Eötvös Loránd Tudományegyetem
Informatikai Kar

Programozási Nyelvek és
Fordítóprogramok Tanszék

Séma alapú refaktoráló DSL adaptálása objektumorientált programozási nyelvre

Tudományos Diákköri Dolgozat

Témavezető:

Horpácsi Dániel

tanársegéd
Programozási Nyelvek és
Fordítóprogramok Tanszék

Készítette:

Németh Dávid János

programtervező informatikus MSc
szoftvertechnológia szakirány
II. évfolyam

Budapest, 2018

Tartalomjegyzék

1. Bevezetés

A szoftverfejlesztés folyamata jellemzően iteratív jellegű, azaz a megvalósítandó program nem egyetlen lépésben készül el, hanem több, iterációkban finomított prototípus formájában teljesedik ki. Egy új szoftver első kiadását követően, az életciklusának meghatározó részét kitevő karbantartási és továbbfejlesztési fázis ugyancsak meglévő programkód javítását, bővítését teszi szükségessé. Az egyes iterációs lépések közötti módosítások halmazára tekinthetünk forráskód-transzformációkként is.

Ezen átalakítások közül refaktorálásnak nevezzük azokat, melyek a szoftver jelentését nem változtatják meg. Refaktorálásokat jellemzően valamely nemfunkcionális mutató – például karbantarthatóság vagy hatékonyság – javítására alkalmaznak. Napjainkban már a legtöbb fejlesztői környezet biztosít egyszerűen használható refaktorálásokat, azonban ezek nem adnak garanciát arra, hogy az implementációjuk valóban teljesíti a jelentésmegőrzés kitételét. Továbbá biztonságkritikus és komplex rendszerek esetén valós igény támasztható a refaktorálások helyességének formális verifikációjára is.

A funkcionális paradigmát támogató Erlang programozási nyelvhez elérhető egy olyan, refaktorálások definiálására és formális helyességbizonyítására használható módszer [7][6], amely alkalmas a fenti követelmények teljesítésére. Dolgozatomban ennek egy objektumelvű paradigmára való adaptálási lehetőségét mutatom be a Java programozási nyelven keresztül. Ehhez ismertetek egy intuitív jelentésfogalmat közelítő, többszintű ekvivalenciadefiníciót; meghatározok a célparadigma absztrakcióit karakterizáló, elsősorban előfeltételek megadásához használható szemantikus függvényeket és predikátumokat; illetve azonosítok új refaktorálási sémákat. Az így konstruált prototípust egy komplex refaktorálási szabály dekomponálására és formális definiálására használom fel, végül az előálló mikroátalakítások működését egy konkrét programon illusztrálom.

2. Kapcsolódó munkák

A helyességbizonyított refaktorálás témakörében született eredmények jellemzően a mikrolépésekre bontást veszik alapul. A módszer alapötlete az, hogy a komplex transzformációkat a lehető legelemibb lépésekre bontjuk szét, az elemi lépéseket pedig valamilyen vezérlési logika mentén építjük össze. A minimálisra csökkentett méretű refaktorálásokat könnyű felírni, megérteni és verifikálni, az összetett definíciók helyessége pedig a kompozicionalitás következményeként adódik[1].

[1]Egy összetett refaktorálás helyességéhez természetesen gondoskodni kell arról, hogy a benne felhasznált elemi átalakítások előfeltételei az egyes szabályok végrehajtásának kezdetén teljesüljenek.

Opdyke disszertációjában [9] objektumorientált rendszerekre jellemző refaktorálásokat mutat be. Műve a mikrolépésekre bontás egyik megalapozójaként három nagy átalakításhoz ad dekompozíciót, részletesen ismertetve a mikrolépések meghatározásához szükséges stratégiát és az egyes refaktorálások felírásához használt metaelméletet. Ezentúl a bemutatott transzformációk helyességét informális módon bizonyítja is.

Schafer munkájában [11] szintén a mikrolépésekre bontás fontosságát hangsúlyozza, a megvalósításhoz pedig két újszerű technikát is felhasznál. Egyrészt ahhoz, hogy a refaktorálásokat intuitívabban lehessen megfogalmazni, olyan nyelvi kiterjesztéseket vezet be, amelyek a célnyelvben nem léteznek, csak a transzformáció folyamatában játszanak szerepet, az eredmény előállása előtt pedig eliminálásra kerülnek. Másrészt, bonyolult előfeltételek meghatározása helyett a jelentés megőrzését invariánsok teljesüléséhez köti. Ehhez a refaktorálások alkalmazása előtt eltárol jelentést karakterizáló tulajdonságokat – adat- és vezérlésfolyam, illetve kötések –, majd a transzformáció végrehajtása után ellenőrzi ezek megmaradását.

3. Séma alapú refaktorálás

Az alapul vett módszer [7] egy olyan nyelvet kínál, amely lehetőséget ad refaktorálások feltételes termátírások formájában történő definiálására, támogatva a mikrolépésekre bontás technikáját. Az így megfogalmazott átalakítások a módszer célnyelvének szemantikáján keresztül válnak helyességbizonyíthatóvá. Az újonnan megalkotott definíciók verifikációját az teszi lehetővé, hogy ezek alapjául kizárólag előre meghatározott és bizonyított sémák használhatók fel. A továbbiakban röviden bemutatom a módszer leírónyelvét, illetve a verifikációhoz használt háttérelméletet.

3.1. Leírónyelv

A leírónyelv magja a feltételes termátírás, amely segítségével a konkrét szabályokat kiinduló és transzformált programminták formájában lehet megadni, ahol ezen programmintákban konkrét forráskódra illeszkedő metaváltozók felhasználásával lehet a kívánt szerkezetű programokat zárt módon kijelölni. Az ilyen átírási szabályok előfeltételeit előre definiált, az Erlang metaelméletét körülíró szemantikus függvények és predikátumok, illetve ezekből elsőrendű logikai összekötőkkel előálló formulák alakjában lehet megadni.

Tekintsük például a következő feltételes átírási szabályt, amely egy Erlang listakifejezés fejrészét emeli ki egy új változóba:

```
1 ||    [Head | Tail]
2 ||    --------------------
```

```
3       X = Head, [X | Tail]
4   when
5       fresh(X)
```

Ennek első, **when** előtti részében két, törtvonallal elválasztott mintával tudjuk leírni a tényleges transzformációt. A vonal feletti minta határozza meg azt, hogy szerkezetileg milyen *alakú* programrészleten hajthatjuk végre az átírást, a vonal alatti minta pedig azt írja le, hogy egy illeszkedő konkrét programrészt milyen alakúra transzformálunk. A mintaillesztés szokásos koncepciójának megfelelően az illeszkedés vizsgálatakor az alakot kijelölő metaváltozókat (a példában `Head` és `Tail`) kötjük is a konkrét program illeszkedő részeihez. A transzformáció előfeltételét a definíció második, **when** utáni részében adhatjuk meg (a példában azt várjuk el, hogy az implicit bevezetett `X` változó valóban új legyen).

Ahogy a konkrét algoritmusrészletekkel az előre bizonyított programozási tételeket példányosítjuk, úgy az így definiált feltételes termátírásokat az előre verifikált sémák példányosítására lehet felhasználni, eredményül pedig (egy konkrét algoritmus helyett) egy konkrét refaktorálási szabály áll elő.

Például, a függvényszignatúrát transzformáló sémát egy első két paramétert megcserélő átírási szabállyal példányosítva megkapjuk azt a konkrét refaktorálást, amelyet tényleges függvénydefinícióra alkalmazva megcserélhetjük a definíció paramétereinek sorrendjét.

```
1   function signature refactoring swapFirstTwo
2       F(A, B, Ps..)
3       -------------
4       F(B, A, Ps..)
```

Vegyük észre, hogy a fenti átírás csak akkor lesz refaktorálás, ha a formális paraméterlistának megfelelően, a függvény hívásaiban is megcseréljük az első két konkrét paramétert. Annak ellenére, hogy ezt explicit nem fogalmaztuk bele a fenti sémapéldányba, az mégis megőrzi a jelentést. A háttérben ugyanis a szükséges kompenzációs transzformációkat maga a séma tartalmazza, a nyelv pedig így éri el, hogy a felhasználó által megfogalmazandó átírások a lehető legegyszerűbbek lehessenek.

A sémapéldányosítással előálló, tényleges mikrorefaktorálásokat összetett refaktorálások segítségével lehet komponálni. Ezekben az összeépített szabályok célnódusait – azaz, azokat a szimbolikus programrészleteket, melyeken egy-egy átírást alkalmazunk – és végrehajtási módját rendre különböző szelekciós és vezérlést meghatározó nyelvi konstrukciókkal tudjuk megfogalmazni. Az előző, jórészt deklaratív definíciós módszertől eltérően az összetett refaktorálásokat már imperatív jellegben, utasítások kiadásával tudjuk meghatározni. Ha egy ilyen, összeépített refaktorálás valamely komponensének alkalmazása sikertelen – azaz, nem illeszkedik a kiinduló

átírási minta, vagy nem teljesül az előfeltétel –, az addigi módosítások visszagörgetésre kerülnek, az összetett refaktorálás pedig sikertelenül terminál.

Például, a következő összetett refaktorálás (`f()`) előbb a választott célkifejezésen alkalmazza a `g()`, majd az eredeti cél befoglaló függvénydefinícióján alkalmazza a `h()` refaktorálást.

```
1 composite refactoring f
2 do
3         g()
4         function().h()
```

3.2. Verifikáció

A helyességbizonyítás Ciobaca nyelvfüggetlen, programminták ekvivalenciájának verifikálására lehetőséget adó bizonyítási rendszerén [3] keresztül történik, amely Stefanescu szemantika alapú programtulajdonság-verifikáló rendszerét használja fel. Utóbbi a célnyelvet formálisan, egzakt módon definiáló operációs szemantikát használja a területet jellemzően domináló, Hoare-hármasokra építő axiómák helyett. A módszer motivációja az, hogy az operációs szemantika könnyebben írható le, szélesebb körben használható fel, illetve lehetőséget ad arra, hogy egyszerre nyelvi definícióként és helyességbizonyításra használható, logikai eszközként is szolgáljon.

Az operációs szemantika és a logika közötti kapcsolatot az illesztési logikára épülő elérhetőségi logika teremti meg [10]. Az illesztési logika a mintaillesztéshez köthető problémák leírására használható, ahol a formulák valamilyen nyelv feletti minták, az interpretációk a nyelv konkrét mondatai, a következményfogalmat pedig a mintára való illeszkedéssel fejezzük ki. Az elérhetőségi logika minták közötti elérhetőséget formalizál, lehetőséget teremtve arra, hogy a helyességbizonyításban használt, $\{\{Q\}\}\ S\ \{\{R\}\}$ alakú előfeltétel–program–utófeltétel hármasokat olyan elérhetőségi követelményekre fordítsuk le, amelyek aztán a célnyelv egy operációs szemantikájával lesznek formálisan vizsgálhatók.

Stefanescu általános, programtulajdonságok bizonyítására használható rendszerét [12] Ciobaca úgy bővítette ki, hogy két program ekvivalenciáját egy speciális, aggregált programnak egy aggregált szemantika mentén vizsgált programtulajdonságaként tekintette. Az aggregált program és szemantika rendre az eredeti programok és szemantikák felhasználásával konstruálható meg. A verifikáláshoz először definiálni kell az eleve ekvivalensnek tekintett programpárokat leíró mintákat, majd a rendszer levezetési szabályai használhatók ekvivalencia igazolására.

Az alapul vett refaktoráló nyelvben a sémák előre verifikálásra kerülnek. Erre a sémákhoz tartozó előfeltételek, illetve a sémákhoz a példányosításukra használható feltételes átírások körének szerződések mentén történő megszorítása ad lehetőséget.

A sémák bizonyítása jellemzően a szemantika mentén, strukturális indukcióval történik, ahol az indukció alapeseteit a sémákhoz tartozó szerződések igazolják. Ekkor a sémapéldányok helyességéhez már csak azt kell belátni, hogy a példányosításra használt átírási szabályok teljesítik a sémában foglalt szerződést. Ezt már jellemzően automatizáltan lehet megtenni a korábban ismertetett bizonyítórendszert támogató eszközökkel, így elérhető az, hogy a sémákat alapul vevő, utólag definiált refaktorálási példányokat is automatikusan lehessen verifikálni.

