Attributum nyelvtanok és alkalmazásaik a szoftvertechnológia és a mesterséges intelligencia területén

Tézisek

Gyimóthy Tibor

Szeged

1995.

1 Bevezetés

Az attributum nyelvtanok a kőrnyezetfüggetlen nyelvtanok kiterjesztésének tekinthetők, ahol az attributumokat a nyelvtani szimbólumokhoz rendeljük , és a szintaktikus szabályokhoz kapcsolódó szemantikus egyenletekkel definiáljuk ezen attributumok értékeit[Knu68]. Ezt a formalizmust elterjedten használják fordítás orientált nyelvek specifikálására és implementálására-[DJL88]. Elmondhatjuk, hogy az attributum nyelvtanos módszer tekinthető a legelfogadottabb elméleti, de gyakorlatban is alkalmazható modellnek fordítóprogramíró rendszerek, nyelvi processzorok megvalósítására[AlM91].

Dolgozatunk 2. fejezetében egy attributum nyelvtan alapú fordítóprogramíró rendszert (PROF-LP) és ennek korábbi verzióit ismertetjük. A PROF-LP rendszer alkalmas valódi felhasználói kőrnyezetben hatékonyan működő szoftverek generálására. Ennek bemutatására a fejezet végén röviden ismertetjük a rendszer néhány alkalmazását.

A 3. fejezetben az attributum nyelvtan alapú természetes nyelvi interfészekkel kapcsolatos kutatásainkat írjuk le. Vizsgáltuk, hogy a hagyományos programozási nyelvek fordítási modelljének statikus szemantikai része hogyan értelmezhető természetes nyelvi interfészek esetén.

Svéd kutatókkal közösen kifejlesztett módszert (GADT) mutatunk be a 4. fejezetben, amivel procedurális nyelveken írt programok algoritmikus nyomkövetése és tesztelése valósítható meg.

Logikai programok esetében nehezebb hatékony algoritmus nyomkövetés megvalósítása, mivel az adatáramlás iránya nincs definiálva. Azonban, az attributum nyelvtanos modell alkalmazásával [BPM93] lehetőség van logikai programok függőségi gráfjának megadására és ez alapján a GADT módszerhez hasonló algoritmus fejlesztésére. Az 5. fejezetben finn kutatóval közösen kidolgozott módszert (IDTS) ismertetünk logikai programok algoritmikus nyomkövetésére és tesztelésére.

A 6.fejezetben logikai programok statikus szeletelési módszereit mutatjuk be. A szeletelési módszer alapjául az 5. fejezetben említett attributumos annotálási technika szolgál. A bevezetett statikus szeletelési módszerek alkalmazhatók logikai programok párhuzamosítása, tesztelése, karbantartása esetén.

A 7. fejezetben ismertetett módszer (IMPUT) a SPECTRE specializációs algoritmus és az IDTS technika integrálásával egy általánosan alkalmazható *theory revision* algoritmust nyújt logikai programok tanulására.

Végezetül egy algoritmust ismertetünk (AGLEARN) atrributum nyelvtanok tanulására. A módszer az Induktív Logikai Programozás területén alkalmazott transzformációs technikán alapul.

A dolgozatban minden fejezet végén megadom a fejezetben közölt eredményekhez kapcsolódó publikációim listáját, és részletezem mely eredmények alapulnak döntő mértékben az én munkámon.

Köszönetet mondok azon munkatársaimnak, akikkel együtt dolgoztam az ezen tanulmányban szereplő rendszerek, eredmények létrehozásában. Külön köszönettel tartozom dr. Simon Endrének, Alexin Zoltánnak, Horváth Tamásnak, Kocsis Ferencnek és dr. Toczki Jánosnak, akikkel több éven keresztül intenzív csoportmunkában dolgoztunk attributum nyelvtan alapú rendszerek fejlesztésén és kutatásán. Köszönet illeti meg dr.Makay Árpádot az egyetemi doktori dolgozatom témavezetéséért, valamint dr. Turán Győrgyöt, amiért inspirált a gépi tanulással kapcsolatos kutatásokra.

2 Integrált kőrnyezet attributum nyelvtan alapú alkalmazások fejlesztésére

Az attributum nyelvtanokat D.E.Knuth [Knu68] definiálta 1968-ban fa-struktúrában való számítások specifikálására. Az azóta eltelt időben bebizonyosodott, hogy az attributum nyelvtanok sikeresen alkalmazhatók programozási nyelvek analízisére, fordítóprogramok, nyelvi processzorok előállítására. Az attributum nyelvtanok hatékony implementálására számos módszert fejlesztettek ki és több attributum nyelvtan alapú fordítóprogramíró rendszer készült. Az attributum nyelvtanos módszerek és fordítóprogramíró rendszerek átfogó ismertetése megtalálható [DJL88]-ban. A fejezet további részében három attributum nyelvtan alapú nyelvi processzorokat generáló rendszert ismertetünk.

2.1 HLP/SZ

A HLP/SZ rendszer[GSM83],[DJL88] alapjául a Helsinkiben kifejlesztett HLP78 rendszer[RSS78] lexikális illetve szintaktikus-szemantikus metanyelve szolgált. A lexikális metanyelven leírható a forrás nyelv lexikális struktúrája, amiből a rendszer automatikusan generálja az adott nyelv lexikális elemzőjét. Egy nyelv lexikális leírása tartalmazza a karakter halmazok, token osztályok és a szükséges transzformációk, szűrések definiálását.

Az attributum nyelvtanos specifikáció a rendszer szintaktikus szemantikus metanyelvén írható le. Egy ilyen specifikáció tartalmazza az örökölt és szintetizált attributumok definícióját, szintaktikus szabályok és a hozzájuk kapcsolódó szemantikus egyenletek és kódgeneráló utasítások megadását.

A HLP/SZ rendszer LALR(1) típusú [ASU85] elemzőt generál a a derivációs fa előállítására. Az attributumos elemzési fák kiértékelését az irodalomból ismert ASE[JaW75] és OAG[Kas80] algoritmusok módosított változataival valósítottuk meg. Felhasználva az implementációs nyelv (SIMULA 67) hatékony tárkezelési technikáját javítottuk az eredeti ASE kiértékelés memória felhasználását. Az attributum nyelvtan alapú fordítóprogramíró rendszerek leghatékonyabb kiértékelési stratégiája a vizit-orientált OAG módszer. Ezt a módszert módosítottuk úgy, hogy az eredményül kapott MOAG algoritmus [GSM83] a nemcirkuláris attributum nyelvtanok nagyobb osztályára képes attributum kiértékelőt előállítani.

2.2 HLP/Pascal

A HLP/Pascal rendszer [TSG86],[GyT87],[TGH86] fejlesztését egyrészről az motiválta, hogy a Simula nyelvet kevés helyen használták, másrészt tapasztalataink azt mutatták, hogy egymenetes fordítók használata sok esetben előnyösebb lehet mint a HLP/SZ rendszer által generált többmenetes fordítóké (egymenetes fordításnál nem kell az attributumos fát tárolni). Ezért a HLP/Pascal rendszer képes egymenetes és többmenetes nyelvi processzorokat előállítani Pascal nyelven.

Az egymenetes alrendszer LL(1), L-attributumos[Boc76] nyelvtanok feldolgozására alkalmas. A generált fordítók rekurzív-leszálló elemzők, ahol a szemantikus akciók (atributum kiértékelés, kódgenerálás) az elemzés alatt kerülnek végrehajtásra.

A többmenetes alrendszer LALR(1) típusú elemzőt és OAG/MOAG attributum kiértékelőt képes generálni. A generált elemző felépít egy elemzési fát és a szemantikus akciók a vizitsorozatokkal vezérelt fabejárás során hajtódnak végre. Mindkét alrendszer által generált fordítók tartalmazzák az u.n. *panic* módszert a szintaktikus hibajavításra (error recovery).

2.3 PROF-LP

A PROF-LP rendszer [GHK88],[TGH88],[GyK88],[Gyi91a],[GASz92] fejlesztésének fő célkitűzése az volt, hogy egy olyan eszközt állítsunk elő, ami valódi felhasználói kőrnyezetben is hatékonyan működő nyelvi processzorok generálását biztosítja. További feltétel volt, hogy a generáló eszközt is kényelmesen tudják használni olyan alkalmazók is, akik csak kevéssé járatosak a formális nyelvek területén. Ezért a HLP/Pascal rendszerhez képest optimalizáltuk a generált kód szerkezetét, valamint biztosítottuk a C nyelvre való generálást is. Az attributum nyelvtanokat kevéssé ismerő felhasználók számára kifejlesztettünk egy interaktív alrendszert. Ezen alrendszerben a teljes attributumos specifikáció u.n. item definíciók sorozatából áll elő. A felhasználó minden item definiálásakor azonnal visszajelzést kap a specifikáció helyességéről és a rendszer on-line segítséget nyújt egy korrekt attributum nyelvtanos leírás elkészítéséhez. Ez a változat tartalmaz egy inkrementális generáló modult a LL(1) elemzési tábla előállítására. Ez különösen akkor hasznos, ha egy nagy nyelvtanon kis módosítást hajtunk végre és gyorsan szeretnénk látni a módosítás hatását. Ilyenkor az inkrementális elemző generátor modul, ami használja az előzőleg kiszámított *first* és *follows* halmazokat gyorsabban tudja előállítani az elemző táblát, mintha újra kellene számolni ezeket a halmazokat.

