Eötvös Loránd Tudományegyetem

Informatikai Kar

Tézisfüzet

dr. Tejfel Máté

2017

Forráskód feldolgozást végző alkalmazások fejlesztése és helyességvizsgálata.

1. Bevezetés

Napjainkban reneszánszát éli az új programozási nyelvek fejlesztése, mivel számos olyan specifikus alkalmazási terület létezik, amely az eddigiektől eltérő, speciális nyelvek használatát igényli. Ilyen igény lehet az energiafelhasználás vagy futásidő szempontjából hatékony programok készítése. Erre a két példára jellemző területként említhető a beágyazott rendszerek programozása, illetve a hálózati forgalmat kezelő alkalmazások (ide kapcsolódó nyelvek például a HUME [Ham07], illetve a P4 [Bos14]).

Természetesen ezekben a specifikus esetekben is előállíthatóak megfelelő alkalmazások, alacsony szintű, általános célú nyelvek (jellemzően a C nyelv) segítségével. Viszont ilyenkor az implementációk elkészítése nagyon magas fokú szakértői tudást igényel és az alkalmazott hardvereszközök megváltoztatása általában a teljes implementáció újbóli elkészítését vonja maga után. Ebből adódóan folyamatosan jelentkezik az igény magas szintű, alkalmazási terület specifikus programozási nyelvek készítése iránt.

Részben a fent is említett optimalizálási igények, részben a nagyméretű alkalmazások fejlesztése során általánosan jelentkező karbantarthatósági, továbbfejleszthetőségi igények miatt, szintén népszerűek az elemzéssel, illetve refaktorálással, valamint az optimalizálási céllal végrehajtott, forráskód szintű transzformációval foglalkozó kutatások.

A fenti témák közös vonása, hogy az elkészített alkalmazások a tevékenységük során valamilyen forráskód feldolgozását végzik el. Ez teszi a tesztelésüket is hasonlóvá, hiszen vizsgálatukhoz elő kell állítanunk a forráskódot, melyet feldolgozhatnak. Speciális funkciók ellenőrzése esetén pedig a bemeneti forráskódot (refaktorálás, transzformáció esetén akár a kimeneti forráskódot is) elemeznünk kell.

Jelen dolgozat célja, hogy a szerzőnek a PhD fokozat megszerzését követően a fent leírt témakörben végzett kutatásait, azok eredményét összefoglalja. A bemutatott eredmények többszerzős cikkekben, jellemzően ipari partner által inspirált kutatási projektek keretében készültek, melyek létrehozása során folyamatosan együtt gondolkoztunk és alkottunk, így nem lehet azokat kizárólag egyetlen személyhez kötni. A bemutatás során azon eredményeket fogom részletezni, melyeket fontosnak és lényegi hozzájárulásnak tekintek a témához, és melyek létrejöttében jelentős szerepem volt.

2. Programozási nyelvek fejlesztése

Az elmúlt évek kutatásai során áttekintettük, milyen lehetőségek adódnak meglévő nyelvek kibővítésére, illetve új (vélhetően alkalmazási terület specifikus) programozási nyelvek készítésére. Ezen felül azt is megvizsgáltuk, hogy ezek a lehetőségek hogyan viszonyulnak egymáshoz, hogyan lehet esetleg az egyik megközelítés segítségével orvosolni a másik gyengeségeit.

2.1. A Haskell nyelv kiterjesztése a Reference típussal

A Haskell nyelv kibővítése a Reference-nek nevezett új nyelvi elemmel.

Amikor valamilyen új funkcionalitást ellátni képes nyelvet szeretnénk létrehozni, talán a legegyszerűbb megközelítés, ha kiterjesztünk egy már meglévő nyelvet, új programkönyvtárakat létrehozva. Ebben az esetben nagy előnyt jelent, hogy (esetleg megfelelő bővítéseket alkalmazva) felhasználható a meglévő nyelvhez tartozó teljes eszközkészlet (például típusellenőrző, interpreter, fordító stb.).

Ezt a megközelítést alkalmaztuk, amikor a Haskell nyelvet bővítettük a *Reference*-nek nevezett új nyelvi elemmel. Ebben az esetben nem egy teljesen új nyelvet akartunk létrehozni, csak egy nem támogatott funkciót hozzáadni a nyelvhez. Ez a bővítés összetett adatkonstrukciók elemeinek egyszerűbb, egységesített elérését teszi lehetővé.

Imperatív nyelvekben megszokott, hogy rendelkezésünkre állnak struktúrák mezőinek elérését, felülírását lehetővé tevő lekérdező, illetve beállító függvények. Hasonlóan Haskellben is léteznek automatikusan generált függvények, melyeket használhatunk a rekordokban szereplő értékek lekérdezésekor, illetve rekord-felülírások alkalmával. Azonban ezek nem első osztályú nyelvi elemek, nem jelenhetnek meg kifejezésként és nem kombinálhatóak. A *Lens* típus (lásd [Kme15]) részben megoldást kínál erre a problémára. A Lens-ek első osztályú nyelvi elemek és alkalmazhatóak összetett struktúrák elemeinek elérése és felüldefiniálása során, azonban nem használhatóak Haskell monádokat alkalmazó kifejezésekben.

A Haskell egy tisztán funkcionális nyelv, így ha a program kiértékelése során bármiféle mellékhatást szeretnénk elérni (például csak kiírni valamit a képernyőre), akkor szükségünk van egy speciális nyelvi elemre, a monádra. A monádokat egy konténerként képzelhetjük el, ami elrejt valamilyen mellékhatást, számítást és egy adott típusú adatot reprezentál. Monádok segítségével számos, az imperatív programozás világából ismert nyelvi elem integrálható Haskellbe (például felhasználói interfész kezelése, kivételkezelés, állapotok tárolása).

A fentiekből látható, hogy számos olyan alkalmazás létezik, ahol nem kerülhetjük el a monádok használatát. Az általunk bevezetett Reference-ek (lásd [Nem15]) monádokat alkalmazó kifejezésekben is használhatóak, így például alkalmasak akár más szálak, folyamatok vagy gépek által tárolt adatok elérésére is. A Reference-ek szintén első osztályú nyelvi elemek, kompozíciónálisak, automatikusan generálhatóak bármilyen előre definiált típushoz és visszafele kompatibilisek a Lens-ekkel.

Egy Reference típusú elem egy adatelérési absztrakciónak tekinthető, amely három függvényt bocsájt rendelkezésünkre, az adatot lekérdező (*refGet*), beállító (*refSet*) és (egy paraméterben megadott függvény segítségével) módosító (*refUpdate*) függvényeket. A Reference-ekre vonatkozóan a következő öt szabály teljesülését várjuk el:

- get-get szabály – a lekérdezés nem okozhat mellékhatást, így két lekérdezés egymás utáni végrehajtása ekvivalens egyetlen lekérdezés végrehajtásával,
- set-get szabály – egy beállítás utáni követlen lekérdezés a beállított értéket kell visszaadja (bár a beállításnak lehet mellékhatása),
- get-set szabály – egy lekérdezés után, ha a kapott értékkel hajtunk végre egy beállítást, akkor nem változik az eredeti adatstruktúra,
- set-set szabály – a beállítás felülír minden előző változtatást, azaz két egymás után elvégzett beállító művelet végrehajtása ekvivalens a második művelet végrehajtásával,
- update szabály – egy adott *f* függvény segítségével történő módosítás ekvivalens a lekérdezésnek, a kapott érték *f* függvény szerinti módosításának és az új értékkel végzett beállításnak a sorozatával.

A Reference-ek egyik nagy előnye a tradicionális mező elérő függvényekkel szemben, hogy

kombinálhatóak. Az egyik legfontosabb kombinátor, ami alkalmazható rájuk a kompozíció (`&`) kombinátor. Segítségével egy összetett adatstruktúra mélyén lévő érték is elérhető. Például egy n-es első elemében lévő lehetséges értéket érhetjük el a `just` referencia (mellyel egy `Maybe` típusú adatból tudunk egy lehetséges értéket kinyerni), illetve a `_1` referencia (mellyel egy n-es első elemét tudjuk elérni) kompozíciójával a (`_1 & just`) referenciával.

A másik érdekes kombinátor az additív kombinátor (`&+&`), amellyel több Reference által elért elem együttesét tudjuk elérni, abban az esetben, ha az elemek megegyező típusúak. Például a (`_1 &+& _2`) referencia egy n-es első két elemét tudja elérni.

Az additív kombinátor és a kompozíció disztributív:

`r & (p &+& q) ≡ (r & p) &+& (r & q)`, illetve

`(r &+& p) & q ≡ (r & q) &+& (p & q)`.

Azért, hogy az újonnan létrehozott programkönyvtár valóban hasznos legyen, azt kell elérnünk, hogy bármilyen létrehozott felhasználói adattípusra alkalmazható legyen. Emiatt megalkottunk egy Template Haskell [She02] generátort, amely képes referenciákat létrehozni adott adattípushoz. A generálás során külön referenciák jönnek létre az adattípusban szereplő minden konstruktor minden mezőjéhez.

Esettanulmányként, bizonyítandó a Reference típus használhatóságát, azt is megmutattuk, hogy referenciák segítségével is megvalósítható a programkönyvtár ezen része, amely generálni tud referenciákat bármilyen felhasználói adattípushoz.

Kapcsolódó publikáció:

Boldizsár Németh, Zoltán Kelemen, Máté Karácsony, Máté Tejfel: **Extending Haskell with Effectful Property Abstraction** In: Valerie Novitzka , Štefan Korečko , Anikó Szakál (szerk.) *Proceedings of 13th International Scientific Conference on Informatics.* Poprad, Szlovákia, IEEE, 2015. pp. 183-188. (ISBN:978-1-4673-9867-1)

2.2 Beágyazott nyelvek

A Miller és a Feldspar nyelvek létrehozása beágyazott nyelvként a Haskell, mint gazdanyelv felhasználásával.

Egy teljesen új nyelv létrehozásakor az egyik általános megoldás a beágyazás. Ilyenkor a nyelvet egy másik (legtöbbször általános célú) programnyelv könyvtáraként hozzuk létre, azaz a programfejlesztés egy jelentős méretű programkiterjesztésnek is felfogható, ami így is jóval kisebb költséggel jár, mint egy önálló, független nyelv fordítóprogramjának elkészítése. Beágyazott nyelvekkel kapcsolatban számos kutatást végeztek, például, hogy hogyan kell megtervezni ([Kam98], [Lei99]), implementálni ([Hud98]), illetve fordítani őket ([Ell00]). A technika hátránya, hogy a beágyazott nyelv fordítóprogramjának nincs közvetlen hozzáférése a forráskódhoz, így a pontos hibaüzenetek készítése, illetve az elemző, hibakereső funkciók támogatása sokkal körülményesebb.

A beágyazási technikákat tekintve megkülönböztethetünk sekély és mély beágyazást. Előbbi esetén a beágyazott nyelven írt programot a gazdanyelv programjaként futtatva kaphatjuk meg a beágyazott nyelvű program eredményét. A második esetben viszont ilyen módon futtatva a programot a beágyazott nyelvű program absztrakt szintaxisfáját kapjuk meg, amit a beágyazott nyelv fordítója vagy interpretere dolgoz fel.

