`[e, ...xs]` *új tömb konstruálása egy elem és a mögé kibontott sorozat segítségével*	`x.beszur(e)`

2. táblázat: A gyűjtemények absztrakt osztályának megvalósítása

Ezen megfeleltetés alapján bármelyik tétel átírható úgy, hogy a működjön a *Felsorolo* vagy *Gyujtemeny* típusú adatszerkezeteken is. A továbbiakban három konkrét tétel implementációját mutatjuk be, a *sorozatszámítás*, *másolás* és *kiválogatás* tételét. Azért erre a három tételre esett a választás, mert ezen tételek köré szerveződik a többi tétel magasabb rendű függvényekkel (*higher order functions)* történő megvalósítása a funkcionális programozásban [1]. A tételek közül a *sorozatszámítás* tetszőleges *Felsorolo* objektumon végrehajtható, ezért azt a *Felsorolo* osztály metódusaként valósítjuk meg, míg a *másolás* és a *kiválogatás* tételt – mivel azok építenek a *.beszur()* műveletre – a *Gyujtemeny* absztrakt osztály metódusaként implementáljuk. Ebből a tulajdonságból következik, hogy a *Felsorolo* típusba tartozó, de nem *Gyujtemeny*-ből származtatott osztályok esetén az egy konkrét értékre képező tételek mindegyikét (*megszámolás, eldöntés, kiválasztás, lineáris keresés)* a *sorozatszámítás* tételre szükséges visszavezetni.

4.1. Sorozatszámítás

```
abstract class Felsorolo<H> {
  // ...
  public sorSzam: <M> (f: (s: M, e: H) => M, k: M): M =
    (f, k) => this.vegeE()
      ? k
      : f(this.maradek().sorSzam(f, k), this.kovetkezo())
}
```

4. ábra: A sorozatszámítás tétel szignatúrája és megvalósítása a *Felsorolo* absztrakt típusban

A *sorozatszámítás* tételt megvalósító metódust (`sorSzam`) a **4. ábra** mutatja be TypeScript nyelven. A megvalósítás egy egyszerű esetszétválasztáson alapszik, mely attól függően, hogy végére értünk-e az adott felsorolónak, a k kezdőértéket adja eredményül, vagy a maradék részre rekurzívan kiszámolt részeredmény és a következő elemre alkalmazott f függvény eredményét. A tétel egyértelmű, **2. táblázat** szerinti átirata a tömbökre történő megvalósításnak.

4.2. Másolás

```
abstract class Gyujtemeny<H> {
  // ...
  public masol: <M> (f: (e: H) => M): Gyujtemeny<M>
    (f) => this.vegeE()
      ? this.constructor()
      : this.maradek().masol(f).beszur(f(this.kovetkezo()))
}
```

5. ábra: A másolás tétel szignatúrája és megvalósítása a *Gyujtemeny* absztrakt típusban

A *másolás* tételt megvalósító metódust (`masol`) (**5. ábra**) a *Gyujtemeny* osztályon belül valósítjuk meg, eredménye szintén egy *Gyujtemeny*, de már nem H, hanem tetszőleges M típusú elemek gyűjteménye, amennyiben a leképezést végző f függvény $H \rightarrow M$ típusú.

A tétel megvalósításában megjelenik a TypeScript nyelv egy sajátossága a *this.vegeE()* feltételes kifejezés igaz ágában: *this.constructor()*. A TypeScript nyelvben az úgynevezett prototípusos objektumorientált [5] jelleg miatt egy osztály konstruktora egy példányon keresztül is meghívható. Ez a konstruktor paraméter nélkül meghívva létre fog hozni egy új, üres *Gyujtemény* objektumot. A sablonparaméter (M) a metódus szignatúrájából következik, mivel ott kikötöttük, hogy a metódus M típusú elemek gyűjteményére képez.

4.3. Kiválogatás

```
abstract class Gyujtemeny<H> {
  // ...
  public kivalogat: (T: (e: H) => boolean): Gyujtemeny<H> =
    (T) => this.vegeE()
      ? this.constructor()
      : T(this.kovetkezo())
        ? this.maradek().kivalogat(T).beszur(this.kovetkezo())
        : this.maradek().kivalogat(T)
}
```

6. ábra: A kiválogatás tétel szignatúrája és megvalósítása a *Gyujtemeny* absztrakt típusban

A *kiválogatás* tételt megvalósító metódusban (`kivalogat`) (**6. ábra**) szintén megjelenik az új üres gyűjtemény létrehozása, de ebben az esetben H típusú elemekből jön létre az új gyűjtemény. A logi-

ka hasonló, mint a *másolás* tételnél, csak kiegészül egy további elágazással, melyben azt vizsgáljuk, hogy a soron következő elemre teljesül-e a *T* tulajdonság. E szerint ágazik el a függvény, hogy beszúrja-e a keletkező gyűjteménybe az aktuálisan feldolgozott elemet vagy nem.