A mesterséges intelligencia területén számos estben nemegyértelmű nyelvtanokkal írhatók le az adott problémák (például szintaktikus alakfelismerés, természetes nyelvi elemzés). A PROF-LP rendszer tartalmaz egy backtrack típusú elemző generáló modult az ilyen nyelvtanok feldolgozására[GyT87]. A generált backtrack elemzők sajátossága, hogy a használnak LL(1) elemző táblát, ezzel a minták egyértelmű részein jelentősen megnövelhető az elemzők hatékonysága. Mivel az attributumos rendszerekben a szintaktikus elemzéssel párhuzamosan általában attributum kiértékelés is folyik, ezért az ilyen típusú elemző alkalmazása nagy méretű minták feldolgozására különösen hasznos. A generált backtrack elemzők további javítását értük el azzal, hogy az elemzés során használjuk a kiszámított attributum értékeket.

2.4 Alkalmazások

A továbbiakban röviden ismertetünk néhány nagyobb méretű PROF-LP alkalmazást.

1. Egy attributum nyelvtanos specifikáció alapján elkészült egy C-interpreter. Ez a C-interpreter a MicroSegams[MKM89] orvosi képfoldolgozó rendszer részeként számos helyen működik. A felhasználók C nyelvű utasítások interaktív használatával paraméterezhetik illetve vezérelhetik a képfeldolgozó eljárásokat.

2. A PROF-LP rendszerrel elkészült egy előfordító a teljes COBOL nyelvre. Ez az előfordító, ami egy COBOL programot elemez és belső kódot generál a programra a MICROTEST [SFH89] tesztelő rendszer részeként került forgalmazásra.

3. Elkészült egy attributum nyelvtanos specifikáció a teljes LOTOS nyelv (Language of Temporal Ordering Specifications)[BoB88] statikus szemantikájára. A specifikáció alapján PROF-LP-vel generált LOTOS előfordító [TGJ88] egy program statikus szemantikájának ellenőrzésén túl belső adatformában előállítja a program kanonikus viselkedési sruktúráját. Ezzel a rendszerrel lehetőség van hálózati protokol leírások szintaktikus vizsgálatára és szimulátorokhoz input adatok előállítására.

4. A PROF-LP használatával elkészült egy természetes nyelvi interfész a síkgeometriai szerkesztések oktatására szolgáló THALES [AFD90],[FAG90],[AFG90],[Gyi91a] rendszerhez. A THALES rendszer és a természetes nyelvi interfész bővebb leírása a következő fejezetben található.

5. Az attributum nyelvtanok sikeresen aklkalmazhatók az alakfelismerés területén is, mivel segítségükkel sok esetben integrálhatók az alakfelismerés statisztikus és szintaktikus módszerei [Fu82],[TsF80]. A PROF-LP rendszer backtrack elemző generátor moduljának alkalmazásával vizsgáltuk EKG hullámok felismerését[GyT87].

6. A PROF-LP rendszer egyik legnagyobb alkalmazási területének az oktatás bizunyult. A József Attila Tudományegyetem több mint 30 programozó, programtervező matematikus hallgatójának diplomamunkája kapcsolódott a PROF-LP rendszerhez. A hallgatók diplomamunkáikban vizsgálták hagyományos programozási nyelvek (Ada, Pascal, C) struktúráinak fordítási modelljeit, speciális nyelvek fordítóprogramjait(adatbáziskezelő nyelvek, grafika leíró nyelvek), különböző programgeneráló rendszereket(felhasználói interfész, helpgenerátor), természetes nyelvek és attributum nyelvtanok kapcsolatát, szintaktikus alakfelismerési problémákat.

Kapcsolódó publikációim: [AFG90],[GAS92], [GHK88],[GSM83],[GyH94], [Gyi91a],[Gyi91b],-[GyK88],[GyT87],[TGH88],[TGJ88],[TGK90],[TSG86].

A HLP/SZ rendszer tervezését és megvalósítását döntően ketten végeztük el. Az én feladatom volt a szintaktikus-szemantikus feldolgozás és a kódgenerálás tervezése illetve implementálsa. A HLP/Pascal és a PROF-LP rendszerek megvalósítása további munkatársak bevonásával történt. Ezen rendszereknél a fejlesztési munkák irányítását, valamint az interaktiv alrendszer és a backtrack elemző modul tervezését végeztem el. A szintaktikus alakfelismerési alkalmazás kidolgozása tőlem származik, jelentősen résztvettem a THALES NLI és kisebb mértékben a LOTOS precompiler illetve a C-interpreter tervezésében. Témavezetője voltam több mint húsz olyan diplomamunkának illetve szakdolgozatnak, amelyek témája a fejezetben felsorolt rendszerekhez illetve alkalmazásokhoz kapcsolódik.

3 Attributum nyelvtan alapú természetes nyelvi interfészek

A természetes nyelvi interfészek (NLI-Natural Language Interface) az információ feldolgozás több területén sikeresen helyettesítik a hagyományos felhasználói interfészeket (menük, formális lekérdező nyelvek)[JAK88]. Az NLI-k fő előnye, hogy rugalmas, könnyen elsajátítható eszközt biztosítanak nem számítógépes szakembereknek programok, adatbázisok kezeléséhez[Sch88].

A természetes nyelvek bonyolultsága miatt általános célú, de egy adott területen hatékonyan alkalmazható NLI-k létrehozása nagyon nehéz feladat. Ezért a gyakorlatban egy adott területre részletesen kidolgozott NLI-k terjedtek el, amelyek az aktuális problémát lefedő résznyelv teljes megértését és feldolgozását valósítják meg. Azonban, még az ilyen egyszerűbb NLI-k létrehozása is nagy ráfordítást igényel, ezért kulcsfontosságú az alábbi két szempont vizsgálata:

1. Az NLI-k mely részei vihetők át egyéb alkalmazásokra.

2. Az NLI-k mely részei használhatók fel különböző nyelvi kőrnyezetekben ugyanazon alkalmazási területre.

Ezen kérdések vizsgálatát egy több éves fejlesztési munkával elkészített NLI alapú rendszer ($THALES$) tapasztalatai alapján ismertetjük. A $THALES$ rendszer [FAG90],[AFG90],[Gyi91a] síkgeometriai szerkesztések megvalósítására szolgál, amelyben a felhasználó egy *attributum nyelvtanos specifikáció* alapján generált NLI segítségével írhatja le a geometriai szerkesztési lépések angol nyelvű utasításait.

A NLI az alábbi fő funkcionális egységekből áll:

1. Lexikális-morfológiai elemző

2. Szintaktikus elemző

3. Szemantikus elemző

A nyelvi feldolgozás alapjául egy attributum nyelvtanos specifikáció szolgál, amelyből a PROF-LP nyelvi processzor generálja az elemzőt és a szemantikus kiértékelőt.
Szintaktikailag a *THALES NLI*-je a következő részekre osztható:

1. Command Werb

2. Object Word

3. Specifier

Az *NLI* komplexitásának szemléltetésére bemutatunk néhány mondatot, amelyeket a rendszer fel tud dolgozni:

1. *Draw two circles with radius EF at a distance equal to the difference between the base of the triangle and the side RS of the heptagon.*

2. *Inscribe a quadrangle that is tangent to the circle on the left part of the screen.*

Tapasztalataink azt mutatták, hogy egy *NLI* elkészítésének legbonyolultabb része a szemantikus elemző megvalósítása (esetünkban a teljes rendszer 80 százalékát tette ki ez a rész) [AFG90]. Ezért megpróbáltuk a az *NLI* szemantikus specifikációját jól definiált részekre elkülöníteni. Hagyományos programozási nyelvek esetében megkülönböztetünk *statikus* és *dinamikus* szemantikát.
A statikus szemantika olyan kőrnyezetfüggő tulajdonságokat jelől, amelyek fordítási időben vizsgálhatók. A legfontosabb ilyen tulajdonságok a következők:

1. A változók (objektumok) definiáló és alkalmazott előfordulásainak megkülönböztetése.

2. Azonosítási probléma (scope): az alkalmazott előfordulásokhoz hozzárendelni a megfelelő definiáló előfordulásokat.

3. Típuskompatibilitás.

Dinamikus szemantikán a program futása során végrehajtott tevékenységeket értjük.

A *THALES* típusú *NLI*-k (imperatív jellegű NLI) statikus szemantikájának megadását két részre oszthatjuk:

1. Az első rész feladata a nyelvi objektumok (és ezek attributumainak) származtatása az attributum nyelvtanos struktúra definíciók alapján.

2. A második rész feladata az objektumok azonosítása illetve a típuskompatibilitás eldöntése.

   (a) Az azonosítási probléma hagyományos programozási nyelvekre általában egyszerű algoritmussal megvalósítható (például blokk-struktúrájú nyelvek), azonban *NLI*-k esetében jóval bonyolultabb módszer szükséges. Az azonosítás történhet az objektum nevével, típusával, attributumaival, objektumok közötti relációkkal, illetve ezek kombinációival.

   (b) A típuskompatibilitás kérdése programozási nyelvekre a típusok közötti kapcsolatok explicit megadása után az attributumos modellben hatékonyan kezelhető. A *THALES* típusú *NLI*-knél is szükség van az objektum típusok megengedett attributumainak(melléknevek, mértékegységek, részobjektumok) illetve az objektumok közötti érvényes relációk definiálására. Ezekután a hagyományos attributum nyelvtanos módszereket lehet alkalmazni a típuskompatibilitás eldöntésére.

Az azonosítási és típuskompatibilitási probléma megoldására a hagyományos programozási nyelvekhez hasonlóan NLI-k esetében is hatékony szimbólumtábla kezelés szükséges.

