Definition/Use reprezentáció Java nyelven TDK-dolgozat

SZEGEDI TUDOMÁNYEGYETEM

TERMÉSZETTUDOMÁNYI ÉS INFORMATIKAI KAR

SZOFTVERFEJLESZTÉS TANSZÉK

Szerző:

Tóth Bojnik Tibor

Programtervező Informatikus BSc

4. évfolyam

Témavezető:

Soha Péter

PhD hallgató

Szeged, 2024

Tartalomjegyzék

1. fejezet

Bevezetés

Ebben a dolgozatban azt mutatom be, hogy hogyan készítettem el a hatékony programszeletelőhöz a Definition/Use Táblámat, amely lehetővé teszi az újabb Java programkódok hatékony szeletelését, és kiküszöböli az előzőek problémáit, hiányosságait, korlátait.

A program szeletelő (angolul program slicer) egy olyan a program terjedelmének lekicsinyítésére szolgáló eszköz, melynek célja az adott program egy részét kiemelni, vagy elkülöníteni és külön részként vizsgálni. Ezt olyan módon teszi, hogy egy változót vesz alapul, és azokat a kódrészleteket tartja meg, melyek befolyásolják az adott változónak az értékét egy adott ponton/pontig. A program segíthet a tesztelőknek, vagy akár a fejlesztőknek is a program hibáinak feltárásában, azok javításában, vagy alapvetően a program működésének a megértésében, mivel könnyebb kihagyni azokat, amelyek nem befolyásolják a hibás részeket, és a program egy részét átvizsgálni az egésszel szemben [1].

A második fejezetben bemutatom, a programszeletelést, mint hibaallokációs technika, összefoglalom a legfontossabb irányait, változatait, melyiket mire használják, azok hogyan működnek.

A harmadik fejezetben bemutatom, hogy mik lehetnek a buktatóik a program szeletelőknek, és felvezetem, hogy a legfőbb nehézséget okozó tárigényre irányuló problémát hogy tudtam megkerülni, és ez mért lesz hatékonyabb.

A negyedik fejezetben áttekintem, milyen Java nyelvű implementációk jöttek létre eddig. Megnézem mik voltak azok a részek, amelyek gondot okoztak. Kitérek a hiányosságaikra, és korlátaikra. Megfigyelem mik voltak a közös problémák, amelyek a használhatatlanságukat okozták a nagyobb, újabb nyelvi elemeket tartalmazó projektekre nézve.

Az ötödik fejezetben bemutatom az általunk fejlesztett szeletelő technológia hátterét. Milyen könyvtárakat használtam hozzá. Hogyan készült el a Def/Use táblám, amely lehetővé teszi a nagyobb, újabb nyelvi elemeket tartalmazó projektek szeletelését is, és miért lesz hatékonyabb, valamint használhatóbb a többinél.

A hatodik fejezetben bemutatom egy példán keresztül hogyan, és milyen információkat nyújt a Def/Use táblám.

A hetedik fejezetben elmagyarázom mik jelentették a projektben a legnagyobb kihívásokat, és ezeket hogyan tudtam megoldani.

A nyolcadik fejezetben pedig felvázolom a jövőbeni terveket, hogy milyen javítások lesznek még eszközölve, illetve milyen újabb funkciókkal lesz még kibővítve a szeletelő program.

2. fejezet

Programszeletelés

A program szeletelés egy olyan hibaallokalizáló módszer, amely arra fókuszál, hogy egy meghatározott kritérium alapján elhagyja a nem releváns részeket [1, 2, 3]. A szeletelési kritérium egy változó lesz, amely függően mely irányba szeletelünk fogja meghatározni, hogy mely részek lesznek érdekesek a számunra. A szeletelés segíthet hibakeresésben, újratervezésben, az adott program megértésében, szoftvermérésekben.

2.1. Statikus és dinamikus szeletelők

Megkülönböztetünk statikus, illetve dinamikus szeletelőket. Ezek attól függnek, hogy az adott programot az aktuális paraméterekkel futtatva dinamikusan elemezzük, vagy statikus elemzést hajtunk végre rajta. A statikus szeletelésnél az adott program szelet az előre meghatározott kritériumok, mint a control és data flow vagy a def-use láncok alapján jönnek létre. A dinamikus program szeletelésnél pedig a végrehajtás alapján gyűjtődik össze az információ a változók értékeiről és a különböző statement-ek végrehajtása alapján. Itt a probléma az, hogy control és dependence gráfok építődnek, amik exponenciális növekedése végett nagyon lekorlátozódnak a szeletelhető programok méretei. Ezen felül van még a releváns szeletelés, ahol figyelembe vesszük mind a statikus és dinamkus szeletelést. Ez segíthet még jobban leszűkíteni az adott programrészeket, és csak a releváns részeket veszi bele a szeletbe.

