

TDK Dolgozat



Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Parse transform-ok vizsgálata

Szabó Richárd

Programtervező Informatikus MSc, II. évfolyam

Témavezető: **Tóth Melinda**

egyetemi tanársegéd

Budapest, 2013

Tartalomjegyzék

1. Parse transform-ok és vizsgálatuk	3
1.1. Általános ismertető	3
1.2. Létező alkalmazások	4
2. Motiváció	9
2.1. RefactorErl	9
2.2. Feladat általánosítása	10
3. Megoldás ismertetése	12
3.1. Megoldási lehetőségek	12
3.2. Abstract Format ismertetése	13
3.3. Illeszkedési szabályok	17
3.4. Algoritmus	19
4. Esettanulmányok	23
5. Kapcsolódó munkák	27
Irodalomjegyzék	31

Kivonat

A parse transform segítségével az Erlang nyelvben fordításidőben tetszőleges átalakításokat végezhetünk a kód szintaxisfáján. Ezzel egy szintaktikusan helyes kódhoz tetszőleges, akár az eredetitől eltérő szemantikát rendelhetünk. A RefactorErl Erlang programok statikus elemzésének és transzformálásának eszköze. A programok megértésének támogatásához fontos, hogy helyesen tudjon elemezni tetszőleges Erlang kódot, így parse transform-ot tartalmazó kódot is. Az alkalmazás eddig nem volt erre felkészítve, ezért dolgozatomban parse transformok elemzésének lehetőségeivel foglalkozom.

Bevezetés

A parse tranform [12] az Erlang [16] fordítójának egy – hivatalosan nem támogatott – lehetősége, amellyel a fordítás folyamatában, a forrás előfeldolgozása, szintaktikus ellenőrzése után átalakításokat végezhetünk a program szintaxisfáján. Ezzel az eszközzel bármely szintaktikusan helyes Erlang kódhoz tetszőleges új szemantikát rendelhetünk.

Lehetséges alkalmazása ennek az eszköznek különféle szintaktikus cukorkák implementálása a nyelvre. Egy létező felhasználás például az Erlang nyelv bonyolult rekord szintaxisa helyett lehetővé teszi az egyszerűsített, imperatív nyelveknél megszokott jelölést.

A RefactorErl [14] programok statikus elemzésének és transzformálásának eszköze, fontos célja, hogy programok megértését segítse. Az alkalmazás a forráskódot egy absztrakt szintaxisfára épülő szemantikus gráfban tárolja. Ezen a gráfon szemantikus lekérdezéseket futtathatunk, vizsgálhatjuk az elemzett kód tulajdonságait.

A rendszer eddig nem volt felkészítve parse transform-ok vizsgálatára, nem vette figyelembe annak esetleges változtatásait. Így lehetséges, hogy az az új szemantika, amit a transzformáció vezetett be, elveszett a programozó szeme elől, nem tudta a kód érintett részeinek valódi jelentését megérteni. Ezért szükséges, hogy a program elemzése során kezeljük a parse transform-okat is, figyelembe vegyük azok változtatásait, így tároljuk el és jelenítsük meg a kódot az adatbázisban.

A probléma egy kézenfekvő megoldása lenne, ha egyszerűen lefuttatnánk

a vizsgált kódon a parse transform-ot, majd az átalakított kódot tárolnánk el. Egy probléma azonban, hogy a RefactorErl nem a fordító és a parse transform formátumát használja, hanem saját reprezentációja van a szintaxisfára. Így valahogy a parse transform szintaxisfán végrehajtott műveleteit még át kellene fordítani a saját szintaxisfánk műveleteire. Ez sem lenne azonban elegendő, hiszen a programozó számára az eredeti forráskódon kell megmutatnunk a kódra vonatkozó szemantikus információkat, nem pedig a szintaxisfán. Szükséges tehát, hogy a parse transform hatását meg is vizsgáljuk valamilyen módon, hogy össze tudjuk kapcsolni az eredeti kódot a transzformált kóddal.

Dolgozatomban a parse transform-ok elemzésének lehetőségeit fogom vizsgálni. Ezek közül az egyik legkézenfekvőbb a következő. Meg kell vizsgálnunk a transzformáció előtti, leellenőrzött szintaxisfa és az átalakított szintaxisfa közötti különbségeket. A különbségek alapján különböző heurisztikákkal, illesztésekkel megpróbáljuk megadni a kapcsolatot az eredeti és az átalakított fa csúcsai között. Ezzel később meg tudjuk mutatni a felhasználónak a két állapot közötti kapcsolatokat, segítjük megérteni a transzformáció hatását, a kód tényleges szemantikáját. A választott módszer előnye, hogy nem szükséges a transzformációs kódot értelmeznünk, anélkül próbáljuk kitalálni annak a kódra való hatását, hogy a szemantikáját vizsgálunk kéne.

1. fejezet

Parse transform-ok és vizsgálatuk

1.1. Általános ismertető

A parse transform egy olyan lehetőség az Erlang nyelvben, amivel a fordítás során a program forráskódján tetszőleges átalakításokat tudunk végrehajtani. Miután a fordító elvégzi az előfeldolgozást, majd a kód szintaktikus ellenőrzését, a szintaxisfát reprezentáló listát megkapva egy parse transform függvény segítségével átalakíthatjuk a szintaxisfát. Ehhez szükséges egy parse transform modul, aminek van egy `parse_transform` nevű függvénye. Ebben egy szintaxisfához egy másik szintaxisfát rendelhetünk. A szintaxisfa reprezentációja az úgynevezett Abstract Format [15]. Fontos kiemelni, hogy az átalakításra vonatkozóan semmilyen megkötés nincs, a programozó egy teljesen más, az eredetitől eltérő értelmű kódot is rendelhet hozzá. Az is lehetséges, hogy egy eredetileg szemantikusan hibás program így válik helyessé. Az Erlang oldalán példaként megtalálható egy identitás transzformáció [12].

A transzformáló függvény alkalmazásának két módja lehetséges. Egyik, hogy az átalakítani kívánt kódban `-compile` attribútumban adjuk meg a parse transform modult: `-compile({parse_transform,PtModule})`. Másik lehetőség, hogy a fordítás során hasonló módon megadjuk a paraméterek listájában a fenti két paramétert egy tuple-ben:

```
c(File, [{parse_transform, PtModule}])).
```

Szükséges, hogy a parse transform modulja ekkor már le legyen fordítva.

A parse transform tehát gyakorlatilag egy korlátlan lehetőséget ad arra, hogy tetszőleges szintaktikusan helyes Erlang kódnak egy tetszőleges szemantikát adjunk, tetszőlegesen átalakítsuk a szintaxisfát. Erre a következőkben adunk néhány lehetséges, létező alkalmazást.