Összefoglalóan az imperatív jellegű NLI-k egyes részeinek alkalmazás illetve nyelvi függetlenségére az alábbi megállapításokat tehetjük:

1. Lexikális-morfológiai elemző
   - nagy mértékben független az adott alkalmazástól
   - nyelv-függő
2. Szintaktikus elemző
   - parancs-orientált alkalmazásokra nagy részben átvihető
   - nyelv-függő
3. Nyelvi objektumok származtatása
   - nagy mértékben független az adott alkalmazástól
   - nyelv-függő
4. Szimbólumtábla kezelés, típuskompatibilitás,azonosítás
   - kevéssé függ az adott alkalmazástól (a típus és reláció definíciókat kell újra írni, az általános kezelő rész átvihető).
   - az általános kezelő rész átvihető más nyelvi kőrnyezetre.
5. Dinamikus szemantika
   - alkalmazás függő
   - független a nyelvi kőrnyezettől

Kapcsolódó publikációim: [AFD90],[AFG90],[FAG90],[Gyi90a],[HAF90].

Az imperatív jellegű NLI-k statikus szemantikájának vizsgálatára az attributum nyelvtanos modell alkalmazása tölem származik.

4 Algoritmikus nyomkövetés és tesztelés

A nyomkövetés a szoftver fejlesztés egyik legköltségesebb fázisa, ezért már sok kisérlet történt hatékony, automatikus nyomkövetési módszerek kidolgozására. A Shapiro [Sha83] által bevezetett algoritmikus nyomkövetési módszer nyújtotta az első elméletileg is megalapozott megoldást logikai programok részben automatikus nyomkövetésére.

Shapiro rendszerében a programozó (oracle) a hiba azonosítása során információt nyújt a program viselkedéséről a a rendszer által feltett kérdésekre válaszolva. Bár a Shapiro féle modellt elviekben imperativ nyelvekre is alkalmazni lehetne, azonban ez a modell nem kezeli például a mellékhatásokat illetve a ciklusokat így a gyakorlatilag ezen a területen nem volt alkalmazható. Egy további súlyos hátránya a módszernek, hogy a felhasználónak általában nagyon sok kérdésre kell válaszolnia a nyomkövetés során.

Az algoritmikus nyomkövetési módszert kombinálva a program szeletelési (slicing) eljárással csökkenteni lehet a hibakeresés során feltett kérdések számát[SKF90]. A program szeletelési módszer a program adatáramlási és vezérlési gráfja alapján összefüggő komponenseket határoz

meg az adott programban. A létrejövő komponensek (szeletek) mérete rendszerint program és input függő, azonban egy szelet általában sokkal kisebb mint az eredeti program. Ez különösen blokk-strukturájú nyelvekre igaz.

Az algoritmikus hibakeresésben további javulást lehet elérni a *CPM* (Category Partition Method) [OsB88] funkcionális tesztelési módszer alkalmazásával. Reális méretű programok algoritmikus nyomkövetése során a felhasználónak sok *nehéz* kérdésre kell válaszolnia. Például, tegyük fel, hogy a programban van egy eljárás, ami egy tömb elemeinek az összegét számítja ki, és ezt az eljárást meghívjuk egy száz elemű tömbbel. A felhasználó nehezen tudja ellenőrizni a számítás helyességét. Azonban, ha az eljárást már teszteltük a *CPM* tesztelési módszerrel, akkor a tesztelési eredmények használhatók a nyomkövetés során.

A svéd kutatókkal közösen kifejlesztett *GADT* (Generalized Algorithmic Debugging and Testing) módszer integrálja a program szeletelési eljárást és a *CPM* algoritmus egy kiterjesztett változatát a Shapiro féle algoritmikus nyomkövetési módszerrel[FGK91].

4.1 A CPM módszer kiterjesztése

A funkcionális tesztelés során a programokat nem tudjuk tesztelni az input paraméterek összes tulajdonságára. Ezért a tesztelőnek meg kell határozni a paraméterek kritikus tulajdonságait. Ezeket a tulajdonságokat (*kategoriákat*) vizsgáljuk a tesztelés során. A kategoriákat osztályokra (*choice*) bonhatjuk azon feltételezés alapján, hogy az azonos osztályba eső elemek viselkedése megegyezik a tesztelés szempontjából. A kategoria és osztály specifikációk felhasználásával *teszt vázak* generálhatók. Egy teszt váz pontosan egy osztályt tartalmaz minden kategoriából.

Rendszerint nagyon sok értelmetlen és felesleges teszt váz van a generált vázak között. Ezek a felesleges teszt vázak kiszűrhetők az osztályokhoz rendelt *szelektor kifejezések* alkalmazásával. Egy osztály csak akkor kerülhet be egy teszt vázba, ha a hozzá rendelt szelektor kifejezés értéke igaz. A szelektor kifejezésekben szereplő 'logikai változók' (*properties*) az osztályokhoz kapcsolódnak és értékük akkor igaz, ha az adott osztály szerepel az aktuális teszt vázban.

Egy program általában nagyon sok eredményt állít elő ezért a tesztelőnek definiálni kell az érdekes teszt vázakat. A program eredményei kategoriákra és osztályokra bonthatók szelektor kifejezések használatával. Teszt esetek futtatása rendszerint környezeti paraméterek időigényes beállításával valósítható meg. Az azonos környezeti paramétereket használó teszt vázak összegyűjthetők *tesztelési csoportokba* (*test sript*) szelektor kifejezések segítségével. A specifikáció különböző pontjain elhelyezhető deklarációs és végrehajtható utasítások segítik a futtatható teszt esetek előállítását.

Az eredeti *CPM* módszer csak a teszt vázak előállítását ismertette korlátozott formájú szelektor kifejezések alapján. A *GADT* módszerben alkalmazott új jellemzők (bővített szelektor kifejezés, eredmény kategóriák, futtatható teszt esetek, teszt riportok) kiterjesztették a *CPM* módszer alkalmazási lehetőségét[SzG90],[TGK90].

Egy hatékony algoritmikus nyomkövető algoritmus célja, hogy minél kevesebb felhasználói segédlettel azonosítson egy hibás program részt. A *GADT* algoritmusban a hiba azonosítási folyamat három fő részre osztható : egyszerű algoritmikus nyomkövető, teszt eset kereső és program szeletelő.

4.2 Egyszerű algoritmikus nyomkövető

Ez a komponens a végrehajtási fa bejárása során kérdéseket tesz fel a felhasználónak a program egységek elvárt eredményeire vagyis a program *elvárt viselkedésére*. Mielőtt feltenné az adott

csomópontnak megfelelő kérdést meghívja a teszt eset kereső komponenst, annak eldöntésére, hogy a teszt adatbázis alapján az aktuális kérdés megválaszolható-e. Az algoritmikus nyomkövető komponens addíg folytatja működését, ameddig az alábbi két feltétel valamelyike nem teljesül:

1. A hiba azonosítása megtörténik eljárás szinten.
2. Egy hiba hatására a felhasználó meghívja a program szeletelő komponenst. Ez akkor fordulhat elő, amikor egy eljárás több outputot eredményez és csak egy output érték hibás.

4.3 Teszt eset kereső

Az eljárásokra adott teszt specifikációk és teszt esetek használhatók a nyomkövetés során. Hiba kereséskor az eljárások input, output paramétereinek konkrét értéke adott. Az aktuális input értékre a megfelelő teszt váz kétféle módon határozható meg. Számos eljárásra megadható egy függvény, ami automatikusan kiválasztja a megfelelő teszt vázat. Lehetnek azonban olyan eljárások is amelyekre nagyon nehéz ilyen függvényt definiálni. Ebben az esetben a teszt specifikáció alapján a felhasználó választja ki a megfelelő teszt vázat. Ezután a generált teszt riport adatbázis alapján történik a hibakereső vezérlése. Ha a teszt riport az adott teszt vázra *false*, akkor az aktuális eljárásban folytatjuk a nyomkövetést, *true* esetén pedig kihagyjuk ezt az eljárást a hibakeresésből.

Természetesen funkcionális tesztelés során a tesztelés megbízhatósága nagy mértékben függ a tesztelőtől. Ezért ha egy hibát nem tudunk azonosítani az integrált módszerrel, akkor meg kell ismételni a nyomkövetést a teszt adatbázis alkalmazása nélkül.

4.4 Program szeletelés

A program szeletelő komponens levágja az irreleváns részeket a végrehajtási fából a hibakeresés során. Ez a komponens akkor kerül meghívásra, amikor a egy program pontnál egy változó értékét a felhasználó hibásnak ítéli meg. A szeletelő komponens az adott változóra vonatkozólag kiszámít egy program szeletet. Ennek a szeletnek van egy megfelelő végrehajtási fája, amin az algoritmikus nyomkövető komponens folytatja a hibakeresést.

Kapcsolódó publikációim: [FKG91],[FKS92],[Gyi91b],[SzG91].

Az algoritmikus nyomkövetés és a CPM funkcionális tesztelés integrálását én dolgoztam ki valamint én végeztem el a CPM módszer kiterjesztésének tervezését.

5 Logikai programok algoritmikus nyomkövetése és tesztelése

Az előző fejezetben röviden ismertettük a GADT módszert, ami imperatív nyelvek algoritmikus nyomkövetésére szolgál. Ebben a fejezetben egy algoritmikus nyomkövetési módszert (IDTS) [GA93],[PGH94a],[PGH94b] ismertetünk, ami logikai programokra illetve funkcionális logikai programokra alkalmazható. Az IDTS (Integrated Debugging ,Testing and Slicing) módszer fő előnye, hogy a CPM algoritmus illetve a statikus program szeletelés alkalmazásával csökkenti a felhasználói kérdések számát.

