

Forrás szintű hibakeresés támogatása beágyazott nyelvek esetén

Béres Ferenc Gergely

MSC 2. évfolyam

Témavezetők:

Dévai Gergely

Egyetemi tanársegéd

Dr. Tejfel Máté

Egyetemi adjunktus

Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és

Fordítóprogramok Tanszék



Eötvös Loránd Tudományegyetem

Informatikai Kar

2012.

Tartalomjegyzék

Tartalomjegyzék	1
1. Bevezetés	2
1.1. A dolgozatról	3
2. Hibakeresés	4
2.1. Hibakeresést segítő eszközök	5
2.2. Hibakeresési technikák	6
3. Beágyazott programozási nyelvek	8
3.1. Beágyazott programozási nyelvek tervezése	9
3.2. Beágyazott alkalmazásterület specifikus programozási nyelvek	10
3.3. A probléma	11
4. A módszer	12
4.1. Alapfogalmak és előfeltételek	12
4.2. A beágyazott nyelvünk kibővítése	13
4.3. A módszer	14
5. A fordítási folyamat lépései	16
5.1. Gazda reprezentáció manipulálása	16
5.2. Kódgenerálás	17
6. Egy lehetséges implementáció	20
6.1. While nyelv beágyazása	20
6.2. Beágyazott While kiterjesztése	25
7. Eredmények	38
Irodalomjegyzék	40

1. fejezet

Bevezetés

A szoftverfejlesztés során a programozók idejük számottevő részét töltik az általuk írt programok hibamentesítésével. Egy programot jelen esetben akkor tekintünk hibásnak, mikor a futás közben egy abnormalis viselkedés következtében terminál. Ilyen abnormalis viselkedés lehet például nem élő változóra hivatkozás, vagy olyan gépi utasítás használata, mely nem található az adott hardveren. A hibakereső eszközöknek alapvetően két fajtájuk van.

- Forrás szintű, mely program terminálás esetén pontosan meg tudja mondani, hogy a vezérlés folyamán forráskód mely pontján terminált a program.
- Alacsony szintű, mely csak a forrásból fordított assembly kódban tudja megmondani a hiba helyét.

Nem lehet eléggé hangsúlyozni a hibakeresési tevékenységet támogató módszerek, illetve eszközök hasznosságát. Ebből kifolyólag számos fejlesztőkörnyezet esetén léteznek a környezetbe integrált, a program futását megfigyelő, illetve szabályozó eszközök. Ezek általában lehetőséget adnak a forráskódban törési pontok elhelyezésével az adott pillanatban létező változók, objektumok értékének megfigyelésére, illetve a program futásának megállítására, továbbindítására. Ezen túl léteznek fejlesztő környezetektől független eszközök melyekkel hasonló működést érhetünk el. Ezek általában támogatják a forráskód szintű hibakeresést.

Az utóbbi években egyre népszerűbbek lettek a beágyazott programozási nyelvek. Beágyazott nyelvek fejlesztése sokkal gyorsabban történik egy önálló fordítóprogrammal rendelkezőhöz képest, mivel az felhasználja a gazdanyelv lexikális és szintaktikus elemzőjét, típus ellenőrzőjét. Viszont cserébe a beágyazott nyelv fordítóprogramja csak egy konstans adatstruktúrát lát, nem fér

hozzá semmilyen információhoz a beágyazott forráskódhoz kapcsolódóan. Az, hogy egy beágyazott programozási nyelv is hozzáférhessen ezekhez a hibakereséshez elengedhetetlen információkhoz egy nem triviális, komplex feladat.

1.1. A dolgozatról

Ahogy a bevezetésből kiderült, beágyazott programozási nyelveknél nem tudunk egyértelmű leképezést mondani a beágyazott forráskód és a belőle generált tárgykód között. Dolgozatomban egy módszert adok, mely segítségével, kibővítve egy létező beágyazott nyelvet, már lesz lehetséges a forrás szintű hibakeresés támogatása. Alapötlet az, hogy a gazdanyelv fordítóprogramjának szintaktikus elemzőjével lefordítjuk a beágyazott forráskódot. Az így előállított szintaxisfában már megtalálhatóak a számunkra szükséges sor és oszlop információk amelyeket el kell tárolni a beágyazott nyelv belső reprezentációjában. Következő lépésként a nyelv fordítóprogramját úgy kell módosítani, hogy a fordítóprogram a kódgenerálás során, miközben eltárolja a beágyazott szintaxisfa adott részéből generált szöveg pozícióját, összepárosítja azt a hozzá tartozó forráskód pozíciókkal, melyet a gazdanyelv szintaktikus elemzője biztosít számunkra.

A dolgozat felépítése a következő. A második fejezetben hibakeresési módszereket, illetve ebben a témakörben született cikkeket ismertetek. A harmadik fejezetben a beágyazott programozási nyelvek fejlesztésével, illetve azok sajátosságaival foglalkozom. A negyedik fejezetben definiálásra kerül néhány általam használt fogalom, valamint ismertetem, hogy milyen bővítéseket kell eszközölni egy beágyazott programozási nyelven, hogy az már támogatni tudja a forrás szintű hibakeresést. Az ötödik fejezet részleteibe menően foglalkozom a beágyazott nyelv fordítóprogramját érintő változtatásokkal. A hatodik fejezetben egy konkrét példán keresztül mutatom be a problémára általam felvázolt megoldást, végül pedig a módszerrel kapcsolatos eredményeimet ismertetem.

2. fejezet

Hibakeresés

Egy programban rejtőző, illetve futás közben történő hibák felderítése és azok kijavítása a szoftverfejlesztés legnehezebb feladata. Nem hiába a programozók egy szoftver fejlesztésére szánt idő túlnyomó részét azok javításával töltik, és nem ritka az a jelenség se, mikor a kész, tökéletesnek tűnő program nem várt működést produkál. Ez az oka annak, hogy egy kész szoftver átadása után legalább ugyanannyi, ha nem több időt szokás a program hibáinak javítására, mint további funkciókkal való bővítésére fordítani. Igaz manapság a helyzet jobb, hiszen a különböző módszertanok segítségével csökkenteni tudjuk az esetleges hibák számát, ilyen például a rendszeres automatizált tesztelés. Ezzel azt tudják vizsgálni, hogy adott bemenő paraméter esetén a megfelelő kimenetet produkálja-e a program. Ezt és az ehhez hasonló módszereket nevezik a programok verifikálásának, amely mostanra önálló kutatási területté nőtte ki magát. Hasonló lehetőség a modell ellenőrző használata. Ebben az esetben a programunk alapján egy modellt készítünk, majd különböző állításokat fogalmazhatunk meg vele kapcsolatban. Ezen állítások leírása általában temporális logika segítségével történik. Az ellenőrző pedig különböző heurisztikák segítségével megpróbál a modellből az állítást megcáfoló működést produkálni. Amennyiben nem talál ilyen esetet, a programunkat helyesnek ítéli. Egy hasonló tudományos módszer a programok szintézise. A módszer lényege, hogy kezdeti specifikációs feltételekből kiindulva, azokat folyamatosan finomítva próbálunk meg bizonyítottan helyes programot előállítani. Ez lenne az ideális megközelítés, azonban ez rendkívüli szakértelmet igényel, nem is beszélve arról, hogy egyszerűbb programok helyességének a bizonyítása is terjedelmes lehet, komplexebb rendszerek esetén ez szinte lehetetlen feladatnak számít.

2.1. Hibakeresést segítő eszközök

Kivételkezelés

Nem hiába, a programozási nyelvek fejlesztői próbálnak minnél több eszközt adni a programozók kezébe, mellyel segíteni tudják őket a hibakeresésben. Modern programozási nyelveknél például nyelvi elemmé vált a kivételkezelés, melynek a lényege a következő. Amennyiben olyan kód részletet írunk, melynek futása során hiba váltható ki, például olyan fájlt szeretnénk megnyitni olvasásra amely nem is található a fájlrendszerben, akkor azt egy try-catch vagy try-catch-finally blokkba helyezzük. Try-ba kerül a hibát kiváltható program részlet, a catch ágban található kódrészlet kivétel kiváltása esetén hajtódik végre. Természetesen itt van lehetőségünk egyszerre több kivétel lekezelésére is. Az opcionális finally blokkban található kódok pedig minden esetben végrehajtásra kerülnek. Jó példa erre a hálózati kommunikáció, hiszen akár sikeres vagy sikertelen a kommunikációs csatorna kiépítése, utána mindenképpen le szeretnénk zárni azt a kapcsolatot.

Fejlesztőkörnyezetek

Ahogy azt a bevezetőben említettem, a legtöbb fejlesztőkörnyezet már integrált hibakeresési eszközökkel kerül forgalomba, és lehetőséget ad külön hibakeresési üzemmódban futtatni programjainkat. Mielőtt így futtatnánk a programunkat lehetőségünk van úgy nevezett töréspontok elhelyezésére a forráskódban. Amennyiben ilyen helyre ér a vezérlés a program futása felfüggesztődik és lehetőségünk lesz az adott pillanatban élő változók és objektumok értékeinek vizsgálatára. Ezt követően folytathatjuk a végrehajtást egész addig, amíg újabb törési ponthoz nem érkezünk. Röviden ezen környezetekben lehetőség van már a program végrehajtását is valamilyen szinten befolyásolni. Továbbá ha a fejlesztői környezetből futtatjuk a kódunkat és a futás során abnormalis viselkedés következtében az terminál, egy külön konzolra kiüríti a végrehajtási verem aktuális tartalmát, vagy egy lekezeletlen kivétel esetén megmutathatja, hogy az a kód mely pontját váltódott ki. Fontos megjegyezni, hogy egyes keretrendszerek esetén külön névtérben található osztályok sokasága áll a programozó rendelkezésére melyek feladata kimondottan a hibakeresés megkönnyítése.

Debuggerek

A debuggerek eggyel alacsonyabb szinten tudnak dolgozni mint a fejlesztői környezetekbe integrált hibakeresők. Ezek ugyanis nem tudnak és nem is feltételezhetnek semmit a vizsgálandó kóddal kapcsolatban. Persze ez csak az általános debuggerekre igaz, léteznek külön nyelvekhez implementált szoftverek, a [2] cikkben például speciálisan az Ada taskok hibamentesítését segítő eszköz implementálását részletezik a szerzők. Tehát a debuggerek általában a megfigyelendő programhoz rendelt memória területet illetve annak végrehajtási vermét tudják figyelemmel kísérni. Természetesen itt is van lehetőség a program végrehajtásának manipulálására.