2.2. Forward és backward szeletelés

Másik felosztásban pedig szét kell választani a forward slicing-et és a backward slicing-et. Az előbbinél egy adott pontból indulva haladunk tovább a programban, és határozzuk meg, hogy az adott változó, vagy kifejezés értéke mely részeket fogja befolyásolni. Ezt akkor használjuk, ha arra vagyunk kíváncsiak, hogy az adott változó hogyan befolyásolja a program további részeit. Illetve az utóbbinál pedig az adott ponttól visszafelé haladunk, és határozzuk meg, hogy mely korábbi részek befolyásolják az adott változó, vagy kifejezés értékét. Ez pedig akkor lehet hasznos, ha azt keressük, mi, mely változók, részek okozhattak hibát az adott helyen.

2.3. Példa a forward és backward szeletelésre

A következőkben bemutatom az előre és hátra irányuló dinamikus programszeletelés menetét az alábbi példán. Itt az egyszerűség kedvéért az adott futáshoz tartozó inputot beégettem a kódba:

```
1  public class Example {
2      public static void main(String[] args) {
3          int i = 0;
4          int a = 20;
5          int n = 5;
6          int s = 12;
7          if(n < 10)
8              n += 10;
9          while (i <= a){
10             if (a > 0)
11                 s += 2;
12             else
13                 s *= n;
14             i++;
15         }
16         System.out.println(s);
17         System.out.println(n);
18     }
19 }
```

2.1. forráskód. Egyszerű példakód a szeleteléshez

2.3.1. Backward slicing

A 2.1-es forráskód példán látható kódnak a $C := \langle 16; s \rangle$ kritériumhoz tartozó hatrafelé irányban számított dinamikus szelete az alábbi módon áll elő:
Alapvetően függ magától az első létrehozástól, ami a *(6)*-os sor. Statikus elemzés esetén minden értékadását is bele kellene vennünk, tehát a *(11)* és *(13)* sorokat is. Viszont dinamikusan tekintve a jelenlegi értékadásokkal beláthatjuk, hogy az elágazásban az igaz ágra fogunk ráfutni, tehát csak a *(11)*-es sor kerül bele a szeletbe. Ez a sor viszont függ a feltételtől, tehát a *(10)*-es soról, illetve a ciklus feltételétől, ami a *(9)*-es sort jelenti. Ez kibővül még a *(14)*-es sorral, mivel a feltételben szereplő i változónak az értéke itt változik. Valamint ezáltal függeni fog a feltételekben felhasznált változók *(a,i)* értékeitől, amelyek a *(3)*-as és *(4)*-es sorban adottak. Tehát a *(16)*-os sorban lévő kiírás által használ s változó értéke függ a *(3)(4)(9)(10)(11)(14)* soroktól.

2.3.2. Forward slicing

Amennyiben a 2.1-es példán a $C := \langle 5; n \rangle$-hez tartozó előre irányuló dinamikus szeletet akarjuk kiszámolni, azt az alábbi módon történik: Itt azt nézem, mely sorok azok, amelyek értékének kiszámolásához közvetlen, vagy közvetett módon felhasználja a program az n változó értékét. Az első utasítás, amely függőségben áll a kritériumbeli változóval, az a *(7)*. sorban található feltételes elágzás predikátuma. Tekintve, hogy az adott utasításban $n = 5$, így a predikátum értéke igaz, emiatt az `if`-hez tartozó *(8)*-es sorbeli utasítás végre fog hajtódni. A *(9)*-es sorhoz tartozó utasítás az i és az a változók aktuális értékei miatt szintén igaz a `while` feltétele, viszont mivel a *(10)*-es utasítás, vagyis az $a > 0$ kifejezés az $a = 20$ miatt szintén igazra értékelődik ki, így a n változót felhasználó *(13)*. sorra nem adódik át a vezérlés. Tehát a kritérium által definált dinamikus szelet a *(7)(8)(17)*.

3. fejezet

Új megközelítés

A legfőbb probléma a programszeletelésben, hogy a különböző függőségeket gráfokban kezelték. Ezek "robbanását" okozta az, hogy a tranzitív függéseket is kellett tárolniuk, és minden változót minden függő résszel összekötniük. Ezáltal bármilyen kiterjesztése az adott programnak, akár egy-egy fünkcióhívással is, nagy mértékben növelte a gráf méretét, köszönhetően a redundáns, vagy a számunkra irreleváns részek belevételének.