Az IDTS módszer program szeletelési része az u.n. annotációs következtetési algoritmuson alapul [BPM93]. Ennek az a lényege, hogy egy logikai program argumentumainak input/output

irányultsága automatikusan származtatható a logikai program funkcionális részéből. Az annotáció alapján egy függőségi gráf állítható elő a logikai programot alkotó klózokra. Ezen függőségi gráf felhasználásával levágható egy hibás bizonyítási fa azon része, ami nincs hatással a hibára.

Az IDTS módszer az interaktív nyomkövetés során használja a CPM teszt adatbázist. A felhasználói válaszok alapján ez a teszt adatbázis folyamatosan aktualizálódik, így ez a módszer hatékonyan alkalmazható logikai programok tesztelésére is.

5.1 Jelölések

A továbbiakban P egy logikai programot M pedig ezen program interpretációját jelöli.

Definició 5.1 [Bizonyítási fa] (Proof tree) A P logikai program *bizonyítási fája* egy T cimkézett fa, amire teljesül, hogy

1. T minden csomópontja egy atommal van cimkézve.
2. Tartalmazzon T egy csomópontot a a_0 cimkével és legyen ezen csomópont n darab leszármazottjának cimkéje $a_1, \ldots, a_n (n \geq 0)$. Akkor P-ben létezik egy klóz $a_0' :- a_1', \ldots, a_n'$ amire teljesül, hogy minden $(i = 0, 1, \ldots, n)$-re, $a_i = a_i'\theta$ a θ átírásra.

A következő definícióban a $< p, x, y >$ hármas a p predikátum terminális hívását jelöli az x input és az y output értékkel.

Definíció 5.2 [Felső-szintű trace] (Top-level trace) A $< p, x, y >$ hármas *felső-szintű trace*-e egy olyan $(< p_1, x_1, y_1 >, < p_2, x_2, y_2 >, \ldots, < p_n, x_n, y_n >)$ véges sororozat (üres lehet), ahol p az x input hatására hívja p_1-et az x_1 inputtal és az output értéke y_1 ,..., végül hívja p_n-et az x_n inputtal és az output értéke y_n, és p terminal az y output értékkel.

A felső-szintű trace megfelel egy rezolúciós szintnek a bizonyítási fában.

Definíció 5.3 [Korrekt klóz] Legyen $C = (a :- a_1, \ldots, a_n)$ egy klóz P-ben és a' egy *ground* atom. Azt mondjuk, hogy a C klóz *fedi* az a' atomot akkor és csak akkor, ha van egy olyan θ helyettesítés, amire $a\theta = a'$ és $a_i\theta \in M$ minden $(i = 1, \ldots, n)$-re. C *korrekt* M-ben akkor és csak akkor, ha minden C által fedett a atom M-ben van; különben C nem korrekt M-ben.

Legyen q egy predikátum P-ben. Számozzuk meg q argumentum pozícióit és jelölje q_k a k-adik argumentum pozícióit. Jelölje $Argpos(q)$ a q argumentum pozícióinak halmazát és legyen $Argpos(P) = \bigcup_{q \in P} Argpos(q)$.

Definíció 5.4 [Program annotáció] Egy P program *annotációja* egy függvény $\mu : Argpos(P) \rightarrow \{\downarrow, \uparrow, \updownarrow, \square\}$,ahol a $\downarrow, \uparrow, \updownarrow, \square$ flag-eket *mód*-oknak nevezzük. (Elnevezés: $\downarrow$ *inherited*, $\uparrow$ *synthesized*, $\updownarrow$ *dual*, $\square$ *unannotated*). Egy annotáció *parciális* ha vannak dual vagy unannotated pozíciók.

Definíció 5.5 [Input és output pozíciók] Legyen C egy klóz és a_j egy atom C-ben. Legyen q az a_j atomhoz tartozó predikátum. Ha $\mu(q_k) = \downarrow$ és a_j a C head atomja vagy ha $\mu(q_k) = \uparrow$ és a_j egy body atom C-ben,akkor (C, j, q, k)-t egy *input* pozíciónak nevezzük. Ha $\mu(q_k) = \uparrow$ és a_j a C head atomja vagy ha $\mu(q_k) = \downarrow$ és a_j egy body atom C-ben, akkor (C, j, q, k)-t egy *output* pozíciónak nevezzük.

Jelölje $\mathcal{I}(C)$ és $\mathcal{O}(C)$ a C klóz input és output pozícióit. Az $\mathcal{I}(P) = \bigcup_{C \in P} \mathcal{I}(C)$ és $\mathcal{O}(P) = \bigcup_{C \in P} \mathcal{O}(C)$ halmazok a P program input,output pozícióit jelölik.

Jelölje $C_1 \mapsto C_2$ ha a C_2 klóz hívható C_1-ből, vagyis van egy a body atom C_1-ben, ami unifikálható C_2 head atomjával. A $\mapsto$ relációt *statikus hívási gráfnak* nevezzük. A továbbiakban feltételezzük, hogy adott egy $\mu : Argpos(P) \rightarrow \{\downarrow, \uparrow, \updownarrow, \square\}$ annotáció. Az attributum nyelvtanokhoz hasonlóan ezt az annotációt felhasználhatjuk az információ áramlási irányának meghatározására. A következő definíciókban β és γ argumentum pozíciókat jelölnek.

Definíció 5.6 [Lokális függőségi gráf] Minden C klózra definiáljuk a $\rightsquigarrow_C \subseteq \mathcal{I}(C) \times \mathcal{O}(C)$ *lokális függőségi gráf*-ot az alábbiak szerint:

$\beta \rightsquigarrow_C \gamma$ akkor és csak akkor, ha β és γ tartalmaz közös változót.

Definíció 5.7 [Átmenet gráf] Legyen C és D két klóz, b_0 D head atomja, a_j C egy body atomja és legyen a_j és b_0 unifikálható. A $\rightsquigarrow_{C,D}$ $\mathcal{O}(C \cup D) \times \mathcal{I}(C \cup D)$ feletti *átmeneti gráf*-ot az alábbiak szerint definiáljuk:

$$\gamma \rightsquigarrow_{C,D} \beta \quad \text{csakkor} \quad \begin{cases} \gamma = (C, j, q, k) \\ \beta = (D, 0, q, k) \\ \mu(q_k) = \downarrow \end{cases} \quad \text{vagy} \quad \begin{cases} \gamma = (D, 0, q, k) \\ \beta = (C, j, q, k) \\ \mu(q_k) = \uparrow \end{cases}$$

Definíció 5.8 [Globális függőségi gráf] A $\rightsquigarrow_G$ *globális függőségi gráf*-ot a következőképpen definiáljuk:

$$\rightsquigarrow_G = \bigcup_{C \in P} \rightsquigarrow_C \cup \bigcup_{C,D \in P} \rightsquigarrow_{C,D}$$

Jelölje $\rightsquigarrow_G^*$ a $\rightsquigarrow_G$ tranzitív és reflexív lezártját. A statikus szeletelés bevezetéséhez szükségünk van a duális és az unannotated pozíciók által indukált adat függőségekre, mivel ezek is okozhatják a hibás eredményt. Ezért bevezetjük a klóz függőségi gráfot, ami tartalmazza ezen adatáramlási éleket is.

Definíció 5.9 [Klóz függőségi gráf] Legyen C egy klóz és jelölje $\mathcal{U}(C)$ a C duál és unannotated pozícióit. Legyen $\beta \in \mathcal{U}(C)$, és $\gamma \in \mathcal{P}os(C)$, $\beta \neq \gamma$. A $\leftrightarrow_C \subseteq \mathcal{P}os(C) \times \mathcal{P}os(C)$ gráfot az alábbiak szerint definiáljuk:

$\beta \leftrightarrow_C \gamma$ és $\gamma \leftrightarrow_C \beta$ akkor és csak akkor, haf β és γ tartalmaz közös változót.

Ezekután a $\Leftrightarrow_C$ *klóz függőségi gráf* definíciója:

$$\Leftrightarrow_C = \rightsquigarrow_C \cup \leftrightarrow_C$$

Definíció 5.10 [Annotált bizonyítási fa] Legyen T a P program bizonyítási fája. Egy T_a *annotált bizonyítási fa* az alábbiak szerint definiált:

1. T_a-nak a csomópontjai megegyeznek T csomópontjaival.
2. Legyen a_0 egy T-beli csomópont n leszármazottal $a_1, \ldots, a_n$ $(n \geq 0)$, és legyen $C = (a_0' :- a_1', \ldots, a_n')$ a megfelelő klóz P-ben.Legyen β egy pozíció a_i-ben és γ egy pozíció a_j-ben, $0 \leq i \leq n$, $0 \leq j \leq n$. Legyenek $\beta' \in \mathcal{P}os(C)$ and $\gamma' \in \mathcal{P}os(C)$ a megfelelő pozíciók C-ben. A T_a-ban van egy irányított él β-ból γ-ba és ezt $\beta \rightarrow \gamma$ jelöli akkor és csak akkor, ha $\beta' \Leftrightarrow_C \gamma'$.

Jelölje $\rightarrow^*$ a $\rightarrow$ tranzitív reflexív lezártját.

Definíció 5.11 [T-szelet] Legyen T_a a P annotált bizonyítási fája és legyen γ egy pozíció T_a-ban. Egy *T-szelet* definíciója T_a felett az alábbi:

1. Minden csomópont a szeletben T_a-beli csomópont.

2. A T_a n csomópontja a szeletben van akkor és csak akkor, ha n tartalmaz egy β pozíciót, amire $\beta \rightarrow^* \gamma$.