2.2. Hibakeresési technikák

Amennyiben nem áll rendelkezésre hibakereső szoftver, úgy is van lehetőség az általunk írt forráskód hibamentesítésére. Ebben az alfejezetben a teljesség igénye nélkül néhány technika kerül ismertetésre.

printf()[1]

Ez a hibakeresési módszer a legegyszerűbb mind közül. Lényege csupán annyi, hogy a forráskód különböző pontjain elhelyezett függvények segítségével a szabványos kimenetre, amennyiben lehetőségünk van rá, akkor a szabványos hiba kimenetre szöveget, esetleg változók és objektumok értékeit írjuk ki. Ennek a módszernek jelentős gyengeségei a következők:

- Ez a módszer ad-hoc. Ideiglenesen bővítjük a kódunkat, amint megszűnt a hiba eltávolítjuk, majd a következő hibánál újra bővítjük stb.
- Jelentősen megnövelheti a program futási idejét.
- Gyakran nem is hasznos. A szabványos kimenet puffertelt csatorna, tehát mikor a program terminál az összes benne lévő információ elveszik.

Fontos, hogy amennyiben van lehetőségünk, akkor mindenképp a szabványos hibakimenetet használjuk, hiszen az a többivel ellentétben nem puffertelt, így kevésbé valószínű, hogy a programunk esetleges összeomlása esetén elvesztenénk a kiírni kívánt információt.

Távoli hibakeresés

Ezt a hibakeresési módszert általában beágyazott rendszerek esetén[3] használják, de manapság komolyabb fejlesztőkörnyezetek is támogatják. Ebben az esetben a futó program és a hibakereső eszköz nem ugyanazon a számítógépen találhatóak. Ezért a hibakeresést végző személynek hálózaton keresztül kapcsolódnia kell a programot futtató rendszerhez, majd a saját gépén keresztül tudja annak futását ellenőrizni illetve manipulálni, amennyiben szükséges.

Postmortem hibakeresés

Ilyen technikákat akkor alkalmaznak, miután a program abnormális működés következtében terminált. Ilyenek lehetnek például a visszalépéses technikák[4] vagy a memória szemét vizsgálatával foglalkozó módszerek. Nagy segítséget nyújthatnak ebben az esetben az egyes monitorozó alkalmazások vagy akár magának a rendszernek a naplófájljai is.

3. fejezet

Beágyazott programozási nyelvek

Beágyazott programozási nyelvek alapötlete a következő: válasszunk egy gazdanyelvet, amely gazdag nyelvi eszközkészlettel rendelkezik, mint például típus ellenőrzés, függvény és adat absztrakció, modularitás, és gyarapítsuk ezt további adattípusokkal, illetve függvényekkel, melyek kizárólag ilyen típusú értékekkel dolgoznak. Ezek után a nyelv fejlesztőinek nem kell implementálniuk ezen szolgáltatásokat, mert ezeket a gazdanyelv biztosítja a számunkra, így a nyelvünk belső felépítése nem bonyolódik el, mondhatni minimális lesz, és könnyebben karbantarthatóbbá, valamint gyorsabban fejleszthetővé válik. A gazdanyelvekre jellemző, hogy ezek általában általános célú programozási nyelvek, mint a C, C++, Haskell. Egy nyelvet kétféleképpen tudunk beágyazni: sekélyen és mélyen. A sekély beágyazás azt jelenti, hogy megadunk egy beágyazott nyelvi kifejezést, majd annak a kimenetele egy szöveg, a tárgykód. Nincs köztes reprezentáció, tehát ebben az esetben a beágyazott nyelvünkön semmilyen optimalizációt nem tudunk végrehajtani. Például az `a + b` kimenetele az `"a + b"` szöveg mint tárgykódbeli kifejezés. Mély beágyazás esetén a kifejezésből, a gazdanyelv szemszögéből, egy adatstruktúra épül. Az előző példát megvizsgálva, az `a + b` kifejezése értéke most már lehet `Add (Var "a") (Var "b")`, ahol az `Add` illetve a `Var` jelen esetben már a beágyazott nyelvünk reprezentációjából származó adatkonstruktorok. Így azonban már képesek vagyunk különböző transzformációkat illetve optimalizációkat végrehajtani. Sok beágyazott nyelv létezik, ilyen például a Lex és Yacc[5], PL/SQL[6] és a Feldspar[7]. Természetes módon ezek után felmerül bennünk a kérdés, hogy mélyen beágyazott nyelvek esetén hogyan ismerjük fel, hogy nem a gazdanyelven programozunk? A válasz roppant egyszerű: a nyelv saját típusokkal dolgozik. Például egy Feldspar függvényt onnan ismer fel a programozó, hogy a függvény nem a Haskellbe beépített `Int` típust használja, hanem egy `Data a`

típust, ahol az a már egy tetszőleges Haskellbe beépített alap típus lehet.

```
map :: (a -> b) -> [a] -> [b]
-- Haskell-ben implementált map függvény.
map :: (Data a -> Data b) -> Data Vector a -> Data Vector b
-- Feldsparban implementált map függvény.
```

A fenti példában bemutatott map egy magasabb rendű függvény. Ez onnan látható, hogy egy olyan függvényt vár paraméterként, mely a típusú értékeket b típusú értékekre képezi le. A map függvény megfelel az elemenkénti alkalmazásnak. Azaz az első paraméterként kapott függvényt elemenként alkalmazza a második paraméterként kapott listára, az így készült lista lesz a függvény visszatérési értéke.

3.1. Beágyazott programozási nyelvek tervezése

Mostantól mélyen beágyazott nyelvekkel foglalkozunk. Mélyen beágyazott nyelvek esetén első körben definiálnunk kell a beágyazott reprezentációnkat. Tegyük fel, hogy egy tetszőlegesen bonyolult aritmetikai műveleteket leíró nyelvet szeretnénk beágyazni Haskellbe, akkor első lépésben definiálnunk kell egy, a műveleteket reprezentáló adattípust. Ebben az esetben a reprezentációnk így fog kinézni.

```
data Arithmetic a where
  Plus :: (Arithmetic a) -> (Arithmetic a) -> (Arithmetic a)
  Minus :: (Arithmetic a) -> (Arithmetic a) -> (Arithmetic a)
  Multiply :: (Arithmetic a) -> (Arithmetic a) -> (Arithmetic a)
  Const :: (Num a, Show a) => a -> Arithmetic a
```

Tehát a nyelvünk beágyazott reprezentációját a Plus, Minus, Multiply, Const adatkonstruktorok tetszőlegesen mély egymásba ágyazásával építhetjük fel. Látható, hogy az Arithmetic a típus általánosított algebrai adattípusként került definiálásra. Ennek jelen esetben csak az az oka, hogy így megszorítást tehetünk a Const adatkonstruktorra, mely így már csak olyan értékekből tud típusértékeket előállítani, melyekre alkalmazhatóak a Num a típusosztályban definiált (aritmetikai) függvények illetve a Show a-ban definiált show függvény, melynek feladata a paraméterül kapott típusérték szöveggé alakítása.

Tehát eddig a beágyazott nyelvünkben lehetőség van két aritmetikai kifejezés összeadására, kivonására, összeszorzására valamint konstans értékek definiálására. Szükség van egy fordítóprogramra is, amely a fent definiált repre-

zentációt tárgykódra képezi le. Jelen esetben nincs nehéz dolgunk, hiszen a fordítónk eképpen fog kinézni.

```
compile :: (Num a, Show a) => Arithmetic a -> String
compile (Const const) = show const
compile (Plus lhs rhs) = (compile lhs) ++ "+" ++ (compile rhs)
compile (Minus lhs rhs) = (compile lhs) ++ "-" ++ (compile rhs)
compile (Multiply lhs rhs) = (compile lhs) ++ "*" ++ (compile rhs)
```

A példában bemutató compile függvény a paraméterül kapott beágyazott reprezentációt bejárva, a Plus, Minus, Multiply konstruktorok esetén az operandusokra rekurzívan meghívja önmagát, hogy az azokból generált tárgykódot konkatenálja az általuk reprezentált művelet szöveges megfelelőjével. A rekurzió a Const adatkonstruktornál megáll, ebben az esetben a visszatérési értékünk a benne található konstans érték szöveges reprezentációja.

Továbbá szükségünk van olyan függvényekre, amelyek elrejtik a programozó elől a belső reprezentációkat. Még mindig nincs nehéz dolgunk.

```
(+) :: Arithmetic a -> Arithmetic a -> Arithmetic a
(+) lhs rhs = Plus lhs rhs
(-) :: Arithmetic a -> Arithmetic a -> Arithmetic a
(-) lhs rhs = Minus lhs rhs
(*) :: Arithmetic a -> Arithmetic a -> Arithmetic a
(*) lhs rhs = Multiply lhs rhs
```

Amennyiben a Num a típusosztályt példányosítjuk a saját típusunkra, akkor például konstans egész értékekkel meghívva az aritmetikai műveleteinket is képesek leszünk tárgykódot generálni. Továbbá itt már lehetőségünk is lenne a nyelvünk szemantikáját megadni, hiszen definiálhatnánk az egyes függvények precedenciáját illetve asszociativitását. Ezek után már írhat is a programozó tetszőlegesen bonyolult aritmetikai kifejezéseket, és azokat le is fordíthatja. Tovább fejleszthetnénk a nyelvünket például kiértékelő funkcióval, de ezt most mellőzzük. Vegyük észre, hogy alig pár sor gépelésével már van egy programozási nyelvünk. Nem kellett se lexikális elemzőt, szintaktikus elemzőt, típus ellenőrzőt stb. írunk, mert ezeket a gazdanyelv, jelen esetben a Haskell, biztosítja számunkra!