Ennek a problémának az elkerülésére jött létre egy megoldás elsődlegesen C programokra, amely reális méretű programokra is használható szeletelőt kínált. A megoldás leredukálta a szeletelő tárigényét azzal, hogy a gráfok használata helyett egy másik megközelítést alkalmaztak, mely jóval kisebb méreteket öltött.

Ebben az új megközelítésben a gráfok helyett egy Definition/Use táblát hoztak létre, mellyel képesek voltak valódi méretű C programok szeletelésére, melyekben voltak pointerek, funkció hívások, ugrások.

A szeletelő elsőnek analizálta a bejövő programot, és létrehozta ezt a táblát a statikus függések alapján, és a program instrumentálva lett, hogy megkapja a szükséges runtime információkat. Majd az instumentált program fordítva és futtatva lett, ami által létre lett hozva a Trace, ami tartalmazta a dinamikus információkat, amelyek szükségesek voltak a dinamikus szeletelőhöz.

A Tábla tartalma minden sorra az adott változó(k) definíciója, valamint a definiáláshoz használt változók set-je. A különböző definiált és használt változók lehettek:

- `skalár`
- `predikátum`

- `output`
- `dereference`
- `funkció hívásban argumentum`
- `return`

A skalár változókat a konstans címükkel azonosították, a predikátum változókat *pn* jelöléssel látták el, ahol az *n* volt a sorszáma a predikátumnak. Az output változók *on* jelöléssel lettek ellátva, ahol az előzőhöz hasonló módon az *n* jelentette a sorszámot. Ezek olyan *dummy* változók voltak, ahol a use set-ben található változók értékei nem voltak sehol tovább használva. A dereference-ket *dn*-el jelölték. Az argumentum változókat a függvényhívásoknál egy *arg(f,n)*-el jelölték, ahol az "f" volt a funkció neve, n a funkcióban lévő paraméternek a száma. Ez a változó definiálva volt a funkcióhívásnál és használva a függvény deklarációnál. Return változók pedig *ret(f)* formában voltak hivatkozva, ahol az *f* volt a függvény neve. Ez a függvények végén voltak definiálva, és a függvényhívások végén használva. [4]

4. fejezet

Kapcsolódó kutatások

Az eddigi szeletelők legfőbb hibája, hogy mind gráfokat építettek, és azok segítségével határozták meg a különböző függőségeket. Emiatt nagyon korlátozottak voltak a szeletelendő programok méretére, a gráfok exponenciális növekedése végett [5]. A következőkben felsorolok néhány létező program slicert Java nyelvre, és hogy melyik verziót támogatták, miért jöttek létre, mik voltak a limitációik, hibáik:

4.1. JavaSlicer

JavaSlicer az okból jött létre, mert előtte csak a JSlice létezett, melyet nehéz volt beindítani, és csak nagyon kevés programra működött. JSlice-t úgy oldották meg, hogy módosítottak egy úgynevezett “Kaffe VM”-et (Virtual Machine), ami egy régebbi Java verzióhoz készült. Ez teljesen C-ben íródott, és arra lett kitalálva, hogy minden információt kiírjon a trace rekonstruktálásához. Ez még a Sun-JDK 1.4-et használta, amit manapság már nagyon régi, és elavult. Hogy ne függjön a VM-től, a JavaSlicer Java Agent-et használt a bytecode instrumentálásához. A megvalósításhoz a “hammacher-bachthesis”-ben foglaltakat használták fel. Így a programban a változók azonosításához ők is hashCode-okat használtak, viszont ők kihasználták a gyenge hivatkozásokat (weak references), és egy olyan Map-et építettek fel, amelyekben a Garbage Collector segítségével mindig törlődtek a nem használt változók [6, 7].

4.2. Slicer4J

A Slicer4J azért jött létre, mivel előtte csak a JavaSlicer létezett, amely akkoriban még csak a Java 6-ot támogatta, illetve nem volt megoldásuk a többszálú programokra (multithreading). Ők úgynevezett “low-overhead instrumentation”-t használtak, hogy nyomon kövessék a program futását. Ennek segítségével hoztak létre egy “thread-aware, interprocedural dynamic control-flow graph”-ot, amely figyelembe vette a szálak kezelését, és a függvények közötti ugrásokat is [8, 9].