A továbbiakban néhány formális definíciót adunk az előző fejezetben bevezetett CPM funkcionális tesztelési módszerre. Ezen definíciókat használjuk az $IDTS$ algoritmus ismertetése során.

Definíció 5.12 [Teszt váz] Legyen p a P program predikátuma és legyen $\Sigma_p = \sigma_{p1}, \ldots, \sigma_{pk}$ egy teszt specifikációja p-nek ahol σ_{pi}-k choice-kat jelölnek. Jelőlje $F(\Sigma_p)$ a $\sigma_{p1} \times \ldots \times \sigma_{pk}$-nek egy részhalmazát. $F(\Sigma_p)$ a D^n input tér egy osztályozása (ekvivalencia reláció D^n felett). $F(\Sigma_p)$ egy elemét *teszt váznak* nevezzük.

Definíció 5.13 [CPM tesztelés] Legyen p egy predikátum és Σ_p egy teszt specifikációja p-nek. A Σ_p *CPM tesztelését* a ϕ_p: $F(\Sigma_p) \rightarrow \{correct, incorrect, undefined\}$ leképezés definiálja.

Definíció 5.14 [CPM teszt konfiguráció] A P program egy *CPM teszt konfigurációja* a $F(\Sigma_p)$ felett egy véges halmaz az alábbi formában: $T(F(\Sigma_P)) = \{(p, f, i, o, e) \mid p$ egy predikátum, $f \in F(\Sigma_p)$, i egy teszt esete f-nek, o a p predikatum outputja az i inputon, és $e = \phi_p(f)\}$.

5.2 IDTS algoritmus

Először röviden ismertetjük Shapiro eredeti *single-stepping* nyomkövető algoritmusát[Sha83]. Legyen p egy olyan predikátum, ami terminál az x inputon, eredményül y értéket ad és $< p, x, y >$ nem eleme M-nek. Ebből következik, hogy a P programnak van legalább egy hibás klóza. Ennek megkeresésére postorder módon bejárjuk a T bizonyítási fát, amelynek $< p, x, y >$ a gyökere. A bejárás során minden $< q, u, v >$ csomópontra egy kérdés aktiválódik annak eldöntésére, hogy $< q, u, v >$ eleme-e M-nek. Tegyük fel, hogy az első negatív válasz a $< q, u, v >$ csomópontra érkezett. Jelőlje $< q_1, u_1, v_1 >, \ldots, < q_n, u_n, v_n >$ a $< q, u, v >$ direkt leszármazottait T-ben. Mivel a postorder bejárás biztosítja, hogy $< q_i, u_i, v_i > \in M$ minden $(1 \leq i \leq n)i$-re ezért ebből következik,hogy a $q(\ldots) :- q_1(\ldots), \ldots, q_n(\ldots)$ klóz fedi $< q, u, v >$ hármast, ami nem eleme M-nek. Az algoritmus megáll és hibás klózként visszaadja a $< q, u, v >$: $- < q_1, u_1, v_1 >, \ldots, < q_n, u_n, v_n >$ példányt.

A továbbiakban ismertetjük az $IDTS$ algoritmust, ami a CPM teszt adatbázis és a T-szelet használatával csökkenti a Shapiro féle algoritmus felhasználói kérdéseinek számát.

Algoritmus 5.1 [IDTS]

Input: $< p, x, y >$: egy hibás atom;
S: a T_a bizonyítási fa T-szelete az y hibás γ pozíciójára.
Output: r: a hibás klóz példány.

procedure IDTS($< p, x, y >, S, r$);
begin
Debug($< p, x, y >, S, r$,*found*);
let $r = (< q, u, v >:- < q_1, u_1, v_1 >, \ldots, < q_n, u_n, v_n >)$;
if (**not** *found*) **or** (Query($< q, u, v >$) = *correct*) **or**
(Query($< q_i, u_i, v_i >$) = *incorrect* néhány $(i = 1, \ldots, n)$-re **then**
/* nem konzistens teszt adatok */
fp($< p, x, y >, T, r$); /*single-stepping algoritmus */
/* különben r a hibás klóz példány */
end if;
end IDTS.

Algoritmus 5.2 [Nyomkövetés + CPM tesztelés + Szeletelés]

Input: $< p, x, y >$: egy hibás atom ;
S: T_a annotált bizonyítási fa T-szelete.
Output: q: hibás klóz példány;
found: *true* ha q azonosításra kerül,
false különben.

procedure Debug($< p, x, y >, S, q, found$);
begin
legyen ($< p_1, x_1, y_1 >, \ldots, < p_n, x_n, y_n >$) a $< p, x, y >$ felső-szintű trace;
found:= *false*; q := *empty*; i:= 1;
while (i $\leq$ n) **and** (**not** *found*) **do**
if $< p_i, x_i, y_i >$ **is in** S **then** Debug($< p_i, x_i, y_i >, S, q, found$);
i:= i+1;
end while;
if not *found* **then**
if $\phi_p(\chi_p(x)) = undefined$ **then** Query($< p, x, y >$); **end if**;
if $\phi_p(\chi_p(x)) = incorrect$ **then**
found := *true*;
$q := (< p, x, y >:- < p_1, x_1, y_1 >, \ldots, < p_n, x_n, y_n >)$;
end if;
end if;
end Debug.

Algoritmus 5.3 [Oracle]
Input: $< p, x, y >$: egy atom, amit vizsgálunk.
Output: *correct* ha $< p, x, y > \in M$,
incorrect ha $< p, x, y > \notin M$.

function Query($< p, x, y >$): {*correct*, *incorrect*};
begin
if $\exists(p, f, x, y, e) \in T(F(\Sigma_P))$ **where** ($e \neq undefined$) **then return** e
else
cpm:= $\phi_p(\chi_p(x))$;
oracle:= Benne van $< p, x, y >$ M-ben?; /* felhasználói konzultáció */
if *cpm* = *undefined* **then**
$T(F(\Sigma_P)) := T(F(\Sigma_P)) \cup \{(p, \chi_p(x), x, y, oracle)\}$
else
if ($cpm \neq oracle$) **then** /* nem konzisztens teszt adatok */
finomítsuk $T(F(\Sigma_p))$ CPM teszt konfigurációt $T(F(\Sigma_p^{'}))$-re
bevezetve a $f_1, f_2, \ldots, f_n$ teszt vázakat
/* $F(\Sigma_p^{'}) = F(\Sigma_p) \setminus \{\chi_p(x)\} \cup \{f_1\} \cup \{f_2\} \cup \ldots \cup \{f_n\}$ */
end if;
end if;
return *oracle*;
end if;
end Query.

Kapcsolódó publikációim: [HGA93],[PGH94a],[PGH94b],[AGK94].

A GADT módszer alkalmazását logikai program kőrnyezetben én vezettem be. Én javasoltam a fejezetben ismertetett T-szelet IDTS-be való bevezetését. (Korábban egy gyengébb statikus szelet módszer került alkalmazásra az IDTS-ben [PHG94a]).

6 Logikai programok statikus szeletelése

A szeletelési módszer a szoftver technoloógia több területén sikeresen alakalmazható imperatív nyelvekre: pl. algoritmikus nyomkövetés,verziókezelés,tesztelés,reverse engineering, karbantartás [BaH93],[BeE93],[HoR93]. Intuitíven egy program szelet egy adott váltózóra vonatkozóan (egy adott program pontnál) tartalmazza a program azon részeit, amelyek befolyásolhatják a változó értékét, vagy amelyek függhetnek a változótól. Az első esetben *backward* a második esetben *forward* szeletelésről beszélünk. Imperatív nyelvek esetén lehetőség van a változók közötti függőségek vizsgálatára, mivel az adatáramlás az értékadó utasításokkal és az input,output paraméterekkel meghatározott[HBR90],[Wei84]. Ez nem teljesűl logikai programokra, ahol az adatáramlás és adat függés implicit, így nehezebb vizsgálni.

A logikai programokra kifejlesztett statikus szeletelési módszerünk [GyP94] a program predikátumok (részleges) annotációján alapul. Ez az annotáció automatikusan generálható logikai programok funkcionális részeire. A funkcionális részek annotációja alapján következtetni lehet a program bizonyos további predikátumainak lehetséges annotációjára is. Az annotáció alapján egy program függőségi gráf konstruálható, és ezen gráf az alapja a szeletelési módszernek.

A fejezet további részében különböző szeletelési módszereket ismertetünk logikai programokra. Az ismertetésben használjuk a 5. fejezetben bevezetett definíciókat.

Definíció 6.1 [Program függőségi gráf] Legyenek C,D klózok és jelőlje $\mathcal{U}(C)$,$\mathcal{U}(D)$ a klózok dual és unannotated pozícióinak halmazát. Legyen b_0 a D head atomja, és a_j a C egy body atomja úgy, hogy a_j és b_0 unifikálható. Legyen $\beta = (C, j, q, k) \in \mathcal{U}(C)$, $\gamma = (D, 0, q, k) \in \mathcal{U}(D)$. A $\leftrightarrow_{C,D} \subseteq \mathcal{U}(C \cup D) \times \mathcal{U}(C \cup D)$ gráfot az alábbi formula definiálja:

$$\beta \leftrightarrow_{C,D} \gamma \text{ és } \gamma \leftrightarrow_{C,D} \beta$$

A $\rightsquigarrow_P$ *program függőségi gráf* a következőképpen definiált:

$$\rightsquigarrow_P = \rightsquigarrow_G \cup \bigcup_{C \in P} \leftrightarrow_C \cup \bigcup_{C,D \in P} \leftrightarrow_{C,D}$$

A $\rightsquigarrow_P^*$ gráf a $\rightsquigarrow_P$ tranzitív,reflexív lezártját jelöli.