3.2. Beágyazott alkalmazásterület specifikus programozási nyelvek

Alkalmazásterület specifikus nyelveken azokat értjük, melyek egy előre meghatározott problémakör könnyebb megoldását célozva kerülnek fejlesztésre,

így egyéb területen kevésbé, vagy egyáltalán nem is használhatóak. Alkalmazásterület specifikus nyelv például a Cryptol[8], mely kimondottan kriptográf algoritmusok könnyebb implementálását segíti, vagy a Feldspar[7] amelyet jel-feldolgozó algoritmusok hatékony implementálására fejlesztettek. Ezen nyelvek egyértelmű előnyei az általános célú nyelvekkel szemben, hogy magasabb absztrakciós szintet biztosítanak a programozók számára, így az ilyen nyelven írt forrásszövegek tömörebben megfogalmazhatók, könnyebben karbantarthatók, könnyebb transzformációkat végezni rajtuk, illetve a helyességüket bizonyítani. Nem utolsó sorban, az adott terület szakértői is könnyebben megértik a programozók által írt kódot. Ilyen nyelvek is íródhatnak saját, önálló fordítóprogrammal, ám az ok, amiért ebben a fejezetben külön szót kell ejteni róluk, hogy manapság egyre több alkalmazásterület specifikus programozási nyelv kerül implementálásra beágyazott nyelvként. Ezeket beágyazott alkalmazásterület specifikus nyelvnek, röviden EDSL-nek (Embedded Domain Specific Language) nevezzük. Ezek fejlesztéséről sok érdekes cikk született, mint például [9] [10] [11]. Az ok nagyon egyszerű: rendkívül gyorsan lehet egy prototípust kifejleszteni a gazdanyelv által szolgáltatott funkciók felhasználásával, valamint így a magasabb absztrakciós szintet is könnyebb biztosítani.

3.3. A probléma

Ahogy a fejezet bevezetésében már említettük, egy új programozási nyelvet beágyazott nyelvként implementálni megkönnyíti a nyelv fejlesztőinek a dolgát. Viszont hiába nem kell magunknak leimplementálni egy saját lexikális illetve szintaktikus elemzőt, mivel a gazdanyelv ezeket biztosítja számunkra, így a fordítóprogramunk már nem is fog hozzáférni semmilyen információhoz a beágyazott forrásszöveggel kapcsolatban (például a szintaktikus egységek forrásban elhelyezkedő pozícióihoz), így a forrás szintű hibakeresés első körben nem tűnik triviális feladatnak. Ezért a nyelv fejlesztőinek más egyéb eszközöket kell kifejleszteniük, amikkel segíteni tudják a programozókat a hibakeresésben. A hibakeresés beágyazott nyelvek esetén nyitott kutatási terület, tudomásom szerint ebben a témában tudományos cikk még nem született. Dolgozatomban erre a problémára vázolok fel egy lehetséges megoldást.

4. fejezet

A módszer

A dolgozatomban egy módszert adok a beágyazott programozási nyelvek forrás szintű hibakeresésének támogatására. Ebben a fejezetben első lépésként definiálásra kerül néhány fogalom, valamint egy megkötés a gazdanyelvre vonatkozóan. Ezt követően pedig lépésről lépésre felvázolom azokat a módosításokat melyeket a saját nyelvünkön kell elvégeznünk.

4.1. Alapfogalmak és előfeltételek

A bemutatott módszer esetén azzal a feltételezéssel élünk, hogy lehetőségünk van a gazdanyelv fordítóprogramjának szintaktikus elemzőjéhez külső könyvtárként hozzáférni. Amennyiben ez nem lehetséges akkor találnunk kell egy olyan függvény csomagot, mely lehetőséget ad a kiválasztott gazdanyelvű forrás szöveg lexikális és szintaktikus elemzésére, valamint rendelkezésünkre bocsátja a belőle generált szintaxisfát. Ez több okból is fontos számunkra :

- Mivel a nyelvünk nem rendelkezik lexikális és szintaktikus elemzővel, így csak a gazdanyelv fordítóprogramjának segítségével tudunk hozzáférni a hibakeresés szempontjából elengedhetetlen sor és oszlop információkhoz.
- Valamint így a gazdanyelv reprezentációján keresztül lehetőségünk van manipulálni a beágyazott nyelven írt forráskódunkat. Erről egy későbbi fejezetben részletesebben fogunk szót ejteni.

Gazdanyelv reprezentációja alatt a beágyazott nyelven írt forráskódunknak a gazdanyelv fordítóprogramja által előállított szintaxisfáját értjük. Hasonló fogalom a **beágyazott reprezentáció**, melyen pedig forráskódunknak a be-

ágyazott nyelvi reprezentációját értjük.

Innentől feltételezzük, hogy van egy beágyazott nyelvünk, amely rendelkezik

- Saját belső reprezentációval
- Magasszintű függvény- és/vagy osztálykönyvtárral
- Fordítóprogrammal, mely a beágyazott reprezentációból tárgykódot generál

4.2. A beágyazott nyelvünk kibővítése

Első lépésben egy olyan nyelvi elemet kell biztosítanunk, amelynek segítségével tárolni tudjuk a nyelvünk egy nyelvi elemének elhelyezkedését a beágyazott forrásszövegben. Ehhez minden egyes nyelvi konstrukciót (vezérlési szerkezeteket, kifejezéseket, változókat) ki kell bővítenünk egy csomagoló nyelvi elemmel, ami a forrásszövegbeli sor és oszlop pozíciók, valamint egy adott nyelvi konstrukció segítségével egy újabbat állít elő. Egyszerű példaként, ha a nyelvünkben lehetőség van ciklusok, valamint feltételes utasítások definiálására, akkor a beágyazott reprezentációnkat leíró nyelvtan, mint matematikai struktúra a következőképpen néz ki.

$$G = \langle N, T, P, S \rangle, ahol$$

$$N = \{S, L\}$$

$$T := \{\text{while, do, if, then, else, skip, and, or, false, true}\}$$

$$P := \{S \rightarrow \text{while } L \text{ do } S$$

$$| \text{if } L \text{ then } S \text{ else } S$$

$$| \text{skip}$$

$$, L \rightarrow L \text{ and } L$$

$$| L \text{ or } L$$

$$| \text{true}$$

$$| \text{false}\}$$

$$G' = \langle N', T', P', S' \rangle, ahol$$

$$N' := \{S', L'\}$$

$$T' := T \cup \{\text{labelS, labelL, position}\}$$

$$P' := \{S' \rightarrow \text{while } L' \text{ do } S'$$

$$| \text{if } L' \text{ then } S' \text{ else } S'$$

$$| \text{skip}$$

```

| labelS position  $S'$ 
,  $L' \rightarrow L'$  and  $L'$ 
|  $L'$  or  $L'$ 
| true
| false
| labelL position  $L'$ 

```

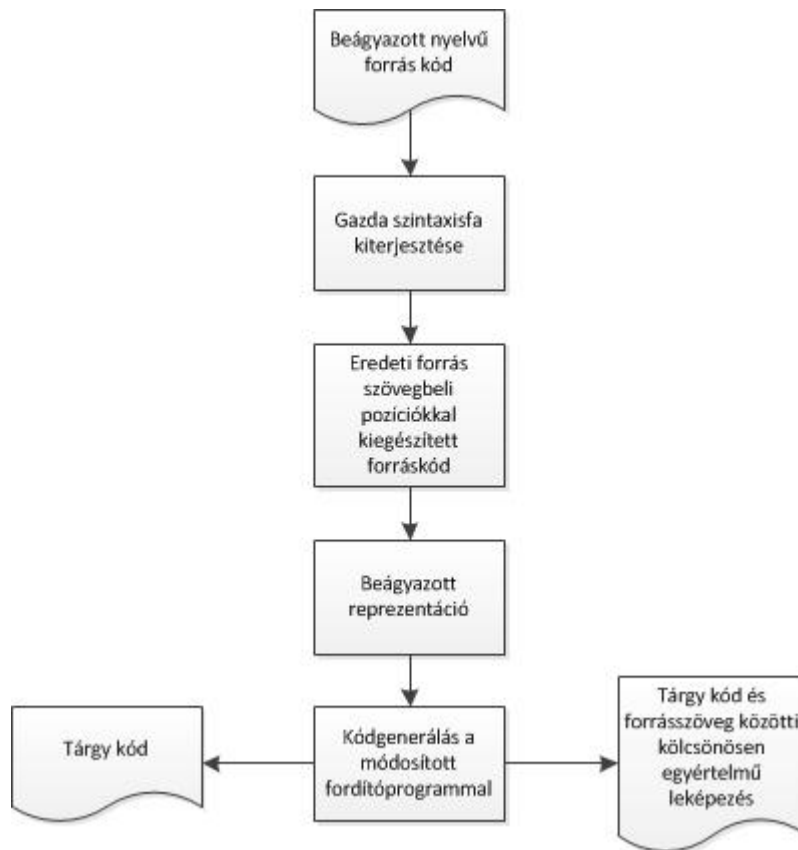
Az S nemterminális szimbólum determinisztikus programot, míg az L nemterminális szimbólum logikai kifejezéseket jelölnek. Tehát a G nyelvtan alapján egy ciklus az alábbi módon épül fel: while terminális szimbólumot egy logikai kifejezést követ, ez a ciklus feltétel, majd a do terminális szimbólum után egy determinisztikus program következik. A G' módosított nyelvtan által generált $L(G')$ nyelvnek egy lehetséges mondata a `labelS position skip`, ahol a `positions` nemterminális szimbólum jelöli majd a beágyazott nyelvünk adott nyelvi elemének a forráskódbeli pozícióját. Következő lépésként a felhasználók számára biztosított magas szintű függvényeinket kell kibővítenünk egy csomagoló függvénnyel, mely a forráskódbeli sor és oszlop pozíciókból, valamint egy nyelvi elemből előállít egy ugyanolyan típusú nyelvi konstrukciót. Objektum orientált nyelvek esetén generic-ek illetve a template-k, funkcionális nyelveknél pedig a típusosztályok pont a megfelelő szolgáltatást nyújtják számunkra. Természetesen a csomagoló függvényünket az összes lehetséges nyelvi konstrukcióra implementálnunk kell.

Ezek után a kódgenerálás módosítása következik. Egyrészt a kimenetünk már nem csak a tárgykód, hanem egy kölcsönösen egyértelmű leképezés is a generált tárgykód, valamint a beágyazott nyelvű forráskód között, valamint maga a kódgenerálás is módosul hiszen már nyilván kell tartanunk, hogy az általunk generált kódrészletek hol helyezkednek el. Erről a következő fejezet ide vonatkozó alfejezetében részletesebben beszélünk.

