0. Címlap

Üdvözlöm a bíráló bizottság tagjait, a kar oktatóit és hallgatóit, valamint a megjelent érdeklődőket. Bagossy Attila vagyok, a dolgozatom és az előadásom címe *Generátorok előállítása CPS-transzformációval Java nyelven.* A témavezetőim Dr. Battyányi Gyula Péter és Balla Tibor.

1. Áttekintés

Az előadásom összeállításakor úgy döntöttem, hogy nem fogom szigorúan követni a dolgozatom felépítését. A generátorokkal rokon szerkezetek, a *continuation passing style*, valamint a kidolgozott eljárás mögötti megfontolások részletes ismertetése helyett inkább a gyakorlati oldalt szeretném megmutatni.

Természetesen ahhoz, hogy az előadás önmagában is értelmezhető legyen, szükséges az alapfogalmak bevezetése. Ezt követően azonban a dolgozattól eltérve a transzformációs eljárás pontos bemutatása helyett érvelni szeretnék annak léte, s újdonságtartalma mellett. Először a generátorok más nyelvekben való elterjedtségére és széleskörű felhasználási lehetőségeire rámutatva fogom indokolni a *Java*ban való bevezetésük szükségességét. Mivel nem az én eljárásom az első, mely ezt kívánja megvalósítani, a meglevő *Java* nyelvű implementációk ismertetésével folytatom, kiemelve az eltéréseket. Az előadás második felében egy összetett kódrészlet transzformálásának lépésein kalauzolom végig a hallgatóságot, mely bemutatja az eljárást, s egyben rávilágít arra, hogy a *CPS* mennyire elegáns megoldást ad egy ilyen komplex problémára.

2. A generátor és rokonai

Az áttekintést követően vegyük szemügyre, hogy mi az a *generátor*, és milyen más eszközökkel van kapcsolatban.

3. Szubrutin (*subroutine*)

1. A szubrutin az egyik leggyakrabban használt eszköz a programok szervezésére. Emiatt bemutatása szükségtelennek tűnhet, azonban ez az első lépcsőfok a generátorokhoz vezető úton. A szubrutinok alkalmasak gyakran használt kódrészletek kiemelésére, ami csökkenti a kódduplikációt. Emellett az implementáció elrejtésével növelhetik az absztrakciós

szintet, és akár könyvtárak képezhetőek belőlük. Jelen esetben azonban a programvezérlésre kifejtett hatásukat szeretném kihangsúlyozni. Amikor meghívunk egy szubrutint, a vezérlés végighalad annak első utasításától az utolsóig, majd visszatér a hívóhoz. A hívó kód végrehajtása csak ezután folytatódhat tovább.

2. Az ábra két oldala a hívót és a meghívott szubrutint ábrázolja. A nyilak a vezérlés irányát jelzik. Bár az ábrán nem látszik, de ki kell emelni, hogy a szubrutin végrehajtása minden egyes meghívásakor a legelső utasításánál kezdődik.

4. Korutin (*coroutine*)

1. A szubrutin mindig alárendelt szerepet játszott a hívó kódhoz képest. Ezzel szemben a korutinok alárendeltségi viszony mellett mellérendeltség kialakítására is alkalmasak, akár meghatározhatatlanná téve, hogy valójában ki hív kit. Ezt az teszi lehetővé, hogy futásuk felfüggeszthető és folytatható, és a hívások között megőrzik a lokális változóik értékét. Attól függően, hogy egy korutin tetszőleges másik korutint hívhat meg, vagy pedig muszáj visszaadnia valamikor a vezérlést az őt hívó korutinnak, beszélhetünk szimmetrikus és aszimmetrikus korutinokról.

2. A fólián látható kódrészletekben szimmetrikus korutinok szerepelnek, melyek a `yield` utasítás segítségével adják át egymásnak a vezérlést. Miután `A` meghívta `B`-t, felfüggesztésre kerül, mindaddig, míg `B` vagy egy másik korutin újra meg nem hívja. Ekkor a szubrutinoktól eltérően ugyanonnan fog folytatódni a végrehajtás, ahol abbamaradt, tehát a `B`-t hívó `yield`-től.