5. Nevezetes felsorolók

A felsorolókat és gyűjteményeket leíró, tételeket megvalósító absztrakt osztályok segítségével lehetőségünk van konkrét, nevezetes *felsoroló típusokat* [3] megvalósítani. A továbbiakban a teljesség igénye nélkül néhány fontos felsoroló típus esetén mutatjuk meg a *Felsorolo* vagy a *Gyujtemeny* absztrakt metódusainak megvalósítását.

5.1. Egész-intervallumot felsoroló típus

Az egész intervallumot felsoroló típus egy adott [*m*..*n*] intervallum elemeit sorolja fel *m*-től, *n*-ig egyesével. Ennek mintájára sok hasonló felsoroló készíthető. Ez a felsoroló nem tekinthető gyűjteménynek, mivel csak a két végpontja és egy szabály (+1 hozzáadás) alkotja. Egész számokat sorol fel, ezért a *Felsorolo* osztály altípusa *number* típusértékkel.

```
class Intervallum extends Felsorolo<number> {
  constructor (private m: number, private n: number) { super() }

  public vegeE: () => boolean =
    () => this.m > this.n
  public kovetkezo: () => number =
    () => this.m
  public maradek: () => Intervallum =
    () => new Intervallum(this.m + 1, this.n)
}
```

7. ábra: Az egész-intervallumot felsoroló típus megvalósítása

A megvalósítás alapja, hogy az *Intervallum* objektumokat mindig egy *m* és egy *n* intervallum-határ értékkel hozzuk létre. Ezeket az értékeket kapja az osztály konstruktora paraméterként. A konstruktor paraméterei mellett szereplő, láthatóságot jelölő *private* kulcsszó a TypeScript-ben egyben azt is jelöli, hogy az így kapott értékek egyből tárolásra is kerülnek a létrejövő objektum belső állapotában.

A *maradek* metódus implementációjában fontos, hogy nem az aktuális objektum m és n értékeit módosítjuk, hanem egy új *Intervallum* felsorolót adunk vissza a már módosított határoló értékekkel. Ez azért szükséges, mert enélkül változna a belső állapot, ami sértené az immutábilis tulajdonságot.

5.2. Sorozatot felsoroló típus

A *sorozatot felsoroló típus* valamely adott típusú (*H*) elemek olyan *gyűjteményét* sorolja fel, melyben lehetséges index alapján egy elemet olvasni vagy módosítani, melynek ismerjük a hosszát, és mely sorozat bővíthető és melyből elemeket ki lehet venni. Az alább megadott implementáció azt az egyszerű esetet mutatja be, amikor a sorozatot az elejétől a végéig járjuk be, habár más fajta bejárások is lehetségesek lennének. Az ilyen sorozatok – a belső reprezentációtól függetlenül – az alábbi műveleteket kell megvalósítsák:

```
class Sorozat<H> extends Gyujtemeny<H> {
  public olvas:   (index: number) => H = //...
  public modosit: (index: number, ertek: H) => Sorozat<H> = //...
  public hossz:   () => number = //...
  public hozzaad: (ertek: H) => Sorozat<H> = // ...
  public kivesz:  () => Sorozat<H> = // ...
}
```

8. ábra: A sorozat típus műveletei

Amennyiben a sorozat műveletei (**8. ábra**) megvalósításra kerülnek, akkor a felsoroló műveletek ezek segítségével az alábbi módon implementálhatók.

```
class Sorozat<H> extends Gyujtemeny<H> {
  // ...
  public vegeE: () => boolean =
    () => this.hossz() === 0
  public kovetkezo: () => H =
    () => this.olvas(0)
  public maradek: () => Sorozat<H> =
    () => this.kivesz()
  public beszur: (e: H) => Sorozat<H> =
    () => this.hozzaad(e)
}
```

9. ábra: A sorozat-felsoroló műveletei

Ebben az általános implementációban nincs megadva a belső reprezentáció, illetve a *Sorozat* típus saját műveleteinek megvalósítása. Ennek oka, hogy a felsoroló műveletek ezektől függetlenek. Azt azonban elmondhatjuk, hogy a *beszur* és a *kivesz* műveletek működésétől függően (a sorozat elejével vagy végével dolgozik) a sorozat lehet *verem* vagy *sor* adatszerkezet is.