4.3. A módszer

Az előbb említett bővítéseket végrehajtva a beágyazott nyelvünkön a hibakeresést támogató fordítás folyamata a következőképpen kerül végrehajtásra.

- A beágyazott forráskódunkon lefuttatjuk a gazdanyelv fordító program-



4.1. ábra. A fordítási folyamat folyamatábrája

ját.

- A gazda reprezentációt mélységében bejárva úgy módosítjuk azt, hogy a csomópontokat kibővítjük egy olyan közbenső részfával, amely egy csomagoló függvényt reprezentál, feltöltve azt a gazdanyelv szintaxisfájából kinyert sor és oszlop információkkal.
- A módosított szintaxisfát szöveggé alakítjuk, mely így mind a gazdanyelv, mind a beágyazott nyelv szempontjából legális kódot reprezentál.
- Az így kapott forrásszöveget újrafordítjuk, így a beágyazott nyelvünk fordító programja számára már értelmezhető adatstruktúrát kapunk.
- Az adatstruktúrára meghívjuk a beágyazott nyelvünk kibővített fordító programját, amelynek kimenete immáron a tárgykód mellett egy kölcsönösen egyértelmű leképezést megvalósító adatstruktúra, a generált kód, valamint a beágyazott nyelvű forráskód között.

A fent leírt fordítási folyamatot szemlélteti a 4.1 ábra.

5. fejezet

A fordítási folyamat lépései

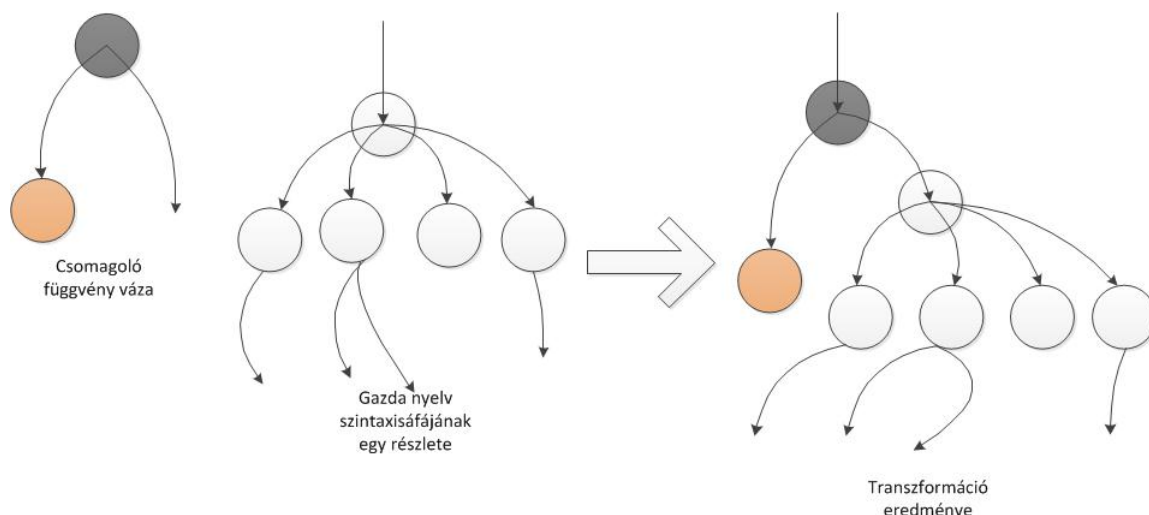
Ebben a fejezetben a fordítási folyamatot komolyabban érintő változásokat nézzük végig. Először megvizsgáljuk, hogy hogyan lehet a gazdanyelvünk belső reprezentációját úgy módosítani, hogy annak forráskóddá visszaalakítása után mind a gazdanyelv, mind a beágyazott nyelvünk szempontjából érvényes kódot kapjunk. Érvényes kód alatt a gazdanyelv szempontjából a szintaktikusan és típusosan helyes, beágyazott nyelv szempontjából pedig az általunk definiált nyelvi konstrukcióknak megfelelő kódot értjük. Következő lépésként a kódgenerálásban bekövetkező változásokat vizsgáljuk meg.

5.1. Gazda reprezentáció manipulálása

Fontos megjegyeznünk az elején, hogy a különböző paradigmájú programozási nyelveknek nem csak a belső reprezentációjuk, de azok logikai felépítése is merőben eltér egymástól. Ezért első lépésként fontos megértenünk, hogy a kiválasztott gazdanyelv szintaktikus elemzője hogyan építi fel a saját szintaxisfáját, utána a manipulálás logikája könnyen implementálható.

Gazda reprezentáció felcímkézése

Miután megértettük a gazdanyelv szintaxisfa építésének a logikáját két segédműveletet kell definiálnunk. Az egyszerűbb segédművelet feladata, a szintaxisfánk adott csúcsának forrásszövegben való elhelyezkedő pozíció információinak kinyerése. Következő függvény dolga már kicsit összetettebb. Fő feladata



5.1. ábra. A transzformációt bemutató ábra

egy részfa építése, ami a csomagoló függvényt reprezentálja, majd annak befűzése a gazda reprezentációba. Ennek az elvégzéséhez szükségünk van egyrészt az adott csúcsra, valamint az adott csúcsból származó információkra. Ezen két művelet segítségével egyszerűen definiálható az általunk használni kívánt manipuláció. Minden becsomagolni kívánt csúcsnál felépítjük a részfat, amely a csomagoló függvényünket reprezentálja, a részfánk egyik gyereke a forrás kódban található pozíciókat fogja tartalmazni, másik gyereke pedig az adott részfa gyökere. Nem utolsó sorban az általunk épített részfa szülőjének beállítjuk a transzformáció adott lépésénél található csúcspont szülőjét. A transzformációt az 5.1 ábra szemlélteti.

Mivel a fenti transzformációval nem sértjük meg a gazdanyelv szintaktikai szabályait, illetve mert a beágyazott nyelvünk belső reprezentációját, illetve a programozó felé biztosított magas szintű függvény könyvtárát kibővítettük a megfelelő csomagoló nyelvi konstrukciókkal, ezért amennyiben a fenti transzformáció segítségével kibővített szintaxisfából szöveget generálunk, az mind a gazdanyelv, mind a beágyazott nyelvünk szemszögéből legális kódot kapunk.

5.2. Kódgenerálás

Ahogy korábban említettük, a kódgenerálás jelen esetben összetettebbé válik, hiszen most már nem csak az adott csúcsból kell nekünk tárgykódot generálni, hanem egyéb tevékenységeket is kell végeznünk.

- Ha az adott csúcsból tárgykód kerül generálásra, akkor ki kell számíta-

nunk hogy ő éppen az eddig már generált kódhoz képest hol helyezkedik el.

- Meg kell valósítanunk egy kölcsönösen egyértelmű leképezést a tárgykód és a beágyazott forráskód között.

Ahogy látjuk így már a fordítás ezen részén egy adott csúcsnak több információra is szüksége van.

- Tudnia kell, hogy mekkora mértékű indentálással kell elkezdenie a kódgenerálást.
- Tudnia kell, hogy melyik pozícióról kell elkezdenie a kódgenerálást.
- Tudnia kell, hogy melyik pozíción fejezte be a kódgenerálást.

Ezeket az információkat tovább kell adni a csúcspontok között, de némelyiket, például az indentálás mértékét minden gyerekének meg kell kapnia, viszont a kódgenerálást más pozícióról kell kezdeniük.

Ezen adatok struktúrálását illetve továbbterjesztését

Hajós Márton diplomamunkája[13] alapján dolgoztam ki. A kódgenerálás elején definiálásra kerülnek lefelé terjedő információk, állapotok, illetve felfelé terjedő információk. Egy adat lefelé terjedő információ, amennyiben egy csúcs minden gyereke ugyanazt kapja meg, ilyen az indentálás mértéke. Egy adat állapot, amennyiben a gyerek a szülőtől kapja és abból új értékeket számol, ilyen például a kódgenerálás kezdő pozíciója. Felfelé terjedő információ az amit a szülő a gyerek csúcstól kap és azt nem módosítja, csak a sajátjait számolja ki belőle, ilyen az adott csúcsból generált forráskód, a kódgenerálást befejező és befejező pozíciók, valamint a már meghatározott tárgy- és forráskód közötti kapcsolatok listája.

A szintaxisfa bejárása

A beágyazott reprezentáció bejárása mélység szerint történik. Kezdetben indentálás nem szükséges és a kódgenerálás a első sor első oszlopától kezdődik. A csúcsonál amennyiben az nem csomagoló csúcs, akkor ott kódot generálunk, és a szülőtől kapott lefelé terjedő, valamint állapot információ segítségével új állapotot számolunk magunknak, illetve meghatározzuk a felfelé terjedő információkat. Csomagoló csúcspontok esetében kód nem generálódik, csak a már kiszámolt tárgykódbeli pozíciók, illetve a gazdanyelv szintaktikus elemzője segítségével kinyert információk kerülnek összepárosításra. Tárgykódbeli pozíciókat a többi csúcs esetén számolunk.

Tárgykódbeli pozíciók meghatározása

A generálkód pozícióinak meghatározása egyszerűen történik. A kezdősor és oszlop pozíciókat a szülőcsúctól kapjuk meg állapot jellegű információként. Befejező pozíciók meghatározására az alábbi algoritmust használom.

- Az utolsó sor az a kezdő sor és a generált kódban található sortörést reprezentáló escape szekvenciák összege.
- Az utolsó oszlop a generált kód végétől az utolsó sortörést reprezentáló escape szekvenciág található karakterek száma.

Utolsó lépésként az általunk generált szöveget hozzáillesztjük a már az állapotban lévőhöz, a szülőtől kapott pozíciókat pedig lecseréljük az általunk kiszámoltakkal.

Csomagoló csomópontok

A csomagoló csúcspontoknál nem történik kódgenerálás. Itt a gyerek csúctól kapott tárgykódbeli kezdő és befejező pozíciókat reprezentáló rendezett négyest összepárosítja a gazdanyelv fordítóprogramjától szerzett forrásbeli pozíciókkal, majd a felfelé terjedő információk között található kölcsönösen egyértelmű leképezést megvalósító lista elejére fűzi.