5. Generátor (*generator*)

A korutinok ismeretében már kifejezhető a generátor is, ami egy olyan aszimmetrikus korutin, mely egy sorozatot állít elő. A legnagyszerűbb tulajdonsága, hogy egyszerre csak egy elemet generál, ami azt jelenti, hogy lustán képzi a sorozatot. Ennek köszönhetően akár végtelen sorozatok reprezentálására alkalmas eszközt kínál.

Az ábra jobboldalán az ábécét előállító generátor szerepel, a baloldalán pedig ennek a meghívása egy `for` ciklus segítségével. Minden egyes iteráció alkalmával egy újabb elemet kérünk a generátortól, mindaddig, amíg a generált sorozat véget nem ér.

6. Generátorok ismert nyelvekben

Miután megismertük, mi az a generátor, folytassuk a támogatás vizsgálatával. Mely nyelvekben jelenik meg, milyen szerepet játszik? Hogyan könnyíti meg a programozók munkáját?

7. Támogatást biztosító nyelvek

A ma is ismert generátorokhoz hasonló szerkezetek először az *IPL-V* és az *Alphard* nyelvekben jelentek meg. Jelenleg használatos formájukat pedig a Barbara Liskov és munkatársai által kifejlesztett *CLU* nyelvben nyerték el, emellett itt jelent meg először *yield* kulcsszó használata.

A TIOBE minden hónapban kiadott, programozási nyelvek népszerűségét mérő listájának 10 legnépszerűbb nyelve között 5 olyan is van, mely nyelvi szintű támogatást ad a generátorok írásához. Ezek a C#, a Python, a VB.NET, a PHP és a JavaScript. Ez mindenképpen figyelemre méltó, hiszen ezek közül például a C# a Java közvetlen konkurensének tekinthető.

8. Felhasználási lehetőségek – 1

A generátor egy elegáns szerkezet lehet a programozók eszköztárában. Nem csak elegáns azonban, hanem rendkívül erőteljes is, számtalan felhasználási lehetőséggel.

A sorozat fogalmat itt most nem matematikai értelemben használjuk, hanem egyszerűen csak egymást követő, azonos típusú elemeket jelent. Alkalmas a generátor végtelen sorozatok előállítására, ilyen például a Fibonacci-sorozat, vagy az összes prímszám. Használható véletlen értékek forrásaként. Ilyen formában alkalmazza a generátorokat a QuickCheck tesztelési keretrendszer is.

Véges sorozatok képzésére gyakran ott is bevethető a generátor, ahol nem is sejtenénk. Ilyen lehet például a reguláris kifejezések illesztése. A generátor mindig a következő illeszkedő karaktersorozatot adja vissza, lustán képezve valójában az összes illeszkedést. Fájlok beolvasására is alkalmazhatóak a generátorok, mindig csak egy sort vagy darabot beolvasva, *on demand* módon, azaz igény szerint. Egy teljesen különböző ötlet paraméteres görbék pontjainak előállítása. A kliens kódnak semmit sem kell tudnia a görbe paramétertartományáról és megvalósítási részleteiről, csak sorban kéri a pontokat, amiket a generátor egyenként kiszámol.

9. Felhasználási lehetőségek – 2

Az előző példák a generátorok hagyományos felhasználási lehetőségeit szemléltették. Azonban a generátorok aszimmetrikus korutin voltát még sokféleképpen ki lehet aknázni.

Meglepő lehet, de szimmetrikus korutinok megvalósítására is képesek. Ehhez egy *trampoline* szükséges, melyen keresztül a generátorok hívni tudják egymást. Az `async/await` szerkezet leegyszerűsíti az aszinkron kód írását. A *Babel JavaScript transpiler* valójában generátor-alapú kódot hoz létre az `async/await` szimulálására. Konkurens végrehajtásra is lehetőséget adnak a generátorok, mikroszálak segítségével. Ezek az operációs rendszer szintű szálaknál sokkal alacsonyabb erőforrásigénnyel rendelkeznek, és kooperatív multitaszkingot valósítanak meg.

10. Korábbi *Java* implementációk

A generátorok *Java*ban történő megvalósítására már több próbálkozás is történt. Mi ezekkel a probléma, miért érdemes még egy implementációt készíteni?