4. A módszer adaptálása

A módszer adaptálásához először ki kell jelölni egy konkrét paradigmát, illetve célnyelvet. Ezután meg kell határozni, hogy a kiválasztott célnyelvben mikor tekintünk két programot ekvivalensnek. Ezt követően azonosítani kell olyan összetett refaktorálásokat, amelyek mikrolépésekre történő bontásával, illetve az így kapott alaplépések feltételes termátírási szabályokkal való leírásával meghatározhatók a refaktoráló nyelv azon módosításai, amelyek az adódó mikrolépések megfogalmazhatóvá tételéhez szükségesek. A lépések előfeltételeit a kijelölt ekvivalenciadefiníció ismeretében lehet megkonstruálni, az így előálló formulákból pedig szintetizálhatók azok a szemantikus függvények és predikátumok, melyek az új célnyelv metaelméletének kifejezését teszik lehetővé. Ezután a meghatározott mikroátalakításokból általánosíthatók a célnyelvhez illeszkedő sémák, ezek előfeltételeivel és példányosításra tett szerződéseivel együtt. Itt már használhatók a korábbiakban körülírt metaelméleti fogalmak.

A célnyelvvel együtt érdemes kijelölni annak egy operációs szemantikáját is, melynek ugyan a verifikációs fázisban lesz igazán fontos szerepe, de már korábban, az ekvivalencia és a metaelmélet vizsgálatában is hasznosnak bizonyulhat, hiszen elősegíti a nyelv fogalmainak mélyebb és precízebb megértését. A verifikációhoz formálisan, a szemantika felhasználásával kell modellezni mind az ekvivalenciadefiníciót, mind a korábban kijelölt szemantikus függvényeket és predikátumokat. Ezt követően az operációs szemantikából meg kell konstruálni a bizonyítórendszer használatához szükséges, aggregált konfigurációt és szemantikát, ezzel együtt kijelölve az ekvivalenciafogalom formális definíciójához illeszkedő, eleve ekvivalensnek tekintett programpárokat karakterizáló mintahalmazt. Végül a meghatározott sémákat kell manuálisan helyességbizonyítani, szintén az operációs szemantikát és az ekvivalenciadefiníciót alapul véve.

4.1. A célnyelv meghatározása

A célnyelvet úgy célszerű meghatározni, hogy az kellően tiszta és magasszintű legyen, ezáltal biztosítva lehetőséget a paradigma fogalmainak tömör vizsgálatára. Kerülni szeretnénk tehát az olyan nyelveket, amelyek a paradigma szempontjából feleslegesen bonyolult nyelvi konstrukciókat tartalmaznak. Ezen követelményeket megfontolva kézenfekvően adódnak a paradigmát általánosan bemutató, oktatási céllal létrehozott nyelvek. Ilyen például az Aiken által ismertetett COOL [1], vagy a Corliss által bemutatott Bantam Java [4]. A probléma ezekkel viszont az, hogy az adaptáláshoz kritikus formális szemantika, illetve az implementáció szempontjából lényeges eszköztámogatás nem, vagy csak erősen korlátozottan áll hozzájuk rendelkezésre, továbbá a láthatóságuk, ismertségük sem széleskörű.

Megfelelő alternatívát jelent viszont a tisztán objektumorientált és kellően magasszintű Java, amelyhez elérhető formális szemantika [2], illetve refaktorálások és metaelmélet implementálására alkalmas eszköztámogatás is, továbbá, a legnépszerűbb programozási nyelvként [13] széleskörű láthatóságot is magáénak tudhat. A Java választásának hátrányait a bonyolultsága adja: a további vizsgálathoz erősen le kell szűkíteni a felhasználható nyelvi elemek körét, ennek megfelelően pedig a szemantikát is. Újabb korlátozásás adódik abból, hogy a rendelkezésre álló formális szemantika a Java 1.4-es verzióját támogatja, így például hiányoznak belőle a típusparaméterek és a lambda függvények.

A továbbiakban egy olyan, 1.4-es verziójú Java magnyelvvel fogunk dolgozni, amely nem tartalmazza vagy engedélyezi a következőket:

- ugró utasítások (a `return` kivételével),
- kivételkezelés,
- láthatóságon kívüli módosítók,
- mezőinicializáló kifejezések,
- osztályinicializáló blokkok,
- lokális osztálydefiníciók,
- csomagok (az alapértelmezett csomag kivételével),
- reflekció,
- annotációk,
- szálkezelés és
- JVM manipuláció (például dinamikus osztálybetöltés).

4.2. Az ekvivalencia definiálása

A programmintákra kimondott ekvivalenciafogalom meghatározása kulcsfontosságú a módszer adaptálásában, hiszen ez alapján döntjük el, hogy egy transzformációt mi-

kor tekintünk refaktorálásnak, vagyis olyan átalakításnak, amely garantáltan megőrzi a célprogram jelentését. Az ekvivalencia különféle karakterizációiról Horpácsi [6], Schafer [11], Opdyke [9] és Ciobaca [3] is értekezik. Java programok esetén kézenfekvően adódik, hogy az ekvivalenciát egy-egy főprogramhoz viszonyítva vizsgáljuk. Viszont, ahogy arra Opdyke is utal, a programhierarchiában különböző szinten lévő entitásokat – például utasításokat vagy osztályokat – módosító refaktorálásoknál a jelentés megőrzésének intuitív értelmezéséhez kevéssé illeszkedik az a megközelítés, hogy az ekvivalenciát főprogramok mentén határozzuk meg.

Vegyük például azt az átalakítást, amely egy metódusdefiníció belsejében megcserél két utasítást. Világos, hogy általánosan nem adható garancia arra, hogy az így adódó definíció jelentése a kiindulóéval megegyezik. Viszont ha az ekvivalencia alapjául főprogramok jelentésének egyezését tekintenénk, refaktorálásnak nevezhetnénk ezt az átalakítást is például abban az esetben, ha a módosítandó metódus nem lenne a főprogramból indulva elérhető. Általánosítva ezt a gondolatot, egy ilyen ekvivalenciát alapul véve minden metódust vagy metódustörzs tartalmát módosító refaktorálás előfeltételéhez diszjunkcióval kapcsolhatnánk azt a kitételt, hogy a transzformáció céljának befoglaló metódusa főprogramból nem elérhető, hiszen ebben az értelmezésben ezeken az entitásokon bármilyen, akár jelentést megváltoztató transzformáció is alkalmazható, feltéve persze, hogy a transzformáció nem idéz elő fordítási hibát.

Ezen inkonzisztencia feloldására tekintsünk egy olyan, összetett ekvivalenciafogalmat, amely a kifejezések és utasítások, a blokkok, a főprogramok és az osztályok szintjén különböző módon ragadja meg a jelentés megőrzésének tulajdonságát. A klasszikus ekvivalenciafogalmat itt a formális szemantika ismeretében határozzuk meg, a szemantikában definiált konfigurációkat általánosító mintákat alapul véve.

$$\langle\langle K\rangle_{code}\langle C\rangle_{classes}\langle M\rangle_{methods}\langle S\rangle_{stack}\langle H\rangle_{heap}\langle E\rangle_{s.effects}\cdots\rangle_{cfg}$$

A formális szemantikákra jellemző módon ezen konfigurációk nemcsak a vizsgált programkódot, hanem a teljes végrehajtási környezetet – osztályhierarchiát, stacket, heapet, mellékhatásokat, stb. – is modellezik, így mikor a tárgyalt programkódok ekvivalenciáját vizsgáljuk, valójában ezen, teljes végrehajtási környezetet leíró konfigurációkat hasonlítjuk össze úgy, hogy azok kód cellájába az összehasonlítandó programentitásokat helyezzük. Két programmintát akkor nevezünk ekvivalensnek, ha a korábban ismertetett bizonyítórendszer [3] szabályai felhasználásával ekvivalensnek tekintett formára hozhatók. Opdyke nyomán ezt a forma-ekvivalenciát programentitásonként definiáljuk úgy, hogy a legszigorúbb ekvivalenciafogalmat a vizsgált entitásokra refaktorálási célként tekintve, azok programhierarchiában elfoglalt helye alapján gyengítjük. A cél itt az, hogy az egyes átalakítások jelentésmegtartását

mindig a transzformáció hatókörén már éppen kívülről szemléljük, hiszen így tudjuk biztosítani, hogy kellően sokféle átalakítás úgy legyen leírható, hogy a lokalizáltabb transzformációk modulárisan ellenőrizhetők maradjanak.

Definiáljuk a fent körülírt forma-ekvivalenciát a következő szabályokkal:

- A legalacsonyabb szinten, a kifejezések és utasítások esetén nem tételezhetünk fel semmilyen, ekvivalenciát befolyásoló környezeti információt, így itt a klasszikus ekvivalenciafogalmat kell használnunk, tehát – megfelelő absztrakciók mellett – minden cellának egyeznie kell.

- Eggyel magasabb szinten, blokkok esetén már felhasználhatjuk azt a többletinformációt, hogy a blokkok lokális változói közvetlenül nem befolyásolhatják a program jelentését. Ebből kiindulva két blokk ekvivalenciájának vizsgálatakor elvonatkoztathatunk a blokkok lokális változóitól, vagyis elég csak a blokkon kívül is elérhető elemeket összehasonlítanunk.

  Megjegyzés. Általánosságban, a mintákban előforduló változókat úgy tudjuk kezelni, hogy a változóra történő első hivatkozás elérésekor a hivatkozott változót hozzáadjuk a befoglaló konfiguráció veréhez. Ha a változó értéke nem ismert (mert például a blokkon kívül lett meghatározva), a veremben egy értékmintát rendelünk hozzá.

  Ezt az ekvivalenciafogalmat kiterjeszthetjük a metódusokra is, a törzsüket mint blokkokat, a paramétereiket mint lokális változókat alapul véve. Ha a vizsgált metódus visszatérési típusa nem `void`, a visszaadott értékmintát a kód cellába helyezzük, az ilyen értékmintákra szintén szemantikus egyezést elvárva.

- A blokkekvivalencia speciális esetét képezi a főprogram-ekvivalencia is, hiszen az előzőek alapján a főprogramként kezelt metódusra blokként is tekinthetünk. Ugyanakkor különbséget jelent itt az, hogy a blokkon kívül látható elemekre további engedményt tehetünk: az osztálystatikusok adattartalma a főprogram lefutása után már nem releváns a jelentés karakterizálása szempontjából, így ilyenkor elég a mellékhatásokat összehasonlítanunk.

- A legspeciálisabb ekvivalenciaágat az osztályhierarchiákra vezetjük be. A helyzet különlegessége annak köszönhető, hogy az eddigiektől eltérően itt két, összehasonlítandó metódusdefiníció helyett osztályok és ezek szerkezetének jelentését kell összevetnünk. A nehézség ebben az, hogy *jelentést* kell *szerkezethez* rendelnünk. Vegyük észre, hogy a szokásos ekvivalenciát használva kizárnánk például az olyan transzformációkat, amelyek a metódusok jelentésének megtartásával új szerkezeti elemeket, például új metódusokat vagy osztályokat vezetnek be.

Az intuitív értelmezés közelítéséhez gyengítsük az egyébként használt ekvivalenciarelációt részbenrendezéssé, vagyis a szimmetria helyett válasszunk antiszimmetriát. Az alapötlet egy olyan kapcsolat megfogalmazása, amely azt fejezi ki, hogy a *módosított* osztályszerkezet publikus interfészén keresztül *legalább* a *kiinduló* osztályszerkezet publikus interfésze mentén elérhető szolgáltatásokat tudja nyújtani, ráadásul azokat ekvivalens módon. Ezek tükrében tekintsük a következő fogalmakat.

Definíció. Azt mondjuk, hogy a H' hierarchia *megfelel* a H hierarchiának, ha minden, H-ban definiált T típushoz létezik olyan, H'-ben definiált T' típus, hogy T' *megfelel* T-nek.

Definíció. Azt mondjuk, hogy a T' típus *megfelel* a T típusnak, ha

- egyezik a nevük,
- és vagy egyiknek sincs, vagy mindkettőnek van (akár implicit, ld. `Object`) őstípusa, utóbbi esetben pedig T' S' őstípusa *megfelel* T S őstípusának.
- és (amennyiben ezen típusok nem primitívek[2]) minden, T-ből elérhető, nyilvános F mezőhöz létezik olyan, T'-ből elérhető, nyilvános F' mező, hogy F' *megfelel* F-nek, továbbá minden, T-ből elérhető, nyilvános M metódushoz létezik olyan, T'-ből elérhető, nyilvános M' metódus, hogy M' *megfelel* M-nek.

Definíció. Azt mondjuk, hogy az F' mező *megfelel* az F mezőnek, ha

- egyezik a nevük,
- és F' típusa *megfelel* F típusának.

Definíció. Azt mondjuk, hogy az M' metódus megfelel az M metódusnak, ha

- egyezik a nevük,
- és M' visszatérési típusa *megfelel* M visszatérési típusának,
- és M' $(p'_1, \ldots, p'_m)$ formális paraméterlistája, valamint M $(p_1, \ldots, p_n)$ formális paraméterlistája olyan, hogy $m = n$ és minden $1 \leq i \leq n$ esetén p'_i típusa *megfelel* p_i típusának,
- és M és M' törzsei ekvivalensek (ld. blokk-ekvivalencia).