6. fejezet

Egy lehetséges implementáció

Dolgozatom ezen fejezetében az eddig felvázolt módszer egy lehetséges implementációját mutatom be. Első lépésként egy egyszerű programozási nyelv kerül mélyen beágyazásra Haskellbe. A kiválasztott nyelv a While[14]. Bemutatásra kerül a nyelv beágyazott reprezentációja, a programozó számára biztosított magas szintű függvények, valamint egy beágyazott programkód, illetve a belőle generált tárgykód.

6.1. While nyelv beágyazása

A While nyelv

A While egy egyszerű imperatív programozási nyelv. A nyelvben lehetőségünk van aritmetikai, illetve logikai kifejezések írására, egész típusú változók definiálására. Egyszerű utasításként lehetőségünk van változóknak értékül adni tetszőleges aritmetikai kifejezéseket, valamint írhatunk üres utasítást, szekvenciákat, elől tesztelős ciklusokat, illetve feltételes utasításokat. A While nyelv definíciója a következő.

```
S ::= x := a
    | s1; s2
    | If L Then s1 Else s2
    | While L do S
    | Skip

L ::= a1 == a2
    | a1 < a2
    | l1 or l2
    | l1 and l2
```

```

    | not l
  A ::= a1 + a2
    | a1 - a2
    | a1 * a2
    | a1 / a2

```

A fent használt definícióban használt jelölések jelentése a következő.

- S, s1, s2 : determinisztikus program
- L, l1, l2 : logikai kifejezések
- A, a1, a2 : aritmetikai kifejezések, vagy egész típusú értékek

Belső reprezentáció

Első lépésként a fent leírt nyelvtant kell Haskellben algebrai adattípusként definiálnunk. Szerencsére ez nem okoz túl nagy nehézséget. Egy lehetséges implementáció lehet a következő.

```

data Program where
  (:=)      :: Variable a -> Expr a -> Program
  Declare   :: [Variable a] -> Program
  Sequence  :: [Program] -> Program
  Loop      :: Expr Bool -> Program -> Program
  IfThenElse :: Expr Bool -> Program -> Program -> Program
  Skip      :: Program

data Expr a where
  --aritmetikai kifejezések
  Plus      :: Expr Int -> Expr Int -> Expr Int
  Minus     :: Expr Int -> Expr Int -> Expr Int
  Mul       :: Expr Int -> Expr Int -> Expr Int
  -- logikai kifejezések
  And       :: Expr Bool -> Expr Bool -> Expr Bool
  Or        :: Expr Bool -> Expr Bool -> Expr Bool
  Neg       :: Expr Bool -> Expr Bool

  Compare   :: Expr Int -> Expr Int -> Expr Bool
  Eq        :: Expr a -> Expr a -> Expr Bool

  Value     :: (Show a) => a -> Expr a
  Var       :: Variable a -> Expr a

data Variable a where
  Variable :: Name -> Variable a

type Name   = String

```

A fenti kóddal kapcsolatban fontos megjegyezni, hogy nem algebrai adattípusként lett definiálva a beágyazott reprezentációnk, hanem általánosított algebrai adattípusként. Ennek oka a következő: amennyiben egyszerű algebrai adattípusként definiálnák a Program típust, akkor ott meg kéne jelennie egy további típus változónak. Tehát a programjaink maguk típusosak lennének, így viszont ezt könnyen elkerülhetjük. Továbbá ha a kifejezéseket egységesként szeretnénk kezelni, azaz nem szeretnénk külön típust definiálni a logikai, illetve az aritmetikai kifejezéseknek, akkor is hasonlóképpen kell eljárunk.

Programozók számára biztosított felület

Ahogy a beágyazott nyelvekkel foglalkozó fejezetben említettük, szükség van egy olyan felületre, amely elrejtí a programozó elől a belső reprezentációkat és kényelmes használhatóságot biztosít a számukra. Jelen esetben néhány típusosztály segítségével került definiálásra ez az interfész.

```
class Compare a where
  (<) :: a -> a -> Expr Bool

class Equal a where
  (==) :: a -> a -> Expr Bool

class Logic a where
  (&&), (||) :: a -> a -> Expr Bool
  (!)      :: a -> Expr Bool
```

Amennyiben a fenti típusosztályokat példányosítjuk az Expr Bool, valamint a Haskell által biztosított Bool típusokra, akkor már kényelmesen írhatunk logikai kifejezéseket a beágyazott nyelvünkön. Felmerülhet azonban a kérdés, hogy miért nem a Haskell által már biztosított hasonló típusosztályokat példányosítjuk a sajátjainkra? A válasz egyszerű : a választott gazdanyelv által definiált függvények szignatúrája nem megfelelő a számunkra. Példaként a Haskell Prelude moduljában definiált egyenlőségvizsgálat függvény szignatúrája a következő.

```
class Eq a where
  (==) :: a -> a -> a
```

Könnyen láthatjuk, hogy két logikai érték esetében nincs lehetőségünk azt beágyazott nyelvi konstrukcióként értelmezni. Más a helyzet azonban az aritmetikai műveleteknél. A Num a típusosztály megvalósításához négy függvényt definícióját kell megadnunk, melyek a következők.

```
instance Num (Expr Int) where
  (+) = Plus
  (-) = Minus
  (*) = Mul
  fromInteger i = Value (fromInteger i)
```

Amennyiben a fent leírt módon járunk el, már írhatunk Integer típusú értékeket beágyazott nyelvünk kifejezésekként, mert azokat majd automatikusan konvertálódnak számunkra megfelelő konstans értékke.

Fordítóprogram

Egy beágyazott While nyelven írt függvény fordítására a `compile` függvényt használhatja a programozó, melynek a definíciója a következő.

```
compile :: Program -> IO ()
compile p = putStrLn $ generate p 0
```

A függvény tehát egy `Program` típusú adatstruktúrát vár paraméterül, eredménye pedig a belőle generált tárgy kód. Látható, hogy jelen esetben a fordítóprogramunk nincs elbonyolítva, csak a kódgenerálásból áll. Ennek oka, hogy nem csak a nyelvnél, de annak a fordítóprogramjánál is a minimális bonyolultságra törekedtem.

A kódgenerálásért felelős típusosztályok definíciói a következők.

```
class CodeGenerator a where
  generate :: a -> Int -> String

class (CodeGenerator a) => CodeGenerator1 a where
  generate' :: [a] -> Int -> String
  generate' [] _ = ""
  generate' (x:[]) indent = cSource where
    ((), cSource) = flip runState "" $ do
      code $ generate x indent
      code $ "\n"

  generate' (x:xs) indent = cSource where
    ((), cSource) = flip runState "" $ do
      code $ generate x indent
      code $ generate' xs indent
```

A `CodeGenerator` típusosztály egy függvényt, a `generate`-t tartalmazza melynek paraméterei rendre a következők.

- A beágyazott szintaxis fa csúcsa.
- A kódgenerálásnál alkalmazott indentálás mértéke.

A `CodeGenerator1` típusosztályra csak azért van szükségünk, hogy szekvenciák és változó deklarációkból is tudjunk tárgykódot generálni. A tárgykód jelen esetben C kód részletek lesznek, mely alatt olyan forráskódot értek, ami önmagában még nem fordul le, viszont amennyiben körbe vesszük azt a `main` függvénnyel, akkor már legális C kódot kapunk. A dolgunk tehát nem más, mint a belső reprezentációban definiált algebrai adattípusokra példányosítani a típusosztályainkat. A következő kódrészletek mutatnak erre példát.

```
instance CodeGenerator Program where
  generate (:=) (Variable name) expr indent = cSource where
    ((), cSource) = flip runState "" $ do
      code $ indenter indent ++ name
      code $ " = " ++ generate expr 0 ++ ";\n"
  ...

instance CodeGenerator (Expr a) where
  generate (Plus lhs rhs) _ = cSource where
    ((), cSource) = flip runState "" $ do
      code $ "(" ++ generate lhs 0 ++
        " " ++ generate rhs 0 ++ ")"
  ...
```

Értékadás esetén a kódgenerálás menete a következőképpen zajlik. Megfelelő indentálás után leírjuk annak a változónak a nevét amelynek értéket szeretnénk adni, majd egy értékadó egyenlőséget, majd legeneráljuk az értékadás jobb oldalán álló kifejezésnek a kódját. Fontos, hogy magában a kifejezésben nem szeretnénk indentálni, ezért annak a mértéke nulla. Hasonló a logikája két aritmetikai kifejezés összeadását reprezentáló csúcsból történő kódgenerálásnak. Ennek eredménye egy nyitó zárójel, majd az összeadás bal oldalát reprezentáló részfából generált kódrészlet, csukó zárójel, összeadás jel, nyitó zárójel, az összeadás jobb oldalát reprezentáló részfából generált kódrészlet végül egy újabb csukó zárójel.

Példa kódok

Tehát van már beágyazott reprezentációnk, magas szintű függvényeink, továbbá kódot is tudunk generálni. A nyelv első változata elkészült, tehát lehetőségünk van már beágyazott While nyelven programokat írni. Példának okáért, két egész szám legnagyobb közös osztóját kiszámoló algoritmus egy implementációja a következő.

```
gcd :: Program
gcd = (Declare [x, y]) ++
  (Loop
```

```
((!)(x == y))
(IfThenElse
  (x < y)
  (y := ((Var y) - (Var x)))
  (x := ((Var x) - (Var y)))
)
)

x :: Variable Int
x = Variable "x"

y :: Variable Int
y = Variable "y"
```

Ebből a forráskódból a következő tárgykód lesz a fordítóprogramom kimenete.

```
int x;
int y;
while(!(x == y))
{
    if((x < y))
    {
        y = (y) - (x);
    }
    else
    {
        x = (x) - (y);
    }
}
```

6.2. Beágyazott While kiterjesztése

Készen van a beágyazott nyelvünk, ám a programozó számára nincs még semmiféle lehetőség hibakeresésre. Az eddig felvázolt módszer segítségével kibővítésre kerül a beágyazott nyelvem, az ezzel kapcsolatos implementációs kérdésekkel foglalkozik az alfejezet. Első lépésben módosítjuk a beágyazott reprezentációt, illetve a felhasználók felé biztosított felületünket. Ezt követően megvizsgáljuk, hogy a gazdanyelv, jelen esetben a Haskell, hogyan építi fel a szintaxisfáját, illetve abból milyen módszerrel tudjuk kinyerni a forrás szövegben elhelyezkedő nyelvi elemek pozícióit, majd annak függvényében implementálásra kerül a gazdareprezentációt manipuláló transzformáció. Végző lépésként a fordítóprogramom kódgenerátorát egészítem ki úgy, hogy a tárgyszöveg generálása során feljegyzésre kerüljön az adott belső reprezentációból generált tárgykód abszolút pozíciója.