11. Általános problémák

1. A rendelkezésre álló könyvtárak többségét már nem támogatják, a legutolsó módosítás jellemzően még a *Java SE* 6-os vagy 7-es verziója idején történt. Ennek folytán bizonyos szerkezetek támogatása, mint például a *try-with-resources*, a lambda függvények, vagy az interfészek *default* metódusai nincsenek jelen ezen könyvtárakban.

2. Többségében kényelmetlen, akár átgondolatlan interfészt kell a programozónak használnia. Természetesen a *yield* kulcsszót csak a *Java* fordító módosításával lehetne bevezetni, amire egyik könyvtár sem vállalkozik. Más megoldáshoz folyamodva, legtöbbször különféle függvényhívásokkal érhető el a generátorokra jellemző működés.

3. A különböző megvalósítások mögött sokféle, eltérő technika áll. Van példa bájtkód-instrumentációra, szálak használatára, illetve egyszerű programkönyvtárakra. Ezek egyike sem mondható optimálisnak. A bájtkód-instrumentáció kevésbé megbízható, mint az annotáció-feldolgozás, mivel a fordító nem ellenőrzi a generált kódot. Érvénytelen bájtkód esetén a *JVM* megtagadhatja az osztály betöltését. A szálak elsősorban

erőforrás-igényüket tekintve okoznak gondot. Az egyszerű programkönyvtárak egy ősosztályt biztosítanak, melynek gyermekei lesznek a generátorok. Ez nem kifejezetten elegáns és rugalmas megoldás.

12. jyield

A kódrészlet a *jyield* könyvtár példakódjainak egyike. Ez a könyvtár 7 éve lett frissítve utoljára, még a *Java SE* 6 idején. Bájtkód-instrumentációt használ, és a *Yield* nevű osztály statikus metódusait kell hívni, ahogy a 8. és a 10. sorokban látható. A metódusok használata szemantikailag hibás, hiszen míg a *yield* egy utasítás, addig a metódushívások kifejezések. Ez félreértéseket, hibás működést okozhat.

13. java-generator-functions

A *java-generator-functions* könyvtár egy ősosztályt biztosít, melynek gyermekosztályai lehetnek a generátorok. Ez kisebbfajta kényelmetlenséget jelent, hiszen nem elég csak egy annotációt elhelyezni a megfelelő metóduson. A könyvtár a háttérben szálakat használ, amiknek a száma korlátozott, és memóriaigényük sem elhanyagolható.

14. lombok-pg

A *lombok-pg* az elterjedt *Lombok* könyvtár kiegészítéseit tartalmazza. Azonban a fejlesztése mintegy 5 éve megszakadt, és a jelenlegi *Lombok* verzióval már nem kompatibilis. Itt is a megjelenik a szemantikailag megkérdőjelezhető függvény használata. Emellett a generátor paramétereit kötelező a *final* módosítóval ellátni. Előnye viszont ennek a megoldásnak, hogy annotáció-feldolgozást használ, és egy állapotgépnek megfelelő kódot generál. Ez tekinthető az egyik legjobb könyvtárnak.

15. Az eljárás egy példán keresztül

Az előadás következő részében a dolgozatom eredményét jelentő eljárást fogom bemutatni egy példán keresztül.

16. Legfontosabb jellemzők

A példa előtt vázolnám az eljárás legfontosabb jellemzőit.

1. A módszer a *Pluggable Annotation Processing API*-t használja, mely a fordítási folyamatba ékelve teszi lehetővé az annotációkon alapuló kódanalízist, és forráskódgenerálást. Az implementáció része egy annotáció, melyet azokon a metódusokon kell elhelyezni, melyeket generátorrá szeretnénk alakítani. Ezután a könyvtár a fordítás közben megkeresi ezeket a metódusokat, és elvégzi a szükséges transzformációkat.

2. Különböző függvényhívások helyett a könyvtárat használó programozó a jól ismert `return` utasítással adhat vissza értékeket a generátorokból. E módon a `yield` szemantikájával ruházzuk fel a `return`-t, ezeknél fog megállni a generátor, majd innen fog tovább folytatódni.

3. A *CPS* elsősorban a funkcionális programozási nyelvekben elterjedt stílus, de használják fordítóprogramok belső reprezentációjaként is. Előnye, hogy alkalmazásakor a programvezérlés teljesen explicitté válik, könnyen manipulálható a futás iránya. Egy ilyen könyvtár esetén pont erre van szükségünk. Fordítási időben folyamatosan követni fogjuk a *continuation*öket, futási időben pedig egy *trampoline* segítségével valósítjuk meg a technikát. Az említett *continuation* nem más, mint a hátralevő teendőket jelképező struktúra.