4.3. JavaSDGSlicer

A JavaSDGSlicer fejlesztői is szintén a gráfos megoldás mellett döntöttek. Leírták, hogy alapvetően annyira kiforrott a gráfos slicerek létrehozása, hogy szinte minden nyelvi elemre készültek bővítmények (extensions) mint például a PPDG (pseudo-predicate program dependence graph), amely kifejezetten a “feltétel nélküli ugrások” (unconditional jumps) készült. Viszont a probléma ezekkel az, hogy alapból nem kompatibilisek egymással. Az ő slicerük próbált olyan megoldásokat nyújtani, amely ezeket kompatibilissé tette egymással [10, 11].

4.4. Indus Kavari Slicer

Ez a szeletelő nem épít gráfokat, helyette fókuszál a control- és data-flow analizálására a szimbolikus végrehajtás útján. A program végrehajtását szimbolikusan trace-li végig, ahol a bemenet és a változók szimbolikusan vannak reprezentálva, nem pedig konkrétan. Tehát a program figyeli a különböző útvonalakat, és a feltételeknek, és ezeknek az irányoknak megfelelően generálja a szeleteket. Így ők egészen a párhuzamosságig elértek a támogatással. Viszont mivel a 2000-res évek közepén voltak ezek a fejlesztések, ezért az újabb nyelvi elemeket ők sem támogatják [12, 13].

5. fejezet

D/U reprezentáció építése Javaban

5.1. Használt toolok

5.1.1. ASM

Az ASM egy Java bytecode manipuláló, és analizáló keretrendszer. Vele képesek vagyunk módosítani a már létező class filejainkat, de akár teljesen újakat is létrehozhatunk. Maga a tool körülbelül ugyanazt tudja, mint más Java bytecode manipuláló toolok, csak az ASM fókuszál a teljesítményre, mert amit csinál, azt megpróbálja olyan kicsiben és olyan gyorsan, amennyire csak lehet. Ezért is tökéletes számunkra a dinamikus szeletelőnkhöz.

5.1.2. JavaParser

A JavaParser egy forráskód analizáló, transzformáló, generáló könyvtár. Ez a tool kínál nekünk egy `Abstract Syntax Tree-t (AST)` a Java kódunkból, amivel aztán könnyedén tudunk dolgozni. Ez által tudja analizálni a forráskódot, kinyerve értékes információkat a különböző mintákon keresztül. Akár megváltoztatni bizonyos részeket a kódban egy megfelelő minta alapján, vagy még generálni is kódrészeket a kódunkba.

5.2. Program szeletelő kialakítása

5.2.1. Def/Use tábla létrehozása

A Java és a C nyelvek közötti különbségeket figyelembe véve hoztam létre a saját program szeletelőmet. A szeletelő a Def/Use táblát a JavaParser nevű könyvtár segítségével hoztam létre. A legtöbb munkát az `ASM` könyvtár kezeli, valamint a futtatott sorok kinyerésére használtam a `Java Debug Interface (JDI)`. A Def/Use táblámat a `Parser` által létrehozott visitorok felülírásával valósítom meg. A folyamat lépései:

1. A táblát először egy ideiglenes táblába hozzuk létre. Elsőnek üres `DU` és `Variable` elemekkel. Ez azért szükséges, mivel a használati változóknál a visitorokban nincs elegendő információm arról, melyik változónak a használatai, így ha nem csak az adott változó definíció lenne ott, nem tudnám melyhez kell hozzáadni.

2. Majd a visit metódusokból kinyert információk alapján létrehozok különböző, a Variable osztályból létrehozott változót.

3. Ez után a táblából a megfelelő sorindexű változóhoz létrehozok egy referenciát.

4. Utána feltöltöm a kapott `DU`-t a megfelelő definícióval, vagy használattal.

5. A változók kinyerése mellett töltöm fel a végleges táblánkat is, amely több definíciót, és azok használatát képes tárolni. Ebbe a táblába elsőnek a függvény argumentum definíciókat és azok használatait, valamint a függvényhívások adatait tárolom.

6. Utolsó lépésként pedig a végső táblához hozzáadom az ideiglenes táblám változóit is.

Ezeken felül predikátum változókat is bevezettünk, melyek vezérlési szerkezetek, ciklusok, elágazások, stb. kezelését segítik. Példák: p ciklusoknál és elágazásoknál, o az output-nál, tr try-nál, stb.