Az elmúlt évek kutatásai során több magas szintű, alkalmazási terület specifikus nyelvet is készítettünk a mély beágyazás módszerét alkalmazva. Az egyik ilyen nyelv a Miller [Nem13], melynek célja futásidő tekintetében hatékony alkalmazások készítése szoftver-vezérelt gyors elérésű memóriát tartalmazó architektúrákra. A nyelv támogatja a végrehajtási verem nélküli programokat a függvények helyett a „bubble”-nek nevezett végrehajtási egységek felhasználásával. Emiatt a Millerben készített programok vezérlésfolyama a hívási verem helyett folytatás átadáson alapszik. A nyelv másik specialitása, hogy támogatja az adatstruktúrák konfigurálását, amellyel többszintű memóriahierarchiával rendelkező hardverek esetén megadható, hogy a struktúra mely része, melyik memóriában helyezkedjen el (lásd a 4.2 fejezetet). A nyelv fordítója az általános MIPS32 architektúrát, illetve annak egy ipari környezetben használt változatát támogatja és optimalizált assembly kódot állít elő. Az optimalizálások során ablak és prefetch optimalizációt végez.

Egy másik ilyen nyelv a Feldspar, amely digitális jelfeldolgozó algoritmusok absztrakt, deklaratív stílusú leírását teszi lehetővé mellőzve a hardverfüggő, alacsony szintű konstrukciók használatát. A Feldspar programok a `Vector` típust alkalmazzák, amely egy indexezett sorozatnak tekinthető, és Haskell lista operátorokhoz hasonló függvényekkel kezelhető.

Példaként tekintsük a következő Feldspar nyelven írt kódot, amely egy olyan függvényt definiál, mely az 1 és n közötti értékek négyzetének összegét számolja ki. Az első sor egy típusszignatúra, amely megadja, hogy a függvény egy egész értéket vár paraméterül és az eredménye is egy egész érték.

```
sumSq :: Data Int -> Data Int
sumSq n = sum (map square (1 ... n))
  where
    square x = x*x
```

A fenti kódra úgy is tekinthetünk, hogy az egy program generátor (vagy makró), ami előállítja a függvényt ténylegesen kiszámító, Feldspar core nyelvű programot. Az előbbi példakódból a következő core program fog keletkezni:

```
program v0 = v11_1
  where
    v2 = v0 - 1
    v3 = v2 + 1
    v4 = v3 - 1
    (v11_0,v11_1) = while cont body (0,0)
      where
        cont (v1_0,v1_1) = v5
         where
           v5 = v1_0 <= v4
        body (v6_0,v6_1) = (v7,v10)
         where
           v7 = v6_0 + 1
           v8 = v6_0 + 1
           v9 = v8 * v8
           v10 = v6_1 + v9
```

Az a gráfleírás, amely ezt a core programot reprezentálja köztes reprezentációként szolgál a fordító backend-je számára. A backend először egy absztrakt imperatív kódot készít ebből a gráfból.

Ezen az absztrakt kódon különféle optimalizációkat tudunk elvégezni, illetve viszonylag egyszerűen lehet belőle C kódot előállítani, ami ennek a fázisnak a végső célja. Az optimalizációk egy általános plugin architektúra segítségével készültek el, így tetszőlegesen bővíthetőek. Az optimalizációk egy része hardverfüggő, különböző célhardverek esetén különböző optimalizációk végezhetőek el.

Az előbb látott core programból néhány egyszerű optimalizálás (mint például a copy propagálás) elvégzése után az alábbi C kód állítható elő:

```
void sumSq(signed int var0, signed int *out)
{
   signed int var11_0;
   var11_0 = 0;
   *out = 0;
   {
     while( (var11_0 <= (((var0 - 1) + 1) - 1)) )
     {
         signed int var8;
         var8 = (var11_0 + 1);
         var11_0 = (var11_0 + 1);
         *out = (*out + (var8 * var8));
     }
   }
}
```

Néhány egyszerű, jelfeldolgozás során is használt algoritmus segítségével összehasonlítottuk a Feldspar nyelven írt programok és hardverfüggetlen ANSI C szabvány szerint készített programok teljesítményét. Az összehasonlításnál a célhardver egy digitális jelfeldolgozáshoz széles körben használt Texas Instruments eszköz a TMS320C64xx volt, az elemzéshez a Code Composer Studio programot és annak elemző eszközeit használtuk. Eredményül azt kaptuk, hogy néhány speciális esettől eltekintve a Feldspar programból kapott kód a kézzel írt C kóddal összevethető teljesítményű.

Mindkét bemutatott nyelv esetén a Haskell funkcionális nyelvet használtuk gazdanyelvként. A Haskell kiválóan alkalmas ilyen célra, mivel jól használható különböző adatstruktúrák kezelésére, mert nem jelennek meg benne mutatók és a fordítója is számos optimalizációt képes elvégezni, ezen felül statikusan típusos és deklaratív, illetve a szintaxisa is elég rugalmas ahhoz, hogy lehetővé tegye különböző stílusú beágyazott nyelvek szintaxisának elkészítését.

Általános esetben egy nyelv mély beágyazása a következő tevékenységeket rejti magában:

- az absztrakt szintaxisfa adattípusainak definíciója;
- az absztrakt szintaxisfa elkészítését segítő frontend létrehozása, amely megkönnyíti a felhasználóknak a programok leírását a beágyazott nyelven, valójában a frontend határozza meg a felhasználók számára érzékelhető szintaxisát a nyelvnek;
- a backend elkészítése, ami lehet egy fordító, ami az absztrakt szintaxisfát célnyelvű forráskódra alakítja, vagy egy interpreter, ami végrehajtja azt.

Így egy beágyazott nyelv elkészítése általában jóval egyszerűbb, mint egy önálló nyelvé, hiszen nem kell lexikális elemzőt, szintaktikus elemzőt, típusellenőrzőt készíteni, továbbá, ha a gazdanyelv típusrendszere megfelelő, az új nyelv szemantikus szabályainak egy részét megadhatjuk a frontend függvényeknek, illetve az absztrakt szintaxisfa elemeinek típusleírásaiban. Ezáltal a szemantikus elemzés egy részét automatikusan elvégzi a gazdanyelv fordítója.

A technika hátránya, mint már említettem, hogy a fent leírtaknak megfelelően a beágyazott nyelv fordítóprogramja (backend-je) nem fér hozzá közvetlenül a forráskódhoz, csak az absztrakt

szintaxisfához. Így általános esetben nehéz meghatározni, hogy például egy, az absztrakt szintaxisfában szereplő definícióhoz a forráskódnak pontosan melyik része tartozik, azaz például egy, a backend által felfedett esetleges hiba esetén nehéz pontos hibajelzést adni.

Ez a probléma több szinten és mélységben orvosolható. Az egyik megoldás egy előfeldolgozó (preprocesszor) alkalmazása, melynek segítségével létrehozhatunk olyan makrókat, melyek feldolgozáskor automatikusan extra információkat generálnak a forráskódból (például a forrásfájl neve, az aktuális sor sorszáma, a forráskódbeli azonosító neve stb.). Kiegészítve a frontend függvényeket és az absztrakt szintaxisfát oly módon, hogy képesek legyenek továbbadni, illetve tárolni ezt a többletinformációt, ezek elérhetővé tehetőek a backend számára. Ezen megközelítés előnye a könnyű megvalósíthatóság, hátránya, hogy a makrók használata miatt megváltozik a szintaxis, illetve, hogy a gazdanyelv fordítójának hibaüzenetei a makrókat már nem tartalmazó, kibontott kódra fognak hivatkozni, így ezek a hibaüzenetek nehezen értelmezhetőek is lehetnek.

Egy másik megoldás a gazdanyelv szintaxisfájának manipulációján alapszik. A megoldás során a beágyazott nyelvben olyan új, burkoló függvényeket (nevezzük ezeket a továbbiakban címke függvényeknek) vezetünk be (kiterjesztve ezzel a frontend-et és a belső reprezentációt), amelyek a különböző elemek mellé paraméterül megkapják azok forráskódbeli pozícióját is. Ezek után a beágyazott nyelven írt programhoz tartozó gazdanyelvi reprezentációt bővítjük ki oly módon, hogy minden szintaktikus elemhez tartozó csúcshoz készítünk egy, az elem pozícióját tartalmazó címke függvényt reprezentáló csúcsot is. Ezt a módosított szintaxisfát visszaalakítjuk forráskóddá, amelyben így megjelennek a megfelelő címke függvények, majd ezt a módosított forráskódot alakítjuk át beágyazott nyelvi absztrakt szintaxisfává. Ily módon a pozíció információk (függvényparaméterként) megjelennek a beágyazott nyelv backend-je számára elérhető absztrakt szintaxisfában. Ezen megközelítés érdekessége, hogy az első lépés, azaz a gazdanyelv szintaxisfájának manipulációja független a konkrét beágyazott nyelvtől, azaz ha ugyanazt a gazdanyelvet több beágyazott nyelvhez is felhasználjuk, elég csak egyszer definiálni.

Kapcsolódó publikációk:

Gergely Dévai , Dániel Leskó , Máté Tejfel: **The EDSL's Struggle for Their Sources** *LECTURE NOTES IN COMPUTER SCIENCE 8606*: pp. 300-335. (2015) Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers. (ISBN 978-3-319-15940-9)

Dévai G , Gera Z , Horváth Z , Páli G , Tejfel M: **Feldspar - A Functional Embedded Language for Digital Signal Processing** In: Attila Egri-Nagy , Emőd Kovács , Gergely Kovásznai , Gábor Kusper , Tibor Tómács (szerk.) *Proceedings of the 8th International Conference on Applied Informatics*: January 27-30, 2010, Eger, Hungary. 2011. pp. II./149-156. (ISBN:978-963-9894-72-3)

Gergely Dévai , Máté Tejfel , Zoltán Gera , Gábor Páli , Gyula Nagy , Zoltán Horváth , Emil Axelsson , Mary Sheeran , András Vajda , Bo Lyckegard , Anders Persson: **Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs** In: Jagadeesh Sankaran , Tom Vander Aa (szerk.) *Proceedings of 8th Workshop on Optimizations for DSP and Embedded Systems (ODES-8)* Toronto , Kanada , 2010.04.25 pp. 12-20.

2.3 Önálló nyelv készítése

A Miller nyelv elkészítése önálló nyelvként.

Mint az előzőekben láthattuk, egy új nyelv beágyazott nyelvként történő elkészítése több előnnyel is jár, ugyanakkor erős hátrányt jelent, hogy a beágyazott nyelv fordítója nem fér hozzá közvetlenül

a forráskódhoz, így például a pontos hibajelzés, hibakeresés és kódelemzés komoly gondot jelenthet, valamint (mint azt például a Feldspar esetén magunk is tapasztaltuk) nehéz az új nyelvet elfogadtatni olyan felhasználókkal (például egy ipari környezetben), akik nem járatosak a gazdanyelv (esetünkben a Haskell) szintaxisában.