1.2. Létező alkalmazások

Recless [17]. A recless egy olyan parse transform, ami az Erlang bonyolult rekordszintaxisa helyett lehetővé teszi, hogy a más pl. imperatív nyelveken megszokott módon, jóval egyszerűbben írjuk le a rekordkifejezéseket, módosítsuk azokat. Mutatok erre két példát, amihez először megadom a felhasznált rekordok definícióját.

```
-record(address, {street, city, country = "USA"}).  
-record(person, {name, address = #address{}}).  
-record(project, {name, owner = #person{}}).
```

A két kifejezésben az első a hagyományos jelölést használják, az utána következők pedig az új, egyszerűsített formát.

```
City = ((Project#project.owner)#owner.address)#address.city,  
City = Project.owner.address.city,
```

```
NewProject =  
    Project#project{owner =  
        (Project#project.person)#owner{address =  
            ((Project#project.person)#owner.address)#address{city =  
                'Boston'}}}.  
NewProject = Project.owner.address.city = 'Boston'.
```

Ahogy láthatjuk nem kell minden újabb szintél, újabb rekortípusnál a típus nevét kiiírnunk a kifejezésben, azt a program kikövetkezteti. A második példa azt mutatja, hogy egy mező értékének – egy új változóban való – módosításához nem szükséges többszörösen egymásba ágyazott kifejezéseket írunk, minden mezőt közvetlen hivatkozással elérünk.

A recless tehát a háttérben az egyszerűbb – szintaktikusan helyes, de szemantikusan hibás – kifejezést egy szemantikusan helyes, az eredeti jelölésnek megfelelő kifejezéssé alakítja át. Pontosabban a szintaxisfa egyenlő lesz a bonyolultabb kifejezésnek megfelelő szintaxisfával.

XML. Egy parse transform segítségével egyszerű szöveggént, inline módon írhatunk XML literálokat[18], amiket a modul feldolgoz, felépíti az XML szerkezetének megfelelő rekordot. Nézzünk egy egyszerű kódot, amiben használunk ilyen XML literált.

```
Doc = '<greeting>Hello!</greeting>',  
io:format("~p~n", [Doc]).
```

Erre a program a parse transform után a következő eredményt adja:

```
{xmlElement,greeting,  
  greeting,  
  [],  
  {xmlNamespace,[],[]},  
  [],  
  1,  
  [],  
  [{xmlText,[{greeting,1}],1,[],"Hello!",text}],  
  [],  
  "/tmp",  
  undeclared}
```

Lager [19]. A Lager egy naplózást segítő keretrendszer Erlang-hoz. Ez a parse transform segítségével olyan naplózást valósít meg, ami mind a kódban, a naplózás írását, emellett az adminisztrátor számára a visszaolvasást is kényelmesebbé, olvashatóbbá teszi. Alapértelmezésben a könnyebb áttekinthetéshez elrejt a nem minden esetben szükséges információkat, a naplózáshoz használt bonyolult tuple-öket. A megoldásban a parse transform azért jelent nagy előnyt, mert így a logolás közben egyben tesz elérhetővé sok hasznos információt, mint az aktuális modul, függvény neve, sorszám információk, folyamat azonosító. A naplózáshoz elkülönít többféle üzenet típust: debug, info, notice, warning, error, critical, alert, emergency. Az egyszerű használatot és a könnyen olvasható eredményt mutatja a következő példa.

```
lager:error("oh no!")
lager:warning("~s, ~s and ~s, oh!", [lions, tigers, bears])
```

```
2013-05-19 18:02:02 [error] @test2:start:8 oh no!
2013-05-19 18:02:02 [warning] @test2:start:9 lions,
                        tigers and bears, oh!
```

QLC [20]. A mnesia adatbázisrendszerhez használt lekérdezőnyelv a QLC (*Query List Comprehension*), aminek a megvalósításában szintén parse transform-ot használtak. Ennek segítségével egyszerű lista-kifejezésekkel írhatjuk le a lekérdezéseinket. Egy egyszerű példán megmutatom a lekérdezések megadásának két lehetséges módját. Példában használt – alkalmazottak tárolását szolgáló – emp nevű tábla definíciója a következő:

```
-record(emp, {empno,      % az alkalmazott azonosítója
              surname,    % az alkalmazott vezetékeve
              givenname,  % az alkalmazott keresztnéve
              sex,        % az alkalmazott neme
              dept,       % osztály: {dev,sales,prod,adm}
              empyear}). % felvétel éve
```

Tegyük fel, hogy meg akarjuk kapni az összes nőnemű alkalmazott nevét. Először nézzük meg a lekérdezést a `mnesia` beépített függvényeivel.

```
mnesia:select(employee,  
               [{#employee{sex = female,  
                  name = '$1', _ = '_'}, [], ['$1']}]
```

Ugyanezt az eredmény `qlc`-kifejezéssel a következőképp írhatjuk le.

```
Q = qlc:q([E#employee.name  
          || E <- mnesia:table(employee),  
          E#employee.sex == female]),  
qlc:e(Q),
```

Tehát itt is pontosan azt látjuk, amit kiemeltünk a `parse transform`-ok általános jellemzésénél, hogy egy a nyelvben megszokott, ismert, kényelmes szintaxist más szemantikára használunk, ezzel sokkal kényelmesebb, tömörebb írásmódot biztosítva. Ez rendkívül fontos eszköz a `mnesia` használatához, enélkül ugyanis meglehetősen kényelmetlen módon tudnánk az adatokat lekérdezni, módosítani.

Ms_transform [21]. Az `ms_transform` szintén az adatbázisok kezelésével kapcsolatos modul, ami `parse transform`-mal alakít függvény-kifejezéseket – az előző `mnesia:select` példában is használt – mintaillesztési kifejezésekre (*match specification*). Ez szintén egy kényelmi funkciót jelent, amivel például sokkal rövidebben, tömörebben írhatunk le egy lekérdezést egy `ets` táblán. Tekintsük a következő példát.

Adott az előző példában már ismertetésre került `emp` tábla.

Például, ha ezen meg akarjuk kapni a `sales` osztályon dolgozók azonosítóját, az `ms_transform` nélkül a következőképp kellene leírunk:

```
ets:match(emp_tab, {'_', '$1', '_', '_', sales, '_'})
```

Ezzel szemben az `ms_transform`-mal sokkal olvashatóbban, kényelmesebben írhatjuk le ezt a lekérdezést:

```
ets:foldr(fun(#emp{empno = E, dept = sales},Acc) -> [E | Acc];
          (_,Acc) -> Acc
        end,
        [],
        emp_tab).
```

Legegyszerűbb példa. Egy egyszerű példán megmutatom, hogy kell megvalósítani egy parse transform-ot. Ebben nem végzünk tényleges átalakítást a szintaxisfán, csak egyszerűen kiírjuk a tartalmát.

```
-export([parse_transform/2]).
```

```
parse_transform(Forms, _Options) ->
  io:fwrite("Forms = ~p~n", [Forms]), Forms.
```

Egy `parse_transform` nevű két paraméteres függvényt kell megvalósítani és exportálni, ami megkapja a form-okat, valamint a fordítási paraméterek listáját. A fordítási paramétereket parse transform-okban ritkán használjuk, de bizonyos esetekben szükség lehet ezekre. Például feltételes fordítás esetén a paraméterekből a makrókra szükség lehet, hogy a parse transform-ban is egy feltételes átalakítást valósítsunk meg.