5.2.2. Variable osztály és Scope

A változók eltárolása különbözhet, például:

- `AnnotationVariable`

- `MethodVariable`
- `OutputVariable`
- `PredicateVariable`
- `SimpleVariable`

A változók kezelését segíti a `Variable` osztály, melynek adattagjai: `name`, `lineNumber`, és `fullScope`. A `Scope` további részleteit is tároljuk, például a változó fájlját, osztályát, metódusát, valamint a belsőbb scope-okat.

5.3. SymbolSolver használata

A SymbolSolver segítségével a változók típusát tudjuk meghatározni. A konfiguráció lépései:

- Létrehozunk egy `TypeSolver`-t, amely kombinálja a `JavaParserTypeSolver`-t és a `ReflectionTypeSolver`-t.
- Ezt a kombinált `TypeSolver`-t átadjuk a `JavaSymbolSolver`-nek.
- A `ParserConfiguration`-ben meghatározzuk a Java verziót és a SymbolSolvert.

5.4. Identitások kezelése

5.4.1. HashCode alapú azonosítás

Az `identityHashCode` segítségével a változók egyedileg azonosíthatók. Ez nem primitív típusok esetén bizonyult hatékonynak. Mivel Java-ban nem érhetőek el közvetlenül a memóriacímek, ezért az ASM könyvtár kód injektálási lehetőségeit használjuk.

5.4.2. Mezőváltozók kezelése

A mezőváltozók HashCode-jainak kinyeréséhez a statikus inicializáló blokkot használjuk, mivel ezek nem metóduson belül jönnek létre. A megfelelő kód injektálásával ezek is azonosíthatóvá váltak.

6. fejezet

Példa a saját reprezentációmból

```
1  public class testCustomClass {
2    public static void main(String[] args) {
3      Pair b = new Pair(9,47);
4      Pair a = new Pair(7, 6);
5      a.x = b.getY();
6      b.y = 64;
7      a.setX(55);
8      a.addToX(4);
9      b.addToX(9);
10     int valami = a.getX();
11     System.out.println(valami);
12   }
13 }
14
15 class Pair {
16   int x;
17   int y;
18
19   Pair(int x, int y) {
20     this.x = x;
21     this.y = y;
22   }
23
24   public void addToX(int x){
25     this.x += x;
26   }
```

```
27
28    public int getX() {
29      return x;
30    }
31
32    public void setX(int x) {
33      this.x = x;
34    }
35
36    public int getY() {
37      return y;
38    }
39
40    public void setY(int y) {
41      this.y = y;
42    }
43  }
```

6.1. forráskód. testCustomClass példa kód

Ennél a példánál, amely kicsit bonyolultabb egy általánosabb programnál extra figyelmet igényelt, hogy helyesen tudjuk kezelni a belső osztályokat is. Mint említettem, ezért kell külön kezelni a file, és clazz értékeinket, mivel itt is egy fileban, de különböző osztályban lévő változókat is találunk. A Def/Use táblában az egyszerű értékadás, mint a példában a 6-os sor ily módon jelenik meg:

```
1  {
2    "dus": [{
3      "def": {
4        "name": "b.y",
5        "lineNumber": 6,
6        "fullScope": {
7          "file": "src/main/resources/files/sources/
               testCustomClass.java",
8          "clazz": "testCustomClass",
9          "method": {
10           "begin": 2,
11           "end": 12,
```

```
12            "name": "main"
13          },
14          "statementScopes": []
15        },
16        "type": "int",
17      },
18      "uses": []
19    }],
20    "lineNumber": 6
21  }
```

6.2. forráskód. Egy objektum és mezőjének a definiálása DU-ban

Itt látható, hogy csak egy definiálásunk van a 6os sorban. Itt a *type* azért int típus lesz, mivel konkrétan a *b* objektum *y* változójának a típusa lesz érdekes.

A getter setter metódusokat szintén tudom statikusan is átalakítani. Ezt egyszerűen egy névellenőrzéssel oldom meg, mivel a visit metódus, amely a *MethodCallExpr*-öket ismeri fel, és kezeli, ott egy név ellenőrzéssel meg tudom állapítani, hogy az adott metódus getter vagy setter, és mely változóra vonatkozik. Például a 7-es sorban lévő kódra a Def/Use táblában az alábbi bejegyzés jön létre:

```
1  {
2    "dus" : [ {
3      "def" : {
4        "name" : "a.x",
5        "lineNumber" : 7,
6        "fullScope" : {
7          "file" : "src/main/resources/files/sources/
               testCustomClass.java",
8          "clazz" : "testCustomClass",
9          "method" : {
10           "begin" : 2,
11           "end" : 12,
12           "name" : "main"
13         },
14         "statementScopes" : [ ]
```

```
15        },
16        "endLineNumber" : 7,
17        "owner" : "a",
18        "getSet" : true
19      },
20      "uses" : [ ]
21    } ],
22    "lineNumber" : 7
```

6.3. forráskód. Setter metódus feloldása a DU-ban

A különbség mindössze, hogy ez metódusból jön, ami miatt *MethodVariable* lesz, aminek más változói vannak, viszont névileg és ez által a slicerben is összeegyeztető lesz, hogy az *a* változó *x* mezőjére hivatkozunk általa.