Részben ezen okokból döntöttünk úgy, hogy a már említett Miller nyelv végső változatát önálló nyelvként készítjük el. Így a nyelv fejlesztése több fázisból állt. A kezdeti tervezési fázisban, amikor még nem alakultak ki a pontos nyelvi koncepciók egy beágyazott nyelvet készítettünk, így ki tudtuk használni a beágyazásból adódó rugalmasságot, nem kellett előre meghatároznunk a konkrét szintaxist.

Az önálló nyelvhez ugyanazt a belső reprezentációt használtuk fel, mint a beágyazott nyelv esetén. Az önálló nyelv lexikális és szintaktikus elemzője először egy konkrét szintaxisfát épít fel, majd ezt alakítjuk át absztrakt szintaxisfává, amely megegyezik a beágyazott nyelv esetén a gazdanyelv fordítója által előállított absztrakt szintaxisfával. Ezáltal a backend rész, azaz az optimalizálás és a kódgenerálás fázisa teljesen megegyezik a két változat esetén, így a beágyazott nyelv készítésekor létrehozott fordítóprogram jelentős részét fel tudtuk használni a végső, önálló változathoz.

A teljes fejlesztés, azaz a beágyazott és az önálló nyelvváltozat elkészítéséhez szükséges teljes kódmennyiség tekintetében megvizsgáltuk, hogy az effektív kódsorokat figyelembe véve, a kód 58 százaléka volt olyan, amelyet mindkét változatnál felhasználtunk, 13 százalék olyan, melyet csak a beágyazott változatnál és 29 százalék olyan, melyet csak az önálló nyelvnél. Ez alapján megállapítható, hogy a konkrét fejlesztés esetén megfelelő volt az alkalmazott módszer, azaz hogy első lépésben egy beágyazott nyelv készült el és csak a fontosabb nyelvi koncepciók pontos letisztázása után jött létre az önálló nyelv.

Általánosságban tanulságként megfogalmazható, hogy, amennyiben lehetséges, érdemes kezdetben egy beágyazott nyelvet elkészíteni, mivel így viszonylag hamar (esetünkben 9 hónap alatt) elkészülhet a nyelv első verziója. Ennek segítségével pedig megvizsgálható, hogy megvalósított nyelv mennyire felel meg az elvárásoknak, meg tudjuk-e fogalmazni benne a kívánt algoritmusokat. Ezáltal először a nyelvi elemek belső reprezentációját tervezzük meg (és használjuk fel) és csak később alkotjuk meg hozzájuk a konkrét szintaxist. Így kezdetben fókuszálhatunk a tervezési kérdésekre és kevésbé kell foglalkoznunk a technikai problémákkal. Illetve, mivel a végső verzió önálló nyelvként készül el, a beágyazott nyelv esetén is csak a funkcionalitást kell szem előtt tartanunk, használhatjuk a legegyszerűbben megvalósítható szintaxist. Ugyanakkor a fejlesztés végső fázisában kellő időt kell szánni a konkrét szintaxis megvalósítására és esetünkben is elmondható, hogy a konkrét szintaxisfa és az absztrakt szintaxisfa közötti transzformáció jelentős átalakításokkal járt, azaz komoly figyelmet és energiát kell fektetni ennek implementálásába is.

Kapcsolódó publikációk:

Gergely Dévai , Dániel Leskó , Máté Tejfel: **Language Design and Implementation via the Combination of Embedding and Parsing** *COMMUNICATIONS IN COMPUTER AND INFORMATION SCIENCE* 457. pp. 131-147. (2014) (ISBN 978-3-662-44919-6)

Gergely Dévai , Dániel Leskó , Máté Tejfel: **Embedding and Parsing Combined for Efficient Language Design** In: *International Conference on Software Engineering and Applications (ICSOFT-EA 2013)*. Reykjavik, Izland, SciTePress, pp. 244-250.

3. Tesztelés

A fejlesztési lehetőségekkel párhuzamosan azt is megvizsgáltuk, hogy a nyelvek fejlesztése, illetve más forráskód feldolgozást végző alkalmazások fejlesztése során milyen tesztelési technikák alkalmazhatóak, hogyan lehet a teszteléshez szükséges forráskódokat előállítani, illetve elemezni.

3.1 Általános tesztelő keretrendszer

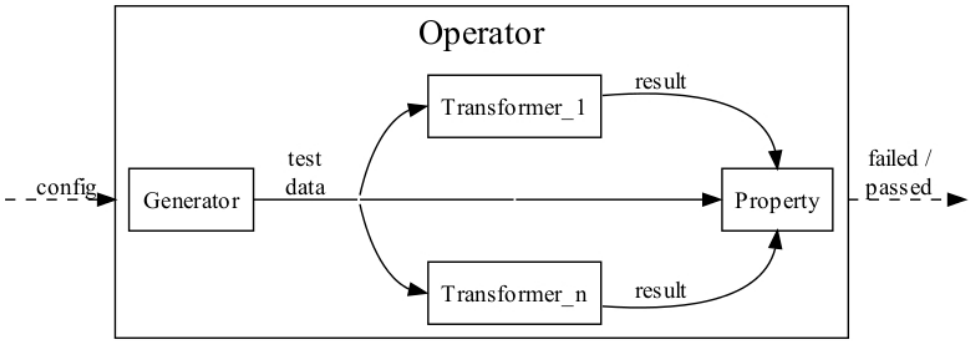
Tesztelő keretrendszer létrehozása beágyazott nyelvek teszteléséhez.

Egy új nyelv elkészítésekor a tesztelés szempontjából két, külön is vizsgálható fontos kérdés merül fel. Egyrészt vizsgálnunk kell, hogy a nyelvhez készített fordító vagy interpreter helyes-e, azaz az általunk elvártaknak (jó esetben valamilyen formális specifikáció által rögzítettek szerint) megfelelően működik-e. Másrészt, ha egy jól használható nyelvet szeretnénk létrehozni, fontos kérdés, hogy milyen fejlesztést támogató eszközöket (tesztelő, hibakereső, elemző) tudunk adni a nyelv mellé, az adott nyelven elkészített alkalmazások vizsgálatának elősegítésére. Megvizsgálhatjuk például, hogy milyen tesztelési lehetőségeink vannak. Ezt tette meg például Maria Grima és Gordon J. Pace egy új, geometriai problémák leírására készített, Haskell-be ágyazott nyelv esetén ([Gri07]). Természetesen hatékonyabb lenne, ha az újonnan létrehozott nyelvhez a fejlesztést támogató eszközöket nem kellene teljesen újra elkészítenünk, hanem felhasználhatnánk már meglévő megoldásokat.

A fenti két kérdést, részben egybevonva, beágyazott programozási nyelvek esetén vizsgáltuk meg részletesen. Létrehoztunk egy olyan kiterjeszthető és moduláris tesztelő keretrendszer modellt, amely lehetővé teszi, hogy egy új beágyazott nyelv támogatásához csak egy jól meghatározott interfészt kelljen kiterjesztenünk.

Az elkészített modell egy absztrakt váznak tekinthető, melyben négy komponenst különböztetünk meg:

- generátor (ez a komponens állítja elő a teszteseteket)
- transzformátor (a program absztrakciója, több is lehet a rendszerben)
- tulajdonság ellenőrző (a tesztadatok és a transzformátorok eredményei alapján vizsgálja a helyességet, egy logikai értéket ad eredményül)
- operátor (összefogja az előző hármat, kezeli az adatfolyamot és a felhasználói interakciókat)



1. Ábra: Tesztelő keretrendszer

A koncepció előnye, hogy amennyiben a fenti váz megvalósításra kerül egy adott gazda nyelven, akkor minden, abba a nyelvbe beágyazott új nyelv esetén csak egy új transzformátort kell készítenünk, a többi komponens változatlan maradhat. Az 1.ábra bemutatja a keretrendszer általános adatfolyam modelljét.

A generátor megvalósításához felhasználtuk a QuickCheck [Cl00] és a SmallCheck [Run08] eszköz meglévő eszközeit. Ez nem jelent túlzott megszorítást a lehetséges gazdanyelv tekintetében, hiszen például a QuickCheck eszköznek több mint 20 nyelvhez készült implementációja. A

generátor kimenete egy heterogén lista, amivel elérhető, hogy bár mindig egyetlen elem a kimenet, valójában akárhány, különböző típusú elemet generálhatunk, a transzformátorok igényének megfelelően.

Egy `Transformer a b` típusú transzformátor általánosan egy egyszerű `a→b` függvénynek tekinthető, amely bármilyen `a` típusú adatból előállít egy `b` típusú adatot. Adott beágyazott nyelv esetén transzformátor minták segítségével állíthatunk elő egyszerűen ilyen transzformátorokat. Egy ilyen minta általános alakja a következő

```
exampleTP :: f → Transformer a b
```

Ez a minta az adott nyelv bármilyen függvényéből egy transzformátort állít elő. A meglévő transzformátorokból nyelvfüggetlen, általános transzformátor kombinátorokkal állíthatunk elő bonyolultabb transzformátorokat. Ilyen kombinátorok például:

- `(>>>)` – adatfolyamként összekapcsol két transzformátort,
- `pairT` – két transzformátor egyszerű párhuzamos végrehajtása, ugyanazon a bemeneti adaton kell dolgozzanak, de a kimeneti értékek típusa lehet különböző,
- `listT` – bármennyi, de ugyanolyan típusú transzformátor párhuzamos összekapcsolása, a transzformátorok ugyanazt a bemeneti adatot kapják meg, a kimeneti adatok listája lesz az összetett transzformátor eredménye.

Az ellenőrző komponens pontos megalkotása a felhasználó feladata, de könnyítésül létrehoztunk néhány előre definiált verziót a leggyakoribb esetekre. Ez a komponens első paraméterül mindig megkapja a generált tesztadatot, második paraméterül pedig egy listát, ami a transzformátorok eredményeit tartalmazza. Ez utóbbiak típusa meg kell egyezzen. Az előre definiált ellenőrző komponensek a következők:

- `strictEquality` – pontos egyenlőséget vizsgál,
- `epsilonEquality` – sablon, közelítő egyenlőség vizsgálathoz,
- `elemWise` – magasabb rendű ellenőrző, elemenkénti ellenőrzést végez egy iterátor struktúrán egy egyszerű ellenőrző alapján,
- `sumWise` – magasabb rendű ellenőrző, egy iterátor elemeinek összegét vizsgálja egy egyszerű ellenőrző alapján,
- `base` – magasabb rendű ellenőrző az eredménylista egy elemét hasonlítja össze a többivel egy egyszerű ellenőrző alapján,
- `total` – magasabb rendű ellenőrző az eredménylista minden elemét összehasonlítja a többivel egy egyszerű ellenőrző alapján.

Arra is lehetőségünk van, hogy egyszerre több ellenőrzőt használjunk. Ebben az esetben az ellenőrző komponensek eredményeinek összekapcsolására az `and, or, not, imply` magasabb rendű ellenőrzők használhatóak.