2. fejezet

Motiváció

2.1. RefactorErl

A RefactorErl kódok refaktorálását és statikus elemzését, programok megértését támogató eszköz. Célja, hogy ipari méretű kódok átalakítását kézi megoldások helyét automatizált mechanizmusokkal, refaktorálások különféle típusaival támogassa. Fontos felhasználási területe emellett a statikus elemzés. Az alkalmazás egy kód elemzésekor felépít egy szemantikus gráfot, amit egy adatbázisban eltárol. Az elemzett kódokról szemantikus lekérdezésekkel tudhatunk meg különféle információkat. A modulok közötti kapcsolatokat például függőségi gráfokkal jeleníti meg.

A parse transform-ok esetében láttuk, hogy az adott szintaxisnak a megszokottól eltérő, más szemantikája lehet. A szemantika megértéséhez ezért elengedhetetlen, hogy értelmezzük a megváltozott kódot, megvizsgáljuk a transzformáció hatását, megértsük azt a kódot, ami valójában le fog fordulni. Ezek után tudjuk megoldani, hogy a fejlesztőnek is a tényleges szemantikával mutassuk meg a vizsgált kódot.

Jelenleg a RefactorErl nem kezeli a parse transform-okat, nem vizsgálja azok hatását, így lehetséges, hogy egy szemantikus lekérdezés helytelen választ ad a programozónak, félrevezeti őt a kód megértésében. A fenti példák

közül a `recless` egyszerűsített rekord-kifejezéseit például a `RefactorErl` nem is tudná elemezni, mert az eszközben definiált nyelvtan nem fogadná el azokat.

Ha egy parse transform-ot használó kódot sikerül is elemeznünk, lehetséges, hogy helytelen szemantikus információt ad a felhasználó számára, mivel nem az átalakított kódot elemzi. Ilyen fordulhat elő például, ha a parse transform egyszerűen lecseréli egy modul minden `f` függvényhívását `g` függvényhívásra. Ekkor a függvényhívási gráfban hibás eredményt jelenítünk meg: `f` jelenik meg ott is, ahol `g`-nek kellene lennie. Ezenkívül egy refaktorálásnál is hibás eredményhez vezethet az, hogy nem helyesen elemeztük a kódot. Például a `g` függvény átnevezése figyelmen kívül hagyná az `f`-ből `g`-re átalakított hívásokat. Ha pedig az `f` függvényt neveznénk át, utána a parse transform nem is alakítaná át a függvényhívásokat `g`-re.

Célunk, hogy az eszközt kiegészítsük egy olyan funkcióval, ami kiküszöböli ezeket a hiányosságokat, valamint megmutatja az eredeti kód mellett a transzformáció hatására történt, megváltozott szemantikát is.

Ehhez dolgozatomban azt keresem, hogy lehet megvizsgálni az átalakítás hatását, hogy lehet megadni a kapcsolatot az eredeti és az átalakított kód között. Később ennek segítségével megvalósítható lesz az eszközben a fent leírt funkció.

2.2. Feladat általánosítása

A következő fejezetekben ismertetett algoritmus megoldást ad a fenti, `RefactorErl`-ből kiinduló problémára, azonban a megoldás valójában független attól, jóval általánosabb. Az általam választott megoldásban ugyanis a program eredeti és átalakított szintaxisfáját hasonlítom össze, ezzel pedig a problémát egy általánosabb térbe helyezzük. Így ugyanis általános értelemben kell megvizsgálunk két program, vagy egy program két különböző változata közti különbséget, kapcsolatot. Két szöveg összehasonlítása a ha-

gyományos diff algoritmussal megoldható, mi azonban szöveges szintről egy magasabb absztrakciós szintre lépünk: szintaxisfákat hasonlítunk össze. Így az összehasonlításnál olyan szemantikus összefüggéseket is felismerhetünk, amit egy hagyományos szöveges vizsgálatnál nem. Felismerhetjük például, hogy a transzformáció egy beszúrási művelete nem egy egyszerű beszúrást volt, hanem egy meglevő csúcs másolása. A feladatot tehát tekinthetjük két szintaxisfa összehasonlításának. Egy másik megközelítésben pedig a problémát a verziókezelés irányába is elterelhetjük. Az eredeti és az átalakított kódot két verzióknak tekinthetjük és speciális módon a különbségeket a szöveges hasonlítás helyett a szintaxisfa alapján számítjuk ki.

A probléma nehézségéhez az is hozzátartozik, hogy – mint említettem – az átalakításra vonatkozóan nincs semmilyen megkötés, így szélsőséges esetben az is elképzelhető, hogy nincs semmilyen tényleges kapcsolat a két kód között. Ez az eset áll fenn például egy konstans transzformációnál. Ezeket figyelembe véve megszorításokat teszünk a lehetséges kapcsolatokra, olyan összefüggéseket próbálunk felismerni, amelyek értelmes – és létező – parse transform-okban előfordulhatnak, előfordulnak.

3. fejezet

Megoldás ismertetése

3.1. Megoldási lehetőségek

A probléma megoldására több módszer is lehetséges. Egyik megközelítésben a transzformációs kódot magát vizsgáljuk. Megpróbálhatnánk például adatfolyam-elemzéssel kitalálni, hogy az szintaxisfában melyik csúcsból mi lesz. Ez azonban nem lehetséges, hiszen a bemenő adatok több olyan csatornán is keresztülmennek, amit mi nem tudunk megvizsgálni. Mire a forrás szintaxisfáját megkapjuk az már végigment előfeldolgozón, a lexikális elemzőn és a szintaktikus elemzőn is. A szemantika ilyen módon való értelmezése tehát statikus elemzéssel nem lehetséges.

Másik lehetőség, hogy a parse transform kódját felannotáljuk, az átalakítás műveleteit átfordítjuk a RefactorErl gráfjának műveleteire, abba való beszúrásokra, törlésekre. Az így kapott annotált műveletsorozatot az eredeti kód gráfján végrehajtva pedig megkapnánk az átalakított kódnak megfelelő, azzal ekvivalens gráfot. Ezenkívül az annotációk segítségével a csúcsok közti kapcsolatokat is felépíthetjük.

Ezekkel szemben mi azonban egy olyan utat választottunk, amiben nem kell értelmezni a parse transform kódját, nem kell foglalkozunk a kód szemantikájával. Ehelyett megvizsgáljuk a két állapot közötti különbségeket,

a változások alapján pedig megpróbáljuk megadni a kapcsolatot. Ez a reverse engineering módszeréhez hasonlít, ugyanis a program működését nem annak elemzésével próbáljuk megérteni, hanem a hatásából próbálunk visszakövetkeztetni az értelmére. Ez láthatóan egyszerűbb feladat annál, hogy a kódot magát kellene értelmeznünk. Ezenkívül az esetek nagy többségében így kevesebb elemzést kell elvégeznünk, nem kell a transzformáció teljes kódját értelmeznünk, hanem csak azokkal a részekkel foglalkozunk, amik az aktuális kódban lefutottak, amiknek a hatása ebben megjelent. Ez a módszer abból a szempontból lehet költségesebb, hogy ugyanazt az átalakítást többször is újra fel kell ismernünk. A transzformációs kód elemzésénél elegendő lenne mindent egyszer értelmezni. Mi a megoldásban mégis az előbbi módszert választottunk, hiszen ez jóval rugalmasabb, dinamikusabb.