A további definíciókban p, q, r a P logikai program pozícióit T pedig egy bizonyítási fát fog jelőlni.

Definíció 6.2 [p-szelet] Egy $\rightsquigarrow_P^*$ feletti *p-szelet*-et az r pozícióra vonatkozóan olyan halmaznak tekintjük, amelyre teljesül, hogy egy q pozíció eleme a p-szeletnek akkor és csak akkor, ha van egy irányított út q-ból r-be.

Definíció 6.3 [Egy p-szeletből származtatott atomok halmaza] Legyen s a P program p-szelete egy r pozícióra vonatkozóan. Az s-ből származtatott $P'(s)$ *atom halmaz* tartalmazza a P olyan atomjait, amelyeknek van s-beli pozíciója.

Definíció 6.4 [Izomorf p-szeletek] Legyen s_1, s_2 P-beli p-szeletek és legyen $P'(s_1), P'(s_2)$ a származtatott atomok halmaza. Az s_1, s_2 p-szeletek *izomorfak* akkor és csak akkor, ha $P'(s_1) = P'(s_2)$. Továbbá $s_1 \leq s_2$ akkor és csak akkor, ha $P'(s_1) \subseteq P'(s_2)$.

Definíció 6.5 [PT-szelet] Egy T feletti *PT-szelet* a γ pozícióra vonatkozóan egy olyan fa, amire:

1. A szelet minden csomópontja T-beli.
2. Egy n T-beli csomópont eleme a szeletnek akkor és csak akkor, ha n tartalmaz egy olyan β pozíciót, hogy β-nak megfelelő valamelyik program pozíció benne van a γ-nak megfelelő valamelyik program pozíció p-szeletében. (A bizonyítási fa minden belső csomópontjának két programbeli atom felel meg, a gyökérnek és a leveleknek egy-egy).

Definíció 6.6 [pa-szelet] Legyen T_a a P program annotált bizonyítási fája és γ egy pozíció a fában. Egy *pa-szelet* a P-beli pozíciók olyan halmaza, hogy egy q pozíció eleme a halmaznak akkor és csak akkor, ha van olyan β pozíció a fában, ami megfelel q-nak és $\beta \rightarrow^* \gamma$.

A következő állítás összehasonlítja az ebben a fejezetben bevezetett szeleteket és a Definíció 5.11-ben ismertetett T-szeletet.

Tétel 6.1 [A pa-szelet,p-szelet és a PT-szelet,T-szelet összehasonlítása] Legyen γ a P program T_a annotált bizonyítási fájának egy pozíciója és legyen r egy P-beli program pozíció. Jelölje $pa(\gamma)$ a pa-szeletet, $p(r)$ a p-szeletet, $T(\gamma)$ a T-szeletet és $PT(\gamma)$ a PT-szeletet. Akkor,

1. $pa(\gamma) \leq p(r_1) \cup p(r_2)$, ahol r_1,r_2 a γ-nak megfelelő pozíciók.
2. $T(\gamma) \subseteq PT(\gamma)$.

Látjuk, hogy a T-szelet precízebb mint a PT-szelet , azonban a T-szelet számítása nagyon tárigényes lehet, mivel tárolni kell az annotált bizonyítási fát. A továbbiakban egy olyan módszert ismertetünk, amivel elkerülhető az annotációs fa tárolása a T-szelet számításakor.

Definíció 6.7 [R-sorozat] Legyen T egy bizonyítási fa és n, m T csomópontjai. Jelölje a $n_1, n_2, \ldots, n_k$ sorozat az m-ből n-be tartó utat, ahol $m = n_1$ és $n = n_k$. Az m-ből n-be tartó *R-sorozatot* a $n_1, c_1, n_2, c_i, \ldots, c_{k-1}, n_k$ sorozat jelöli, ahol c_j $(1 \leq j \leq k-1)$ egy P-beli klózt azonosít az alábbiak szerint:

1. Ha n_j az n_{j+1} leszármazottja, akkor c_j az n_{j+1} csomópontnál alkalmazott szabályt jelöli.
2. Ha n_{j+1} az n_j leszármazottja, akkor c_j az n_j csomópontnál alkalmazott szabályt jelöli.
3. Ha az n_j,n_{j+1} csomópontoknak n_l a közös őse, akkor c_j az n_l csomópontnál alkalmazott szabályt jelöli.

Definíció 6.8 [P-sorozat] Legyen m, n a P program T bizonyítási fájának két csomópontja. Jelölje p_{n_j} a P egy olyan atomját, ami megfelel a T-beli n_j csomópontnak. Legyen $n_1, c_1,$, $\ldots$,c_{k-1}, n_k egy R-sorozat m-ből n-be. Az m, n csomópontok $a_1, b_1, a_2, b_2, \ldots, b_{k-2}, a_{k-1}$ *P-sorozata* olyan sorozat,amire teljesül:

1. Az a_1 él a p_m pozícióból indul $a_1 \in \Leftrightarrow^*_{c_1}$ (lsd. Definíció 5.9)
2. Az a_{k-1} él a p_n pozícióban végződik és $a_{k-1} \in \Leftrightarrow^*_{c_{k-1}}$.
3. Minden a_j $(1 \leq j \leq (k-1))$ él r kezdő és q vég pozíciójára teljesül,hogy $r \in Pos(p_{n_j})$, $q \in Pos(p_{n_{j+1}})$, r a b_{j-1} él végpontja és $a_j \in \Leftrightarrow^*_{c_j}$,ahol $p_{n_j}, p_{n_{j+1}}$ atomok a c_j klózban vannak.
4. Minden b_j $(1 \leq j \leq (k-2))$ él r kezdő és q vég pozíciójára teljesül,hogy $r \in Pos(p_{n_j})$ a c_j klózban, $q \in Pos(p_{n_j})$ a c_{j+1} klózban, r az a_j él végpontja és $b_j \in \Leftrightarrow^*_{u,v}$,ahol $u = c_j, v = c_{j+1}$ ha a p_{n_j} atom a c_{j+1} feje és $u = c_{j+1}, v = c_j$ ha a p_{n_j} atom a c_j feje.

Definíció 6.9 [NT-szelet] Legyen T_a a P program annotált bizonyítási fája és legyen γ egy T-beli pozíció. Egy *NT-szelet* a γ-ra vonatkozóan olyan fa, amire:

1. A szelet minden csomópontja T_a-beli elem.
2. Az m csomópont a szeletben van akkor és csak akkor, ha van olyan P-sorozat m-ből n-be, aminek végpontja γ.

A következő állítás megmutatja,hogy a NT-szelet megegyezik a T-szelettel, így ezzel a módszerrel lehetőség van a T-szelet számítására az annotált bizonyítási fa tárolása nélkül.

Tétel 6.2 [Az NT-szelet és aT-szelet összehasonlítása] Legyen T_a egy annotált bizonyítási fa a γ pozícióval. Ekkor $NT(\gamma) = T(\gamma)$, ahol $NT(\gamma)$ az NT-szeletet, $T(\gamma)$ a T-szeletet jelöli.

Kapcsolódó publikációim: [GyP94],[JGH94a],[JGH94b],[HGA93].

A fejezetben bevezetett statikus szeletelési módszerek definiálása és kapcsolatuk vizsgálata tőlem származik.

7 Interaktív Induktív Logikai Programozás

Az Induktív Logikai Programozás (ILP) feladata reláció definíciók megtanulása logikai programok alakjában[Mug92], [MuR94]. Interaktív ILP esetében a tanuló rendszer használhat u.n. *oracle*-t a bevezetett szabályok érvényesítésére. A Shapiro által ismertetett ILP módszer a 4. fejezetben vázolt algoritmikus nyomkövetési technikán alapul. Ezért az általunk bevezetett $IDTS$ módszer, ami csökkenti a Shapiro féle nyomkövetési algoritmus kérdéseinek számát hatékonyan alkalmazható ILP környezetben is.

Ebben a fejezetben az $IDTS$ algoritmus egy további alkalmazását ismertetjük. Az olyan ILP módszereket, amelyek egy kezdeti hipotézist (theory) elfogadnak és ezt módosítják (revise) a tanulás során Theory Revision (TR) módszereknek nevezzük [Mug92].

Egy ismert TR módszer az u.n. $SPECTRE$ algoritmus, ami az unfolding transzformációs szabállyal specializálja a kezdeti hipotézist [BoI94]. Az általunk bevezetett $IMPUT$ módszer az $IDTS$ és a $SPECTRE$ algoritmusok integrálásával optimalizálja a klózok kiválasztását az unfolding transzformációra [AGB94]. Ezzel a módszerrel olyan logikai programok is helyesen specializálhatók, amelyeket az eredeti $SPECTRE$ algoritmus nem tudott elvégezni.

Definíció 7.1 [Deriváció, resolvens] Legyen G_i egy $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ *goal*, és C_{i+1} egy $A \leftarrow B_1, \ldots, B_q$ klóz és legyen R egy számítási szabály. Azt mondjuk, hogy G_{i+1} a G_i-ből és C_{i+1}-ből *derivációval* áll elő a θ_{i+1} mgu (most general unifier) és az R alkalmazásával, ha az alábbi feltételek teljesülnek:

1. A_m az R által meghatározott atom.
2. $A_m\theta_{i+1} = A\theta_{i+1}$ (vagyis θ_{i+1} egy mgu-ja A_m-nek és A-nak).
3. G_{i+1} egy $(\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_{i+1}$ goal.

A rezolúciós terminológiában G_{i+1} egy *resolvense* G_i-nek és C_{i+1}-nek.