A fenti definíciókkal a kiinduló hierarchiát H, a módosítottat H' helyében tekintve megkapjuk a vizsgálandó relációt. Vegyük észre, hogy bizonyos refaktorálások nemcsak a kódot, hanem például az osztályhierarchiát tartalmazó cellát is

[2] A névegyezés miatt nincs szükség ezt a tulajdonságot külön-külön feltenni.

módosíthatják (például, ha bevezetnek egy új metódusdefiníciót). Azt is fontos észrevennünk, hogy az általános verifikáció szintjén a transzformációk hatását – azok definíciójához hasonlóan – eleve egy általánosított mintából kiindulva vizsgáljuk, vagyis amikor általános és transzformált konfigurációkat vetünk össze, az ekvivalenciát általános formában, programminták szintjén kell belátnunk.

4.3. A leírónyelv bővítése

A 3.1. részben ismertetett nyelvet nemcsak az objektumorientált paradigmához kapcsolódó fogalmak kifejezésére alkalmas elemekkel bővítjük ki, hanem bevezetünk olyan konstrukciókat is, melyek a refaktorálási definíciókat egyszerűbben, kényelmesebben megfogalmazhatóvá teszik.

A kiinduló nyelvben a `this` pszeudováltozó jelöli a refaktorálási aktuális célját. Mivel ugyanez a pszeudováltozó az objektumorientált paradigmában az aktuális objektum jelölésére szolgál, a félreértelmezések elkerülése végett az aktuális célt a `target` pszeudováltozóval tesszük a refaktorálási definíciókban meghivatkozhatóvá.

A majdani sémákhoz, sémapéldányokhoz a következő záradékokat adjuk hozzá:

- `target`: A sémát példányosító átírási szabály kiinduló mintája és a refaktorálás célját kijelölő minta függetlenítésére használható záradék. A korábban ismertetettek alapján a példányosító átírási szabály törtvonal feletti részét illesztjük a refaktorálás céljára. Ennek következményeként, ha a kiinduló minta része a célmintának, a kiinduló mintát újra meg kell ismételni a célmintában.

```
1 S ; S'
2 ----------
3 { S ; S' }
```

  Továbbá, mintaszerűen a bemenetnek csak a kiinduló mintára illeszkedő részeit tudjuk elérni, azaz például egy `S;S'` alakú program esetén, ha egy refaktorálást `S`-re alkalmazunk, a definíción belül `S'`-t legfeljebb szemantikus függvények segítségével tudjuk elérni

  A tárgyalt záradék mögötti motiváció éppen ezek áthidalása:

```
1     target
2     ----------
3     { target }
4 target
5     S ; S'
```

```
1     target ; S'
2     -----------
3     S' ; target
4 target
5     S
```

- `shadowed references`: Valamilyen kódmozgatás hatására megváltozott kötésű hivatkozások kompenzációs átírásának megadására (jellemzően az eredeti kötés minősített névvel történő visszaállítására) használható. Például, ha

egy adattag nevével megegyező változó deklarációját egy eredetileg a kérdéses adattagra feloldódó hivatkozás elé mozgatjuk, a megváltozó kötést ebben a záradékban az eredetileg adattagra hivatkozó referencia `this` minősítővel történő ellátásával tudjuk helyreállítani.

- `top level definition`: Fájlszintű programentitás (például osztály vagy interfész) hozzáadására használható definíciós záradék[3].
- `definition in class`: A refaktorálás célját befoglaló osztályban új osztályelem (például mező vagy metódus) hozzáadására használható definíciós záradék.
- `definition in super`: A refaktorálás célját befoglaló osztály ősosztályában új osztályelem hozzáadására használható definíciós záradék.

Az átírási szabályok alapját képező mintákat a ; rugalmas értelmezésével tesszük kifejezőbbé. Ezt a mintát a konkrét programokban előforduló ; karakterektől függetlenül értelmezzük szekvenciát jelölő vezérlési szerkezetként. Például, az `S;S'` mintára illeszkedőnek tekintjük az `{}{}` és az `{};{}` programokat is.

4.4. Új szemantikus függvények és predikátumok bevezetése

A szemantikus függvények és predikátumok teszik a refaktorálási definíciókon belül hozzáférhetővé a célnyelv statikus szemantikáját, amit főként a sémák és sémapéldányok előfeltételeinek meghatározásához használhatunk fel. Ugyan az alapul vett módszer törekszik a nyelvfüggetlenségre, a célnyelv statikus szemantikájához ilyen szorosan kapcsolódó részei jelentős átalakításokat igényelnek, ha a paradigmát is le szeretnénk cserélni. A továbbiakban először ismertetem az újonnan bevezetett szemantikus függvények meghatározásához szükséges reprezentációt és formulanyelvet, majd áttérek néhány tényleges, paradigmára jellemző szemantikus kapcsolatot leíró fogalom bemutatására.

Reprezentáció és formulanyelv

A reprezentáció alapját a célnyelv absztrakt szintaxisfáit leíró modell adja, hiszen ez jól illeszkedik a definiálandó statikus szemantikus tulajdonságok absztrakciós szintjéhez. Ennek megfelelően vezessük be a `Node` alaptípust, amely a szintaxisfák egy tetszőleges csúcspontját reprezentálja. Specializáljuk ezt úgy, hogy a célnyelv főbb entitásaihoz tartozzon egy-egy altípus, például definiáljuk a `Class`, `Method`, `Field`, `Block`, `Parameter`, `Variable` és `Reference` típusokat rendre egy-egy osztály, metódus, mező, blokk, formális paraméter, változó és hivatkozás ábrázolásához.

[3]Definíciós záradéknak azon záradékokat nevezzük, melyekben csak csereminta (tört vonal alatti, ld. 3.1. rész) adható meg, kiinduló minta (tört vonal feletti, ld. 3.1. rész) nem.

Az előfeltételek megfogalmazásához gyakran lesz szükségünk programentitások gyűjteményének kezelésére, így vezessünk be ezek reprezentálására szolgáló, paraméteres típusokat is, például a rendre halmazt, listát, gráfot és fát ábrázoló `Set<T>`, `List<T>`, `Graph<T>` és `Tree<T>` gyűjteményeket. Ezekhez definiálhatjuk a szokásos műveleteket, például a nemüres lista utolsó elemét visszaadó `last`, a halmazt egy predikátum (itt: speciális névtelen függvény) alapján megszűrő `filter`, vagy a gráf ciklikusságát eldöntő `isCyclic` függvényeket.

Vegyük észre, hogy a reprezentáció és a segédfüggvények elnevezésében a Java konvencióihoz közelítettünk. Ez egy általános tervezési döntés, amely azt irányozza elő, hogy az előfeltételek formáját minél inkább a célnyelvhez hasonlóvá tegyük. Ugyanakkor, ahol ez indokolt, a célnyelvhez való lehető legszorosabb illeszkedés helyett az olvashatóságot helyezzük előtérbe. Például, a konjunkciós logikai összekötőt a Java-ban megszokott `&&` helyett az `and` kulcsszóval jelöljük, továbbá a magasabbrendű függvényeket is a megszokottnál egyszerűbb szintaxissal, `paraméterek : törzs` formájában tesszük leírhatóvá.

Szemantikus függvények és predikátumok

A szemantikus függvényeket és predikátumokat az alapján csoportosítjuk, hogy milyen statikus szemantikai tulajdonság kifejezésére szolgálnak. Ennek megfelelően kijelölünk az ekvivalencia karakterizálásához általánosan felhasznált adat- és vezérlésfolyamról, illetve kötésekről információt adó függvényeket, de meghatározunk olyan eszközöket is, melyek ugyan besorolhatók lennének az előző három kategória egyikébe, de önálló tárgyalásukat az objektumorientált paradigmához való erős kötődésük indokolja.

- **Adatfolyam:** Az adatfolyam elemzése arról szolgáltat információkat, hogy a program egy adott pontján (például egy függvényhívásnál) milyen lehetséges változóértékek (például konkrét argumentumok) fordulhatnak elő. Ezen tulajdonságokra elsősorban változók és mezők használatának vizsgálatához lehet szükség, de a vizsgálandó értékek megszorítása révén a verifikációt is egyszerűsíthetik. Itt célszerű lekérdezhetővé tennünk egy entitás lokális változóit (`localVariables`), változó olvasásait és írásait (rendre `variableReads` és `variableWrites`), illetve elért mezőit (`accessedFields`).
- **Vezérlésfolyam:** A program vezérlésfolyama a benne szereplő utasítások lehetséges végrehajtási sorrendjeit írja le. Jellemző fogalma a végrehajtási út (`Path`), amely utasítások sorozatából áll. Például a `paths` függvény segítségével lekérdezhetjük egy programentitás összes végrehajtási útját, vagyis az entitásban szereplő utasítások összes lehetséges futtatási sorozatát. Ezen utak vizsgálatát a bonyolultabb vezérlési szerkezetek (elágazás, ciklus) nehezítik meg.

Schafer [11] nyomán bevezetjük a vezérlésfolyam-rákövetkező `controlSuccessor`, illetve a metódustörzsek utolsó (utáni) vezérlési pontját jelképező, szimbolikus `exitNode` csúcsot. Lekérdezhetővé tesszük továbbá a metódusdefinícióból induló hívási gráfot (`callGraph`), amely az a maximális, osztálynévvel minősített metódusdefiníciókkal élcímkézett, irányított gráf, amely a kezdetben vizsgált definícióból indul, két csúcs között pedig pontosan akkor vezet él, ha a forráscsúcsnak megfelelő definícióban van olyan metódushívás, melynek szignatúrájára illeszkedik a célcsúcsnak megfelelő definíció, azaz a forrásdefiníció meghívhatja a céldefiníciót.

Tekintsük az alábbi példát:

```
1 class A {
2     public void f(B b) { b.g(); }
3 }
4 class B {
5     public void g() {}
6 }
7 class C extends B {
8     public void g() {}
9 }
```

Itt `A::f` hívási gráfja három csúcsból áll, `A::f`, `B::g` és `C::g` címkékkel, a csúcsok között pedig két él fut, ahol mindkettő `A::f`-ból indul a másik két csúcs irányába.

- **Kötés:** A kötések határozzák meg, hogy a program egy adott pontján egy adott névvel melyik entitást érjük el. Ehhez kapcsolódóan mondjuk ki a hamarosan kulcsfontosságúvá váló definíciós hatókör (`definitionScope`) fogalmát.

  Definíció. Adott D metódusdefiníció definíciós hatókörén azt az osztályhalmazt értjük, amely pontosan azokat az osztályokat tartalmazza, amelyek példányai D szignatúráján keresztül D-t érik el.

  A definíció illusztrálásához vegyük a következő példát:

```
1 class A {
2     public void f() {}
3 }
4 class B extends A {
5     public void f() {}
6 }
7 class C extends B {
8     public void f() {}
```

```
9  }
10 class D extends B {}
```

Itt `A::f` definíciós hatóköre `A`-t, `B::f` definíciós hatóköre `B`-t és `D`-t, `C::f` definíciós hatóköre pedig `C`-t tartalmazza.

- **Paradigma:** Az objektumorientált nyelvek statikus helyességéről való érvelést leginkább a legjellemzőbb dinamikus aspektusok, az öröklődés és a dinamikus kötés nehezítik meg. Az ezektől függő információkra statikusan csak becsléseket tudunk adni, emiatt az érintett előfeltételeket úgy fogalmazzuk meg, hogy velük az összes lehetséges kötés esetén is megőrizzük a jelentést.

  Az öröklődés mentén definiált altípusosság ellenőrzésére bevezetjük az `isSubType` predikátumot, egy adott osztály ős, illetve leszármazotti hierarchiáját (rendre `superHierarchy` és `subHierarchy`) pedig pontosan rendre az összes, közvetlen vagy közvetett ős, illetve leszármazott osztályokkal csúcscímkézett, közvetlen öröklődési kapcsolatoknak megfelelően irányított fával adjuk meg.

  Metódusdefiníciók hozzáadásakor vagy mozgatásakor gyakran kell vizsgálnunk azt, hogy a beszúrás helyén kezdetben még nem létező definíció hogyan fog viselkedni. A továbbiakban az ilyen, ténylegesen még nem hozzáadott definíciót nevezzük elődefiníciónak.

  Ekkor, a 4.2 részben ismertetett hierarchia-megfelelésre építve próbáljuk meg formalizálni azt, hogy egy elődefiníció beszúrása miatt mikor sérül a megfelelés. Vegyük észre, hogy ez pontosan akkor történik meg, ha elődefiníció egy publikus definícióból indulva kerül meghívásra úgy, hogy a már elődefiníciót hívó publikus definíció nem lesz blokk-ekvivalens az elődefiníció beszúrása előtti változatával.