3.2. Abstract Format ismertetése

Az algoritmus megadása előtt röviden ismertetem az Abstract Formatot.

Az Abstract Format az Erlang nyelv szintaxisfájának egy Erlang termeken alapuló reprezentációja. Ez a fordító által használt reprezentáció is, valamint ez szolgál a parse transform-ok be-, illetve kimeneti formátumként is. Az ismertetéshez a formátum hivatalos, az Erlang oldalán található dokumentáció kifejezéseit, fogalmait használok fel [15]. A magyarázatban használt `Rep` függvény adja meg a leképezést a nyelvi elemek és az Abstract Format között.

Modul deklarációk és formok. Egy Erlang modul formok sorozatából áll. Egy form lehet egy függvény definíció, vagy egy attribútum. Egy `Mod` modul nevét leíró attribútum például a következőképp van reprezentálva: `{attribute,LINE,module,Mod}`. A `LINE` minden példában a forrás aktuális sorát jelölő változó. A nagybetűvel kezdődő nevek szintén változókat jelölnek – amit épp reprezentálni akarunk –, a kisbetűs elemek pedig atomok. A többi

attribútum hasonlóan egy `attribute` előtagú tuple-lel van ábrázolva. Egy `Name` nevű, `Name Fc_1 ; ... ; Name Fc_k` klózokból álló függvény reprezentációja: $\{\text{function}, \text{LINE}, \text{Name}, \text{Arity}, [\text{Rep}(\text{Fc}_1), \dots, \text{Rep}(\text{Fc}_k)]\}$, ahol a $\text{Rep}(\text{Fc}_1)$ és $\text{Rep}(\text{Fc}_k)$ jelölik az egyes klózok reprezentációját, az `Arity` pedig a függvény aritását jelenti.

Atomi literálok. Az atomi literálok ábrázolásában egyszerűen megadjuk annak típusát, a tartalmazott értéket, valamint a megfelelő sorszámot. Például egy `I` egész szám ábrázolása: $\{\text{integer}, \text{I}, \text{Line}\}$. A szövegeknél a tartalmat a karakterek listájával írjuk le: $\{\text{string}, \text{LINE}, [\text{C}_1, \dots, \text{C}_k]\}$, ahol C_1 , C_k az egyes karaktereket jelölik.

Minták. Függvények paramétereiben szereplő minták egy `Ps` sorozatát az egyes elemek reprezentációjának sorozatával adjuk meg:

$\text{Rep}(\text{Ps}) = [\text{Rep}(\text{P}_1), \dots, \text{Rep}(\text{P}_k)]$.

Az egyes `P` minták közül néhányra megadom a leképezést:

- egy `L` atomi literálra $\text{Rep}(\text{P}) = \text{Rep}(\text{L})$
- $\text{P}_1 = \text{P}_2$ összetett mintára:

$$\text{Rep}(\text{P}) = \{\text{match}, \text{LINE}, \text{Rep}(\text{P}_1), \text{Rep}(\text{P}_2)\}$$
- ha a minta egy `v` változó: $\text{Rep}(\text{P}) = \{\text{var}, \text{LINE}, \text{A}\}$, ahol `A` a változó neve
- $[\text{P}_h \mid \text{P}_t]$ listára:

$$\text{Rep}(\text{P}) = \{\text{cons}, \text{LINE}, \text{Rep}(\text{P}_h), \text{Rep}(\text{P}_t)\}$$
- $\{\text{P}_1, \dots, \text{P}_k\}$ tuple-re:

$$\text{Rep}(\text{P}) = \{\text{tuple}, \text{LINE}, [\text{Rep}(\text{P}_1), \dots, \text{Rep}(\text{P}_k)]\}$$

Kifejezések. Kifejezések egy E sorozatát szintén az egyes kifejezések képének sorozatával reprezentáljuk: $\text{Rep}(B) = [\text{Rep}(E_1), \dots, \text{Rep}(E_k)]$.

Néhány jelentősebb kifejezéstípus (E) leképezése a következőképp alakul:

- egy $E_0(E_1, \dots, E_k)$ függvényhívásra:
 $\text{Rep}(E) = \{\text{call}, \text{LINE}, \text{Rep}(E_0), [\text{Rep}(E_1), \dots, \text{Rep}(E_k)]\}$
- egy $E_m:E_0(E_1, \dots, E_k)$ függvényhívásra:
 $\text{Rep}(E) = \{\text{call}, \text{LINE}, \{\text{remote}, \text{LINE}, \text{Rep}(E_m), \text{Rep}(E_0)\}, [\text{Rep}(E_1), \dots, \text{Rep}(E_k)]\}$
- ha E egy $\text{if } Ic_1 ; \dots ; Ic_k \text{ end}$ alakú if-kifejezés, ahol Ic_i egy if klóz, akkor
 $\text{Rep}(E) = \{\text{'if'}, \text{LINE}, [\text{Rep}(Ic_1), \dots, \text{Rep}(Ic_k)]\}$
- ha E egy $E_0 \text{ of } Cc_1 ; \dots ; Cc_k \text{ end}$ alakú case-kifejezés, ahol C_i -k case klózek:
 $\text{Rep}(E) = \{\text{'case'}, \text{LINE}, \text{Rep}(E_0), [\text{Rep}(Cc_1), \dots, \text{Rep}(Cc_k)]\}$
- egy $[E_0 \mid W_1, \dots, W_k]$ listakifejezésre, ahol minden W_i egy szűrő, vagy listagenerátor:
 $\text{Rep}(E) = \{\text{lc}, \text{LINE}, \text{Rep}(E_0), [\text{Rep}(W_1), \dots, \text{Rep}(W_k)]\}$, ahol a W generátorok, illetve szűrők reprezentációja:
 - $P \leftarrow E$ generátorra $\text{Rep}(W) = \{\text{generate}, \text{LINE}, \text{Rep}(P), \text{Rep}(E)\}$
 - ha W egy E szűrőkifejezés, akkor $\text{Rep}(W) = \text{Rep}(E)$

Klózok. Az egyes klózek lehetnek függvény klózek, if klózek, case klózek, valamint catch klózek.