Az operátor alapvetően eltér az előző komponensektől. A feladata, hogy gondoskodjon egy teszteset végrehajtásáról, összefogva az előző komponenseket. Minden iterációs lépésben a következő tevékenységeket végzi el:

- végrehajtja a generátort, hogy új tesztadatokat kapjon,
- a tesztadatokat továbbítja minden transzformátornak,
- összegyűjti a transzformátorok eredményeit és a kezdeti tesztadattal együtt továbbítja őket az ellenőrző komponensnek,
- elmenti az eredményt és ha kell végrehajt egy következő iterációt (a kívánt iterációk számát egy konfigurációs paraméterben lehet megadni).

A leírt keretrendszert felhasználtuk az előzőekben már említett, alkalmazási terület specifikus,

Feldspar nyelv interpreterének, fordítóprogramjának, illetve különböző Feldsparban készített alkalmazásoknak a tesztelésére. Ehhez elkészítettük a keretrendszer egy Haskell-es implementációját. Mivel a Feldspar nyelvnek nem állt rendelkezésre formális specifikációja, viszont számos olyan primitív függvénnyel és operátorral rendelkezik, melynek létezik Haskell-es megfelelője, ezért az interpreter ellenőrzésekor ezek Feldspar-os és Haskell-es működését hasonlítottuk össze (feltételezve, hogy a Haskell-es verzió helyesen működik). Ebben az esetben két transzformátort használtunk, az egyik egy Feldspar függvényt hajtott végre a Feldspar-os interpreter segítségével, míg a másik a megfelelő Haskell függvényt a ghci (a Haskell interpreter) segítségével az ellenőrző komponens pedig azt vizsgálta ezek megfeleltethetőek-e egymásnak.

A Feldspar fordító többféle kimenetet képes előállítani, ezek közül az egyik egy standard ANSI C kimenet. A fordítóprogram ellenőrzéséhez ezt a kimenetet használtuk és a kapott C fájlok viselkedését hasonlítottuk össze az eredeti Feldspar függvénynek a (már tesztelt) interpreter által meghatározott viselkedésével. Ehhez az előzőleg használt, az interpreterhez készített transzformátor mellett egy olyan transzformátort alkalmaztunk, amely

- végrehajtotta a Feldspar fordítót,
- a kapott C függvényhez generált egy olyan C `main()` függvényt, amely a szabványos input-ról beolvassa a szükséges adatokat és azokkal meghívja a generált C függvényt, majd a szabványos output-ra kiírja az eredményt,
- az így kapott C fájlokat gcc-vel lefordítja,
- a kapott végrehajtható fájlt egy külső folyamatként végrehajtja úgy, hogy bemeneti adatként továbbítja neki a megfelelő tesztadatokat,
- megvárja a végrehajtás befejeződését, beolvassa a kapott eredményt, majd lezárja a külső folyamatot.

Az interpreter és a fordítóprogram tesztelésén felül ugyanaz a keretrendszer képes Feldsparban írt alkalmazások tesztelésére is. A Feldspar egy digitális jelfeldolgozást végző algoritmusok elkészítéséhez készített alkalmazási terület specifikus nyelv. Ebből adódóan (mint általában az alkalmazási terület specifikus nyelvek esetén) van számos, az adott területre (jelen esetben a digitális jelfeldolgozásra) jellemző algoritmus, melyek helyes működése kritikus lehet az adott nyelven írt alkalmazások esetén és melynek már létezik elérhető implementációja. A Feldspar esetén olyan alkalmazásokat vizsgáltunk, melyeknek elérhető volt valamely referencia C implementációja (Például ilyen volt a gyors Fourier-transzformáció (FFT).) és a Feldspar-os változat, illetve a referencia változat viselkedését hasonlítottuk össze. Ehhez az előzőekben részletesen leírt transzformátor mellett, annak egy olyan módosított változatát alkalmaztuk, amely nem a generált C kódhoz, hanem egy paraméterül megadott C kódhoz készíti el a `main()` függvényt és végzi el a további lépéseket. Mivel a paraméterül adott C kódra nem tudunk típusellenőrzést végezni, ezért megkötés, hogy a referenciaként használt C függvény szignatúrája pontosan egyezzen meg a Feldspar fordítójával generált C függvény szignatúrájával.

A fentiekben bemutatott generátor komponens kellően általános, így nincs megkötés a tesztelendő elemek előállításának módjára. A gyakorlati tapasztalat azt mutatja, hogy érdemes a véletlenszerűen generált elemek használatát ötvözni előre megadott, az adott alkalmazás számára speciális jelentéssel bíró konstans értékek használatával. Például a Feldspar nyelv egy valós számokkal dolgozó könyvtárának vizsgálata során a véletlenszerű értékeket előállító generáló alkalmazásakor csak több heti sikeres „éjszakai teszt” után derült ki, hogy két modul különbözőképpen kerekíti a 0.5 értéket, ami természetesen komoly hibák potenciális okozója volt.

A bemutatott keretrendszer igazi előnye általánosságában rejlik. A konkrét implementáció éppúgy alkalmas volt egy beágyazott nyelv interpreterének és fordítójának vizsgálatára, mint az adott nyelven készített alkalmazások ellenőrzésére.

3.2 Megfelelő tesztadathalmaz létrehozása

Minden tesztelési folyamat esetén fontos kérdés a teszteset halmaz minősége, azaz, hogy a vizsgált tesztesetekkel mennyire átfogóan fedjük le a bemeneti adatok lehetséges halmazát. Természetesen ennek meghatározása elég viszonylagos és általában kellően bonyolult folyamat. A teszteset halmaz minőségének meghatározására különböző metrikákat lehet és szokás használni. Ilyenek például a utasításlefedettség (statement coverage), döntési ág fedettség (branch coverage), feltétel lefedettség (condition coverage), út lefedettség (path coverage) és a relációs operátor lefedettség (relational operator coverage) [Bei90].

Ezek többsége vezérlésfolyam alapú metrika, az adatfolyam figyelembevételével tovább növelhetjük ezek erejét. [Fra88] számos adatfolyam alapú metrikát bemutat, melyek mindegyike abból az alapgondolatból indul ki, hogy lehetőség szerint egy program minden kifejezésénél vizsgáljuk meg az összes lehetséges adatfolyamot, melyen keresztül a kifejezésben szereplő változók értéket kaphattak és minél jobban fedjük le ezeket tesztesetekkel.

A fent leírt metrikák általános esetben jól használhatók, de arányaiban sok kifejezést tartalmazó programok, illetve funkcionális programok esetén kevésbé alkalmazhatók. Ez igaz általában a már említett Feldspar nyelven írt programokra is.

Tekintsük példaként a következő rövid, WHILE nyelven (lásd [Nie92]) írt programot.

```
if ((x - 1 < 10) && (x mod 2 = 0)) then
   res := 100 div x
else
   res := 10 * x
```

Ez a rövid program is tekinthető arányaiban sok kifejezést tartalmazónak, hiszen a benne szereplő tíz nyelvi konstrukcióból kilenc kifejezés. A program hibás, `x==0` esetén rosszul működik, viszont az előzőekben felsorolt mindegyik metrika esetén könnyen tudunk olyan tesztadathalmazt készíteni, amely az adott metrika alapján 100%-os fedést ad és mégsem tartalmazza az `x==0` esetet.

Ebből kiindulva létrehoztunk egy új metrikát, melynél az alapgondolat az volt, hogy a vezérlésfolyam vizsgálatából kiinduló elemzés helyett, (ami leginkább jellemző a meglévő metrikákra) azt vizsgáljuk, hogy az adott tesztadathalmaz alkalmazása esetén a program függvényeinek kimeneti értékei lefednek-e minden, a lehetséges értékkészlet szempontjából érdekes esetet. Azaz például ha egy kétágú elágazásnál egy függvény eredménye alapján ágazunk el, de a függvénynek három lehetséges kimeneti értéke van, akkor nem azt vizsgáljuk, hogy a tesztelés során ráfutunk-e az elágazás mindkét ágára, hanem, hogy megjelenik-e a függvény mindhárom lehetséges értéke.

A metrika alkalmazásakor minden *p* programkonstrukciót függvénynek tekintünk (például egy elágazást is). Ezek után megadjuk ezen függvények viselkedés függvényét (*behavior(p)*), azaz a függvény minden paraméterére az adott paraméter értelmezési tartományának egy olyan felosztását, amely meghatározza, hogy mely bemeneti érték intervallumokat szeretnénk megkülönböztetetten kezelni az adott függvény szempontjából. Ez a felosztás egy adott értelmezési tartományra egy halmazokat tartalmazó halmazt ad, melyet akkor tekintünk helyesnek, ha a benne szereplő halmazok

- az értelmezési tartomány részhalmazai,
- egymáshoz képest diszjunktak,
- uniójuk lefedi a teljes értelmezési tartományt.

Abban az esetben, ha nem tudjuk vagy nem akarjuk az értelmezési tartományt felosztani (például hiányzó ismeretek miatt) egy speciális, □ szimbólummal jelölt felosztást fogunk használni, amely

egyetlen halmazt tartalmaz, magát a teljes értelmezési tartományt. Változók esetén a változó viselkedésfüggvénye mindig a □ felosztást adja eredményül.

Például a fenti kódrészletben szereplő `div` függvény esetén a *behavior(div)* függvény eredménye a következő lehet:

```
{[MIN..(-1)], [0], [1..MAX]} * {[MIN..(-1)], [0], [1..MAX]}
```

Ez azt jelzi, hogy a `div` függvény mindkét paramétere tekintetében három intervallumot szeretnénk megkülönböztetni a negatív számokat, a nullát és a pozitív számokat. Vagyis ezt a függvényt tekintve a tesztadathalmazt akkor fogjuk 100%-osnak tekinteni ha a fenti intervallumok minden kombinációja előáll a tesztelés során.

Természetesen a példaként megadott WHILE program esetén ezt sosem tudnánk elérni, hiszen az abban szereplő `div` függvény első paramétere egy konstans érték, ezért az ilyen esetekben nem a függvényt, hanem annak konstans paraméterrel vett alakját fogjuk használni, azaz a fenti esetben a `"100 div"` függvényt fogjuk alkalmazni és a

```
{[MIN..(-1)], [0], [1..MAX]}
```

felbontást vizsgálni. Ebből adódóan csak akkor fogunk teljesnek tekinteni egy tesztadathalmazt, ha tartalmazza az `x==0` esetet, azaz meg fogjuk találni a hibát a példaprogramban.

Önmagában az új metrikával azonban még nem tudunk feltétlenül javítani a meglévő teszteseteinken, hiszen csak azt tudjuk meghatározni, hogy a belőlük felépített halmaz mennyire tekinthető „jónak”, de ez abban nem nyújt segítséget, hogyan változtassuk meg a jelenlegi halmazt.

Emiatt, az új metrika elvárásait kiindulási alapnak tekintve, elkészítettünk egy olyan automatikus tesztadat generáló eljárást, amely képes a megfelelő (az új metrika alapján közel 100%-os) fedést adó halmaz elkészítésére.