5.3. Halmazt felsoroló típus

A *halmaz* olyan adatszerkezet, melyben azonos típusú értékeket tárolunk sorrendiség és multiplicitás nélkül. Egy ilyen adatszerkezen az alábbi műveletek léteznek: *halmaz ürességének vizsgálata*, *egy elem vizsgálata, hogy benne van-e a halmazban*, *egy elem hozzáadása a halmazhoz* és *egy elem kivétele egy halmazból*. Ezen műveletek szignatúráját a **10. ábra** mutatja.

```
class Halmaz<H> extends Gyujtemeny<H> {
  public uresE:     () => boolean = //...
  public tartalmaz: (ertek: H) => boolean = //...
  public belerak:   (ertek: H) => Sorozat<H> = //...
  public kivesz:    (ertek: H) => Sorozat<H> = // ...
}
```

10. ábra: A halmaz típus műveletei

Ezek mellett szükség van egy olyan műveletre, mely kiválaszt egy tetszőleges elemet a halmazból. Ez a művelet sokféleképpen implementálható, de a legtöbb megvalósításban egy determinisztikus műveletet kapunk.

```
  public kivalaszt: () => H = //...
```

11. ábra: A halmaz típus *kivalaszt* művelete

Ha ez a metódus is adott, akkor a felsoroló műveletei az alábbi módon valósíthatóak meg:

```
class Halmaz<H> extends Gyujtemeny<H> {
  // ...
  public vegeE: () => boolean =
    () => this.uresE()
  public kovetkezo: () => H =
    () => this.kivalaszt()
  public maradek: () => Sorozat<H> =
    () => this.kivesz(this.kivalaszt())
  public beszur: (e: H) => Sorozat<H> =
    () => this.belerak(e)
}
```

12. ábra: A halmaz felsoroló műveletei

6. Konklúzió

A fentiekből látszik, hogy a funkcionális paradigma segítségével bevezetett programozási tételekre építkezve az ismeretek tovább bővíthetők a típusok, típuskonstruckiók, felsorolható adatszerkezetek irányába. Ez a megközelítés kombinálja a funkcionális és az objektumorientált programozási paradigmákat. Módszertani szempontból ez azért lehet előnyös, mert így többféle programozási módszer is bemutatásra kerül, nem szükséges ezeket külön-külön tárgyalni. Ezen az irányvonalon továbbhaladva bemutatható a felsorolók hagyományos, imperatív mintákra épülő megvalósítása is, valamint a különféle paradigmák kombinálásának lehetőségei. Végső soron ez a módszer azt a célt szolgálhatja, hogy kevés eszközzel mutathassunk meg minél több lehetőséget és a tanulókat arra neveljük, hogy a megfelelő célra a megfelelő eszközt válasszák.

A fenti példákban a TypeScript nyelvet választottuk a példákhoz, mivel ez számos paradigmát támogat. Természetesen sok esetben sem a módszer, sem pedig a megvalósítások nem adnak optimális megoldást az egyes problémákra, de a cél itt a tanulási folyamat támogatása.

Köszönetnyilvánítás

EFOP-3.6.1-16-2016-00023: Kutatás-fejlesztési tevékenység megvalósítása az Eötvös Loránd Tudományegyetem szombathelyi kampuszán – A projekt a Magyar Állam és az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

Irodalom

[1] Visnovitz Márton: *Programozási nyelvek funkcionálisan.* INFODIDACT 2017.

[2] Szlávi Péter, Zsakó László: *Módszeres programozás: Programozási tételek.* ELTE Informatikai Kar, 2008.

[3] Gregorics Tibor: *Programozás 1. kötet Tervezés.* ELTE Eötvös Kiadó, 2013.

[4] *TypeScript – JavaScript that scales.* Microsoft, 2018. https://www.typescriptlang.org (utoljára megtekintve: 2018. 11. 15.)

[5] *Prototype-based programming - MDN Web Docs Glossary: Definitions of Web-related terms | MDN.* MDN, 2018. https://developer.mozilla.org/en-US/docs/Glossary/Prototype-based_programming (utoljára megtekintve: 2018. 11. 15.)