- $(Ps) \rightarrow B$ alakú C függvényklóz leképezése:
 $\text{Rep}(C) = \{\text{clause}, \text{LINE}, \text{Rep}(Ps), [], \text{Rep}(B)\}$, ahol Ps minták sorozata, B pedig a törzs

- ha C egy $G_s \rightarrow B$ alakú if klóz, ahol G_s guard-ok sorozata, B pedig a törzs
- $P \rightarrow B$ alakú case klózra:

$$\text{Rep}(C) = \{\text{clause}, \text{LINE}, [\text{Rep}(P)], [], \text{Rep}(B)\}$$
- $P \rightarrow B$ alakú catch klózra:

$$\text{Rep}(C) = \{\text{clause}, \text{LINE}, [\text{Rep}(\{\text{throw}, P, _ \})], [], \text{Rep}(B)\}$$

Guard-ok. Guard-ok egy G_s sorozatát az eddigiekhez hasonlóan az egyes elemek képének sorozatával reprezentáljuk:

$\text{Rep}(G_s) = [\text{Rep}(G_1), \dots, \text{Rep}(G_k)]$. Egy guard őrfeltételek egy nem üres sorozatát jelenti. Az egyes őrfeltételeket (G_t) a következőképp ábrázoljuk:

- egy L atomi literálra $\text{Rep}(G_t) = \text{Rep}(L)$
- egy v változóra: $\text{Rep}(G_t) = \{\text{var}, \text{LINE}, A\}$, ahol A a változó neve
- egy $\{G_{t_1}, \dots, G_{t_k}\}$ tuple-re:

$$\text{Rep}(G_t) = \{\text{tuple}, \text{LINE}, [\text{Rep}(G_{t_1}), \dots, \text{Rep}(G_{t_k})]\}$$
- $[G_h \mid G_t]$ listára:

$$\text{Rep}(G_t) = \{\text{cons}, \text{LINE}, \text{Rep}(G_h), \text{Rep}(G_t)\}$$
- ha a feltétel $G_{t_1} \text{ op } G_{t_2}$ alakú, ahol Op egy bináris operátor:

$$\text{Rep}(G_t) = \{\text{op}, \text{LINE}, \text{Op}, \text{Rep}(G_{t_1}), \text{Rep}(G_{t_2})\}$$

Példa. Az alábbiakban megadom egy egyszerű modul elemzett kódját.

Az eredeti szöveges állapot:

```
-module(ex1).
-export([f/1]).

f(X) -> ok.
```

A szintaxisfa:

```
{attribute,1,file,{"home/ricchie-linux/ex1.erl",1}},  
{attribute,1,module,ex1},  
{attribute,2,export,[{f,1}]},  
{function,4,f,1,[{clause,4,[{var,4,'X'}],[],  
    [{atom,4,ok}]}]},  
{eof,5}
```

3.3. Illeszkedési szabályok

A dolgozatomban megadtam egy az adott feladatra, Erlang nyelv szintaxisfájára [15] specializált algoritmust, amely megadja két szintaxisfa közti különbséget, pontosabban egy leképezést ad a kettő közt. Az algoritmus párhuzamosan haladva egyenként összehasonlítja az eredeti és az átalakított szintaxisfa csúcsait és megpróbálja azokat egymásra illeszteni. Az összehasonlítás során a fa elemeit rekurzívan vizsgáljuk a gyökér szintjéről az alsóbb szintekre, a levelek felé haladva. Egy lépésben két csúcs összehasonlításánál három illeszkedési szabályt definiálunk.

Ekvivalens típusok. Az első szabály illesztésének előfeltétele a két csúcs típusának megegyezése. Ha ez teljesül, elkezdjük rekurzívan lebontani, az adott típusnak megfelelően megvizsgáljuk a részfák egyezőségét. Az egyes csúcsok összehasonlításánál azonban nem követelünk meg minden esetben teljes egyenlőséget, az illeszkedés vizsgálatához egy speciális hasonlósági metrikát alkalmazunk. Infix kifejezéseknél például megengedhetjük, hogy az operátorok különbözőek legyenek.

Ezzel a szabállyal ismerhetjük fel például változók, függvények átnevezését, literálok értékének megváltoztatását, illetve a két csúcs teljes egyezőségét is.

Szerkezeti hasonlóság. Különböző típusú csúcsok esetén először azt nézzük meg, hogy szerkezetileg ekvivalensek, vagy legalább hasonlóak lehetnek-e. Először kinyerjük mindkét csúcsból a részfákat, amit paramétereknek tekinthetünk. Ezek például listánál, rendezett n-eseknél a tartalmazott elemek, vagy egy függvényhívásnál a paraméterek. Ezután megvizsgáljuk, hogy az első csúcs paramétereinek halmazát tartalmazza-e a második csúcs paramétereinek halmaza. Egy-egy elem összehasonlításánál itt nem egyszerű egyenlőségvizsgálatot végzünk, hanem rekurzívan összehasonlítjuk őket az algoritmussal.

Ezzel a szabállyal a következő csúcsok illeszkednek:

- $[X, Y] - \{Y, X, \text{ok}\}$
- $[X, Y] - f(X, Y)$

Ahogy a két példában is láthatjuk, itt tehát a csúcs tartalma lényegileg nem változott meg, megvannak az eredeti elemek. Szemléletesen úgy fogalmazhatunk, hogy csak a „csomagolása”, a külső szerkezete lett átalakítva – például listáról tuple-re, vagy listáról egy függvényhívásra –, még nem is került be egy újabb szint a fában. Ezenkívül azt is megengedjük, hogy új elemekkel bővüljön a részfa.

Szerkezeti tartalmazás. A harmadik szabályban rekurzív lebontást alkalmazunk: megvizsgáljuk, hogy egy szintet lebontva fennállhat-e az első típusú illeszkedés, vagy a szerkezeti hasonlóság. Ezt a lebontást mindkét irányban megpróbáljuk, tehát lejjebb megyünk egy szintet az első paraméterben és így hasonlítjuk össze a másik oldal aktuális szintjével, illetve a másik oldal részfáit is összevetjük az első aktuális szintű részfával. Ebben a lépésben tehát csak leveszünk egy szintet a csúcs szerkezetéből, lemegyünk a fában egyvel alsóbb szintre, és itt újra sorban megpróbáljuk illeszteni mindegyik szabályt. Így olyan transzformációkat ismerhetünk fel, amelyek új szintet, vagy szinteket vezetett be, illetve valamilyen értelemben csúcsokat vont össze. Példák a harmadik szabály illeszkedésére:

- $[X, Y] \rightarrow f([X, Y])$
- $\{X, Y\} \rightarrow [\{X, Y\}, ok]$

Az utóbbi két szabály ötlete a fordítóprogramokban ismert ablak-optimalizációból [11] származik. Ennek a lényege, hogy egy kisebb mintát, egy ablakot karakteresen megpróbálunk illeszteni a kódban, keressük annak előfordulásait. Erre leginkább a második szabályunk hasonlít, azonban a probléma vizsgálata közben arra jutottunk, hogy bővítéssel több és általánosabb kapcsolatot is felismerhetünk. Ahelyett, hogy egy mintával való teljes egyezést keresnénk, megengedjük, hogy egy bővebb kapcsolat, szerkezeti hasonlóság álljon fenn. Így az előző két szabály kombinációjával, egymás utáni alkalmazásával újabb illeszkedésekre is eljuthatunk:

- $[X, Y] \rightarrow f(\{X, Y\})$
- $\{X, Y, ok\} \rightarrow f([X, Y])$

3.4. Algoritmus

Az algoritmus minden lépésben megpróbálja illeszteni a fent ismertetett szabályokat. Ha egy párosnál a három lehetőség közül egyik sem teljesül, az éppen vizsgált páros második elemét egy lehetséges beszúrásnak tekinti, hiszen az illesztés során végig a régi, eredeti állapot csúcsainak próbálunk egy ugyanolyan, vagy egy átalakított megfelelőt találni. Ilyen esetben tehát nem tudtuk eldönteni, hogy párban áll-e a két csúcs, vagy a jobb oldal csúcsa egy új beszúrt elem. Ezért egy második körben egy teljes vizsgálat elvégzésével újra megpróbálja illeszteni. Ugyanis lehet, hogy ez egy duplikátum, vagy egy átmozgatott csúcs, aminek az eredeti fában egy másik helyen találhatjuk a megfelelőjét. Ezután az algoritmus az eredeti kódban még nem lép tovább, hanem az éppen vizsgált elemet összehasonlítja az új kód következő elemével

is. Tehát a bal oldalon maradunk az aktuális elemnél, a jobb oldalon pedig tovább lépünk.