  Az ezt a jelenséget kifejező szemantikus predikátumot viszont túl költséges lenne így megfogalmazni. Gondoljunk csak a fentebb részletezett, dinamikus kötés által indukált statikus bizonytalanságra, továbbá az előfeltételen belüli ekvivalenciavizsgálatra. Ennek tükrében, az alpont kezdő bekezdésével összhangban az ilyen szemantikus függvényektől csak teljességet várunk el, helyességet nem (vagyis, az ilyen szemantikus függvények jelezhetnek tévesen megfelelésséртést), továbbá kerüljük azt, hogy bennük ekvivalenciát vizsgáljunk.

  Tovább boncolva a problémát világossá válik, hogy egy elődefiníció csak akkor ronthat el megfelelést, ha felüldefiniál egy már meglévő metódust, hiszen különben biztosan nem kerülhetett sehol sem olyan szignatúra meghívásra, amely a beszúrandó elődefinícióra is feloldódhatna.

  Vegyük észre, hogy az elérhetőségnek két dimenziója áll elő attól függően, hogy a veszélyesen hivatkozó definíció az elődefiníció osztályhierarchiájában

található-e. A jelenséget könnyebb elképzelni, ha egyelőre csak az elődefinícióban és a potenciálisan hivatkozó, publikus definícióban gondolkozunk. Amennyiben a publikus definíció az elődefiníció osztályhierarchiáján kívüli, úgy pontosan akkor romolhat el a megfelelés, ha azon referenciának, amelyen keresztül a vizsgált publikus definícióban a elődefiníciónak megfelelő szignatúra meghívásra kerül, a statikus típusa olyan, hogy az vagy felmenője az elődefiníció osztályának, vagy az elődefiníció definíciós hatókörén belüli osztály. Hiszen ilyenkor előfordulhat, hogy ezen a referencián keresztül egy olyan objektumot érünk el, amelyben a hivatkozott szignatúra az elődefinícióra oldódna fel, sértve ezzel a megfelelést.

Ha ezzel szemben a veszélyesen hivatkozó definíció az elődefiníció osztályhierarchiáján belüli, pontosan akkor sérülhet a megfelelés, ha a hivatkozó definíciónak és a vizsgált elődefiníciónak nem diszjunkt a definíciós hatóköre. Valóban, ilyenkor a metszetükben lévő osztályokban a hivatkozó publikus definíció szignatúrája a hivatkozó publikus definícióra oldódik fel (mivel ez az osztály benne van a definíciós hatókörében), az ebben a definícióban szereplő, veszélyes szignatúrahivatkozás pedig a vizsgált elődefinícióra fog kötni (mivel ez az osztály az elődefiníció definíciós hatókörében is benne van), elrontva ezzel a megfelelést. Megfordítva a gondolatmenetet, hierarchián belüli szignatúrahivatkozások esetén reménykedhetünk abban, hogy a problémás publikus definíció és a beszúrandó elődefiníció hatóköre diszjunkt.

Továbbgondolva a problémát észrevehetjük, hogy a fentebb két szereplővel lejátszott helyzet akár több köztes definícióval is kiegészülhet, a hierarchián belül és kívül egyaránt. Intuitívan úgy lehet ezt elképzelni, hogy ilyenkor több definíciós hatókör metszetét kell vizsgálnunk, a megfelelés pedig akkor sérülhet, ha egy publikus definícióból indulva, definíciós hatókörök mentén eljutunk az elődefinícióig.

A fentiek tükrében definiáljuk egzakt módon az elődefiníció elérhetőségének fogalmát.

Definíció. Azt mondjuk, hogy a p elődefiníció *elérhető*, ha

- felüldefiniál metódust,
- és *inter-* vagy *intrahierarchia-elérhető*.

Definíció. Azt mondjuk, hogy a p elődefiníció *interhierarchia-elérhető*, ha létezik olyan definíció,

- amely az elődefiníció osztályhierarchiáján kívüli,
- és amely hivatkozik az elődefiníció olyan szignatúrájára,

    - amely az elődefiníció befoglaló osztályának egy felmenőjével,
    - vagy amely az elődefiníció definíciós hatókörében lévő osztállyal minősített,
  - és amely *nemmegszorított intrahierarchia-elérhető*.

Definíció. Azt mondjuk, hogy a p elődefiníció *intrahierarchia-elérhető*, ha

- vagy publikus metódust definiál felül,
- vagy létezik olyan definíció, amely
  - hivatkozik az elődefiníció minősítetlen szignatúrájára,
  - és D_p*-megszorított intrahierarchia-elérhető*, ahol
    - D_p a p definíciós hatóköre.

Definíció. Azt mondjuk, hogy a d elődefiníció D*-megszorított intrahierarchia-elérhető*, ha

- D nem üres, és
  - vagy d láthatósága publikus,
  - vagy létezik olyan d' definíció, amely
    - hivatkozik d minősítetlen szignatúrájára,
    - és D_i*-megszorított intrahierarchia-elérhető*,
  - ahol D_i a D és D_d metszete, ahol
    - D_d a d definíciós hatóköre.

4.5. Új refaktorálási sémák meghatározása

Az előző részekben ismertetettek tükrében minden eszköz rendelkezésünkre áll refaktorálási sémák definiálásához. Egy-egy séma megfogalmazásához meg kell adnunk a séma nevét, lehetséges záradékait – az ezekben példányosításkor megadott átírásokat vezérlő logikával együtt –, előfeltételeit, valamint az átírásparaméterek által teljesítendő szerződéseket. Az előfeltételeket és szerződéseket úgy határozzuk meg, hogy azok az ekvivalenciadefiníció valamely szintjének feleljenek meg. A következőkben 4 sémát vázolunk fel, amelyek Schafer [11], Opdyke [9] és Horpácsi [7][6] példarefaktorálásainak általánosításával jöttek létre.

Lokális séma

A lokális séma olyan egyszerű, blokkra lokális refaktorálások definiálásához használható, amelyek nem igényelnek speciális előfeltételeket vagy vezérlési logikát. Alakja:

```
1  local refactoring <név>
2      <kiinduló minta>
3      ----------------
4      <csereminta>
5  target
6      <opcionális kiinduló minta>
7  when
8      isInsideBlock(target)
9      and <opcionális előfeltételek>
```

Látható, hogy a lokális refaktorálások valójában azt a célt szolgálják, hogy klasszikus feltételes termátírásokat is be tudjuk illeszteni a séma alapú rendszerbe. A séma szerződése ennek megfelelően a kiinduló és a csereminta előfeltételek mentén fennálló ekvivalenciáját írja elő. A 8. sorban olvasható, beépített feltétel azt követeli meg, hogy az alkalmazás célja blokkon belüli entitás legyen.

Megjegyzés. Segíthet a különböző ekvivalenciaszintek megértésében az, ha a bemutatott sémákban megpróbálunk egy klasszikusnak számító refaktorálást, az átnevezést definiálni. Látható, hogy ezt a lokális sémában nem tehetjük meg, hiszen eredeti és átnevezett változót a lokális ekvivalenciával nem lehet egyezőnek tekinteni.

Blokk séma

A blokk séma a lokális séma kibővítésének tekinthető, teljes blokkok refaktorálására használható. Itt már sémába rejtett vezérlési logika is megjelenik. A séma alakja:

```
1  block refactoring <név>
2      <kiinduló minta>
3      ----------------
4      <csereminta>
5  target
6      <opcionális kiinduló minta>
7  shadowed references
8      reference
9      -------------
10     qualifiedName
11 when
12     qualifiedName = reference.qualifiedName()
13     and <opcionális előfeltételek>
```

A kiinduló és cseremintában speciális mintaillesztést teszünk lehetővé. Ha a felhasználó a `target` záradékban megadja a célra illesztendő mintát, az átírási mintapárban pedig úgy tüntetni fel a `target` elemet, hogy a minta szélén tetszőleges sok

elemre illeszkedő mintát (pl. `S..`) helyez el, akkor ezek a multiminták a célt befoglaló blokk adott részéhez fognak kötni. Például, ha a `target` minta `T x`, vagyis egy deklaráció, a kiinduló minta pedig `S.. ; target ; S'..`, akkor `S..` és `S'..` rendre a célt tartalmazó blokk teljes megelőző és következő részére fog illeszkedni.

Blokkon belüli utasításmozgatáskor gyakran sérülhet a névkötés, például mikor egy adattagelérés elé mozgatunk egy adattaggal egyező nevet kötő deklarációt. Ezek az esetek a formális szemantika segítségével azonosíthatók, megfelelő névminősítéssel pedig feloldhatók. A sémában ezt a logikát a 7-12. sorok írják le. Itt észrevehető az alapul vett nyelvre jellemző implicit változókötés is: mivel a 12. sorban a feltétel kiértékelésekor a `qualifiedName` változó még nincs kötve, a sorban összehasonlítás helyett értékkötés történik.

A séma szerződése itt a kiinduló és csereminta blokk-ekvivalenciáját várja el, a megadott előfeltételek és az elfedett változók megfelelő minősítését figyelembe véve. Amennyiben a megadott minták nem fedik le a teljes blokkot, a fentebb részletezett, multimintás technikával implicit blokkmintává alakíthatók.

Megjegyzés. Az előző sémánál felvetett átnevezés refaktorálás itt már működne – természetesen csak akkor, ha bevezetnénk olyan záradékot is, amely az átnevezés miatt sérülő hivatkozásokat kompenzálja automatikusan –, de csak a blokk lokális változói esetén. Ennek hátterében az áll, hogy blokk-ekvivalencia vizsgálatakor a lokális változók egyezésének ellenőrzésétől eltekintünk.

Lambda séma

Schafer megmutatta [11], hogy a lambda függvények jól használhatók kódszegmensek mozgatására, hiszen segítségükkel az egyébként egy lépésben adat- és vezérlésfolyamot is érintő transzformációk – kihasználva, hogy a lambdák adatként és függvényként is viselkednek – felbonthatók adat és vezérlés egymástól független módosítására. Az 1.4-es verziójú, megszorított Java magnyelvünkben ugyan nincsenek lambda függvények, de névtelen osztályokkal szimulálhatók. Ezek kezelésére vezetjük be a lambda sémát, melyet kétféle változatban adunk meg.

Az első változat lambdák bevezetésére szolgál, ahol – a névtelen osztályokra alapuló megoldás miatt – egy új interfészdefiníció hozzáadásáról kell gondoskodni. A változat a következő alakú:

```
1 lambda refactoring <név>
2     <kiinduló minta>
3     -------------------
4     <csere lambdaminta>
5 target
6     <opcionális kiinduló minta>
```

```
7  top level definition
8      <cseremintának megfelelő interfészdefiníció>
9  when
10     isFresh(<csereminta interfészének neve>)
11     and <csere lambdaminta törzse>.variableWrites().filter(write :
12         not write.variable().isDeclaredIn(<csere lambdaminta
               törzse>)).isEmpty()
13     and <opcionális előfeltételek>
```

Ez a változat automatizáltan az alapján különböztethető meg a másiktól (ld. alább), hogy a csereminta lambda, de a kiinduló minta nem. Ilyenkor a cseremintának megfelelő visszatérési értéktípussal, névvel és paraméterekkel kell egy új interfészfüggvényt definiálni, gondoskodva arról, hogy az interfész neve biztosan egyedi legyen (ld. 10. sor). A cseremintában az interfész neve tetszőleges lehet, a rendszer a név egyedisége alapján dönti el, hogy helybenhagyja a felhasználó szándékát, vagy új nevet generál. A 11-12. sorban azt fogalmazzuk meg, hogy a lambdába emelendő kódszegmens nem írhat olyan változót, amely nem benne lett deklarálva, mivel ez fordítási hibát eredményez Java-ban.

Az első változat szerződése a kiinduló és csereminta lokális ekvivalenciáját várja el, gondoskodva az előfeltételek figyelembe vételéről, illetve a módosított konfiguráció definíciós cellájának az új interfésszel történő kiegészítéséről.

A második változat lambdák módosítására használható, itt a már meglévő interfész átírási szabálynak megfelelő átalakítására kell figyelni. A séma alakja:

```
1  lambda refactoring <név>
2      <kiinduló lambdaminta>
3      ----------------------
4      <csere lambdaminta>
5  top level definition
6      <kiinduló mintának megfelelő interfészdefiníció>
7      ------------------------------------------------
8      <cseremintának megfelelő interfészdefiníció>
9  when
10     F.references().size() = 1
11     and F.references().contains(target)
12     and <opcionális előfeltételek>
```

Ebben a változatban a kiinduló és a csereminta is lambdát tartalmaz. A séma feladata itt az, hogy két minta közötti eltéréseket lekövetve azokat a kapcsolódó interfészdefiníciókra is átvezesse. Előfeltételként kikötjük, hogy a módosítandó interfészre pontosan csak a refaktorálás célja hivatkozzon, hiszen lokális refaktorálás lévén nem szeretnénk kiterjedt kompenzációs transzformációkat végrehajtani. Az

opcionális célzáradékot itt elhagyjuk, mivel rögzített alakja van a kiinduló mintának.