17. Prímek generálása

1. A kódrészlet a prímszámok végtelen sorozatát generálja. A megvalósítás egyetlen célja, hogy szemléltesse az eljárás működését, ebből fakad viszonylag primitív volta. Kiemelném a ciklust megelőző `loop` címkét és a `continue` utasítást, melyek kifejezetten azért kerültek a példába, hogy bemutassák, milyen komplexitással kell a transzformációnak megbirkóznia.

2. A generátor először kettőt fog visszadni, az ötödik sorban található `return` utasításnak köszönhetően.

3. Ezt követi úgymond az összes többi prímszám a 20. sor `return`je miatt. Az előállított prímek tárolásra kerülnek; egy soron következő páratlan szám akkor lesz prím, ha az összes eltárolt érték egyike sem osztja.

18. Prímek generálása – Vágási pontok

1. A generátorok jellemzője, hogy a `yield` segítségével futásuk felfüggeszthető, majd ettől a ponttól újra folytatható. Ennek megvalósításához szét kell vágnunk az eredeti metódust kisebb darabokra. A vágás eredményeként előálló darabok között a generátor végrehajtása megszakad, és lehetőségünk nyílik értékeket visszaadni. Hol kell azonban elvégezni ezeket a vágásokat? Úgynevezett explicit és implicit vágási pontok mentén.

2. Explicit vágási pont lesz a `yield`, azaz valójában a `return` utasítás. Ezt látva biztosan vágnunk kell, mivel a `yield` függeszti fel a generátort és ad vissza értéket.

3. Implicit vágási pontokat a különböző vezérlési szerkezetek vezethetnek be. Fontos megjegyezni, hogy ezeket a vágási pontokat csak akkor kell figyelembe venni, ha az adott vezérlési szerkezet tartalmaz egy explicit vágási pontot is, vagy egy megfelelő címkével ellátott `break` vagy `continue` utasítást. Most az egyetlen vezérlési struktúra, mely `return`t tartalmaz, a *do-while* ciklus, így ennek az eleje és a vége biztosan implicit vágási pont lesz. Ha nem bontanánk fel az ilyen struktúrákat, akkor nem lenne lehetséges egy `yield` után az utasítások folytatása.

4. A *do-while* ciklust tehát darabokra kell bontanunk. Ez a tartalmazott más vezérlési szerkezeteket alapesetben nem érinti, azonban a 14. sorban található, `loop` címkével ellátott `continue` utasítás a szétdarabolt ciklust szeretné léptetni.

5. Emiatt szükséges a `continue`-t befoglaló elágazás és `foreach` ciklus feldarabolása is. Az így létrejövő kis metódustöredékek lesznek a *continuation*ök, melyekkel mindig folytatódni fog a futás.

19. A vezérlés útja

Mielőtt elvesznénk a transzformáció során létrejövő metódusok rengetegében, nézzük meg, hogyan is fog alakulni a vezérlés útja ezek között. Összesen 9 új metódus jön létre, melyek mind az eredeti kód megfelelő részleteit tartalmazzák, oly formában, hogy a `return` utasításoknál megállítható legyen a vezérlés. A metódusok nagy számát a vezérlési szerkezetek összetettsége okozza, hiszen egyúttal ezeknek a felfüggeszthetőségét is biztosítani kell.

20. A transzformáció lépései – 1

Az áttekintés után nézzük meg, hogy milyen kód kerül az egyes metódusokba! Először egy minden generátor esetén azonos, lezáró metódusra van szükség. Ez lesz az egész generátor *continuation*je, ami a számítás befejeztét szimbolizálja. Végtelen generátorok esetén sosem kerül végrehajtásra.

21. A transzformáció lépései – 2

1. Folytassuk a transzformációt már valóban a `primes` metódus törzsével! A 3. sor `return` utasítása egy explicit vágási pont, így az, és a megelőző utasítások egy saját metódusba kerülnek.

2. Vegyük észre, hogy a deklarációból egy egyszerű értékadás lett, mivel a változó ki lett emelve a metódusokat befoglaló osztály mezőjévé. Erre azért van szükség, hogy a változó a többi metódusban is hozzáférhető legyen.