Ahogy fentebb említettem vannak részek, amiknél szükségessé váltak a különböző saját változók bevezetése is. Ilyen többek között a 11-es sor, ahol az output változó letsz definiálva az alábbi módon:

```
1  {
2    "dus" : [ {
3      "def" : {
4        "name" : "o11",
5        "lineNumber" : 11,
6        "fullScope" : {
7          "file" : "src/main/resources/files/sources/
               testCustomClass.java",
8          "clazz" : "testCustomClass",
9          "method" : {
10           "begin" : 2,
11           "end" : 12,
12           "name" : "main"
13         },
14         "statementScopes" : [ ]
15       },
16     },
17     "uses" : [ {
```

```
18        "name" : "valami",
19        "lineNumber" : 11,
20        "fullScope" : {
21          "file" : "src/main/resources/files/sources/
                testCustomClass.java",
22          "clazz" : "testCustomClass",
23          "method" : {
24            "begin" : 2,
25            "end" : 12,
26            "name" : "main"
27          },
28          "statementScopes" : [ ]
29        },
30        "type" : "int",
31      } ]
32    } ],
33    "lineNumber" : 11
34  }
```

6.4. forráskód. segéd változók a DU-ban

Itt látható, hogy *on* alapján neveztük el, ahol az *n* a sor száma. Valamint a *valami* változó lesz használva az adott sorban. A függvények argumentumainak a kezelése pedig az alábbi módon történik a 19-es sorban:

```
1   {
2     "dus" : [ {
3       "def" : {
4         "name" : "x",
5         "lineNumber" : 19,
6         "fullScope" : {
7           "file" : "src/main/resources/files/sources/
                testCustomClass.java",
8           "clazz" : "Pair",
9           "method" : {
10            "begin" : 19,
```

```
          "end" : 22,
          "name" : "Pair"
        },
        "statementScopes" : [ ]
      },
    },
    "uses" : [ {
      "name" : "arg(Pair,1)",
      "lineNumber" : 19,
      "fullScope" : {
        "file" : "src/main/resources/files/sources/
           testCustomClass.java",
        "clazz" : "Pair",
        "method" : {
          "begin" : 19,
          "end" : 22,
          "name" : "Pair"
        },
        "statementScopes" : [ ]
      },
      "endLineNumber" : 22,
    } ]
  }, {
    "def" : {
      "name" : "y",
      "lineNumber" : 19,
      "fullScope" : {
        "file" : "src/main/resources/files/sources/
           testCustomClass.java",
        "clazz" : "Pair",
        "method" : {
          "begin" : 19,
          "end" : 22,
          "name" : "Pair"
```

```
        },
        "statementScopes" : [ ]
      },
    },
    "uses" : [ {
      "name" : "arg(Pair,2)",
      "lineNumber" : 19,
      "fullScope" : {
        "file" : "src/main/resources/files/sources/
           testCustomClass.java",
        "clazz" : "Pair",
        "method" : {
          "begin" : 19,
          "end" : 22,
          "name" : "Pair"
        },
        "statementScopes" : [ ]
      },
      "endLineNumber" : 22,
    } ]
  } ],
  "lineNumber" : 19
}
```