A második változat szerződése is a kiinduló és csereminta lokális ekvivalenciáját várja el, szintén figyelembe veszi az előfeltételeket, a konfigurációs cellákban pedig a megfelelő interfészdefiníciókat vizsgálja.

Osztály séma

Az osztály sémával az osztályok tagjaira vonatkozó refaktorálásokat tesszük leírhatóvá. Ennek három változatát mutatom be, melyek rendre metódusdefiníció hozzáadására, továbbá metódusdefiníció és mező ősosztályba mozgatására lesznek használhatók. Az előzőekhez viszonyítva ezen sémák előfeltételei jóval bonyolultabbak lesznek, leginkább amiatt, mert itt a hierarchia-megfelelést használjuk mögöttes jelentésmegőrzési definícióként. Tekintsük először a metódus bevezetésére szolgáló sémát:

```
1  class refactoring <név>
2    <kiinduló minta>
3    ----------------
4    name(params..)
5  target
6    <opcionális kiinduló minta>
7  definition in class
8    visibility T name(params..) body
9  when
10   isUniqueMethodIn(name, params.., target.enclosingClass())
11   and (isOverrideIn(name, params.., target.enclosingClass()) ->
12     not isLessVisible(visibility, overriddenMethodFrom(name, params..,
         target.enclosingClass()).visibility())
13     and isSubtypeOf(T, overriddenMethodFrom(name, params..,
         target.enclosingClass()).type())
14     and overriddenMethodFrom(name, params..,
         target.enclosingClass()).visibility() /= public
15     and publicInterHierarchyReferences(name, params.., union(
16         target.enclosingClass().superclassHierarchy(),
17         definitionScopeFrom(visibility, name, params..,
               target.enclosingClass())
18     )).isEmpty()
19     and boundedPublicIntraHierarchyReferences(name, params..,
20         definitionScopeFrom(visibility, name, params..,
               target.enclosingClass())
21     ).isEmpty())
```

```
22    and for all override in overridesOf(name, params..,
          target.enclosingClass()) :
23      not isLessVisible(override.visibility(), visibility)
24      and isSubtypeOf(override.type(), T)
25    and <opcionális előfeltételek>
```

Itt a definíciós záradékban szerepel az új metódus tényleges bevezetése, ahol a hiányzó információkat a kiinduló mintából nyerhetjük ki. Kompenzációs transzformációként a kiinduló mintára illeszkedő programrészletet, amelyből a metódustörzset emeltük ki, az újonnan létrejött definíció megfelelő paraméterekkel történő meghívására cseréljük.

Az előfeltételek alapvetően úgy lettek megkonstruálva, hogy kiszűrjék azokat az eseteket, mikor a definíciós záradékban látható alakú új metódusdefiníció megsértené a hierarchiamegfelelés valamely kikötését, de gondoskodni kellett olyan elemi tulajdonságokról is, hogy a definíció bevezetése ne idézzen elő fordítási hibát.

Az előfeltételekben gyakran szerepel a `name`, `params..`, `target.enclosingClass()` hármas. Ez a 4.4. részben bevezetett elődefiníció szemantikus függvényekben és predikátumokban használható reprezentációja. Például, az első feltételt (10. sor) úgy kell értelmezni, hogy a *(name, params..)* szignatúrájú elődefiníció a refaktorálás célját befoglaló osztályban egyedi metódus. A 11. sorban kezdődő implikáció egyrészt az elődefiníció elérhetőségének meghatározásából köszön vissza, másrészt felvezet további, fordíthatósághoz kapcsolódó megkötéseket. A 12. és 13. sorban például azt írjuk le, hogy az elődefiníció által felüldefiniálásra kerülő metódusénál nem lehet kevésbé nyilvános a láthatóság, illetve a visszatérési értékek típusainak is kompatibilisnek kell lenniük. A 14. sorban szintén az elődefiníció elérhetőségének feltétele szerepel, miszerint eleve megsértjük a megfelelést, ha publikus metódust definiálunk felül. A 15-21. sorokban rendre az elődefiníció inter-, illetve intrahierarchia-elérhetőségét tiltó kikötéseket teszünk, felhasználva ezekben a 4.4. részben meghatározott definíciós hatókör fogalmat. A 22-24. sorokban a korábban már látott fordíthatósági feltételek ismétlődnek, csak ezúttal nem a bevezetendő elődefinícióra, hanem a bevezetésével kialakuló, felüldefiniálási kapcsolatokra fogalmazzuk meg őket. A feltételek leírására használt `for all ... in ... : ...` konstrukció egy gyűjtemény összes elemére vezet be egy kvantált megkötést.

A séma bonyolult előfeltételei miatt a szerződés egészen egyszerűvé válik. A példányosító átírásnak mindössze abban kell megfelelnie, hogy lokálisan ekvivalens legyen az új metódus megfelelően felparaméterezett változatát meghívó cseremintával, ahol utóbbi konfigurációjában az új metódusdefiníció is elhelyezésre kerül.

A hátralévő két változatban a felhasználói mozgástér már olyan szűk, hogy ezeket sémák helyett inkább értelmezhetjük előre verifikált, konkrét refaktorálásokként.

Következőként tekintsük a metódusdefiníció ősosztályba mozgatására használható sémaváltozatot:

```
1  class refactoring <név>
2    visibility T name(params..) body
3    --------------------------------
4
5  definition in super
6    target
7  when
8    target.enclosingClass().hasSuperclass()
9    and (visibility = private -> target.references().isEmpty())
10   and body.accessedFieldsOfEnclosingClass().isEmpty()
11   and body.accessedMethodsOfEnclosingClass().isEmpty()
12   and isUniqueMethodIn(name, params..,
         target.enclosingClass().superclass())
13   and (isOverrideIn(name, params..,
         target.enclosingClass().superclass()) ->
14     not isLessVisible(visibility, overriddenMethodFrom(name, params..,
           target.enclosingClass().superclass()).visibility())
15     and isSubtypeOf(T, overriddenMethodFrom(name, params..,
           target.enclosingClass().superclass()).type())
16     and overriddenMethodFrom(name, params..,
           target.enclosingClass().superclass()).visibility() /= public
17     and publicInterHierarchyReferences(name, params.., union(
18         target.enclosingClass().superclass().superclassHierarchy(),
19         definitionScopeFrom(visibility, name, params..,
               target.enclosingClass().superclass())
20     )).isEmpty()
21     and boundedPublicIntraHierarchyReferences(name, params..,
22         definitionScopeFrom(visibility, name, params..,
               target.enclosingClass().superclass())
23     ).isEmpty())
24   and for all override in overridesOf(name, params..,
         target.enclosingClass().superclass()) :
25     not isLessVisible(override.visibility(), visibility)
26     and isSubtypeOf(override.type(), T)
```

Az átírási és definíciós záradékot együttesen tekintve láthatjuk, hogy a célmetódust az eredeti pozíciójáról az ősosztályába mozgatjuk. A mozgatás valójában tekinthető speciális bevezetésnek is, ez a kapcsolat pedig tetten érhető a két metódusváltoztató séma előfeltételeinek hasonlóságában is. Figyeljük meg, hogy itt a 12-26. sorokban csak annyi a különbség az előző sémához képest, hogy az elődefiní-

cióhoz köthető információkat nem a refaktorálási cél befoglaló osztályában, hanem annak ősében vizsgáljuk. Vegyük észre, hogy az inter- és intrahierarchia-elérhetőség ellenőrzésénél a felmozgatandó elődefiníció definíciós hatókörének azon részeit kell csak vizsgálnunk, amelyek túlmutatnak a kiinduló definíció hatókörénél. Ugyanakkor ezt nem kell explicit feltüntetnünk, hiszen az elődefiníció ősosztályból számolt definíciós hatókörét egyébként is lezárja az ekkor még a kiinduló osztályban lévő, elődefiníciót felüldefiniáló, eredeti metódus.

A felmozgatás sajátjaként a 8-11. sorokban lévő előfeltételek szolgálnak. Ezekkel rendre azt írjuk le, hogy a célnak legyen ősosztálya, ahova mozgathatunk; hogy privát metódust csak akkor mozgassunk fel, ha nincs rá hivatkozás; illetve, hogy a célt csak akkor mozgassuk fel, ha nem hivatkozik a kiinduló osztály mezőjére vagy metódusára. Joggal gondolhatnánk, hogy ha az utóbbi feltételek nem teljesülnének, egyszerűen felmozgathatnánk a kiinduló osztály hivatkozott elemeit is. Valóban, viszont ez ugyanaz a tevékenység lenne, amit az éppen tárgyalt sémák végeznek. A redundancia elkerülése érdekében a bonyolultabb felmozgatásokhoz szükséges logikát ezen sémáktól függetlenül fogjuk definiálni.

Példányosításra használható átírási szabály hiányában itt nem releváns szerződést vizsgálni, részben emiatt is neveztük ezt a sémát példányokhoz közelebb állónak. Az előfeltételek megfogalmazásakor természetesen a hierarchia-megfelelést tartottuk szem előtt.

Utolsóként vizsgáljuk meg a mezőt ősosztályba mozgató sémát, amely, hasonlóan az előzőhöz, inkább egy konkrét refaktorálást határoz meg:

```
1  class refactoring <név>
2    visibility T name;
3    ------------------
4
5  definition in super
6    target
7  when
8    target.enclosingClass().hasSuperclass()
9    and (visibility = private -> target.references().isEmpty())
10   and isUniqueFieldIn(name, target.enclosingClass().superclass())
11   and publicReferences(name, minus(
12       visibilityFrom(visibility, name,
             target.enclosingClass().superclass()),
13       visibilityFrom(visibility, name, target.enclosingClass())
14     )).isEmpty()
15   and for all reference
16     in references(name, visibilityFrom(visibility, name,
         target.enclosingClass().superclass())) :
```

```
17 ||       isSubtypeOf(T, reference.field().type())
```

Az előző sémával analóg technikával írjuk le a mező ősosztályba mozgatását is. Java-ban a mezők között felüldefiniálás és dinamikus kötés ugyan nincs, de az itt alapul vett hierarchia-megfelelés megsérülhet, ha a felmozgatás miatt kialakuló elfedések megváltoztatják valamely elérhető definíció viselkedését. A definíció 11. sorában szereplő `publicReferences` függvény adja meg azokat a hivatkozásokat, amelyek statikusan a második argumentumában szereplő osztályhalmaz valamely elemével minősített, az első argumentumban megadott nevű mezőre mutatnak egy elérhető metódusdefinícióból. Ha a felmozgatott mező ősosztályból számított láthatóságkülönbségét (ld. 12-13. sor) vesszük alapul, pontosan azokat a lehetséges statikus minősítőket kapjuk meg, melyeken keresztül a felmozgatandó mező nevére hivatkozó, meglévő definíciók jelentése a felmozgatás utáni kötésváltozás miatt módosulhat, megsértve ezzel a hierarchia-megfelelést.

A 8-10., illetve 15-17. sorok szintén fordítási hibák elkerülésére szolgálnak. Az első három sorban szereplő feltételek rendre az ősosztály létezését, privát láthatóságú mező esetén a hivatkozások hiányát, valamint a mező felmozgatás utáni egyediségét fogalmazzák meg. Az utolsó három sor azt írja le, hogy a felmozgatás utáni legális (nem elérhető metódusokon belüli) kötésváltozások típushelyesek lesznek, vagyis azt, hogy az előző hivatkozást elfedő, felmozgatott mező típusa kompatibilis az elfedett mező típusával.

Megjegyzés. Visszatérve az átnevezés refaktorálás példájára, megfigyelhetjük, hogy hierarchia-megfelelést használó sémával már átnevezhetünk rejtett osztályelemeket, de nyilvánosakat, vagy akár osztályokat továbbra sem. Ugyanakkor van lehetőség az utóbbira is, de csak főprogram-ekvivalenciát alapul vevő sémát példányosítva.

5. Esettanulmány

A 4. részben bemutattam a vizsgált módszer adaptálásához szükséges lépéseket, a folyamat ismertetésével párhuzamosan konstruálva meg az átformált rendszer egy prototípusát. Most azt mutatom meg, hogy az így létrehozott eszközkészlet valóban alkalmas egy összetett refaktorálás megfogalmazására. Ehhez először azonosítok egy kellően komplex refaktorálási szabályt, ezt dekomponálom mikrolépésekre, majd az így kapott lépéseket, illetve ezek összeépítési logikáját definiálom az adaptálás során létrejött nyelvben. Végül egy konkrét programon történő végrehajtás segítségével illusztrálom a megfogalmazott szabályok működését.