3. Az eredeti metódus által visszaadott értéket a transzformált metódus visszaadja a *trampoline*-nak, a folytatás kíséretében. Ez a még nem ismert `primes_2` metódus, ami a további teendőket tartalmazza.

22. A transzformáció lépései – 3

1. A `primes_2` most nem kerül részletes vizsgálatra, hiszen nem kifejezetten érdekes. Inkább nézzük meg, hogy a `do-while` ciklussal mi történik! Nyilvánvalóan ezt legalább két részre kell bontanunk. Az egyik a törzs, a másik pedig a fej helyén fog állni. Az utóbbit jelképezi a `primes_4` metódus.

2. A fej feltétele a transzformált kódban egy `if` formájában jelenik meg.

3. Amennyiben a feltétel teljesül, akkor a törzset jelentő metódussal folytatódik a végrehajtás.

4. Ellenkező esetben az előzőleg létrehozott befejező metódus következik. Hiszen ez a `do-while` ciklus az eredeti metódus utolsó utasítása.

23. A transzformáció lépései – 4

1. A ciklusfej után következzen a törzs transzformációja! A törzs első utasítása egy saját metódusban kap helyet, melynek folytatását a rákövetkező `foreach` ciklus transzformáltja jelenti.

2. A `foreach` ciklus utáni utasítások *continuation*je a `do-while` ciklus fejének transzformáltja lesz. Az eredetileg visszaadott `current` érték a transzformált metódusban is visszaadásra kerül.

24. A transzformáció lépései – 5

1. Még hátra van a legbonyolultabb része az eredeti kódnak – a `foreach` ciklus. Ezt három részre kell bontani, egy inicializáló metódusra, egy frissítésre és egy törzsre. A `primes_6` metódus végzi az inicializálást, azaz az iterátor elkérését a `primes` listától.

2. A frissítés `foreach` ciklus esetén az iterátor következő elemének lekérdezését jelenti. Amennyiben van következő elem, a ciklus törzsével kell folytatni. Amennyiben nincs, a ciklust követő utasítások következnek, melyeket a `primes_5` metódus jelképez. Ez a `do-while` ciklus hátralevő utasításait jelenti.

3. Végül következzen az egész `primes` metódus legérdekesebb része, a `foreach` ciklus törzse, azon belül pedig a `continue` utasítás. Ez ugrást tesz lehetővé. Az eljárás részeként folyamatosan tárolásra kerül, hogy a különböző metódusoknak mi a rákövetkezője. A címkék ezt tovább bonyolítják azzal, hogy ugrásokat tesznek lehetővé, akár több szint mélyről is. Itt is pontosan ez történik. A `continue` a külső `do-while` léptetését okozza, emiatt lesz a `primes_4` metódus, ami a ciklusfej transzformáltja, a *continuation*.

4. Ezzel elkészült a teljes metódus transzformációja. Futásidőben egy *trampoline* fogja a generált metódusok meghívását végezni, ez azonban a programozó előtt rejtve marad.

25. A vezérlés útja

Az implementációk kifejtése után talán már világosabb, hogy miért pont így épül fel a létrehozott generátor. Az egyes metódushívások között a vezérlés visszatér a *trampoline*-hoz, ami vagy visszaad egy értéket a hívónak

vagy pedig a következő megfelelő metódust hajtja végre. Előbbi esetben beszélhetünk generált értékről, és a futás tényleges felfüggesztéséről.

26. Összefoglalás

Zárásként szeretném összefoglalni az előadásom tartalmát. Kezdésként szerepelt, hogy a generátorok bevezetése indokolt a *Java* nyelvben. Ezt számtalan példával próbáltam meg alátámasztani. Ezt követően néhány, már nem támogatott könyvtár után bemutattam egy példán keresztül az általam kidolgozott eljárást. Ez az annotációkkal megjelölt metódusokat kis építőkövekre bontja, melyeket futásidőben egy *trampoline* fog össze. A transzformáció az összes vezérlési szerkezetet elágazásokkal, és a *continuation*ök megfelelő szervezésével modellezi. A programozónak csak a megfelelő annotációt kell használnia, minden mást elvégez helyette a létrehozott könyvtár.

Köszönöm a figyelmet!