Ha a rekurzív vizsgálat során az eredeti kód listájából elfogynak az elemek, a jobb oldal maradék csúcsát beszúrt elemeknek vesszük, fordított esetben a baloldali elemeket töröltnek.

Miután egyszer végimentünk ezen a vizsgálaton, végzünk egy újabb iterációt azokon az elemeken, amelyeknek nem sikerült biztosan illeszkedő párt találnunk. Ezzel tehát a fent leírtaknak megfelelően megtalálhatunk duplikátumokat, átmozgatott csúcsokat.

Az alábbiakban megadom az algoritmus egy egyszerűsített leírását. Ehhez pszeudokódot használok.

Az algoritmus első két paramétere a két összehasonlítani kívánt fa. Harmadik paramétere a rekurzív hívásoknál használt lista, ugyanis az eredmény kiszámításához akkumulatív technikát használunk.

- Első lépésben lekérdezzük az éppen vizsgált X és Y csúcsok típusát (2-3. sorok).
- Ezután megpróbáljuk az első szabályt illeszteni a két csúcsra (4. sor). Ennek feltétele, hogy a típusuk megegyezzen. Ha teljesül, megkezdjük a csúcsok rekurzív összehasonlítását, lebontását.
- Amennyiben nem teljesül illeszkedési feltétel, továbblépünk a következő szabályra, ami a szerkezeti hasonlóságot vizsgálja. Ennek illeszkedési feltételét jelzi a 6. sorban `struct_similar` logikai függvény. Ha teljesül a feltétel, betesszük az `{X,Y}` párt az eredménylistába és folytatjuk a vizsgálódást.
- Ha ez a szabály sem illeszkedett, megpróbáljuk a tartalmazási szabályt mindkét irányban (8. és 14. sorok) illeszteni. Ha teljesül valamelyik, egyenként összevetjük a tartalmazni vélt csúcsot a tartalmazó részfáival (9., 15. sor).

- Ha egyik irányú tartalmazás sem teljesül, a vizsgált párt megjelölve egy `insert_maybe` előtaggal az eredménybe tesszük, majd egy újabb iterációban megvizsgáljuk a második elemet, hiszen eddig a vizsgálat során ennek a csúcsnak nem találtunk párt az eredeti csúcsok közt.
- A definíció 22. sorban kezdődő ága azt az esetet jelenti, amikor a baloldalon végigértünk. A jobb oldal maradék elemeit ekkor mind beszúrtuk vesszük.
- Hasonló a helyzet a 24. sorban, csak ekkor az új elemek fogynak el, így törölt elemeknek tekintjük az eredeti fából megmaradt csúcsokat.
- Ha mindkét fa vizsgálatán végigértünk (26.), az eredményt az `Acc` listából kapjuk meg.

Szerkezeti hasonlóság (3.4. ábra). A `subset` függvény egyfajta részhalmoz vizsgálatot végez el a két lista között, azonban annál egy általánosabb műveletet jelent. Az egyes elemek összehasonlításánál ugyanis nem teljes egyezőséget követel meg, hanem rekurzívan meghívja a `diff_tree` függvényt. Ezzel jóval több kapcsolatot, jóval bonyolultabb átalakításokat is képesek leszünk felismerni.

Szerkezeti tartalmazás. A harmadik illeszkedési szabálynál a `contains` függvény csak egy előzetes vizsgálatot végez a típus alapján. Ha egy típus nem egyszerű, azaz van még rajta lebontható szint, akkor mindenképp összeveti a következő szint részeit az éppen vizsgált másik csúccsal.

A fent leírt algoritmus Erlang nyelvű implementációját elkészítettem. Ennek működését a következő fejezet esettanulmányain mutatom be. A megvalósítás még nyilvánosan nem elérhető, azonban az célunk, hogy integráljuk a `RefactorErl`-be a korábban leírt funkciók megvalósításához.

```

01 diff_tree([X|Xs],[Y|Ys],Acc):
02     T1=node_type(X)
03     T2=node_type(Y)
04     if T1==T2 then
05         Result=diff_tree(Xs,Ys,check(X,Y) ++ Acc)
06     elseif struct_similar(X,Y)
07         Result=diff_tree(Xs,Ys,{X,Y} ++ Acc)
08     elseif contains(X,Y) then
09         SubX=subtree(X)
10         Result=[]
11         for I in SubX do
12             Result+=diff_tree(I,Y,[])
13             Result+=diff_tree(Xs,Ys,Acc)
14     elseif contains(Y,X) then
15         SubY=subtree(Y)
16         Result=[]
17         for I in SubY do
18             Result+=diff_tree(X,I,[])
19             Result+=diff_tree(Xs,Ys,Acc)
20     else
21         Result=diff_tree([X|Xs],[Ys],
22                             {insert_maybe,{X,Y}}
23                             ++ Acc)
24 diff_tree([],Ys,Acc):
25     Result={insert,Ys} ++ Acc
26 diff_tree(Xs,[],Acc):
27     Result={delete,Xs} ++ Ac
28 diff_tree([],[],Acc):
29     Result=Acc

```

```

01 similar_struct(X,Y):
02     Xs=expr_arguments(X)
03     Ys=expr_arguments(Y)
04     Result=subset(Xs,Ys)

```

4. fejezet

Esettanulmányok

1. Példa (4.1. ábra). Az első példában a korábban ismertetett `recless` modult használok az egyszerűsített rekord szintaxisra. Ezt tartalmazza az `io:format` függvényhívásban a `Project.owner.name` kifejezés. A kód utáni két ábrában a kifejezés elemzett formáját mutatom meg – Abstract Format-ban – először a parse transform nélkül, majd pedig azzal átalakítva. Ezekben látjuk, hogy a transzformáció beszúrta a megfelelő mezők rekordípusát, tehát helyesen kikövetkeztette azokat. Ezt jelzik a `project` és a `person` atomok.

Az összehasonlító algoritmus helyesen felismeri a kapcsolódó részeket, megadja, hogy az eredeti rekordkifejezés a kibővített kifejezésre lett átalakítva.

2. Példa (4.2. ábra). Ebben a példában a `mnesia` lekérdezéseihez használt `ms_transform` működését próbálok elemezni az összehasonlító algoritmus-sal. Az előző példához hasonlóan itt is megmutatom a kód eredeti formáját, majd az elemzett kódot.