5.1. Informális specifikáció

Az esettanulmányhoz olyan refaktorálási szabályt kell választani, amely úgy dekomponálható, hogy az előálló mikrolépések a 4.5. részben megadott sémák minél sokrétűbb felhasználására adnak lehetőséget. Az alapul vett munkák számos alternatívát kínálnak, kézenfekvően adódik például a Horpácsi [6] által az alapnyelvben definiált *függvényáltalánosítás* vagy a Schafer munkájában [11] tanulmányozható *metóduskiemelés*. Ugyan mindkettő szabály jól alkalmazható a dekompozíció bemutatására, viszont egyik sem teszi szükségessé az objektumorientált paradigmához köthető eredmények felhasználását. Ezzel szemben Opdyke [9] tisztán objektumorientált refaktorálásokat vizsgál, ám ezek túl nyitottak és bonyolultak a céljainkhoz.

A probléma áthidalására konstruáljunk meg egy saját refaktorálást úgy, hogy az ötvözze a procedurális szinten jól illusztrálható mikrolépéseket, illetve az adaptálás lényegét adó objektumorientált koncepciókat. Előbbihez használjuk fel a Schafer által részletesen dekomponált metóduskiemelést, amit gondoljunk tovább az Opdyke disszertációjában olvasható metódusfelmozgatással. A kettő kompozícióját nevezzük *szegmensfelmozgatásnak*.

A refaktorálás egy metódusban kijelölt, összefüggő utasítássorozatnak (szegmens) a metódus ősosztályába egy új függvényként történő átmozgatását írja le (ld. 1. ábra). A létrehozandó függvény neve és láthatósága a refaktorálás paramétere, a többi információt automatikusan kell kikövetkeztetnünk.

5.2. Dekompozíció

Első dekompozíciós lépésként természetesen adódik az a fő illesztési pont, amely mentén a konstrukció során összeépítettük a Schafer-féle procedurális és az Opdyke-féle objektumorientált refaktorálási szabályokat. Ez alapján tehát a kódszegmens kiemelésének és a metódusdefiníció felmozgatásának szabályait kell tovább bontanunk.

Szegmens felmozgatása:

1. kódszegmens kiemelése,
2. metódusdefiníció felmozgatása.

A kódszegmens kiemelésének mikrolépésekre bontási folyamatát Schafer részletesen ismertette munkájában [11]. A kiemelésre megjelölt szegmenstől három fő lépésben jut el a kiemelt metódusdefinícióig, a fő lépések közötti átalakításokkal pedig a részeredményeket finomítja. Módszere az ekvivalencia karakterizációjára használt névkötéseket, illetve vezérlés- és adatfolyamokat potenciálisan megváltoztató transzformációkat választja szét.

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int x = 1;
10         a = x;
11         g();
12         int y;
13         a = y;
14     }
15
16     void g() {
17         a = b = 0;
18     }
19 }
```

```
1  class A {
2      int a;
3      int b;
4
5      void g() {
6          a = b = 0;
7      }
8
9      void h(int x) {
10         a = x;
11         g();
12     }
13 }
14
15 class B extends A {
16     void f() {
17         int y;
18         int x = 1;
19         h(x);
20         a = y;
21     }
22 }
```

1. ábra. Programkód a szegmensfelmozgatás előtt és után. A refaktorálás a bal oldalon kékkel jelölt szegmensre lett alkalmazva a *csomag* láthatósági módosítóval, illetve a `h` függvénynévvel. A refaktorálás által megváltoztatott kódrészleteket a jobb oldalon kékkel emeltük ki.

Kódszegmens kiemelése:

1. szegmens új blokkba mozgatása,
2. blokk lambdába ágyazása,
3. lambda finomítása és kiemelése.

A névkötések megőrzéséhez először a kijelölt szegmenst utasításonként, fordított sorrendben mozgatja át egy újonnan létrehozott, követlenül az eredeti szegmens után beszúrt blokkba. Az utasítások egyesével történő áthelyezése lehetővé teszi a változódeklarációk különválasztását, hiszen ezek felelőtlen mozgatásával kötéseket módosíthatnánk. Mikor a soron következő, blokkba mozgatandó utasítás egy változódeklaráció, akkor attól függően, hogy a deklarációhoz tartozik-e inicializációs kifejezés, rendre vagy kettébontja ezt egy egyszerű deklarációra és egy értékadásra – egy újabb mikrorefaktorálás alkalmazásával –, vagy a deklarációt a feldolgozás alatt álló szegmens legelejére mozgatja. Vegyük észre, hogy a felbontott deklarációból származó értékadás már következmények nélkül átmozgatható a mélyebb blokkba. A deklaráció felmozgatása ugyanakkor bevezethet elfedést, de ezt a refaktoráláshoz használt keretrendszerük segítségével oldja fel.

Szegmens új blokkba mozgatása:

1. új blokk beszúrása,
2. fordított sorrendben egy-egy utasítás blokkba mozgatása, külön kezelve a deklarációkat.

Miután a teljes kijelölt szegmens egy külön blokkba került, a névkötések veszélyeztetése nélkül lehet lambdává transzformálni. Ekkor viszont a vezérlésfolyam megmaradását kell ellenőrizni, hiszen a forrásszegmensben potenciálisan előforduló ugró utasítások szemantikája megváltozhat, ha egy lambdába ágyazzuk őket. Ezen a ponton, amennyiben az eredeti szegmensben értéket visszaadó `return` utasítás is szerepelt, már meghatározható a végeredményként előálló függvény visszatérési értékének típusa is.

Blokk lambdába ágyazása:

1. ugró utasítások vizsgálata,
2. értékadások vizsgálata.

Végül az adatfolyamot módosító transzformációk gondoskodnak arról, hogy a lambdában eddig csak implicit módon szereplő adatfüggőségek bemeneti paramétereken és visszatérési értéken keresztül explicit formában jelenjenek meg. Ezen a ponton sajnos nem tudjuk követni Schafer módszerét, hiszen az általa használt, például lambdán belülről lambdán kívüli változónak történő értékadásokat lehetővé tévő, speciális módú változók számára is csak egyedi nyelvi kiterjesztések formájában érhetők el, mi pedig (a rendelkezésre álló szemantika és a verifikáció miatt) szabványos Java-val dolgozunk. Ennek áthidalására csak olyan lambdákat engedünk létrehozni, melyek törzsében külső változó nem kap értéket.

Lambda finomítása és kiemelése:

1. adatfüggőségek explicitté tétele,
2. lambda kiemelése metódussá.

Miután a bevezetett lambda finomításával gondoskodtunk az adatfolyam megmaradásáról is, a lambdából kinyerhető információk alapján már definiálni tudjuk a megfelelő metódust.

A dekompozíció következő lépéseként a metódusdefiníció felmozgatását bontjuk részekre, amihez Opdyke disszertációját [9] vesszük alapul. A szegmenskiemeléshez képest itt jóval korlátozottabbak a dekompozíciós lehetőségek, a tényleges felmozgatáson túl részlépésként egyedül a függőségek viszontmozgatását tudjuk feltüntetni.

Metódusdefiníció felmozgatása:

1. hivatkozott adattagok felmozgatása,
2. hivatkozott metódusdefiníció felmozgatása[4],
3. metódusdefiníció felmozgatása.

5.3. Formális definíció

A fenti dekompozíciót felhasználva, a 4.5. részben ismertetett sémák konkrét átírási szabályokkal történő példányosításával formálisan is definiáljuk a vizsgált refaktorálást. Ehhez a legelemibb, lokális lépésektől indulunk, végül pedig a mikrolépéseket vezérlési logikával társító, összetett refaktorálásokig jutunk el.

5.3.1. Lokális refaktorálások

Lokális sémapéldányként egyedül az utasítás után új blokkot beszúró refaktorálást definiáljuk.

```
1 local refactoring introduceEmptyBlockAfter()
2     s
3     ------
4     s ; {}
```

Az eredeti dekompozíció alapján itt lehetne megadni az inicializációs kifejezéssel rendelkező értékadásokat felbontó példányt is, de mivel ezek az éppen a szegmensen kívülről elérhető deklarációk kezelésére szolgálnak, mi pedig lambda törzsébe a Java szemantikája miatt egyébként sem fogunk tudni külső változóra vonatkozó értékadásokat hagyni, eltekintünk a definiálásától.

5.3.2. Blokk refaktorálások

Első blokkpéldányként egy utasítást az azt közvetlenül követő blokkba mozgató refaktorálást adjuk meg.

```
1 block refactoring moveIntoNextBlock()
2     target ; { S.. } ; S'..
3     ----------------------
4     { target ; S.. } ; S'..
5 when
6     isSingle(target)
7     and (isVariableDeclaration(target) ->
8         not isReferencedIn(target.variable(), S'..))
```

[4]Vegyük észre a rekurziót.

Az átírási mintán látható, hogyan használjuk fel a mintaillesztést a transzformáció tömör és intuitív megadására. A refaktorálás célját jelölő `target` pszeudováltozót itt a kapcsolódó záradék nélkül használjuk, aminek hatására tetszőleges utasítás illeszkedni fog rá. A speciális `S'..` metaváltozóval a refaktorálási célt befoglaló blokk átmozgatási célt követő utasításaira illesztünk mintát, amit aztán arra használunk fel, hogy a definíció 7-8. sorában megfogalmazzuk azt, hogy a befoglaló blokkban később hivatkozott deklarációkat nem vihetjük mélyebb szintre. Az első előfeltétel gondoskodik róla, hogy valóban csak egyesével lehessen az utasításokat a célblokkba mozgatni.

A következő blokkpéldány a deklarációt a befoglaló blokk elejére mozgató refaktorálás.

```
1 block refactoring moveToTop()
2     S.. ; target
3     ------------
4     target ; S..
5 target
6     T x
```

Ezen a példán is látható a blokk séma által biztosított speciális, a teljes megelőző blokktartalomra illeszkedő multiminták (itt `S..`) haszna, hiszen jelen esetben is ennek segítségével tudtuk kifejezni a blokk elejére mozgatást. A `target` záradék itt a mintaduplikációk elkerülésére szolgál. A háttérben a névkötések megtartásáért kihasználjuk a sémába rejtett `shadowed references` záradékot is.

5.3.3. Lambda refaktorálások

Az első két lambdapéldány egy, már külön blokkba mozgatott szegmens kiemelésére szolgál. A kettő között a különbség a létrejövő lambda visszatérési értékének típusa, amit a kiemelendő blokk alapján határozunk meg.

```
1 lambda refactoring wrapInVoidLambda()
2     { S.. }
3     ---------------------------------------------
4     new F() { public void apply() { S.. } }.apply()
5 when
6     not containsValueReturn(S..)
7     and (containsVoidReturn(S..) ->
8         for all path in target.paths() :
9             path.controlSuccessor() =
                  target.enclosingMethod().exitNode())
```

Az első szabály a `void` visszatérésű lambdák beszúrására használható. Az átírási záradékában látszik, hogy az alapul vett, 1.4-es verziójú Java-ban milyen sok szin-

taktikus zaj társul egy lambda szimulálásához. A séma definíciójának megfelelően az itt, valójában csak minta formájában hivatkozott `F` interfész nevét a rendszer garantáltan egyedire fogja megválasztani. A 6. sorban szereplő előfeltétel azt fejezi ki, hogy a blokkban nem lehet értéket visszaadó `return` kifejezés, a 7-9. sor pedig azt írja le, hogy ha érték nélküli `return` található a blokkban, akkor annak összes lehetséges végrehajtási útjának a befoglaló metódus szimbolikus végrehajtási végpontjába kell vezetnie. Intuitívan, egy `return`-t tartalmazó szegmenst csak akkor lehet lambdába kiemelni, ha a lambda lefutása után már egyébként is kilépnénk a befoglaló metódusból.

```
1 lambda refactoring wrapInValueLambda()
2     { S.. }
3     ------------------------------------------------
4     return new F() { public T apply() { S.. } }.apply()
5 when
6     for all path in target.paths() : containsValueReturn(path)
7     and T = leastCommonType(target.paths().map(path :
          path.last().type()))
```

Az értékkel visszatérő esetben az átírási minta egyrészt a lambda típusában, másrészt a lambda alkalmazása elé helyezett `return` kulcsszóban különbözik, hiszen az egyébként értéket visszaadó `return` már magából a befoglaló metódusból ugrott ki, így ezt egy lambdába zárva az onnan történő kilépést a befoglaló felé is propagálni kell. A 6. sorban azt követeljük meg, hogy a kiemelendő blokk minden végrehajtási útján legyen érték `return` – különben fordítási hibát is kapnánk –, a 7. sorban pedig nem is feltételvizsgálat, hanem a `T` metaváltozónak történő értékadás történik az alapján, hogy az `S..`-ben szereplő `return`-öknek mi a legkisebb közös statikus típusa.

Az utolsó lambdapéldány a már bevezetett lambdák implicit bemeneti adatfüggőségeit teszi explicitté azok paraméterekként történő hozzáadásával.