A generálás során használni fogjuk a felosztás finomítási eljárást, amely egy olyan S felosztásból, amely tartalmaz nem diszjunkt halmazokat is egy új, helyes felosztást készít. A finomítás során ha találunk olyan x, y nem diszjunkt halmazokat, melyek elemei S-nek, akkor S-ből egy új S'= (S \ {x,y}) ∪ {x \ y, y \ x, x ∩ y} felosztást készítünk és ezt addig folytatjuk, amíg az eredményül kapott felosztás helyes nem lesz.

A tesztadat generálás alapja egy alulról felfelé tartó szimbolikus végrehajtás, melynek során egy lépést egy adott *p* programkonstrukcióra fogunk elvégezni és feltesszük, hogy meg van adva a *p* paramétereinek megfelelő számú és a paraméterek típusához illeszkedő bemeneti felosztásunk, illetve egy cél felosztás, ami a *p* eredményének elvárt felosztása. A lépés végrehajtásakor kiszámítjuk minden paraméterre a *behavior(p)* függvény által adott felosztás és a paraméterhez tartozó bemeneti felosztás uniójának finomítását. Majd az így kapott új felosztásokat tovább finomítjuk úgy, hogy garantálni tudjuk, hogy ha egy tesztadathalmaz lefedi a paraméterekhez tartozó felosztások minden osztályát, akkor az eredmény lefedi az eredményhez tartozó cél felosztást.

Egy nyelv fordítóprogramjának és interpreterének tesztelésekor ha véletlenszerű adatokat szeretnénk használni, akkor az adott nyelvnek megfelelő forráskódot kell generálnunk. Ennek érdekében, a meglévő eredményekből kiindulva (lásd [Wang05]) tanulmányoztuk miként lehet típusos lambda kifejezések generálási lehetőségeit felhasználni adott tulajdonságokkal rendelkező funkcionális programok előállításához és ez alapján elkészítettünk egy generátor prototípust.

A generátor két lépésben működik, először előállít egy helyes és típusos lambda kifejezést, majd második lépésben ezt fordítja tovább egy magasabb szintű reprezentációra. A generált értékek, kifejezések névvel is elláthatóak és így későbbi generálásokkor nevekkel hivatkozottan újrafelhasználhatóak. Ezek alapján a generátor képes olyan teljes fájlok generálására, amelyek

funkcionális nyelvű kódot tartalmaznak. A generált kód olyan függvényekből áll, amelyek esetlegesen egymásra is épülhetnek, típushelyesek és a belőlük épített kezdeti kifejezés típusa megfelel az előre megadott követelményeknek.

A generálással kapcsolatos kutatás eredményeiről a közeljövőben szeretnénk egy publikációt megjelentetni.

Kapcsolódó publikációk:

Leskó Dániel , Tejfel Máté: **Testing Framework For Embedded Languages** *COMPUTER SCIENCE AND INFORMATION SYSTEMS 10.* (4) pp. 1647-1659. (2013)

Dániel Leskó , Máté Tejfel: **A Domain Based New Code Coverage Metric And A Related Automated Test Data Generation Method** *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE SECTIO COMPUTATORICA 36* pp. 217-240. (2012)

Leskó Dániel , Tejfel Máté: **Testing framework for embedded languages** *AIP CONFERENCE PROCEEDINGS 1479.* pp. 454-457. (2012)

3.3 Elemző és refaktoráló eszközök tesztelése

Kódelemző és transzformáló eszközök formális specifikáció alapján történő tesztelése.

Az elemző, illetve a refaktoráló eszközök esetén talán még hangsúlyosabb a tesztelés fontossága, hiszen alapvető elvárás, hogy egyrészt az elemzés eredménye pontos legyen, hiszen csak ebben az esetben szolgáltat felhasználható információt a forráskódról (például forráskód transzformációk elvégzéséhez), másrészt pedig, hogy a forráskód transzformációk során csak a kód bizonyos minőségi mutatói (pl. átláthatóság, karbantarthatóság) változzanak (természetesen általában javuljanak) a szemantikája ne. Ahhoz, hogy egy refaktoráló eszközt ténylegesen alkalmazni lehessen akár ipari környezetben is, feltétlenül szükséges, hogy a lehető legnagyobb mértékben garantálni tudjuk az egyes végrehajtott lépések helyességét, azaz, hogy a fentieknek megfelelően nem változtatnak a kód szemantikáján. Tehát alapvető fontosságú, hogy a lehető legátfogóbban teszteljük az egyes lépéseket és ezáltal növeljük a kód minőségét.

Viszont ebben az esetben is elmondható, hogy a tesztelés végrehajtása messze nem triviális, hiszen amellett, hogy mind az elemző, mind a refaktoráló eszközök esetén a bemenet, illetve refaktoráló eszközök esetén a kimenet is forráskód (sőt akár több modulból álló komplett program) általános esetben azt sem egyszerű meghatározni, hogy az eredmény helyes. Azaz, hogy az elemzés esetén ténylegesen valós információkat kaptunk, transzformáció esetén pedig az eredeti, illetve a módosított forráskód által meghatározott program szemantikailag teljesen megegyezik.

Refaktoráló eszközök, pontosabban szólva konkrét refaktoráló lépések, konkrétan megfogalmazott tulajdonságainak vizsgálatával foglalkozik Huiqing Li és Simon Thompson ([Li08]). Ugyanakkor az ilyen eszközök általában rendelkeznek valamely, az elemzett vagy átalakítani kívánt program szintaktikus és szemantikus tulajdonságait leíró belső reprezentációval. Egy jól megtervezett reprezentáció ekvivalensnek tekinthető az eredeti forráskóddal, és precízen képes megadni a program vizsgálható tulajdonságait, így ez a reprezentációs szint is felhasználható a különféle elemzések, transzformálások és akár a tesztelés során is.

Elemző, illetve refaktoráló eszközök tesztelésének vizsgálatát a RefactorErl [Hor09] eszközön keresztül végeztük el, amely egy elemző és kódtranszformáló eszköz az Erlang funkcionális nyelvhez. Az eszköz az Erlang egy szemantikus programgráf reprezentációját használja, amely megfelel egy, a fentiekben leírt tulajdonságú belső reprezentációnak. Ez a reprezentáció egy gyökérrel rendelkező, irányított, címkézett három szintű gráf, amely tartalmazza az Erlang program lexikális, szintaktikus és szemantikus információit. A lexikális réteg tartalmazza az eredeti és az

előfeldolgozott program tokenjeit, a szintaktikus az előfeldolgozott kód absztrakt szintaxis fáját, míg a legfelső, szemantikus réteg a különböző szemantikus elemzések eredményét. Az eszköz egy aszinkron, párhuzamos szemantikus elemző keretrendszerrel rendelkezik, amely különböző statikus elemzéseket (például függvény hívás elemzés, modul hivatkozás elemzés, változó hivatkozás elemzés, függvény mellékhatás elemzés, adatfolyam elemzés, dinamikus függvény hivatkozás elemzés, vezérlésfolyam elemzés) képes modulárisan, egymásra épülve végrehajtani.

Emellett az eszköz egy szintaxis alapú, refaktoráló transzformációkat tartalmazó keretrendszerrel is rendelkezik, a gráfon végzett minden szintaxis alapú változtatás után a szemantikus elemző keretrendszer automatikusan visszaállítja a szemantikus információkat.

A tárolt információ mélységét mutatja, hogy a gráf alapján a kód tördelése és kommentezése is teljes mértékben visszaállítható.

Az alkalmazott belső reprezentáció bemutatására példaként tekintsük a következő egyszerű programot, amely egyetlen, egyparaméteres függvényt tartalmaz, amely egy lista elemeit szorozza össze:

```
-module(example).

prod([]) ->
    1;
prod([H|T]) ->
    Prod = prod(T),
    H * Prod.
```

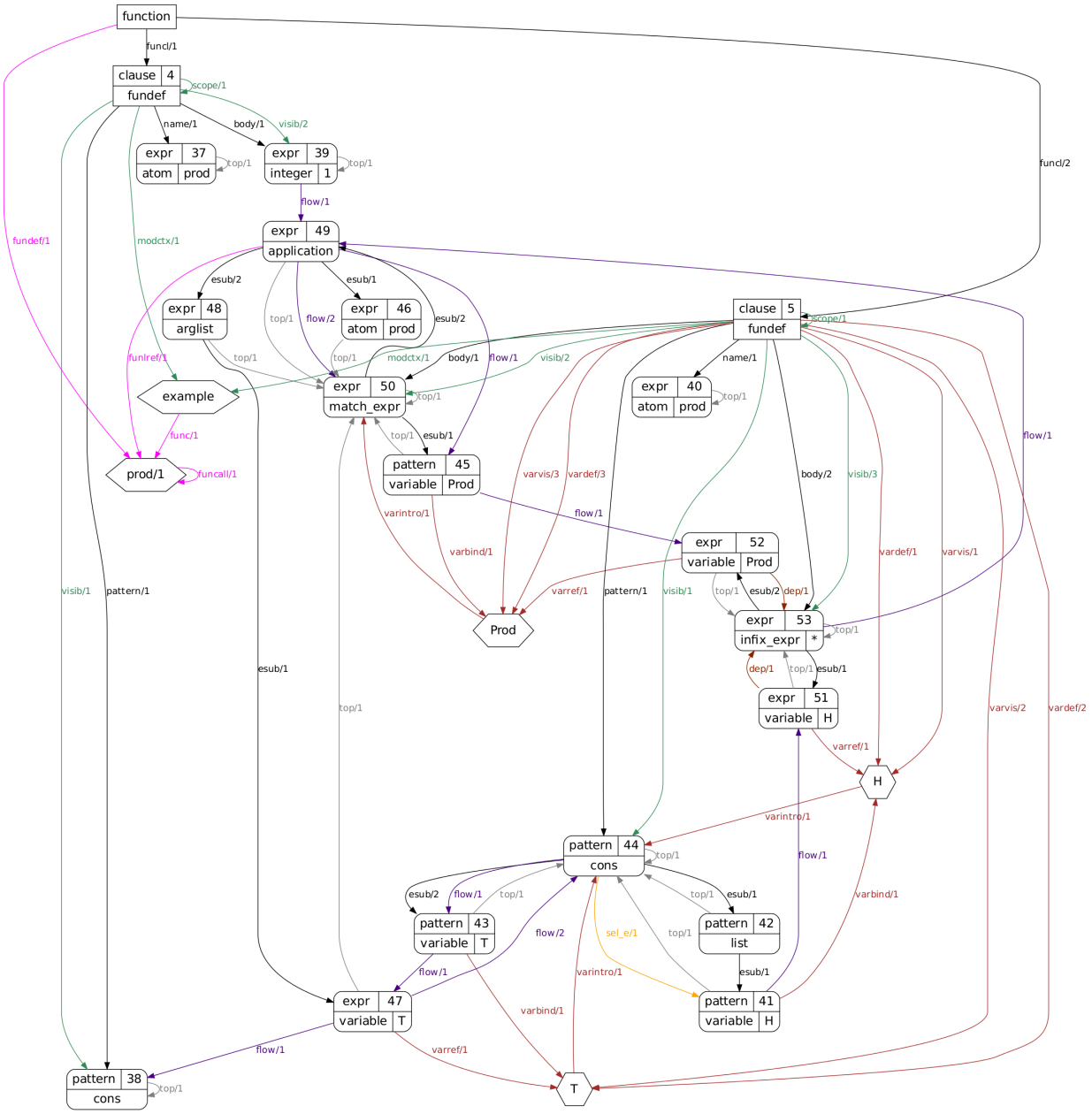
A példaprogramhoz tartozó szemantikus programgráf szintaktikus és szemantikus rétegét mutatja be a 2. ábra.