Beágyazott reprezentáció, és a magas szintű függvények kiegészítése

Ahogy azt a módszer tárgyalásának ide vonatkozó alfejezetéből kiderült, a beágyazott reprezentáció módosítása csak annyiból áll, hogy minden általunk definiált adattípust kiegészítünk egy olyan adat konstruktorral, ami egy rendezett négyesben tárolja a forrás szövegbeli pozícióját annak a csúcspontnak, amit ő becsomagol. Jelen esetben három adatkonstruktorral bővül a belső reprezentációnk.

```
type SrcLoc = ((Int, Int), (Int, Int))

data Program where
  ...
  LabProg :: SrcLoc -> Program -> Program

data Expr a where
  ...
  LabExpr :: SrcLoc -> Expr a -> Expr a

data Variable a where
  Variable :: Name -> Variable a
  LabVar   :: SrcLoc -> Variable a -> Variable a
```

Ezen túl szükségünk van olyan függvény(ek)re, amelyek a megfelelő típusú csúcspontot állítják elő a reprezentációkhoz. Ehhez definiálásra kerül egy `Label` nevű típusosztály, amely csak a `label` nevű függvény szignatúráját határozza meg. Ezt kell majd példányosítanunk az összes általunk definiált algebrai adattípusra.

```
class Label a where
  label :: ((Int, Int), (Int, Int)) -> a -> a

instance Label Program where
  label = LabProg

instance Label (Expr a) where
  label = LabExpr

instance Label (Variable a) where
  label = LabVar

instance Label [Variable a] where
  label = labelList

instance Label [Program] where
  label = labelList
```

```

labelList :: (Label a) => ((Int, Int), (Int, Int)) -> [a] -> [a]
labelList srcLoc []      = []
labelList srcLoc (x:xs) = (label srcLoc x):(labelList srcLoc xs)

```

Példányosítás során a csomagoló függvény visszatérési értéke minden esetben a példányosítani kívánt adattípusban definiált csomagoló adatkonstruktor.

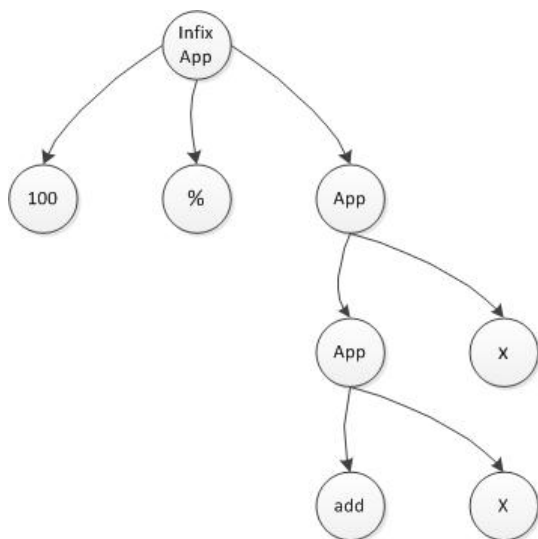
Gazda szintaxisfa manipulálása

A Haskell fordítóprogramja nem rendelkezik olyan reflexiószerű szolgáltatással, mely lehetővé tenné, hogy futási időben információkat szerezzünk a forráskódunkról, mert az így sértené a nyelv tisztán funkcionális voltát. Szerencsére létezik egy csomag, melynek segítségével a Haskell nyelvű forrásszövegünket szintaktikusan elemezni tudjuk. Annak a programnak a kimenete egy Haskell belső reprezentációval megegyező szintaxis fa, amit manipulálni tudunk, illetve rendelkezésünkre bocsátja a nyelvi elemek forrásbeli pozícióit. Ezek után első dolgunk megérteni, hogy miképp épül fel a Haskell belső reprezentációja. Ezt egy egyszerű példán mutatom be.

```

myFunction :: Int -> Int
myFunction x = 100 % add 3 x

```



6.1. ábra. A myFunction függvény definíciójának egyszerűsített szintaxisfája

A Haskell fordítóprogramja természetesen a myFunction függvényt teljes mértékben, típus szignatúrával együtt szintaktikusan elemzi és az is bekerül a szintaxisfába, de nekünk jelen esetben elég megérteni a függvény definíció jobb oldalát reprezentáló részfat, ezt a 6.1 ábra szemlélteti.

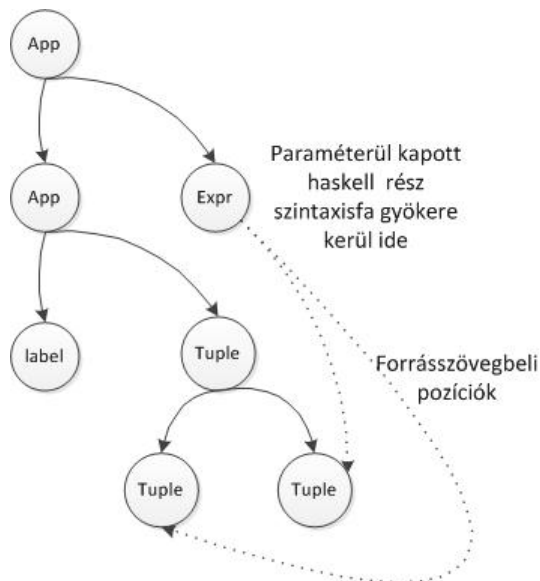
Első lépésként definiálásra kerül egy segédfüggvény, ami egy csúcspontból kiolvassa a számunkra szükséges forrásbéli információkat. A segédfüggvényünk szignatúráját a `Location` a típusosztály rögzíti. Példaként az applikációkat reprezentáló csúcspontokra vonatkozó implementáció az alábbi módon kerül példányosításra.

```
class Location a where
  getStartLine :: a SrcSpanInfo -> Int
  getStartCol  :: a SrcSpanInfo -> Int
  getEndLine   :: a SrcSpanInfo -> Int
  getEndCol    :: a SrcSpanInfo -> Int

instance Location Exp where
  ...
  getStartLine (App inf _ _) = srcSpanStartLine $ srcInfoSpan inf
  getStartCol  (App inf _ _) = srcSpanStartColumn $ srcInfoSpan inf
  getEndLine   (App inf _ _) = srcSpanEndLine $ srcInfoSpan inf
  getEndCol    (App inf _ _) = srcSpanEndColumn $ srcInfoSpan inf
  ...
```

Következő lépésként vegyük jobban szemügyre a `myFunction` függény `add`-ra vonatkozó részfáját, már látjuk is, hogy miképpen kell kézzel felépítenünk a `label` függvényünk reprezentációját. Az ezzel foglalkozó függvény egyszerűsített kódja a következőképpen néz ki.

```
wrap :: Exp SrcSpanInfo -> Exp SrcSpanInfo
wrap exp =
  (App
    (App
      (Var
        (UnQual
          (Ident "label")
        )
      )
    )
    (Tuple
      [Tuple
        [Lit (Int (toInteger $ getStartLine exp))
        ,Lit (Int (toInteger $ getStartCol exp))]
      ,Tuple
        [Lit (Int (toInteger $ getEndLine exp))
        ,Lit (Int (toInteger $ getEndCol exp))]
      ]
    )
  )
  exp)
```



6.2. ábra. A myFunction függvény definíciójának egyszerűsített szintaxisfája

A könnyebb érthetőség érdekében a fenti kódot, azaz magát a manipulációt, az 6.2 ábra szemlélteti.

Természetesen még nem vagyunk kész, hiszen eddig csak egy csúcsra vonatkozóan adtuk meg a transzformációt. Annak tényleges elvégzése a `Transform` a típusosztályban definiált `transform` függvény feladata. A típusosztály definíciója, illetve annak az applikációkra vonatkozó példányosítása a következőképpen néz ki.

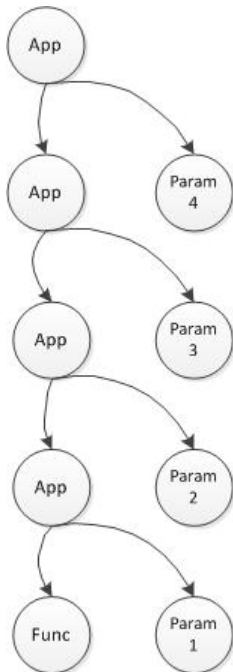
```
class Transform a where
  transform :: (a SrcSpanInfo) -> (a SrcSpanInfo)

instance Transform Exp where
  ...
  transform x@(App _ _ _) = wrap $ transformRec x
  ...

transformRec (App inf f x) = App inf (transformRec f) (transform x)
transformRec x = transform x
```

Fontos megjegyezni, hogy a Haskell belső reprezentációjában nincsenek több változós függvények. De mint tudjuk, funkcionális nyelvekben sincsenek többváltozós függvények, hiszen azokat Curry-féle módszerrel több, egy változós függvények egymásra való alkalmazásával, applikációjával az eredetivel ekvivalens módon leírhatjuk. Példának okáért egy Haskellben definált, négy változós függvény szintaxisfája látható a 6.3 ábrán.

Ezek után már érthető az applikációk esetén a `transform` függvény definíciója,



6.3. ábra. Négy változós függvény szintaxisfájának a váza Haskellben.

és látható, hogy az így definiálva tetszőlegesen sok paraméterrel rendelkező függvény esetén működik.

Fontos még megjegyeznünk, hogy mivel nem sértettük meg a Haskell szintaktikai szabályait, valamint a `label` függvény minden számunkra szükséges adattípusra definiálásra került, ezért mind a gazdanyelvünk és a beágyazott nyelvünk szemszögéből legális kódot kapunk.