6.5. forráskód. Pair osztály konstruktorának változói a DU-ban

Ami lényeges, hogy mindkét változó esetén az adott nevű változót definiálljuk, és az *arg(függvénynév,*hanyadik argumentum). Ilyen módon kössük össze az argumentum változóját a függvényhívásban szereplő értékekkel, illetve változókkal. Függvényhívásnál pedig pont fordítva az alábbi módon:

```
{
  "dus" : [ {
    "def" : {
      "name" : "arg(addToX,1)",
      "lineNumber" : 9,
```

```
      "fullScope" : {
        "file" : "src/main/resources/files/sources/
            testCustomClass.java",
        "clazz" : "testCustomClass",
        "method" : {
          "begin" : 2,
          "end" : 12,
          "name" : "main"
        },
        "statementScopes" : [ ]
      },
      "endLineNumber" : 9,
      "owner" : "b",
    },
    "uses" : [ {
      "name" : "9",
      "lineNumber" : 9,
      "fullScope" : {
        "file" : "src/main/resources/files/sources/
            testCustomClass.java",
        "clazz" : "testCustomClass",
        "method" : {
          "begin" : 2,
          "end" : 12,
          "name" : "main"
        },
        "statementScopes" : [ ]
      },
      "type" : "int",
    }, {
      "name" : "b",
      "lineNumber" : 9,
      "fullScope" : {
        "file" : null,
```

```
38          "clazz" : "Pair",
39          "method" : {
40            "begin" : 9,
41            "end" : 9,
42            "name" : "addToX"
43          },
44          "statementScopes" : [ {
45            "begin" : 9,
46            "end" : 9,
47            "statement" : "addToX"
48          } ]
49        },
50        "type" : "Pair",
51      } ]
52    } ],
53    "lineNumber" : 9
54  }
```

6.6. forráskód. Függvényhívás tárolása a DU-ban

Viszont ahogyan láthatjuk a 9-es sorban található függvényhívásnál nem lesz teljesen helyes megoldásunk, mivel a használatban nem a *b.x* van, csak a *b*. Ezt viszont statikusan nem tudjuk kinyerni az adott sorból, ezt a szeletelő algoritmus fogja összeegyeztetni a függvénytörzsben található adatok alapján.

7. fejezet

Kihívások

7.1. HashCode-k kinyerése

Egyik, ha nem a legnagyobb kihívás az volt, hogy tudjak szerezni egy egyedi, a megkülönböztetésre alkalmas azonosítót a változóknak. Az eredeti cikkben a C nyelvből könnyedén lehetett a változókat azonosítani azok memóriacímével, viszont a Java ezek elérését a legmagasabb szinten korlátozza. Sok kutatás, és próbálkozásom volt különböző könyvtárakkal, natív kódokkal sikertelenül. Ezek után ugrott be az ötlet, hogy a HashCode, és főként az identityHashCode, amely a változó értékváltozásakor sem változik erre megfelelő lehet. Viszont nehézségeket jelentett ezek kinyerése is, mivel semmilyen formában nem tudtam az adott programból kinyerni magukat az objektumokat. Ezért esett a választásom a kód injektálásra. Sok-sok próbálkozás után a legmegfelelőbbnek az bizonyult, ha minden egyes értékadás után beinjektálok egy kódot, mely a konzolra kiírja a változó sorát, nevét, és HashCode-ját. Ezen belül ami még a legnehezebb volt, ennek a három adatnak az összeegyeztetése, mivel a különböző visitorok, amelyeken keresztül ezeket az adatikat elértem, kinyertem, nem teljesen olyan sorrendben hívódtak meg, ahogy nekem arra szükségem lett volna. Viszont a classVisitorokat elég jól lehet sorrendizálni, így külön classVisitorokkal ki tudtam ezt küszöbölni.

7.2. Név feloldások

A másik nagy probléma az volt, hogy a fejlesztők által írt Class file-okban a *this* referenciákat fel tudjam oldani, hogy azok könnyen összeköthetőek legyenek a változókkal.

Itt a legnagyobb problémma az volt, hogy statikusan nem tudtam minden esetben meghatározni, hogy például van két ugyanolyan objektumunk (például *a* és *b*), és hivatkozunk a *this.x* mezőjére, akkor az most az *a* vagy a *b* objektum mezője-e. Ezt a problémát végül a végrehajtások sorainál az "ugrások", ahol egy függvényhívás volt, ami után kerültek elő a *this* referenciák, összeegyeztethetőek voltak az ugrás előtti sorban a változóval, melyre történt a hívás. Ilyen hasonló észrevételekkel, és mintákkal sikeresen fel tudtam oldani őket, ezáltal is elősegítve a szeletelőnk működését, és a változók sikeres azonosítását.

8. fejezet

Jövőbeni tervek

A projektem fejlesztése jelenleg is folyamatban van, viszont megvannak a tervek, amelyeket még meg szeretnék rajta valósítani.

8.1. HashCode-ben történő javítások

Az egyik javításra szoruló dolog az, hogy mé nem teljesen sikerült a sorokat összehozni az adott hash-el, mivel ezek változását úgy tartom számon, hogy eltárolom melyik Hash melyik sortól érvényes, viszont itt az ASM visitorjai nem mindig számomra szükséges sorrendben futnak, és optimalizálni kell ezt, hogy a helyes sorokat tudjam hozzájuk rendelni.