Definíció 7.2 [Unfolding] Legyen p_i egy $H \leftarrow A_1, \ldots, A_m, \ldots, A_k$ klóz a P logikai programban és $C = \{c_1, \ldots, c_q\}$ legyen olyan klózok halmaza ,amelyekre minden $c_j \in C$ klóz feje unifikálható

az A_m atommal. Akkor a P' program az *unfolding* transzformáció után a következő:

$$\mathbf{P'} = \mathcal{U}(\mathbf{P}) = \mathbf{P} \setminus \{p_i\} \cup \Big(\bigcup_{c_j \in C} H \leftarrow A_1, \ldots, A_{m-1}, body(c_j), A_{m+1}, \ldots, A_k \Big)$$

A *SPECTRE* algoritmus a kezdeti logikai programot a pozitív és negatív példák alapján specializálja. Ha egy klóz csak negatív példákat fed le, akkor ezt a klózt töröljük a programból. Ha egy klóz negatív és pozitív példákat is lefed, akkor erre a klózra végrehajtunk egy unfolding lépést. Az algoritmus egy optimalizáló eljárást tartalmaz az adott klóz azon *atomjának* a kiválasztására, amelyre az unfolding végrehajtódik. Azonban a *SPECTRE* módszerben nincs optimalizálás azon *klóz* kiválasztására, amelyre a soron következő unfolding lépést alkalmazzuk.

Az *IMPUT* algoritmusban az *IDTS* módszerrel megkeressük azt a klózt, ami felelős egy negatív példa lefedéséért és erre alkalmazzuk az unfolding transzformációt.

Algoritmus 7.1 [Az IMPUT algoritmus]
Input: Egy $P = \{p_1, \ldots, p_u\}$ kezdeti program, egy $\mathbf{B} = \{b_1, \ldots, b_v\}$ háttér tudás (background knowledge) (olyan klózok halmaza, amelyek nem változnak a tanulás során), az $\mathbf{E}^+$, $\mathbf{E}^-$ pozitív és negatív példák.
Output: A P, P', ..., $P'^{(n)}$ program sorozat, ahol $(P'^{(0)} = P)$ és $P'^{(i+1)} = \tilde{\mathcal{U}}(P'^{(i)})$ $(0 \leq i < n)$, $\tilde{\mathcal{U}}$ jelöli az unfolding operátort.

Az algoritmus:

let i = 0.
while van egy olyan $e^- \in \mathbf{E}^-$, amelyre $\mathbf{P}'^{(i)}$ nem fail e^--n **do**
begin
  Keressünk egy hibás $c \in \mathbf{P}'^{(i)}$ klózt az IDTS használatával
  if c nem fordul elő pozitív példa levezetésében
  then Töröljük c-t $\mathbf{P}'^{(i)}$-ből $(\mathbf{P}'^{(i+1)} = \mathbf{P}'^{(i)} \setminus \{c\})$.
  else begin
    Hajtsuk végre az unfolding lépést a c-n.
    Jelölje $\mathbf{C} = \{c_1, \ldots, c_s\}$ a c resolvenseit.
    let $\mathbf{P}'^{(i+1)} = \mathbf{P}'^{(i)} \setminus \{c\} \cup \mathbf{C}$
  end
  let i = i + 1
end

Kapcsolódó publikációim: [AGB94],[JGH94a],[HGA93].

A SPECTRE és az IDTS módszer integrálását én vezettem be és én definiáltam a fejezetben szereplő IMPUT algoritmust.

8 Attributum nyelvtanok tanulása

Imperatív nyelvek fordítóprogramjai, interpreterjei hatékonyan megvalósíthatók attributum nyelvtanos specifikáció alapján[AlM91]. Azonban attributum nyelvtanos specifikáció készítése reális méretű nyelvekre nagyon komplex feladat[DJL88]. Ezért gyakorlati alkalmazásoknál is hasznos lehet egy olyan módszer, ami példák alapján meg tud tanulni bizonyos attributumos szabályokat.

Az általunk bevezetett *AGLEARN* [GyH94] attributum nyelvtan tanuló módszer fejlesztését az attributum nyelvtanok és a logikai programok közötti szoros kapcsolat motiválta[DeM93]. Az *AGLEARN* módszer az *ILP* alkalmazásokhoz hasonlóan feltételezi a háttér tudás használatát az összetett nyelvi struktúrák és relációk reprezentálására[Mug92]. Az algoritmus jelenlegi változata az attributum nyelvtan kőrnyezet-független szintaxisát ismertnek tekinti, és a háttér attributumos szabályok valamint a példák alapján *S-attributumos* illetve *L-attributumos* [Boc76] szabályok megtanulására képes. A módszer az alábbi három fő lépésből áll:

1. Az attributumos szabályok tanulási problémáját transzformáljuk propozionális tanulási feladattá.
2. Egy propozionális tanulási algoritmussal megoldjuk ezt a feladatot propozionális formában.
3. Az eredményül kapott propozionális hipotézist transzformáljuk attributumos szabályokká.

A módszer ismertetése előtt a jelölések egyértelműsége miatt az attributum nyelvtanokhoz kapcsolódó fogalmakat vezetünk be. (Csupán a későbbiekben használt fogalmak jelölését ismertetjük, az attributum nyelvtanok átfogó, formális definíciója megtalálható például [AlM91]-ben).

Egy *attributum nyelvtan* az egy $AG = (G, SD, AD, R, C)$ ötös, ahol

1. $G = (N, T, P, S)$ egy kőrnyezet-független nyelvtan.
2. $SD = (\mathcal{T}, \mathcal{F})$ a szemantikus domain, ahol $\mathcal{T}$ a (*típusok*) , $\mathcal{F}$ a *függvények* halmaza.
3. $AD = (Attr, Inh, Syn, Type)$ az attributumok leírása.
4. R a szintaktikus szabályokhoz rendelt attributumos szabályok (*szemantikus függvények*) halmaza. Egy szemantikus függvény alakja:

$$(a, p, k) = f((a_1, p, k_1), \ldots, (a_n, p, k_n))$$

5. C a szabályokhoz rendelt szemantikus feltételek halmaza.

Egy attributum nyevtan *S-attributumos* nyelvtan ha csak szintetizált attributumot tartalmaz, *L-attributumos* nyelvtan, ha minden $X_0 \rightarrow X_1 \ldots X_n$ szabály X_i $(1 \leq i \leq n)$ örökölt attributumára teljesül, hogy csak olyan attributumoktól függ, amelyek benne vannak a $\bigcup_{1 \leq j < i} Attr(X_j) \cup Inh(X_0)$ halmazban.

8.1 A tanulási probléma definiálása

Az *AGLEARN* módszer feladata az alábbiak szerint definiált:
Adott:

1. Egy egyértelmű $G = (N, T, P, S)$ kőrnyezet-független nyelvtan. Feltételezzük, hogy a P szabály halmaz két diszjunkt részre van osztva; P_B jelöli a *background* és P_T jelöli *target* szabályokat.
2. Egy SD = $(\mathcal{T}, \mathcal{F})$ szemantikus domain halmaz. Feltételezzük, hogy az attributumok típusa és a szemantikus függvények formája adott.
3. Egy $AD = (Attr, Inh, Syn, Type)$ attributum leírás. Feltételezzük, hogy a nyelvtani szimbólumok attributumai ismertek.

4. Egy R attributum szabály halmaz, ami parciálisan definiált. R teljesen definiált a P_B-beli szabályokra és nem definiált a P_T-beliekre. Ugyanez érvényes a C szemantikus feltételekre is.

5. Minden $p \in P_T$-re egy E_p példa halmaz, amit E_p^+ pozitív és E_p^- példákra osztunk. Legyen a p szabály $X_0 \rightarrow X_1 \ldots X_k$, és $e \in E_p$ egy példa az alábbi formában: $e = (w, (a_1^{(i)}, v_1), \ldots, (a_n^{(i)}, v_n), (a_{n+1}^{(s)}, v_{n+1}), \ldots, (a_m^{(s)}, v_m))$, ahol $X_0 \Rightarrow^* w$, $\bigcup_{1 \leq k \leq n} \{a_k^{(i)}\} = Inh(X_0)$, és $\bigcup_{1 \leq k \leq m} \{a_k^{(s)}\} = Syn(X_0)$, továbbá $a_j^{(t)} \neq a_k^{(t)}$ ha $j \neq k$, $(t \in \{i, s\})$. Egy $(a_j^{(t)}, v_j)$ $(t \in \{i, s\})$ pár az e példában azt jelöli,hogy az X_0 $a_j^{(t)}$ attributuma a v_j értéket kapja a w stringre kiszámított attributumos fában.

Cél:

A $p \in P_T$ szabályokra olyan $R(p)$, $C(p)$ halmazok definiálása, hogy az eredményül kapott attributum nyelvtan konzisztens legyen a példákkal.

8.2 S-attributumos szabályok tanulása

Legyen p: $X_0 \rightarrow X_1 X_2 \ldots X_n$ a P_T egy szabálya, és jelölje, E_p^+ a pozitív, E_p^- a negatív példák halmazát. Minden $a^{(s)} \in Syn(X_0)$-re egy $T(a^{(s)})$ táblát konstruálunk, amelynek sorai megfelelnek egy-egy példának. A tábla oszlopait az alábbi formula adja meg:

$$\{class, word, target\} \cup \mathcal{U} \cup \mathcal{C} \cup \mathcal{F}_{UR} \cup \mathcal{F}_{UCR} \cup \mathcal{F}_{UF} \cup \mathcal{F}_{UCF}$$

ahol egy adott $e_p = (w, (a^{(s)}, v))$ példára az oszlopok definíciója a következő:

1. $class(e_p) = \begin{cases} + & \text{ha } e_p \in E_p^+ \\ - & \text{különben} \end{cases}$

2. $word(e_p) = w$

3. $target(e_p)$ az $a^{(s)}$ értéke e_p-ben, vagyis $target(e_p) = v$

4. Legyen $X_k.a$ egy attributum példány, $k > 0$. Akkor van egy $X_k.a$ oszlop $\mathcal{U}$-ben, és az oszlop értéke P_B-beli attributum értékek alapján kiszámíthatók.