Fordítóprogram

Jelenleg a beágyazott reprezentációnkba sikerült eltárolnunk a forráskódban írt nyelvi elemek elhelyezkedését, viszont még nem tudjuk, hogy a generált kódban azok hol helyezkednek el. Tehát a feladatunk jelen esetben, hogy kódgenerálással egy időben számon tartsuk az aktuális csúcspontból generált tárgykód abszolút elhelyezkedését. Ahogy azt a módszer általános leírásánál említettük itt a kódgenerálás során már szükségünk van több információra is. A felfelé terjedő információkat egy rekordban tároljuk, mely definíciója a következő.

```

data Result = Result
{ source :: String -- eddig generált tárgykód
, positions :: [((Int, Int), (Int, Int), (Int, Int), (Int, Int))]
-- eddig meghatározott kapcsolatok a tárgy- és forráskód között
, position :: (Int, Int, Int, Int)

```

```
-- adott csúcsból generál kód kezdő és befejező pozíciói.
}
```

Lefelé terjedő információ jelen esetben egyedül a kódgeneráláskor szükséges indentáció mértéke, egy egész szám. Állapot jellegű információ pedig, ahogy azt korábban említettük, a kódgenerálás kezdetét reprezentáló sor és oszlop pozícióknak megfelel egész számok rendezett párja, valamint az aktuális csúcsig generált tárgykód részlet. Ezek után már definiálhatjuk a `DebugInfo` a és `DebugInfo1` a típusosztályokat, melyeket majd példányosítanunk kell a beágyazott reprezentációban definiált adattípusokra. Vegyük észre, hogy a függvény szignatúrájában nem szerepel `String` típusú változó.

```
type Location = (Int, Int)

class DebugInfo a where
  generateDebugInfo :: a -> Int -> Location -> Result

class (DebugInfo a) => DebugInfo1 a where
  generateDebugInfo' :: [a] -> Int -> Location -> Result
  generateDebugInfo' (x:[]) indent (line, col)
    = (Result (source newX) (positions newX) (position newX))
    where
      newX = generateDebugInfo x indent (line, col)
  generateDebugInfo' (x:xs) indent (line, col)
    = (Result (source newX ++ source newXs) ((positions newX)
      ++ (positions newXs)) (position newXs))
    where
      newX = generateDebugInfo x indent (line, col)
      newXs = generateDebugInfo' xs indent
              (endPosition $ position newX)
```

A `DebugInfo1` a típusosztályt abban az esetekben szeretnénk használni, amennyiben egy listában találhatóak azok a beágyazott reprezentáció aktuális csúcsának a gyerekei. Ilyen például a programok szekvenciája illetve a változók deklarációja. Szerencsére ilyen esetekben ugyanúgy kell eljárunk: a listafejelemére meghívjuk a neki megfelelő `generateDebugInfo` függvényt, a belőle kódgenerálás során keletkezett felfelé terjedő információból meghatározzuk, hogy a maradék listából generálandó kód kezdő pozícióit, majd arra rekurzívan meghívjuk a `generateDebugInfo'` függvényt az indentálás mértékével valamint az így kapott kezdő pozícióval.

Mielőtt elkezdenénk példányosítani a `DebugInfo` típusosztály a beágyazott reprezentáció adattípusaira át kell gondolnunk néhány részletet az állapot jellegű információkat illetően. Ahogy azt korábban felvázoltuk, ezekből a kódgenerálás során folyamatosan új értékek kerülnek kiszámolásra. Innentől már adódik, hogy ezeket monádba, nevezetesen `State Monad`ba kell becsomagolni.

Tehát a monádunk állapotába a kezdő sor és oszlop pozíciót reprezentáló egész számok, valamint az aktuális C kód részlet kerül be. Annak érdekében, hogy ezen információkhoz a teljes kódgenerálás során hozzáférjünk, a teljes folyamatot be kell ágyaznunk az állapot monádba. Ezért szükségünk van kettő csomagoló függvényre, amik feladata kiolvasni az aktuális állapotot, azaz a lefelé terjedő információkat, majd ezekkel meghívni a `generateDebugInfo` függvényt. Ezek implementációja a következőképpen néz ki.

```
monadicGenerateDebugInfo repr indent = do
  (src, line, col) <- get
  let res = generateDebugInfo repr indent (line, col)
  code $ source res
  return res

monadicListGenerateDebugInfo repr indent = do
  (_, line, col) <- get
  let resList = generateDebugInfo' repr indent (line, col)
  code $ source resList
  return resList
```

Továbbá szükségünk van egy függvényre, ami a monádon belül kap egy szöveget, majd abból az aktuális csúcsból generált kód alapján kiszámolja a befejező pozíciókat, majd a szöveget beteszi a monád állapotába. Ennek a függvénynek az implementációja a következő.

```
code s = do
  (src, line, col) <- get
  let numEOF = length (filter (=='\n') s)
  put (src++s,
       line + numEOF,
       (if numEOF == 0 then col else 0)
       +
       (length $ takeWhile (/=='\n') $ reverse s))
```

Ezen függvények segítségével már könnyen példányosítható a `DebugInfo` a típusosztályunk. Nyelvi konstrukciók esetén azonban figyelembe kell vennünk, hogy csomagoló, esetleg nyelvi konstrukciót reprezentáló csúccsal van dolgunk. Csomagoló csúcs esetében a dolgunk a következő: a csomagolt csúcsra meghívni a monadikus kódgenerátorunkat, az így kapott eredményt pedig beemelni a monádba. Felfelé terjedő információt reprezentáló rekord `source` mezőjébe kerül a csomagolt csúcspontból generált tárgykód. Összepárosítjuk a generált kód által meghatározott tárgykódbéli pozíciókat a beágyazott forrásszövegbéli pozíciókkal, majd az így kapott párosításokat a `positions` mezőben található lista elejére fűzzük. Végül a csomagolt csúcs által generált tárgykód abszolút kezdő, illetve befejező pozíciói bekerülnek a `position`

mezőjébe. Ezt a folyamatot írja le az alábbi implementáció kifejezések esetén.

```
generateDebugInfo (LabExpr ((sl, sc),(el, ec)) cont) ind (l, c)
  = (Result target (newHead:(positions newCont)) (l, c, nl, nc))
  where
    newHead = ((l, sl), (c, sc), (nl, el), (nc, ec))
    (newCont, (target, nl, nc)) = flip runState ("", l, c) $ do
      newContent <- monadicGenerateDebugInfo content indent
      return newContent
```

Nem csomagoló csúcs esetében pedig a következő a teendő. Írjuk le a generálni kívánt szöveget egészen addig, amíg a továbbiakhoz szükség lenne valamelyik gyerek csúcs által generált kódra. Ekkor hívjuk meg a gyerekcsúcsra a monadikus generátor függvényt. Ismételjük ezt az összes gyerekcsúcsra, majd az eredményeket emeljük be a monádba. Fontos megjegyezni, hogy nem mindig kell indentálnunk. Például aritmetikai kifejezések esetén a gyerek csúcsoknál nem szükséges indentálás, elágazásnál azonban az igaz, illetve hamis ágban található programoknál növelni kell az indentálás mértékét. A felfelé terjedő információknál a source mezőbe kerül a generált tárgy kód. A gyerekektől kapott positions listák konkatenációja kerül az aktuális csúcs felfelé terjedő információt reprezentáló rekord megfelelő mezőjébe. Végül pedig a kódgenerálás során kiszámolt befejező pozíciókat a paraméterül kapott kezdő pozíciókkal rendezett négyesként bekerülnek a felfelé terjedő információkat reprezentáló rekordunk position mezőjébe.

```
generateDebugInfo (Compare lhs rhs) indent (l, c) =
  (Result target
    ((positions newLhs) ++ (positions newRhs))
    (l, c, nl, nc))
  where
    ((newLhs, newRhs), (target, nl, nc)) =
      flip runState ("", line, col) $ do
        code "("
        newLhs <- monadicGenerateDebugInfo lhs 0
        code "< ("
        newRhs <- monadicGenerateDebugInfo rhs 0
        code ")"
        return (newLhs, newRhs)
```

Tárgykód és a forrás közötti kapcsolat

Ahogy az előző alfejezetben láttuk, a felfelé terjedő információt reprezentáló rekord positions mezőjében található lista tartalmazza a fordítóprogramunk által generált tárgykód, valamint a beágyazott While nyelvű forráskódja közötti kölcsönösen egyértelmű leképezést. Azonban a fordítási folyamat végé-

vel ez az adatstruktúra eltűnik, nem tudjuk felhasználni a hibakeresés során. Tehát ezt is valamilyen módon el kell tárolnunk. Azt a lehetőséget választottuk, hogy a lista tartalmát egy XML fájlba írjuk ki. A lista egy az egyben olyan szöveggént kerül kiírásra, amely szabványos XML struktúrának tekinthető. Az átalakításért az alábbi kód részlet a felelős.

```
printXML :: [Node] -> String
printXML []      = ""
printXML (x:xs) = (printNodeWithIndent x) ++ (printXML xs)

printNodeWithIndent :: Node -> String
printNodeWithIndent ((tsl, ssl),(tsc, ssc),(tel, sel),(tec, sec))
  = printIndent ++ "<node>\n"
  ++ printIndent ++ printIndent
  ++ "<startLine targetPosition=\"" ++ show tsl
  ++ "\" sourcePosition=\"" ++ show ssl ++ "\"/>\n"
  ++ printIndent ++ printIndent ++
  "<startColumn targetPosition=\"" ++ show tsc
  ++ "\" sourcePosition=\"" ++ show ssc ++ "\"/>\n"
  ++ printIndent ++ printIndent
  ++ "<endLine targetPosition=\"" ++ show tel
  ++ "\" sourcePosition=\"" ++ show sel ++ "\"/>\n"
  ++ printIndent ++ printIndent
  ++ "<endColumn targetPosition=\"" ++ show tec
  ++ "\" sourcePosition=\"" ++ show sec ++ "\"/>\n"
  ++ printIndent ++ "</node>"
```

A választás azért esett az XML struktúrára, mert ez eléggé általános ahhoz, hogy a legtöbb szövegszerkesztő használni tudja. Elképzelésünk szerint a programozó egy két részre osztott ablak egyik felében az általa írt beágyazott forráskódot, másikon a belőle generált tárgykódot láthatja. Amennyiben érdekelné a programozót, hogy egy adott generált kódrészlet a forrásszöveg mely részéből került generálásra, akkor azt kijelöli majd egy billentyű kombináció segítségével egy segédprogram, vagy a szövegszerkesztőhöz implementált plugin elkéri a szövegszerkesztőtől a programozó által kijelölt terület meghatározó pozíciókat. Az alkalmazás az XML struktúrát bejárva megkeresi azt a node-ot, mely a tárgykódban kijelölt területet legszűkebben tartalmazza, majd kiolvassa a megfelelő attribútumokban található értéket és továbbítja a szövegszerkesztőnek, ami az így meghatározott forráskód részletet megjelöli, tegyük fel kiemeléssel a programozó számára.