Az összehasonlítás során sajnos nem sikerül felismerni minden részletet, hiszen a transzformáció itt a teljes `ets:fun2ms` függvényhívást lecseréli egy megfelelő mintaillesztési kifejezésre.

```

recless
-record(address, {street, city, country = "USA"}).
-record(person, {name, address = #address{}}).
-record(project, {name, owner = #person{}}).

test() ->
    Project = #project{
        name = "MyProject",
        owner = #person{
            name = "Bob",
            street = "X",
            address = #address{city = "Miami" }},
        io:format(Project.owner.name).

```

```

szintaxisfa
{record_field,14,
  {record_field,14,{var,14,'Project'},
    {atom,14,owner}},
  {atom,14,name}}

```

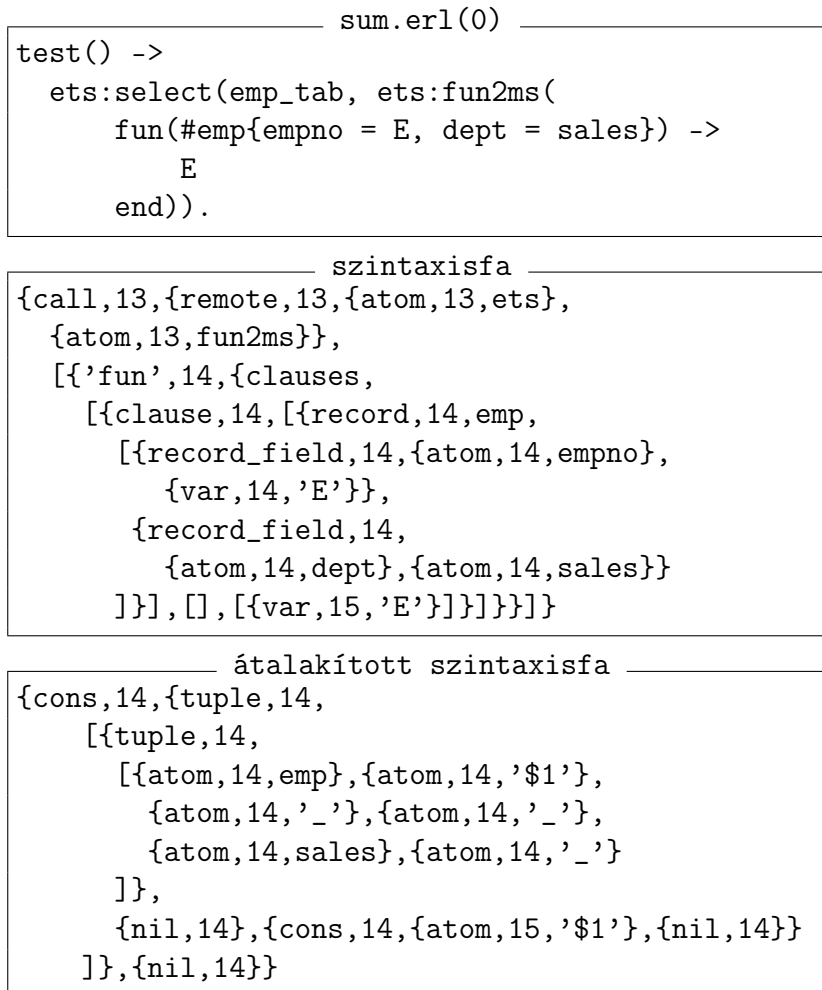
```

átalakított szintaxisfa
{record_field,14,
  {record_field,14,{var,14,'Project'},
    project,{atom,14,owner}},
  person,{atom,14,name}}

```

4.1. ábra. Recless – rekord szintaxis egyszerűsítése

Bár nem sikerült teljes találatot adni, a kifejezéseket befoglaló `ets:select` függvényhívást az eredeti és a transzformált kódban is megtalálja, illeszti egymáshoz, így a felsőbb szinten megkapjuk a kapcsolatot a két csúcs között.



4.2. ábra. mnesia lekérdezések átalakítása

3. Példa (4.3. ábra). A következő példában szereplő a modulban több változatást is végrehajtunk. A szemléltetés kedvéért itt az átalakított kódot is szöveges formában mutatom meg.

Az első változtatás, hogy átmozgattuk a `g` függvény definícióját az `f` függvény elé az első sorba. Ezt az átmozgatást az algoritmus felismeri, összekapcsolja a `g` függvényt az eredeti és az átalakított kódban.

A mozgatás mellett módosítottuk is a függvény törzsét, de ez sem okozott problémát az átalakítás felismerésében.

Két további módosítás az `l` függvény törlése, valamint a `h` függvény beszúrása. Ezeket szintén felismeri az algoritmus.

```

a.erl(0)
f(X) ->
    g(X,X,4) .

g(X,Y,Z) ->
    X * Y - Z .

l(Y) ->
    Y - 6 .

a.erl(1)
g(X,Y,Z) ->
    X * Y * Z .

f(X) ->
    g(X,X,4) .

h(X) ->
    X + 5 .
```

4.3. ábra. Függvény mozgatása, beszúrás és törlés

5. fejezet

Kapcsolódó munkák

A dolgozatom elején felvázolt problémára egy szintaxisfák összehasonlításán alapuló megoldást választottam. Különböző feladatokban egyre nagyobb szerepe van fa típusú adatszerkezetek összehasonlításának, egyre több területen van igény ilyen jellegű feladat elvégzésére.

Napjainkban folyamatosan növekszik az XML-alapú dokumentumok, szolgáltatások felhasználása. Ezekben gyakran van igény a változások felismerésére, különbségek kiszámítására, verziókövetésre, szinkronizálási feladatok elvégzésére. Mivel az XML-dokumentumok fa-szerkezetűek, ezekben a feladatokban szintén fákat kell összehasonlítanunk [5].

Fontos lehet még fák változásainak kezelése bioinformatikai feladatokban – például DNS vizsgálatoknál –, automatikus tételbizonyításoknál, képelemzési feladatokban, fordítók optimalizálásánál.

Bizonyos esetekben egy projekt programjainak verziókövetését is szintaxisfa alapon végzik. Erre akkor lehet szükség, ha mélyebb összefüggéseket akarunk felismerni egy program két verziója között. A szöveges alapon történő összehasonlításnak ugyanis ezen a területen korlátozottak a lehetőségei. Ilyen mélyebb elemzést igénylő kérdések lehetnek például változók, függvények, típusok törlése, vagy hozzáadása, ehhez kapcsolódó információk.

A szakirodalom foglalkozik általános fa adatszerkezetek összehasonlításá-

val, azonban a konkrét feladatok mindig specializációt, optimalizációt igényelnek. A témában a fák összehasonlítására használt angol fogalmak a következők: tree difference [6], tree edit (distance) [4], tree matching [7], tree alignment [3].