RefaktorErl-ben az adatfolyam gráf a szemantikus programgráf része, de különálló gráfként is tekinthetünk rá. Az adatfolyam gráf csúcsait az Erlang program kifejezéseinek szintaktikus csúcsai adják, az adatfolyam relációk pedig címkézett élként jelennek meg benne. A gráfban a következő élek jelennek meg:

- $e_1 \xrightarrow{flow} e_2$ — az e_2 értéke az e_1 értékének másolata,
- $e_1 \xrightarrow{dep} e_2$ — az e_2 értéke függ az e_1 értékétől,
- $e_1 \xrightarrow{cons_i} e_2$, $e_3 \xrightarrow{sel_i} e_4$ — az e_2 összetett adat konstrukciójánál az i-ik felhasznált érték az e_1, illetve az e_3 összetett adatból a szelektoron keresztül elérhető i-edik adat az e_4,
- $e_1 \xrightarrow{call_{g,i}} e_2$, $e_3 \xrightarrow{ret_{g,i}} e_4$ — függvényhíváson keresztül történő adatfolyam kapcsolatot jelöl, $call_{g,i}$ a g függvény i-ik aktuális paraméterének a formális paraméterbe másolását, míg $ret_{g,i}$ a g függvény i-ik eredményének visszamásolását.

A fenti éleket azaz alapvetően az adatfolyam gráfot az Erlang kifejezések közötti közvetlen adatfolyam kapcsolatot leíró formális szabályok alapján hozzuk létre. Illusztrációként két ilyen szabályt mutat be az 1. táblázat, ahol *e* egy Erlang kifejezést, míg *p* egy Erlang mintát jelöl.

Az adatfolyam gráf építésekor a szemantikus programgráfban szereplő mindegyik Erlang kifejezésre alkalmazzuk a hozzá tartozó szabályt.



2. Ábra: Szemantikus programgráf

Kifejezés	Adatfolyam élek
e_0: $p = e$	$e \xrightarrow{flow} e_0$ $e \xrightarrow{flow} p$
e_0: e_1 o e_2	$e_1 \xrightarrow{dep} e_0$ $e_2 \xrightarrow{dep} e_0$

1. Táblázat: Példa adatfolyam éleket létrehozó szabályokra

Az adatfolyam gráf mellett a vezérlésfolyam gráf a másik olyan eszköz, melynek elemzésével fontos információkat kaphatunk a program tulajdonságairól. A vezérlésfolyam gráf a program végrehajtási útvonalait tartalmazza a lehetséges bemeneti adatok alapján. A RefactorErl eszköz a vezérlésfolyam gráfot, az adatfolyam gráftól eltérően, a szemantikus programgráftól teljesen elkülönülten készíti el, viszont a felépítéséhez felhasználja az abban szereplő információkat. A gráfban, az adatfolyam gráfhoz hasonlóan, a csúcsok Erlang kifejezéseket, míg az élek különböző vezérlésfolyam kapcsolatokat reprezentálnak. A gráfban a következő élek jelennek meg:

- $\rightarrow$ — szekvencia,
- $\overset{yes}{\rightarrow}$, $\overset{no}{\rightarrow}$ — elágazás, (például *case* kifejezés),
- $\overset{ret}{\rightarrow}$ — visszatérés egy függvényből, *case* kifejezésből, stb.,
- $\overset{send}{\rightarrow}$, $\overset{rec}{\rightarrow}$ — üzenet küldés és fogadás,
- $\overset{call}{\rightarrow}$ — függvény hívás.

A vezérlésfolyam gráfot az adatfolyam gráf készítésénél alkalmazott formális szabályokhoz hasonló szabályok segítségével állítjuk elő. Egy ilyen szabályra mutat példát a 2. táblázat.

Kifejezés	Vezérlésfolyam élek
e_0: case e of p_1 when $g_1 \rightarrow e_{1,1}, \ldots, e_{1,k1}$; ⋮ p_n when $g_n \rightarrow e_{n,1}, \ldots, e_{n,kn}$ end	$e \rightarrow p_1$, $p_1 \overset{yes}{\rightarrow} g_1$, $p_1 \overset{no}{\rightarrow} p_2$, ⋮ $p_{n-1} \overset{yes}{\rightarrow} g_{n-1}$, $p_{n-1} \overset{no}{\rightarrow} p_n$, $p_n \overset{yes}{\rightarrow} g_n$, $p_n \overset{no}{\rightarrow} error$, $g_1 \overset{yes}{\rightarrow} e_{1,1}$, $g_1 \overset{no}{\rightarrow} p_2$, ⋮ $g_{n-1} \overset{yes}{\rightarrow} e_{n-1,1}$, $g_{n-1} \overset{no}{\rightarrow} p_n$, $g_n \overset{yes}{\rightarrow} e_{n,1}$, $g_n \overset{no}{\rightarrow} error$, $e_{1,1} \rightarrow e_{1,2}, \ldots, e_{1,k1-1} \rightarrow e_{1,k1}$, ⋮ $e_{n,1} \rightarrow e_{n,2}, \ldots, e_{n,kn-1} \rightarrow e_{n,kn}$, $e_{1,k1} \rightarrow ret\ case$, ⋮ $e_{n,kn} \rightarrow ret\ case$, $ret\ case \rightarrow e_0$

2. Táblázat: Példa vezérlésfolyam éleket létrehozó szabályra

Mint már említettük az elemző, illetve refaktoráló eszközök esetén is az egyik legnagyobb feladat a tesztelés esetén a megfelelő bemeneti adatok előállítása. Az általunk alkalmazott tesztelési módszer esetén ahelyett, hogy a megfelelően fedő teszteset halmaz létrehozásához sok eltérő tesztesetet próbálnánk meg definiálni, a kívánt programtulajdonságot kell formálisan specifikálnunk és egy generáló eljárást megadnunk, amellyel a megfelelő bemeneti adatok előállíthatóak. Így mivel ugyanazon tulajdonság felhasználásával számos tesztesetet létrehozhatunk, a tesztelés jóval

költséghatékonyabbá válik. Illetve nagyobb arányban jönnek létre, szokatlan, speciális, esetleg a rendszer által kevésbé kezelt esetek, mint kézzel írt tesztek esetén.

Ez a megközelítés különösen előnyös az olyan eszközök - mint például a RefactorErl - esetén, ahol a kívánt programtulajdonságok formális megadásához kiindulási pontként rendelkezésünkre áll az alkalmazott szabályok elvárt működését megadó, magas szintű specifikáció. Természetesen ezt az absztrakt leírást nem használhatjuk közvetlenül, át kell alakítanunk számítógéppel feldolgozható alakba. Ugyanakkor azt a tényt is kihasználhatjuk, hogy, mint említettük, az elemző és refaktoráló eszközök többsége (így a RefactorErl is) egy magas szintű belső reprezentációt használ, amelyen az elemző, illetve átalakító lépések dolgoznak, így a helyességvizsgálatukat is végezhetjük ezen a magas absztrakciós szinten.

Ezt a tesztelési módszert a RefactorErl esetében úgy hajtottuk végre, hogy a rendszer különböző elemző moduljaira vonatkozó formális specifikációkat átalakítottuk a felépített szemantikus programgráf, illetve a vezérlésfolyam gráf konzisztenciáját leíró tulajdonságokká és ezeket vizsgáltuk.

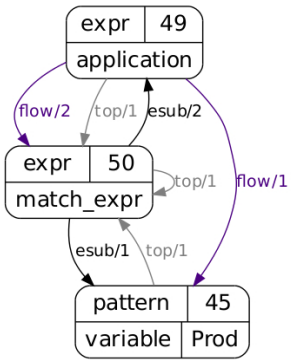
Mintaként tekintsük a változó kötést megvalósító mintaillesztéshez (match expression) tartozó adatfolyam szabályokat, melyeket az 1. táblázat első sora mutatja be. Ezeket a szabályokat a szemantikus programgráf szintjén a következőképpen tudjuk megfogalmazni:

- minden ilyen kifejezést reprezentáló csúcshoz, a csúcs második gyerekétől (amely a kifejezés második operandusát reprezentálja) vezetnie kell egy *flow* címkéjű élnek,
- ugyancsak a vizsgált csúcs második gyerekétől vezetnie kell egy *flow* címkéjű élnek a csúcs első gyerekéhez (amely a kifejezés első operandusát reprezentálja).

A 3. ábra illusztrációként bemutatja a példaprogramban felírt *prod* függvény szemantikus programgráfjának (melyet a 2. ábra mutat be) megfelelő részletét, amely a

```
Prod = prod(T)
```

kifejezéshez tartozik. A részgráfban az 50-es sorszámú csúcs reprezentál egy match expression kifejezést. Mivel a gráfban egy kifejezés operandusait az *esub* címkéjű éleken keresztül lehet elérni, ezért a fenti szabály a konkrét esetben azt jelenti, hogy vezetnie kell egy *flow* címkéjű élnek a 49-es sorszámú csúcsból (az 50-es sorszámú csúcs második gyereke) az 50-es sorszámú csúcsig, illetve szintén vezetnie kell egy *flow* címkéjű élnek a 49-es sorszámú csúcstól a 45-ös sorszámú csúcsig (az 50-es sorszámú csúcs első gyereke).



3. Ábra: Adatfolyam gráf részlete

Ezt a szabályt a RefactorErl belső reprezentációját felhasználva a következőképpen tudjuk Erlangban leírni. Össze kell gyűjtenünk az összes match expression kifejezést reprezentáló csúcsot és minden ilyenre ellenőriznünk kell a megfelelő tulajdonságot:

```
match_expr_test(Module) ->
    MatchExprNodes = getMatchExprNodes(Module),
```

```
        lists:map(fun props/1, MatchExprNodes).
```

Egy adott csúcsra pedig meg kell keressük a két gyerekét és ellenőriznünk, hogy létezik a két *flow* címkéjű él:

```
props(MatchExpr) ->
    case checkTwoChild(MatchExpr) of
       true -> Ch1 = getFirstCh(MatchExpr),
               Ch2 = getSecondCh(MatchExpr),
               check_matchExpr(MatchExpr, Ch1, Ch2);
       false -> {error_in_matchexpr, MatchExpr}
    end.

check_matchExpr(MatchExpr, Ch1, Ch2) ->
    Ch2Flows = getFlows(Ch2),
    case lists:member(Ch1, Ch2Flows) and
           lists:member(MatchExpr, Ch2Flows) of
        true  -> ok;
        false -> {error_in_matchexpr, MatchExpr}
    end.
```

A fenti kódban a *getFlows(x)* függvény az összes *x* csúcsból *flow* élen keresztül elérhető csúcsot adja meg.