```
1 lambda refactoring extractInVariables()
2     new F() { public T name() body }.name()
3     ------------------------------------------------------
4     new F() { public T name(inVars..) body }.name(inVars..)
5 when
6     inVars.. = body.variableReads().filter(read :
7             isBefore(read.variable().declaration(), target))
8         .map(read : read.variable()).reduce()
```

Itt szintén az előfeltételek közé ágyazva határozzuk meg a paraméterré teendő adatok. Ezek pontosan azon olvasott változók lesznek, amelyek a lambdán kívül lettek deklarálva. A 6-8. sorokban lévő formula csak technikai okok miatt ilyen hosszú, itt változóolvasásokból kell paraméterlistákat konstruálnunk. Vegyük észre

a háttérben zajló, implicit típuskonverziókat. Az átírási záradékban kihasználjuk azt is, hogy a paraméterlistát változóolvasásokból állítottuk elő, így azt egyaránt használhatjuk formális és konkrét szerepben is.

5.3.4. Osztály refaktorálások

Az osztály sémákba foglalt, bonyolult előfeltételeknek köszönhetően a tényleges példányok már egyszerűen megfogalmazhatók. Az első példány lambda metódussá emelésére használható.

```
class refactoring extract(visibility, newName)
    new F() { public T name(params..) body }.name(args..)
    ----------------------------------------------------
    newName(args..)
definition in class
    visibility T newName(params..) body
when
    isSubsetOf(body.dataAccesses().map(access : access.target()),
        union(params.., target.enclosingClass().fields(),
            body.localVariables())
```

Vegyük észre, hogy a refaktorálás szignatúrájába két paramétert is felvettünk, hiszen azt a felhasználóra szeretnénk bízni, hogy egy metódust milyen láthatósággal és névvel szeretne bevezetni. A sémában foglaltakon túl további előfeltételként kötjük ki, hogy a kiemelendő lambda által hivatkozott mezők és változók a lambda paraméterei, a befoglaló osztályban elérhető mezők, és a lambda törzsének lokális változói közül kerülnek ki, vagyis nincs közöttük olyan, amely a kiemelés utáni környezetben már nem lenne a törzsből elérhető.

```
class refactoring lift()
    visibility T name(params..) body
    --------------------------------

definition in super
    target
```

A hátralévő két osztály-példány rendre metódus, illetve mező ősosztályba mozgatására használható. A nekik megfelelő sémától csak az elnevezésükben térnek el.

```
class refactoring lift()
    visibility T name;
    ------------------

definition in super
    target
```

5.3.5. Összetett refaktorálások

Az előzőekben formalizált sémapéldányokból összetett refaktorálásokban alkotjuk meg a kezdetben kijelölt refaktorálási szabályt. Az elsőben a szegmenskiemeléshez szükséges vezérlési logikát definiáljuk.

```
1  composite refactoring extract(visibility, name)
2  do
3      introduceEmptyBlockAfter()
4      newBlock = target.nextStatement()
5      for all stmt in reverse(target) :
6          stmt.moveIntoNextBlock()
7              or stmt.moveToTop()
8      newBlock.wrapInVoidLambda()
9          or newBlock.wrapInValueLambda()
10     wrappedLambda = newBlock.enclosingLambda()
11     wrappedLambda.extractInVariables()
12     wrappedLambda.extract(visibility, name)
13 when
14     isSegment(target)
```

A definíción rögtön látható, hogy az eddigi deklaratív stílus helyett a vezérlést imperatív formában fogalmazzuk meg. A leírás törzsében alkalmazhatunk refaktorálási szabályokat, bevezethetünk új változókat, vizsgálhatjuk szemantikus függvények értékét. Itt az előfeltételben meggyőződünk róla, hogy a refaktorálás célja tényleg egy szegmens. Az első sémapéldány előfeltételét tekintve ez ugyan redundáns, de a könnyebb olvashatóság érdekében ettül függetlenül is feltüntetjük.

A fenti definícióban először bevezetjük a cél után az összemozgatáshoz használt üres blokkot (3. sor), majd erre el is tárolunk egy referenciát (4. sor), hogy később is tudjunk hivatkozni rá. Ezután a kiinduló szegmens utasításain visszafelé végighaladva (5. sor) egyesével megpróbáljuk az aktuális elemet a célblokkba mozgatni (6. sor). Ha ez nem sikerül, az **or** vezérlési szerkezet szemantikájának megfelelően a következő, feltételesen kapcsolt tevékenységre lépünk. Ilyenkor itt a blokk elejére mozgatás fog végrehajtódni (7. sor), ami illeszkedik a dekompozíció alapján szükséges lépéshez, hiszen itt a következő blokkba mozgatás pontosan akkor fog abortálni, ha blokk elejére mozgatandó deklarációra próbáltuk alkalmazni.

A blokkba mozgatás után megkíséreljük a lambda kiemelését, aminek két változatát szintén az **or** konstrukcióval fűzzük össze (8-9. sor). Itt valójában egy primitív elágazást szimulálunk a refaktorálási szabályok mintaillesztése és előfeltételei által meghatározott feltételekkel. Bevezethetnénk ugyan dedikált elágazás vezérlési szerkezetet is, de mivel a sémapéldányok már eleve tartalmaznak előfeltételeket, tisztább megoldást kapunk, ha azokat használjuk ki.

Végül a már kiemelt lambdát finomítjuk tovább, melyet, az éppen becsomagolt blokkon keresztül elérve, egy másik referenciában tárolunk el (10. sor). Ezután explicitté tesszük az adatfüggőségeit (11. sor), majd végül az összetett refaktorálás paramétereit továbbadva metódussá is emeljük (12. sor).

A következő összetett refaktorálás a másik nagy kompozíciós lépést, a metódusdefiníció felmozgatását írja le.

```
1  composite refactoring cascadedLift()
2  do
3      for all field in target.body().accessedFieldsOfEnclosingClass() :
4          field.lift()
5      for all method in target.body().accessedMethodsOfEnclosingClass() :
6          if method.enclosingClass() = target.enclosingClass() :
7              method.cascadedLift()
8      target.lift()
9  when
10     isMethod(target)
11     and not isCyclic(target.callGraph())
```

A célmetódus felmozgatása előtt először azokat az általa használt mezőket és metódusokat kell felmozgatnunk, amelyek az eredeti refaktorálási cél befoglaló osztályában lettek definiálva. Mivel a használt metódusok ősosztályba mozgatásához ugyanerre van szükség, tehát ott rekurziót kell alkalmaznunk, először inkább a használt mezőket mozgatjuk fel (3-4. sor). Ezt követően áttérünk a felmozgatandó metódusokra, melyekre az éppen definiálás alatt álló szabályt alkalmazzuk rekurzívan (5. és 7. sor).

Előfordulhat azonban, hogy a rekurzió egy adott szintjén felmozgatandó segédmetódusok egymást is használják, és így, mikorra az iterációban elérünk egy ilyenhez, azt nem kell tovább mozgatni, hiszen ezt egy korábbi rekurzív hívás már megtette. Emiatt hozzáadunk az algoritmushoz egy feltételes vezérlési szerkezetet, ami a felmozgatás szükségességét határozza meg az alapján, hogy az iteráció kezdetén még a célmetódus befoglaló osztályában definiált segédmetódus nem lett-e egy korábbi iterációs lépésben felmozgatva (6. sor).

Megjegyzés. Az előző szabállyal ellentétben itt azért döntöttünk a feltételes programkonstrukció alkalmazása mellett, mert ennek szimulálásához – egy megfelelő előfeltételű refaktorálás hiánya miatt – olyan sémapéldányt kellett volna definiálnunk, amely kizárólag az elágazási feltétel szimulálására szolgált volna.

Miután a használt mezők és metódusok mozgatása végbement, a célmetódust is felmozgatjuk (8. sor). Vegyük észre, hogy a felmozgatási sémák egyébként bonyolult előfeltételeitől a sémák és sémapéldányok által kínált absztrakcióknak köszönhetően el tudtunk vonatkoztatni.

Előfeltételként kikötjük egyrészt azt, hogy a refaktorálás célja valóban egy metódus legyen (10.), másrészt azt, hogy a felmozgatandó metódusok hívási gráfjában ne legyen kör (11. sor). Ha megengednénk a köröket, a fenti naiv algoritmus végtelen rekurzióba kerülne, például akkor, ha két, ugyanabban az osztályban definiált, egymást hívó metódust szeretnénk felmozgatni. A rekurzió aggasztó lehet még a helyességbizonyítás tekintetében is, de a terminálást itt egyrészt a hívási gráfok elvárt körmentessége, másrészt az egy programba ágyazható metódusdefiníciók implicit módon megkövetelt végessége biztosítja.

Az utolsó összetett refaktorálás a legmagasabb szinten kijelölt kompozíciós lépéseket kapcsolja össze, definiálva ezáltal az esettanulmány céljaként kitűzött szabályt.

```
1  composite refactoring lift(visibility, name)
2  do
3      extract(visibility, name)
4      extractedMethod = target.enclosingMethod()
5      extractedMethod.cascadedLift()
6  when
7      isSegment(target)
```

Ebben előfeltételként írjuk elő, hogy a refaktorálási cél egy szegmens legyen (7. sor), melyet először a befoglaló osztály metódusává emelünk (3. sor), az így létrejövő, kiemelt metódust (4. sor) pedig az ősosztályba mozgatjuk (5. sor).

5.4. Kiértékelés

Az előző részben definiált sémapéldányok és összetett refaktorálások működését egy konkrét programon illusztrálom. Példaként tekintsük át lépésenként az 1. ábrán szereplő transzformációt. A következő kódpárok egy-egy lépést mutatnak be: a bal oldalakon a lépés előtti állapot a transzformálandó kódrészlet, a jobb oldalon a lépés utáni állapot a módosított kódrészlet kiemelésével látható, a végrehajtott példány pedig az ábra feliratában kerül megnevezésre. A teljes refaktorálási folyamat a következő ábrákon követhető végig: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int x = 1;
10         a = x;
11         g();
12         int y;
13         a = y;
14     }
15
16     void g() {
17         a = b = 0;
18     }
19 }
```

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int x = 1;
10         a = x;
11         g();
12         int y;
13         {}
14         a = y;
15     }
16
17     void g() {
18         a = b = 0;
19     }
20 }
```

2. ábra. A refaktorálás első lépéseként a `lift` példányt a csomagláthatóság és `h` függvénynév paraméterekkel hívjuk meg. Mivel a `lift` összetett refaktorálás, ezen belül az `extract` hívódik meg, ami szintén összetett, első lépéseként pedig az `introduceEmptyBlockAfter` lesz végrehajtva.

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int x = 1;
10         a = x;
11         g();
12         int y;
13         {}
14         a = y;
15     }
16
17     void g() {
18         a = b = 0;
19     }
20 }
```

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int y;
10         int x = 1;
11         a = x;
12         g();
13         {}
14         a = y;
15     }
16
17     void g() {
18         a = b = 0;
19     }
20 }
```

3. ábra. Ezután a `moveIntoNextBlock` kerül alkalmazásra, azonban nem fut le sikeresen, mivel egy olyan deklarációt kellene lemozgatni, amit a befoglaló blokkban használnak. Az `or` konstrukció miatt a `moveToTop` szabállyal próbálkozunk tovább, ami már sikeres lesz.

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int y;
10         int x = 1;
11         a = x;
12         g();
13         {}
14         a = y;
15     }
16
17     void g() {
18         a = b = 0;
19     }
20 }
```

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int y;
10         int x = 1;
11         a = x;
12         {
13             g();
14         }
15         a = y;
16     }
17
18     void g() {
19         a = b = 0;
20     }
21 }
```

4. ábra. A következő utasítást a `moveIntoNextBlock` alkalmazásával mozgatjuk a célblokkba.

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int y;
10         int x = 1;
11         a = x;
12         {
13             g();
14         }
15         a = y;
16     }
17
18     void g() {
19         a = b = 0;
20     }
21 }
```

```
1  class A {
2  }
3
4  class B extends A {
5      int a;
6      int b;
7
8      void f() {
9          int y;
10         int x = 1;
11         {
12             a = x;
13             g();
14         }
15         a = y;
16     }
17
18     void g() {
19         a = b = 0;
20     }
21 }
```

5. ábra. Az eredetileg kijelölt szegmens első utasítását is blokkba mozgatjuk a `moveIntoNextBlock` alkalmazásával.

```
class A {}
class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        {
            a = x;
            g();
        }
        a = y;
    }

    void g() {
        a = b = 0;
    }
}
```

```
class A {}
class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        new F() {
            public void apply() {
                a = x;
                g();
            }
        }.apply();
        a = y;
    }

    void g() {
        a = b = 0;
    }
}

interface F { void apply(); }
```

6. ábra. Mivel az elkészült blokk nem tartalmaz értéket visszaadó `return` utasítást, a `wrapInVoidLambda` szabállyal alakítjuk lambdává. Figyeljük meg, hogy ennek eredményeként a megfelelő interfész is létrejön.

```
class A {}
class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        new F() {
            public void apply() {
                a = x;
                g();
            }
        }.apply();
        a = y;
    }

    void g() {
        a = b = 0;
    }
}

interface F { void apply(); }
```

```
class A {}
class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        new F() {
            public void apply(int x) {
                a = x;
                g();
            }
        }.apply(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }
}

interface F { void apply(int x); }
```

7. ábra. A következő lépésben a létrejött lambda bemenő paramétereit emeljük ki az `extractInVariables` refaktorálással. Az `x` ilyen, hiszen egy lambdán kívüli lokális változó, viszont az `a` mezőt egyébként is elérjük az osztályban.

```
class A {}
class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        new F() {
            public void apply(int x) {
                a = x;
                g();
            }
        }.apply(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }
}
interface F { void apply(int x); }
```

```
class A {}
class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

8. ábra. A szegmenskimozgatás utolsó lépéseként a lambdát tesszük metódussá az `extract` szabállyal. Ugyan ennek formálisan nem része a már nem használt interfész eltávolítása, azonban egy ilyen refaktorálást is könnyen definiálhatnánk, így olvashatósági okokból inkább megtisztítjuk a kódot.