A fontosabb modellek mindegyike definiál néhány alpműveletet, primitívet, amiknek a segítségével egy fa átalakítható egy másik fába. Egy ilyen műveletsorozatot nevezünk edit szkriptnek. Ez egyben egy leképezést is definiál két fa között. Az alpművelet más diff algoritmushoz hasonlóan itt is a beszúrás, törlés, frissítés. Ezekhez jöhet még a mozgatás, ami azonban gyakran a többi művelettel kerül leírásra. Két fa összehasonlításánál általában több helyes edit szkriptet is előállíthatunk, azonban a cél egy minimális szkript előállítása. Ezért minden alpművelethez rendelnek egy költséget, így egy szkript költsége az azt leíró műveletek költségeinek az összege lesz. A minimalizációt tehát ezen az alapon kell érteni.

A fa csúcsainak illesztésére gyakran valamilyen szöveges távolságmetrikákat – például Levenshtein távolság – használnak. Ebben általában külön kell kezelnünk a fa leveleit és a belső csúcsokat. A leveleknél alkalmazhatunk ilyen szöveges metrikát, a belső csúcsok hasonlóságát pedig a részében levő közös levelek száma alapján becsüljük meg.

AST-k összehasonlításánál ezeket a szabályokat tovább pontosíthatjuk, ha a csúcsokat felcímkézzük típusokkal is. A következő típusok elképzelhetők egy imperatív nyelv esetén: IF – **if** utasítás; THEN – **then** blokk; MI – metódushívás. Így a levelek hasonlóságához megkövetelhetjük, hogy azonos legyen a típusuk. Az algoritmusok pontossága nagyban függ a választott metrikáktól, és korlátoktól. Gyakran szükség van több iterációra is.

TDK-dolgozatomban az első fejezetben leírt problémára egy szintaxisfák összehasonlításán alapuló, a feladatra specializált, az Erlang nyelv Abstract Format-jára illesztett megoldást mutatok. Az algoritmusban a fentiekhez hasonlóan én is használok egy bizonyos metrikát. A többszöri iteráció szükségét szintén bemutattam az algoritmus leírásánál. A szintaxisfák összehasonlítása

után az algoritmussal megkapjuk az eredeti és az átalakított program között a különbséget, ezzel egy leképezést is kapunk köztük.

Diplomamunkámban [10] szintén programok összehasonlításával foglalkozom, azonban egy ennél jóval általánosabb, szélesebb témakörben. Ebben megvizsgálom, hogy a különböző összehasonlítást igénylő feladatokban milyen eredményt érhetünk el, ha a változások kiszámítását szöveges alapon végezzük, milyen finomításai, korlátai lehetnek a szöveges összehasonlításnak, majd összevetem ezeket a szintaxisfa alapú összehasonlítással is.

Összefoglalás

TDK-dolgozatomban az Erlang nyelv parse transform-jának elemzési lehetőségeivel foglalkoztam. Ehhez először mutattam néhány példát a parse transform alkalmazásaira, majd részletesebben ismertettem a RefactorErl-ből kiinduló problémát.

Néhány megoldási lehetőség felvázolása után ismertettem a saját, szintaxisfák összehasonlításán alapuló megoldásomat. Ennek a működését meghatározó három illeszkedési szabály a következő volt: ekvivalens típusok; szerkezeti hasonlóság; szerkezeti tartalmazás. Az Esettanulmányok című fejezetben néhány példán keresztül bemutattam az algoritmus működését.

Fontos kiemelni, hogy a választott megoldás valójában független az eredeti, parse transform-okhoz, RefactorErl-hez kötődő problémától, az algoritmus kellően általános ahhoz, hogy más problémára, más nyelvre is adaptálhassuk. Az implementáció Erlang nyelven és az Erlang nyelv szintaxisfájára készült el, azonban az algoritmus szerkezete, illetve az illeszkedési szabályok könnyen átvihetők más nyelvre, más feladatra is.

Mivel a feladat megoldására szofterváltozatokat hasonlítunk össze, az algoritmust használhatnánk más területeken is, például szoftverevolúció vizsgálatára. Ezzel ugyanis megnézhetnénk, hogy mekkora változás történt két távolabbi verzió között, milyen részletek maradtak meg az eredeti kódból, milyen funkciók lettek átalakítva, törölve, milyen új műveletek lettek megvalósítva.

Irodalomjegyzék

- [1] Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta: *Tracking Your Changes: A Language-Independent Approach* IEEE Software, 26(1), 50-57, 2009
- [2] Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta: *Identifying changed source code lines from version repositories* Proceedings of the 4th International Workshop in Mining Software Repositories, 2007
- [3] Tao Jiang, Lusheng Wang, Kaizhong Zhang: *Alignment of trees — An alternative to tree edit* Lecture Notes in Computer Science, 807, 75-86, 1994
- [4] Philip Bille: *A survey on tree edit distance and related problems* Theor. Comput. Sci, 337, 217-239, 337
- [5] Luuk Peters: *Change Detection in XML Trees: a Survey* 3rd Twente Student Conference on IT., 2005
- [6] Beat Fluri, Michale Wuersch, Martin Pinzger, Harald Gall: *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction* IEEE Trans. Softw. Eng., 33, 725-743, 2007
- [7] Iulian Neamtiu, Jeffrey Foster, Michael Hicks: *Understanding Source Code Evolution Using Abstract Syntax Tree Matching* Proceedings of the International Workshop on Mining Software Repositories (MSR), 1-5, 2005

- [8] Eelco Lempsink, Sean Leather, Andres Löh: *Type-Safe Diff for Families of Datatypes*, Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming, 61-72, 2009, ISBN-978-1-60558-510-9
- [9] ChangeDistiller - Source code change extraction using AST differencing.
<http://www.ifi.uzh.ch/seal/research/tools/changeDistiller.html>,
2013. 05. 14.
- [10] Szabó Richárd: *Forráskódok változásainak vizsgálata* Diplomamunka, ELTE IK, Budapest, 2013
- [11] Peephole optimization – Wikipedia.
http://en.wikipedia.org/wiki/Peephole_optimization
- [12] Erlang – identity parse transform.
http://www.erlang.org/doc/man/erl_id_trans.html, 2013. 05. 14.
- [13] Martin Fowler: *Refactoring - Improving the design of existing code*, Addison Wesley, 2000, ISBN-0201485672
- [14] RefactorErl - Refactoring Erlang programs.
<http://plc.inf.elte.hu/erlang>
- [15] Erlang – The Abstract Format.
<http://www.erlang.org/doc/apps/erts/absform.html>, 2013. 05. 14.
- [16] Erlang Programming Language.
<http://www.erlang.org>
- [17] Recless: A Type Inferring Parse Transform for Erlang
<http://yarivsblog.blogspot.hu/2006/10/recless-type-inferring-parse-transform.html>
- [18] Literal XML in Erlang with parse_transform/2
<http://hyperstruct.net/2007/06/26/literal-xml-in-erlang-with-parse-transform-2/>, 2013. 05. 14.

- [19] Introducing Lager – A New Logging Framework for Erlang/OTP
<http://basho.com/blog/technical/2011/07/20/Introducing-Lager-A-New-Logging-Framework-for-ErlangOTP/>, 2103. 06. 10.
- [20] Erlang – qlc.
<http://www.erlang.org/doc/man/qlc.html>
- [21] Erlang – ms_transform
http://www.erlang.org/doc/man/ms_transform.html