8.2. Szeletelő algoritmus aktualizálása

Mivel nagy figyelmet fektettem az adatok kinyerésére, sajnos egy idő után a jelenlegi szeletelő algoritmus elavultá vált, mivel a jelenlegi csak az egy fileból álló projektekre lett létrehozva, illetve a változókat csak nevük alapján azonosította, ami közel sem elégséges. Ennek aktualizálása fog megtörténni leghamarabb a HashCode-k kijavítása után, hogy legyen egy, ajelenleg rendelkezésünkre álló adatokat feflhasználó programszeletelőnk.

8.3. Szálak kezelése

Másik hiányossága a projektnek még, hogy a többszálú programokat még nem kezeljük semmilyen formában. Ez egy sokkal nehezebb, mint az egyszálú programok szeletelése látható, hogy a jelenleg ismert programszeletelők szinte egyike sem kezeli ezeket. Ettől függetlenül tervezem ezen bővítést is.

Irodalomjegyzék

[1] Mark Weiser. “Program Slicing”. *IEEE Transactions on Software Engineering* SE-10.4 (1984), 352–357. old. DOI: 10.1109/TSE.1984.5010248.

[2] Bogdan Korel és Janusz Laski. “Dynamic program slicing”. *Information Processing Letters* 29.3 (1988), 155–163. old. DOI: 10.1016/0020-0190(88)90054-3.

[3] Hiralal Agrawal és Joseph R. Horgan. “Dynamic Program Slicing”. *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*. 1990, 246–256. old. DOI: 10.1145/93542.93576. URL: https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/slicing.pdf.

[4] Árpád Beszédes és tsai. “Dynamic Slicing Method for Maintenance of Large C Programs”. *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*. Lisbon, Portugal: IEEE Computer Society, 2001. márc., 105–113. old.

[5] Tibor Gyimóthy, Arpád Beszédes és Istán Forgács. “An efficient relevant slicing method for debugging”. *ACM SIGSOFT Software Engineering Notes* 24.6 (1999), 303–321. old.

[6] Clemens Hammacher és tsai. “Profiling Java Programs for Parallelism”. *Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE)*. Vancouver, BC, Canada, 2009. máj., 49–55. old. DOI: 10.1109/IWMSE.2009.5071383.

[7] Clemens Backes. *Backes/javaslicer: JavaSlicer is an open-source dynamic slicing tool developed at Saarland University*. Accessed: 2022-09-18. 2022. URL: https://github.com/backes/javaslicer.

[8] Khaled Ahmed, Mieszko Lis és Julia Rubin. “Slicer4J: A Dynamic Slicer for Java”. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, 1570–1574. ISBN: 9781450385626. DOI: 10.1145/3468264.3473123. URL: https://doi.org/10.1145/3468264.3473123.

[9] Khaled Ahmed, Mieszko Lis és Julia Rubin. *resess/Slicer4J: Slicer4J is an accurate, low-overhead dynamic slicer for Java programs.* Accessed: 2022-09-18. 2022. URL: https://github.com/resess/Slicer4J.

[10] Carlos Galindo, Sergio Pérez és Josep Silva. “Slicing Unconditional Jumps with Unnecessary Control Dependencies”. *Logic-Based Program Synthesis and Transformation: 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7–9, 2020, Proceedings*. Bologna, Italy: Springer-Verlag, 2020, 293–308. ISBN: 978-3-030-68445-7. DOI: 10.1007/978-3-030-68446-4_15. URL: https://doi.org/10.1007/978-3-030-68446-4_15.

[11] Carlos Galindo. *mistupv/JavaSlicer: A program slicer for Java, based on the system dependence graph (SDG).* Accessed: 2022-09-18. 2022. URL: https://github.com/mistupv/JavaSlicer.

[12] Venkatesh Prasad Ranganath és John Hatcliff. “Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs”. *Compiler Construction*. Szerk. Evelyn Duesterwald. Springer Berlin Heidelberg, 2004, 39–56. old. ISBN: 978-3-540-24723-4. DOI: 10.1007/978-3-540-24723-4_4.

[13] Venkatesh Prasad Ranganath és John Hatcliff. *Indus: A program analysis and slicing library for concurrent Java.* Accessed: 2022-09-18. 2022. URL: https://github.com/rvprasad/Indus_archive.

Forráskódjegyzék