Fordítás hibakereső módban

A hibakeresést támogató fordítást végző függvény implementációja a következőképpen néz ki.

```
compileWithDebug :: Program -> String -> String -> IO ()
compileWithDebug prog targetPath xmlPath = do
    let res = generateDebugInfo prog 0 (1,1)
    let src = source res
    writeFile targetPath src
    let xml = "<root>\n" ++ (printXML $ positions res)
        ++ "</root>"
    writeFile xmlPath xml
```

Első paraméterként megkapja akiterjesztési transzformáción már átesett beágyazott adatstruktúrát, melyen végrehajtódik a kibővített fordítási folyamat. Kezdetben az indentálás mértéke 0, továbbá az első sor első oszlopától kezdődik a kódgenerálás. Ennek az eredménye egy rekord, melynek a `source` mezőjében található a tárgy kód. A második paraméterként kapott relatív elérési útvonalon található fájlba kerül ez kiírásra. Harmadik paraméterül kapott elérési útvonalon található fájlba kerül kiírásra a rekord `positions` mezőjében található lista XML adat struktúraként.

A kiterjesztett fordítási folyamat

A fordítás folyamati két, egymástól elkülönülő egységként került implementálásra. Első lépésként a transzformációt végrehajtó egység. Ez önálló, `stand-alone` verzióként használható. Bemenete egy beágyazott While nyelven írt kódot tartalmazó fájl, kimenete pedig a bemeneti forráson alkalmazott kiterjesztési transzformáció által módosított szintaxis fából generált kód. Fájl nevekre a következő konvenciót alkalmazzuk : amennyiben a bemenő fájl neve `XY.hs` a kimenő fájlé `XYDebug.hs` lesz. Az így keletkezett állomány már GHCi segítségével interpretálható, a függvényekre meg lehet hívni a hibakeresést támogató fordító programunkat aminek a kimenete most már kettő fájl: a paraméterül kapott függvényből generált C kód részlet, valamint a generált kód és a függvény forrás pozíciói közötti kapcsolatot leíró XML fájl lesz.

Példa

A fenti változtatások után, a korábban már példaként használt legnagyobb közös osztó kereső algoritmuson szemléltetem a transzformáció, illetve a módo-

sítozott fordítóprogramom működését. Emlékeztetőül a legnagyobb közös osztó forrása beágyazott While nyelven a következőképpen néz ki.

```
gcd :: Program
gcd = (Declare [x, y]) ++
  (Loop
    ((!)(x == y))
    (IfThenElse
      (x < y)
      (y := ((Var y) - (Var x)))
      (x := ((Var x) - (Var y)))
    )
  )

x :: Variable Int
x = Variable "x"

y :: Variable Int
y = Variable "y"
```

A gcd forráskódjára meghívva a szintaktikus elemzőt, az így kapott szintaxis-fán elvégzett transzformációt majd azt szöveggé alakítva a következő kimenetet kapjuk.

```
gcd :: Program
gcd = label ((14, 7), (14, 123))
  ((label ((14, 8), (14, 22))
    (Declare (label ((14, 16), (14, 22)) [x, y]))
    ++
    (label ((14, 28), (14, 122))
      (Loop
        (label ((14, 34), (14, 45))
          ((!)(label ((14, 38), (14, 44)) (x == y))))
        (label ((14, 48), (14, 121))
          (IfThenElse (label ((14, 60), (14, 65)) (x < y))
            (label ((14, 68), (14, 92))
              (y :=
                (label ((14, 74), (14, 91))
                  ((label ((14, 75), (14, 80)) (Var y)) -
                    (label ((14, 85), (14, 90)) (Var x))))))
            (label ((14, 96), (14, 120))
              (x :=
                (label ((14, 102), (14, 119))
                  ((label ((14, 103), (14, 108)) (Var x)) -
                    (label ((14, 113), (14, 118)) (Var y))))))))))
```

Ebből a forrásból generált tárgykód megegyezik a még nem kibővített változat által előállítottal. További kimenetként kapunk egy XML fájlt, amelyből egy részlet az alábbi módon néz ki.

```
<root>
  <node>
    <startLine targetPosition="1" sourcePosition="14"/>
    <startColumn targetPosition="1" sourcePosition="7"/>
    <endLine targetPosition="13" sourcePosition="14"/>
    <endColumn targetPosition="1" sourcePosition="123"/>
  </node>
  <node>
    <startLine targetPosition="1" sourcePosition="14"/>
    <startColumn targetPosition="1" sourcePosition="8"/>
    <endLine targetPosition="3" sourcePosition="14"/>
    <endColumn targetPosition="0" sourcePosition="22"/>
  </node>
  ...
</root>
```

7. fejezet

Eredmények

A dolgozatomban egy módszert adtam mélyen beágyazott nyelvek forrás szintű hibakeresésének támogatására. A módszerhez kapcsolódóan először a While nyelvet beágyaztam Haskellbe. A forráskódból a fordítóprogramom C kód részleteket generál. Következő lépésként a dolgozatomban felvázolt módszer lépéseinek megfelelően a nyelvem beágyazott reprezentációja, valamint a programozó felé biztosított függvények kibővítésre kerültek. A gazdanyelv szintaxisfájára vonatkozó transzformáció is implementálásra került. Nem utolsósorban a fordítóprogramom kódgenerátora is módosításon esett át, melynek már a C kód részlet mellett egy XML fájl is a kimenete, mely a generált tárgykód és a beágyazott forrásszöveg közötti kölcsönösen egyértelmű leképezést valósítja meg. Az eredményeimmel kapcsolatban fontos még megvizsgálni a beágyazott nyelv fordítóprogramjának kódnövekedésének a mértékét. A beágyazott While nyelv implementálása során 230, a kibővített nyelv esetén pedig 721 darab sor Haskell nyelvű kód keletkezett. Ez 213%-os kódnövekedést jelent, azonban vegyük figyelembe az alábbi szempontokat.

- A kibővített nyelvben lehetőségünk van csak tárgykódot, illetve már hibakeresést segítő információkat is előállító fordítási folyamat indítására. A komplexebb kódgenerátor mérete 239 sor, míg az egyszerűbbé 114 sor. Igaz így is kicsivel több mint kétszer annyi forráskódra volt szükség az implementációhoz, viszont az sokkal összetettebb feladatot is végez. Nem utolsósorban a segédfüggvényeik segítségével a beágyazott reprezentációban szereplő adattípusokra a szükséges típusosztályok példányosítása analóg módon történt.
- A gazda reprezentáció manipulálásához szükséges kód mérete 128 sor. Ennek majdnem a negyedét teszi ki annak a típusosztálynak a példányosítása, mely a gazda szintaxisfa egy adott csúcsából kinyeri annak forrás-

beli elhelyezkedésének a pozícióját, ezen függvények törzse szinte teljes mértékben megegyezik.

- Nem kellett lexikális és szintaktikus elemzőt, valamint típus ellenőrzőt implementálnunk, mert a Haskell ezeket biztosította számunkra.
- Nyelvünkben lehetőség van modulok definiálására, továbbá megtehetjük, hogy egy adott modulban található függvények közül néhányat kívülről elérhetetlenné teszünk. Ezt a funkcionalitást sem nekünk kellett megvalósítani, hanem a gazdanyelv rendelkezésünkre bocsátotta.

Ezek után levonhatjuk azt a következtetést, hogy költséghatékonyabb egy új programozási nyelvet beágyazni egy gazdag eszköz készlettel rendelkező általános célú programozási nyelvbe, majd további eszközöket biztosítani a programozó számára, mint önálló nyelvként, saját lexikális, szintaktikus és szemantikus elemzővel, valamint típus ellenőrzővel és egyéb nyelvi funkciókkal implementálni.

Irodalomjegyzék

- [1] Debugging techniques, *The Object Oriented Programming Web*, <http://oopweb.com/CPP/Documents/DebaugCPP/Volume/techniques.html>.
- [2] Keijiro Araki and Zengo Furukawa and Jingde Cheng, *Framework for Debugging*, Kyushu University, 1991.
- [3] Shahabeddin Farokhzad and Gokhan Tanyeri and Trish Messiter and Paul Beckett, *Plug-in Based Debugging For Embedded Systems*, RTES, 2010.
- [4] Ben Liblit and Alex Aiken, *Building a Better Backtrace: Techniques for Postmortem Program Analysis*, University of California at Berkeley, Technical report, 2002.
- [5] The Lex and Yacc Page, <http://dinosaur.compilertools.net>, 2012.
- [6] Oracle PL/SQL, <http://www.oracle.com/technetwork/database/features/plsql/index.html>, 2012.
- [7] Functional Embedded Language for DSP and PARallelism, <http://feldspar.inf.elte.hu/>, 2012.
- [8] Galois - Cryptol, <http://corp.galois.com/cryptol>, 2012.
- [9] Paul Hudak, *Modular Domain Specific Languages and Tools*, ICSR '98: Proceedings of the 5th International Conference on Software Reuse, 134, 1998.
- [10] Daan Leijen and Erik Meijer, *Domain Specific Embedded Compilers*, Proceedings of the 2nd Conference on Domain-Specific Languages, Pages 109-122, 1999.
- [11] Nelio Muniz Mendes Alves and Sergio de Mello Schneider. *Implementation of an Embedded Hardware Description Language Using Haskell*. Journal of Universal Computer Science Volume 9 Issue 8. Pages 795-812. 2003.

-
- [12] Hiralal Agrawal and Richard A. Demillo and Eugene H. Spafford Y, *An execution backtracking approach to program debugging*, In Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference, Pages 283-299, 1988.
- [13] Hajós Márton. *Egységes programozói felület optimalizációs célú programtanszformációk implementációjához*. Diplomamunka 2011
- [14] Syntax of the While language. <http://pag.cs.uni-sb.de/while.html> 2012