Ily módon az elvárt tulajdonságot Erlangban, számítógéppel feldolgozható alakban is meg tudjuk adni, azaz például a QuickCheck (lásd [Cl00]) eszközt is fel tudjuk használni a RefactorErl tulajdonságainak automatikus ellenőrzésére. Ehhez viszont véletlenszerűen generált Erlang programokra van szükségünk tesztadatként.

Az előző fejezet végén már megemlítettük azt a lehetőséget, hogy típusos lambda kifejezések generálási lehetőségeit felhasználva, adott tulajdonságokkal rendelkező funkcionális programok állíthatóak elő. Azonban a RefactorErl QuickCheck segítségével végzett tesztelése során nem ezt az utat választottuk. Kihasználtuk a QuickCheck eszköz azon lehetőségeit, hogy a vizsgált tulajdonságban meg lehet adni az univerzálisan kvantált változókhoz a megfelelő generátorokat, a már meglévő generátorokból magasabb rendű generátor kombinátorok segítségével összetett generátorok készíthetőek, illetve, hogy a támogatott típusokhoz előre definiált generátorok vannak, melyek véletlenszerű értékeket képesek előállítani. A program reprezentálásához ebben az esetben egy absztrakt szintaxisfa (AST) alapú reprezentációt választottunk és megadtuk lehetséges Erlang AST-k halmazát egy attribútum grammatika segítségével. Végül a formális nyelvtan produkciós szabályait ültettük át QuickCheck generátorokká.

A tesztelés hatékonyabbá tételéhez olyan grammatikát írtunk fel, amely összetett adatfolyammal, többszörös kereszt hivatkozásokkal rendelkező programokat állít elő. Így a véletlenszerűen generált programok kellően összetett struktúrájúak lesznek, ami elősegítheti a rejtett hibák megtalálását.

Összességében elmondható, hogy a bemutatott tesztelési módszert könnyen át lehet ültetni más analizáló, illetve refaktoráló eszközre is, amennyiben az rendelkezik valamilyen magas szintű belső reprezentációval.

Kapcsolódó publikációk:

Máté Tejfel , Melinda Tóth , István Bozó , Dániel Horpácsi , Zoltán Horváth: **Improving quality of software analyser and transformer tools using specification based testing** *ANNALES UNIVERSITATIS SCIENTIARUM BUDAPESTINENSIS DE ROLANDO EOTVOS NOMINATAE*

SECTIO COMPUTATORICA 37: pp. 355-368. (2012)

Bozó I , Tóth M , Tejfel M , Horpácsi D , Kitlei R , Kőszegi J , Horváth Z: **Using impact analysis based knowledge for validating refactoring steps** *STUDIA UNIVERSITATIS BABES-BOLYAI SERIES INFORMATICA 56:(3)* pp. 57-64. (2011)

Bozó I , Horpácsi D , Horváth Z , Kitlei R , Köszegi J , Tejfel M , Tóth M: **RefactorErl - Source Code Analysis and Refactoring in Erlang** In: Jaan Phenjam (szerk.) *12th Symposium on Programming Languages and Software Tools*. Tallinn: TUT Press, 2011. pp. 138-148. (ISBN:978-9949-23-178-2)

Bozó I , Tóth M , Tejfel M , Horpácsi D , Kitlei R , Kőszegi J , Horváth Z: **Using impact analysis based knowledge for validating refactoring steps** In: Militon Frentiu , Horia F Pop , Simona Motogna (szerk.) *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2011, Selected papers*. 407 p. Cluj-Napoca: Presa Universitara Clujeana, pp. 325-336.

Horváth Z., Bozó I., Tóth M., Tejfel M.: **First Order Flow Analysis For Erlang** In: Horia F Pop , Antal Bege (szerk.) *8th Joint Conference on Mathematics and Computer Science: MaCS 2010* : Komárno, Slovakia, selected papers 2011. pp. 403-416. (ISBN:978-963-9056-38-1)

Kitlei R. , Bozó I. , Kozsik T. , Tejfel M., Tóth M.: **Analysis of preprocessor constructs in Erlang** In: Scott Lystig Fritchie , Konstantinos F Sagonas (szerk.) *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*. New York: ACM Press, 2010. pp. 45-55. (ISBN:9781450302531)

Tóth M., Bozó I., Horváth Z., Lövei L., Tejfel M., Kozsik T.: **Impact analysis of Erlang programs using behaviour dependency graphs** *LECTURE NOTES IN COMPUTER SCIENCE 6299 LNCS* pp. 372-390. (2010)

4. További kutatási témák

A fentieken leírtakon túl a PhD fokozat megszerzése óta számos - kisebb vagy nagyobb mértékben kapcsolódó - más témával is foglalkoztam. Ebben a fejezetben ezeket szeretném (csak nagyon vázlatosan) bemutatni.

4.1. Biztonságos mobil kód az Intelligens Térben

Az Intelligens Tér [Kor03] egy olyan terület (szoba, folyosó), amely rendelkezik különböző intelligens érzékelőkkel (kamerákkal, mikrofonokkal), illetve a tér megváltoztatására használt aktív, illetve passzív beavatkozó eszközökkel (mint például mobil ágensek, robotok, TV projektorok, hangszórók), ezáltal képes érzékelni és megérteni a térben zajló eseményeket és beavatkozni azokba. Az érzékelőktől származó, előfeldolgozott információt kommunikációs hálózatba kötött, egymással együttműködő, de autonóm robotok és komputerek használják fel, amelyek feladata különböző, akár dinamikusan változtatható feladatok elvégzése emberi környezetben.

A Tér akkor tekinthető intelligensnek, ha elosztott intelligenciájú hálózati eszközöket [Lee04] tartalmaz. Ezen eszközök rendelkeznek

- *érzékelő vagy beavatkozó* funkcióval, amellyel vagy információt gyűjtenek a tér állapotáról, vagy a tér állapotába avatkoznak bele,
- *feldolgozó* funkcióval, amely képes a szenzor(ok) adatait feldolgozni, vagy képes az adatok utófeldolgozására, illetve
- *kommunikációs* funkcióval, amely képessé teszi az eszközöket arra, hogy meg tudják osztani

a feldolgozásaik eredményeit a hálózaton más hasonló eszközökkel.

Megfelelő esetben például a fentiekben leírt intelligens kamerák, robotok ilyen hálózati eszközöknek tekinthetőek.

Kutatásunk keretében azt vizsgáltuk, miként lehet Intelligens Tér részeit képező, többféle feladatot ellátni képes mobil robotok működését biztonságossá tenni. A biztonságon itt azt értjük, hogy a robotok által végrehajtott kód formális helyességvizsgálaton esik át, melynek során előre megadott követelmények teljesülését ellenőrizzük. Ezen követelmények lehetnek például a következők:

- a robot nem mehet olyan helyre (távolságra) ahonnan nem tud visszajutni a töltőállomásáig,
- a robot nem mehet 5cm-nél közelebb a szobában lévő tárgyakhoz,
- az erőforrásigény (memória, idő, energiafelhasználás) bizonyos korlátok között kell maradjon,
- a robot nem mehet be bizonyos területekre (veszélyes helyek, lift, stb.).

Mivel a robotoknak képeseknek kell lenniük többféle feladat ellátására, ezért azt a megközelítést alkalmaztuk, hogy a roboton kezdetben csak egy keretprogram fut és az aktuálisan végrehajtandó feladatot megadó részkomponenseket, mobil kódként, dinamikusan töltjük le a robotokra. Így viszont még hangsúlyosabbá válik a biztonság kérdése.

Annak érdekében, hogy a robotok működése, a fent leírt értelemben biztonságos legyen még a mobil kód alkalmazása mellett is egy általános protokollnak, a CPPCC [Hor02] protokollnak a megadott környezetbe történő integrálása mellett döntöttünk. A protokoll alapján a mobil kód elkészítője formálisan ellenőrzi a kód tulajdonságait és a kód mellé csatol egy leírást a kód tulajdonságairól, illetve mellékeli azok bizonyításait is, majd az egész csomagot elküldi egy központi ellenőrző komponensnek. Az ellenőrző komponens megvizsgálja a kódot és a bizonyítást és amennyiben megfelelőnek találja egy új csomagot készít, amely a kódot, a tulajdonságait és az ellenőrző komponens tanúsítványát tartalmazza. Ezek után az új csomagot fel lehet tölteni egy mobil kód tároló komponensbe, ahonnan a kód letölthető a felhasználó komponens számára, aki megvizsgálja, hogy a program tulajdonságai megfelelőek-e számára, illetve, hogy jó-e, számára elfogadható-e a tanúsítvány és ha ezek teljesülnek, végrehajtja a kódot.

A protokollt olyan módon tudtuk integrálni a rendszerbe, hogy a kód tároló komponens része lett az Intelligens Térnek, a felhasználó komponens szerepét pedig a mobil robotok látják el. Ezen felül bevezettük azt a módosítást, hogy a feladatokhoz tartozó kódokat nem a robotok töltik le a tárolóból, hanem a felhasználók választhatnak a lehetséges feladatok és a rendelkezésre álló robotok közül és a robotok a saját, illetve az Intelligens Téren megosztott információk alapján (a kódhoz tartozó tulajdonságok és tanúsítvány figyelembevételével) hoznak döntést, hogy végre tudják-e hajtani a feladatot. A koncepció alkalmazhatóságát szimulációk segítségével bizonyítottuk.

Kapcsolódó publikációk:

Istenes Z. , Tejfel M., Jeni L. A.: **Verified Mobile Code Repository Simulator for the Intelligent Space** In: Attila Egri-Nagy , Emőd Kovács , Gergely Kovásznai , Gábor Kusper , Tibor Tómács (szerk.) *Proceedings of the 8th International Conference on Applied Informatics* January 27-30, 2010, Eger, Hungary . 456 p. 2011. pp. 79-86.
(ISBN:978-963-9894-72-3)

Jeni L. A., Istenes Z., Tejfel M., Korondi P., Hashimoto H.: **Adaptive, safe mobile robot programming in the Intelligent Space** In: Lucia Lo Bello , Giancarlo Iannizzotto (szerk.) *Human System Interaction. Proceedings of the 2nd International Conference HSI 2009*. IEEE Press, 2009. pp. 421-426.
(ISBN:978-1-4244-3959-1)

4.2 Konfigurálható adatstruktúrák többszintű memória hierarchiával rendelkező hardver architektúrákhoz

Számos olyan hardverarchitektúra létezik, amely, optimalizációs okokból, hierarchikus memóriaszerkezettel rendelkezik, azaz többféle, különböző méretű és hozzáférési idejű memóriatípus együttese adja a rendszer memóriáját. Költséghatékonysági és technikai okokból jellemzően minél gyorsabban hozzá lehet férni egy adott memóriatípusban tárolt elemekhez, annál kevesebb energia kell a memória használatához és annál kevesebb adott típusú memória áll rendelkezésünkre. Ebből adódóan az energiafelhasználást, illetve a futási időt figyelembe vevő optimalizációk esetén fontos kérdés, hogy az adataink közül melyeket tárolunk a gyorsabb elérésű memóriában.