```
class A {
}

class B extends A {
    int a;
    int b;

    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

```
class A {
    int a;
}

class B extends A {
    int b;

    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

9. ábra. Mivel az előző lépésben befejeződött a kiinduló szegmensen hívott `lift` első, szegmenskimozgatásért felelős része, most folytatjuk a második komponenssel, ami a létrejött metódust felmozgató, szintén összetett `cascadedLift`. Ennek első lépéseként azonosítjuk a `h`-ban felhasznált `a` mezőt, amit a `lift` szabállyal mozgatunk fel.

```
class A {
    int a;
}

class B extends A {
    int b;

    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

```
class A {
    int a;
    int b;
}

class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

10. ábra. A következő lépésben azonosítjuk a `h`-ban felhasznált `g` metódust, amire – rekurzívan – meghívjuk az összetett `cascadedLift` refaktorálást. Ennek első lépéseként azonosítjuk `g`-ben használt `b` mezőt, amit a `lift` szabállyal mozgatunk fel.

```
class A {
    int a;
    int b;
}

class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

```
class A {
    int a;
    int b;

    void g() {
        a = b = 0;
    }
}

class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

11. ábra. Mivel `g`-nek nincs további, osztályon belüli függősége, felmozgathatjuk a `lift` szabállyal.

```
class A {
    int a;
    int b;

    void g() {
        a = b = 0;
    }
}

class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }

    void h(int x) {
        a = x;
        g();
    }
}
```

```
class A {
    int a;
    int b;

    void g() {
        a = b = 0;
    }

    void h(int x) {
        a = x;
        g();
    }
}

class B extends A {
    void f() {
        int y;
        int x = 1;
        h(x);
        a = y;
    }
}
```

12. ábra. Az előző lépéssel befejeztük a `h` függőségeinek felmozgatásához szükséges átalakításokat, így a `lift` szabállyal fel tudjuk mozgatni. Ezzel elvégeztük a kiinduló refaktorálásunk második, metódusfelmozgatási részét. Mivel ez egyben az utolsó is volt, ebben a lépésben a teljes refaktorálással elkészültünk.

6. További kutatási irányok

A korábbiakban bemutattam, hogy a módszer más paradigmára történő adaptálása milyen feladatokat jelöl ki, majd ezekre adtam megoldási javaslatokat is. Ezen javaslatok ugyan alkalmasak voltak arra, hogy a folyamat megvalósíthatóságát alátámasszák, de annak érdekében, hogy az ismertetett megoldás kiforrott, gyakorlatban jól használható rendszerré nője ki magát, további finomításra van szükség.

6.1. Újabb esettanulmányok

Az elméleti tervezés során hozott döntések helyességét célszerű lehet a nyelv gyakorlatban való tesztelésével ellenőrizni. Nagy számú, a paradigmára jellemző refaktorálások szempontjából reprezentatív módon megválasztott esettanulmány mikrolépésekre történő dekomponálásával, majd ezen lépések sémapéldányokként való felírásával meg lehetne vizsgálni, hogy a jelenleg javasolt sémák, illetve a metaelméletet karakterizáló szemantikus függvények és predikátumok a gyakorlatban mennyire széleskörűen alkalmasak refaktorálások definiálására. Az esettanulmányok során szerzett tapasztalatokat újabb sémák azonosítására, továbbá a meglévő sémák és metaelmélet finomítására, általánosítására lehetne felhasználni.

6.2. Verifikáció

A sémák és szemantikus függvények azonosítása után lehetőség nyílik a verifikáció alapjainak lefektetésére. Ehhez először formálisan, a korábban ismertetett Java szemantikában is meg kell fogalmazni a többszintű ekvivalenciadefiníciót; a sémákhoz tartozó, átírási szabályokra vonatkozó szerződéseket; továbbá a metaelmélet részeként definiált függvényeket és predikátumokat. A sémák helyességét – az alapul vett módszerben ismertetettekhez hasonlóan – várhatóan itt is strukturális indukcióval lehet majd belátni, ahol az indukció alapeseteit a sémák szerződései fogják igazolni. A bizonyítórendszer feltételeinek megfelelően a programminták ekvivalenciájának belátásához a felhasznált szemantika konfigurációszerkezetéből és levezetési lépéseiből egy aggregált szabályrendszert kell készíteni, ahol az eleve ekvivalensnek tekintett konfigurációpárokat az ekvivalenciadefiníció segítségével lehet majd karakterizálni.

6.3. Implementáció

Jelen dolgozat csupán elemzési és tervezési megfontolásokat ismertet, eredményei tisztán elméleti vonatkozásúak. Egy prototípus elkészítéséhez további kihívásokkal kell szembenézni. Az automatikus verifikációt egy későbbi implementációs iterációra halasztva először a nyelvi támogatást nyújtó fordítóprogramot, a metaelméleti információkat biztosító statikus elemzőt, illetve az előálló refaktorálásokat ténylegesen végrehajtó forráskód-transzformációs eszközt kell megvalósítani.

A fordítóprogram megírását az is nehezíti, hogy itt egy, a Java nyelvre szabott mintaillesztési rendszert is létre kell hozni. Ugyanakkor a teljes nyelvi támogatás és a mintaillesztés akár egy lépésben is realizálható lehet az Xtext [14] nyelvfejlesztő keretrendszert felhasználva. Ebben egy EBNF-jellegű formális nyelvtan megadásával automatikusan előállítható egy teljes, Eclipse-alapú nyelvtámogatási eszközkészlet IDE-integrációval, statikus validációval és valósidejű fordítással. Továbbá az Xtext beépített kifejezésnyelve átalakítható úgy, hogy szintaxisában a Java-t szimulálja [5], az ehhez szükséges nyelvtan megfelelő kiegészítésével pedig megvalósítható lehet egy, a mintákban szereplő metaváltozók felismerésére képes elemző.

A statikus elemző és a transzformációs eszköz szintén megvalósítható lehet egyetlen technológiát, a JDT-t [8] alapul véve. Ez az absztrakt szintaxisfa szintjén teremt lehetőséget Java forráskódok elemzésére és módosítására. Érdekes problémát jelent az utólag, akár a Java virtuális gép futása közben definiált refaktorálások beolvasása, melyhez a dinamikus osztálybetöltési funkciót lehet célszerű megvizsgálni.

7. Összefoglalás

Dolgozatomban ismertettem azt a munkát, amely az Erlang programozási nyelv mikrorefaktorálásainak séma alapú definiálására biztosít lehetőséget, majd részleteztem ennek az objektumorientált paradigmára való adaptálásának lépéseit.

Formális szemantikai fogalomrendszerhez kapcsolva mutattam be egy forráskódtranszformációk jelentésmegőrzésének vizsgálatára alkalmas, többszintű ekvivalenciadefiníciót, melyben Java programok osztályhierarchiájának kompatibilitását a publikus interfészeik szemantikus megfelelésére vezettem vissza.

Az alapvetően refaktorálási előfeltételek leírásához használható szemantikus függvények és predikátumok között kijelöltem olyanokat, melyek paradigmára jellemző absztrakciókat biztosítanak, a bevezetett hierarchia-megfeleléshez illeszkedve pedig definiáltam a metódusok inter- és intrahierarchia-elérhetőségének fogalmát.

A fenti eredményeket felhasználva meghatároztam a lokális, blokk, lambda és osztály sémákat, az így konstruált prototípus használhatóságát pedig egy összetett refaktorálási esettanulmány mikrolépéseinek formális definiálásával, majd az előálló átalakítások egy konkrét programon történő szimulálásával igazoltam.

Hivatkozások

[1] Alexander Aiken: Cool: A portable project for teaching compiler construction. *SIGPLAN Not.*, 31. évf. (1996. július) 7. sz., 19–24. p. ISSN 0362-1340. URL http://doi.acm.org/10.1145/381841.381847. 6 p.

[2] Denis Bogdanas – Grigore Roşu: K-java: A complete semantics of java. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15 konferenciasorozat. New York, NY, USA, 2015, ACM, 445–456. p. ISBN 978-1-4503-3300-9.
URL http://doi.acm.org/10.1145/2676726.2676982. 12 p.

[3] Ştefan Ciobâcă – Dorel Lucanu – Vlad Rusu – Grigore Roşu: A language-independent proof system for full program equivalence. *Formal Aspects of Computing*, 28. évf. (2016. mar) 3. sz., 469–497. p.
URL https://doi.org/10.1007%2Fs00165-016-0361-7.

[4] Marc L. Corliss – David Furcy – Joshua Davis – Lori Pietraszek: Bantam java compiler project: Experiences and extensions. *J. Comput. Sci. Coll.*, 25. évf. (2010. június) 6. sz., 159–166. p. ISSN 1937-4771.
URL http://dl.acm.org/citation.cfm?id=1791129.1791160. 8 p.

[5] Németh Dávid János: XtxtUML – alkalmazásterület-specifikus nyelv végrehajtható UML modellezéshez. BSc szakdolgozat (Eötvös Loránd Tudományegyetem). Budapest, 2016.

[6] Dániel Horpácsi – Judit Kőszegi – Zoltán Horváth: Trustworthy refactoring via decomposition and schemes: A complex case study. In Alexei Lisitsa – Andrei P. Nemytykh – Maurizio Proietti (szerk.): Proceedings Fifth International Workshop on *Verification and Program Transformation,* Uppsala, Sweden, 29th April 2017, Electronic Proceedings in Theoretical Computer Science konferenciasorozat, 253. köt. 2017, Open Publishing Association, 92–108. p.

[7] Dániel Horpácsi – Judit Kőszegi – Simon Thompson: Towards trustworthy refactoring in erlang. In Geoff Hamilton – Alexei Lisitsa – Andrei P. Nemytykh (szerk.): Proceedings of the Fourth International Workshop on *Verification and Program Transformation,* Eindhoven, The Netherlands, 2nd April 2016, Electronic Proceedings in Theoretical Computer Science konferenciasorozat, 216. köt. 2016, Open Publishing Association, 83–103. p.

[8] JDT. `https://www.eclipse.org/jdt/`. Elérés dátuma: 2018. május.

[9] William F. Opdyke: *Refactoring Object-oriented Frameworks*. (Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[10] Grigore Roşu – Andrei Ştefănescu: From hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, LNCS konferenciasorozat, 7436. köt. 2012. Aug, Springer, 387–402. p.

[11] Max Schäfer – Mathieu Verbaere – Torbjörn Ekman – Oege Moor: Stepping stones over the refactoring rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa konferenciasorozat. Berlin, Heidelberg, 2009, Springer-Verlag, 369–393. p. ISBN 978-3-642-03012-3. URL `http://dx.doi.org/10.1007/978-3-642-03013-0_17`. 25 p.

[12] Andrei Stefănescu – Daejun Park – Shijiao Yuwen – Yilong Li – Grigore Roşu: Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016 konferenciasorozat. New York, NY, USA, 2016, ACM, 74–91. p. ISBN 978-1-4503-4444-9.
URL `http://doi.acm.org/10.1145/2983990.2984027`. 18 p.

[13] TIOBE Index. `https://www.tiobe.com/tiobe-index/`. Elérés dátuma: 2018. május.

[14] Xtext. `https://www.eclipse.org/Xtext/`. Elérés dátuma: 2018. május.