Az említett architektúráknál sok esetben hardver-vezérelt cache memória használatával egészül ki a regiszter és a fő memóriahasználat, így hozva létre memóriahierarchiát. Ebben az esetben a programozónak nincs közvetlen befolyása arra, mely adatok kerüljenek a gyors memóriába és, hogy pontosan melyek lesznek ezek, az általában csak nehezen és nem teljes biztonsággal határozható meg előre.

A valósidejű, illetve a beágyazott rendszerek esetén, ahol az energiafelhasználás vagy a gyors válaszidő szigorú követelményei miatt ez a bizonytalanság nem megengedhető sokszor alkalmaznak szoftver-vezérelt gyors elérésű memóriát, úgynevezett scratchpad memóriát [Ban02]. Ebben az esetben teljes mértékben a programozó határozhatja meg, mely adatok és mikor kerüljenek be a scratcpad memóriába, illetve ki onnan. Ezek fontos tervezési döntések, melyek - mivel általában az ilyen rendszereket alacsony szintű nyelvek segítségével programozzák - a program teljes struktúrájára hatással lehetnek (például mely adatstruktúrák mely részeit lehet mutatókon keresztül elérni, vissza kell-e másolni egy értéket a távolabbi memóriába, ha igen mikor, stb.). Azonban sokszor további optimalizációk, illetve a program esetleg továbbfejlesztése, új funkciók bevezetése módosíthatja ezen döntéseket a program jelentős mértékű módosítását eredményezve.

Egy kutatásunk keretében azt vizsgáltuk, miként lehet az ilyen változtatások miatt szükségessé váló módosítások mértékét csökkenteni. Ennek érdekében memória konfigurációval ellátott adatszerkezeteket definiáltunk, melyek esetén az esetleges módosítások csak a konfiguráció megváltoztatásával járnak. Ezek a konfigurációk azt adják meg, hogy az adat mely részét lehet mutatókon keresztül elérni és azok a mutatók mely memóriaterületre mutatnak.

A konfigurációval rendelkező adatszerkezetek deklarálásához, memória foglalásához, felszabadításához és a bennünk tárolt adatok eléréséhez speciális, a konfigurációkat is paraméterül felhasználó utasításokat vezettünk be, így a konfiguráció megváltoztatásával a programban nem kell megváltoztatni ezeket az utasításokat.

A fent leírt koncepcióhoz négy különböző implementációt is készítettünk, bizonyítva annak alkalmazhatóságát. Létrehoztuk a konfigurálható adatszerkezeteket a Miller [Nem13] alkalmazási terület specifikus programozási nyelven, megvalósítottuk őket C makró könyvtárként és C++ template programok segítségével, illetve megmutattuk, hogyan lehet alkalmazni a konfigurációkat forrásról forrásra történő transzformációk végrehajtásával, az eredeti (konfigurációkat nem tartalmazó) kód változtatása nélkül, a konfigurációkat független leírásként a kódból elkészítve.

Kapcsolódó publikációk:

Karácsony M., Tejfel M.: **Configurable Data Structure Layout for Memory Hierarchies** előadás és beküldött cikk a 11th MACS konferencián, 2016, Eger

4.3 Optimalizált fordító a P4 nyelvhez

A hálózati eszközök (switch-ek, router-ek) esetén a legfőbb követelmény a sebesség, hiszen másodpercenként több millió csomagot is fel kell tudniuk dolgozni. Ebből adódóan általában erősen optimalizált, alacsony szintű programozási nyelven (jellemzően C vagy assembly) leírt, hardverfüggő programokat készítenek a kezelésükre. Az ilyen programok karbantartása és készítése azonban nagyon magas szakértelmet igényel és egy-egy új funkció hozzáadása a kód jelentős részének megváltozásával járhat, a hardver megváltozása pedig jellemzően a kód teljes újraírását igényli.

A fentiek tükrében nem meglepő, hogy jelentős igény mutatkozik olyan magas szintű programozási nyelv iránt, melynek segítségével hardverfüggetlenül le lehet írni a hálózati eszközöket vezérlő programokat, ugyanakkor lehetséges hozzá olyan (akár hardverfüggő optimalizációkat is végrehajtani képes) fordítóprogramot készíteni, amely hatékony kódot képes előállítani.

A P4 (Programming Protocol-Independent Packet Processors) [Bos14] egy magas szintű, platform- és protokollfüggetlen, alkalmazási terület specifikus nyelv, melynek segítségével hálózati elemek (switch-ek, router-ek) programozhatók. Egy aktuális kutatási projekt keretében, a nyelvhez egy olyan fordítóprogramot készítünk, mely támogat többféle célhardvert is, ugyanakkor hatékony, teljesítményében a C nyelven, kézzel írt programokkal összevethető teljesítményű C kódot állít elő.

Ezen cél eléréséhez a létrehozott fordítóprogram külön kezeli a program hardverfüggetlen, absztrakt elemeit (például csomag beolvasás, tábla konfigurálás) és a hardverfüggő, konkrét elemeket (például tábla lookup, fejléc mező megváltoztatása). A hardverfüggő elemeket egy hardver-absztrakciós könyvtár írja le, melyet minden hardver esetén csak egyszer kell elkészíteni, míg a hardverfüggetlen részek a P4 programból készített Core programban jelennek meg. A végső program a Core program és a megfelelő hardver-absztrakciós könyvtár összelinkelésével keletkezik.

A fordító a P4 nyelv szabadon hozzáférhető, hivatalos előfordítóját használja fel, amely egy Python nyelven megadott, magas szintű, köztes reprezentációt állít elő. A fordító ezt feldolgozva állítja elő a Core programhoz tartozó C kódot, amely több ponton hivatkozik a hardver-absztrakciós könyvtár elemeire.

Az eddigi esettanulmányaink alapján a hardverfüggő és hardverfüggetlen részek közötti határ megfelelő megválasztásával a fenti módszerrel hatékony kód állítható elő.

Kapcsolódó publikációk:

Laki S. , Horpácsi D., Vörös P., Kitlei R., Leskó D., Tejfel M.:**High speed packet forwarding compiled from protocol independent data plane specifications** In: *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, Florianopolis, Brazil, 2016. pp. 629-630.

Hivatkozások

[Ban02] Banakar,R., Steinke, S., Lee, B.-S., Balakrishnan, M.and Marwedel, P.: **Scratchpad memory: design alternative for cache on-chip memory in embedded systems.** Proceedings of the tenth international symposium on Hardware/software codesign. ACM, 2002. 73–78

[Bei90] Beizer, B.: **Software testing techniques (2nd ed.)**, New York, NY, USA, 1990

[Bos14] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. and Walker, D. **P4: Programming protocol-independent packet processors.** SIGCOMM Comput. Commun. Rev., 44(3):87–95, July 2014.

[Cl00] Claessen, K. and Hughes, J.: **Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs** *ACM SIGPLAN Notices* ACM Press, 2000, pp. 268–279.

[Ell00] Elliott, C., Finne, S. and d. Moor, O.: **Compiling embedded languages**, *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation, ser. SAIG '00*. London, UK: Springer-Verlag, 2000, pp. 9–27. Online: http://portal.acm.org/citation.cfm?id=645845.668623

[Fra88] Frankl, P. G., Weyuker, E. J.: **An Applicable Family of Data Flow Testing Criteria**, *IEEE Transactions on Software Engineering*, 1988, pp. 1483– 1498.

[Gri07] Grima, M., Pace, G.J.: **An embedded geometrical language in Haskell: construction, visualisation, proof.** *Computer science annual workshop,* 2007

[Ham07] Hammond, K., Grov, G., Michaelson, G. and Ireland A.: **Low-Level Programming in Hume: an Exploration of the HW-Hume Level** *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages (IFL'06)*, 2007, pp. 91-107., ISBN: 978-3-540-74129-9

[Hor02] Horváth, Z., Kozsik, T.: **Safe mobile code - CPPCC: Certified Proved- Property-Carrying Code.** G. Czajkowski and J. Vitek, Resource Management for Safe Languages (in: ECOOP 2002 Workshop Reader, LNCS 2548/2002, Springer-Verlag), 2002, pp. 8-10.

[Hor09] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg A., Nagy,T., Tóth, M., Király, R.: **Modeling semantic knowledge in Erlang for refactoring**, *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, volume 54 (2009)* Sp. Issue, Studia Universitatis Babes-Bolyai, Series Informatica, pp. 7–16, Cluj-Napoca, Romania, July, 2009

[Hud98] Hudak, P.: **Modular domain specific languages and tools**, *Proceedings of the 5th International Conference on Software Reuse, ser. ICSR '98.* Washington, DC, USA: IEEE Computer Society, 1998, pp. 134–142. Online: http://portal.acm.org/citation.cfm?id=551789.853532

[Kam98] Kamin, S., N.: **Research on domain-specific embedded languages and program generators**, *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[Kme15] Kmett, E., A.: (2015, Sep.) **Lens: Lenses, Folds and Traversals**. Online: https://hackage.haskell.org/package/lens

[Kor03] Korondi, P., Hashimoto, H. **Intelligent Space, as an Integrated Intelligent System**, Keynote paper of International Conference on Electrical Drives and Power Electronics, Proceedings, 2003, pp. 24-31.

[Lee04] Lee, J., Morioka, K., Ando, N., Hashimoto, H. **Cooperation of Distributed Intelligent Sensors in Intelligent Environment**, IEEE/ASME Transactions on Mechatronics, Vol.9, No.3, pp.535-543, 2004.09, ISSN 1083-4435

[Lei99] Leijen, D. and Meijer, E.: **Domain specific embedded compilers**, *SIGPLAN Not., vol. 35*, pp. 109–122, December 1999. Online: http://doi.acm.org/10.1145/331963.331977

[Li08] Li, H. and Thompson, S.: **Testing Erlang Refactorings with QuickCheck**, *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pp. 19–36, Springer-Verlag, 2008

[Nem13] Németh, B. and Csörnyei. Z.: **Stackless programming in Miller.** Acta Universitatis

Sapientiae, Informatica 5.2, 2013. 167–183.

[Nem15] Németh, B.: (2015, Sep.) **References: Generalization of lenses, folds and traversals to handle monads and addition.** Online: https://hackage.haskell.org/package/references

[Nie92] Nielson, H. R., Nielson, F.: **Semantics with applications: a formal introduction**, John Wiley & Sons, Inc., New York, NY, USA, 1992

[Run08] Runciman, C., Naylor, M. and Lindblad, F.: **Smallcheck and lazy smallcheck: automatic exhaustive testing for small values** *Proceedings of the first ACM SIGPLAN symposium on Haskell*, ser. Haskell '08. New York, NY, USA: ACM, 2008, pp. 37–48.

[She02] Sheard, T. and Jones, S., P.: **Template meta-programming for Haskell** *ACM SIGPLAN Notices*, Volume 37 Issue 12, 2002, pp. 60-75.

[Wang05] Wang, J.: **Generating random lambda calculus terms**. Technical report, Boston University, 2005.