5. Legyen $X_k.a$ egy attributum példány, $k > 0$ úgy, hogy $type(a^{(s)}) = type(X_k.a)$. Akkor van egy $a^{(s)} = X_k.a$ oszlop C-ben. C tartalmazza az u.n. copy szabályoknak megfelelő relációkat.

6. Legyenek $X_{k_1}.a_1, \ldots, X_{k_l}.a_l$ alkalmazott attributum előfordulások, $(0 < k_1, \ldots, k_l \leq n)$ és legyen f: $\tau_1 \times \ldots \times \tau_l \rightarrow \{true, false\}$ egy Boolean függvény $(f \in \mathcal{F})$, amire $type(X_{k_i}.a_i) = \tau_i$ $(1 \leq i \leq l)$. Akkor van egy $f(X_{k_1}.a_1, \ldots, X_{k_l}.a_l)$ oszlop $\mathcal{F}_{UR}$-ben.

7. Legyen $X_k.a$ egy alkalmazott előfordulás $(0 < k \leq n)$ és legyen $\{c_1, \ldots, c_n\}$ azon konstans értékek halmaza, amelyek előfordulnak az U $X_k.a$ oszlopában. Akkor minden c_i-re van egy $X_k.a = c_i$ oszlop $\mathcal{F}_{UCR}$-ben $(1 \leq i \leq n)$.

8. Legyenek $X_{k_1}.a_1, \ldots, X_{k_l}.a_l$ alkalmazott attributum előfordulások $(0 < k_1, \ldots, k_l \leq n)$ és legyen f: $\tau_1 \times \ldots \times \tau_l \rightarrow \tau_0$ egy függvény $(f \in \mathcal{F})$,amire $type(X_{k_i}.a_i) = \tau_i$ $(1 \leq i \leq l)$ és $type(a^{(s)}) = \tau_0$. Akkor van egy $a^{(s)} = f(X_{k_1}.a_1, \ldots, X_{k_l}.a_l)$ oszlop $\mathcal{F}_{UF}$-ben.

9. Legyen $\{c_1, \ldots, c_n\}$ azon konstans értékek halmaza, amelyek előfordulnak U-ban vagy a $target(e_p)$ oszlopban. Akkor minden c_i-re van egy $a^{(s)} = c_i$ oszlop $\mathcal{F}_{UCF}$-ben akkor és csak akkor, ha $type(c_i) = type(a^{(s)})$ $(1 \leq i \leq n)$.

A fentiek alapján megkonstruálható egy propozionális tábla, amiből egy megfelelő következtetési módszerrel *if* szabályok állíthatók elő az alábbi formában:

$\oplus_1$: if $c_{11} = b_{11}$ & ... & $c_{1l_1} = b_{1l_1}$

$\vdots$

$\oplus_k$: if $c_{k1} = b_{k1}$ & ... & $c_{kl_k} = b_{kl_k}$

ahol $c_{ij} \in (\mathcal{F}_{UR} \cup \mathcal{F}_{UCR} \cup \mathcal{F}_{UF} \cup \mathcal{F}_{UCF})$ és $b_{ij} \in \{true, false\}$ $(1 \leq i \leq k, 1 \leq j \leq l_i)$. A $c_{ij} = b_{ij}$ Bool kifejezést *condition item*-nek nevezzük,ha $c_{ij} \in (\mathcal{F}_{UR} \cup \mathcal{F}_{UCR})$. A $c_{ij} = b_{ij}$ Bool kifejezést *assignment item*-nek nevezzük, ha $c_{ij} \in (\mathcal{F}_{UF} \cup \mathcal{F}_{UCF})$. Minden l-re $(1 \leq l \leq k)$ legfeljebb egy $c_{ij} = b_{ij}$ assignment item fordulhat elő $\oplus_l$-ben,továbbá b_{ij} értékének igaznak kell lenni.
Az *AGLEARN* módszer akkor fogad el egy *if* szabály halmazt, ha az alábbi feltételek teljesülnek:

- ha $c_{ij} \in (\mathcal{F}_{UF} \cup \mathcal{F}_{UCF})$ akkor $b_{ij} = true$.
- ha $c_{ij}, c_{qr} \in \mathcal{F}_{UF} \cup \mathcal{F}_{UCF}$ akkor $i \neq q$ vagyis pontosan egy szabály definiálja a *target* attributumot.
- ha valamely i-re $(1 \leq i \leq k)$ $\oplus_i$: c_{i1} úgy,hogy $c_{i1} \in (\mathcal{F}_{UF} \cup \mathcal{F}_{UCF})$ akkor nincs olyan $j \neq i$ $(1 \leq j \leq k)$, amire $\oplus_j$: c_{j1} és $c_{j1} \in (\mathcal{F}_{UF} \cup \mathcal{F}_{UCF})$.

Az módszer által elfogadott *if* szabályok a következő formában adottak:

(i) $\oplus_1$: $c_{11} = b_{11}$ & ... & $c_{1l_1} = b_{1l_1}$

$\vdots$

$\oplus_i$: $c_{i1} = b_{i1}$ & ... & $c_{il_i} = b_{il_i}$

(ii) $\oplus_{i+1}$: $c_{i+1,1} = b_{i+1,1}$ & ... & $c_{i+1,l_{i+1}-1} = b_{i+1,l_{i+1}-1}$ & $a_{i+1,l_{i+1}} = true$

$\vdots$

$\oplus_j$: $c_{j1} = b_{j1}$ & ... & $c_{j,l_j-1} = b_{j,l_j-1}$ & $a_{j,l_j} = true$

(iii) $\oplus_{j+1}$: $a_{j+1,1} = true$

ahol c_{pq} egy condition item-et a_{rt} egy assignment item-et jelől.

Az elfogadott *if*szabály halmazból az *AGLEARN* módszer generálja az attributumos szabályokat az alábbiak szerint:

(a) Az (i)-beli szabályokból szemantikus feltételeket álítunk elő:

if not ($c_{11} = b_{11}$ & ... & $c_{1l_1} = b_{1l_1}$) **then** *error*

$\vdots$

if not ($c_{i1} = b_{i1}$ & ... & $c_{il_i} = b_{il_i}$) **then** *error*

(b) Az (ii)-beli szabályokból feltételes szemantikus függvények generálódnak:

if ($c_{i+1,1} = b_{i+1,1}$ & ... & $c_{i+1,l_{i+1}-1} = b_{i+1,l_{i+1}-1}$) **then** $attr = f_{i+1}(\ldots)$
else
  if ...

  $\vdots$

  else
    if ($c_{j,1} = b_{j,1}$ & ... & $c_{j,l_j-1} = b_{j,l_j-1}$) **then** $attr = f_j(\ldots)$
    else $Statement_c$

(c) $Statement_c = \begin{cases} error & \text{ha } (iii) \text{ rész üres} \\ attr = f_{j+1}(\ldots) & \text{különben} \end{cases}$

ahol az $attr = f_j(\ldots)$ megfelel a $a_j = true$ assignment item-nek.

Kapcsolódó publikációm: [GyH94]

Az attributum nyelvtanok tanulási problémájának definiálása és a fejezetben ismertetett transzformációs algoritmus bevezetése a tanulási feladatra tőlem származik.

References

[AFD90] Alexin Z., Fábricz K., Dombi J., Gyimóthy T., Horváth T.: Constructur: A Natural Language Interface Based on Attribute Grammars. *Acta Cybernetica*, 9. 1990. 247-257.

[AFG90] Alexin Z., Fábricz K., Gyimóthy T., Horváth T.: Grammar Specification for Natural Language Understanding Interface. *Proc. of WAGA*, Paris, LNCS 461.Springer-Verlag 1990. 313-327.

[AGB94] Alexin Z., Gyimóthy T., Boström H.: Interactive Multiple Predicate Revision based on Unfolding Transformation. *submitted to Acta Cybernetica.*

[AGK94] Alexin Z., Gyimóthy T., Kókai G.: IDT: A System for Debugging Testing and Learning of Prolog Programs. *manuscript, will be submitted to Symposium on Programming Language and Software Tools.* Visegrád 1995.

[AlM91] Alblas H., Melichar B. (eds.): *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems* (SAGA),LNCS (Lecture Notes in Computer Science) 545, Springer-Verlag, 1991.

[ASU85] Aho A.V., Sethi R., Ullman J.D.: Compilers:Principles, Tecniques and Tools. *Addison-Wesley,Reading,MA*, 1985.

[BaH93] Bates S., Horwitz S.: Incremental Program Testing Using Program Dependence Graphs, in: *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 1993. 384–396.

[BeE93] Beck J., Eichmann D.: Program and Interface Slicing for Reverse Engineering. In: *Proc. 15th Int. Conference on Software Engineering*, Baltimore, Maryland, 1993. IEEE Computer Society Press, 1993, 509–518.

[BoB87] Bolognesi T., Brinksma H.: Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*,1987.

[Boc76] Bochmann, G.V., Semantic Evaluation from Left to Right, *Comm. ACM* 19(2):55–62 (1976).

[BoI94] Boström H., Idestam-Almquist P.: Specialization of Logic Programs by Pruning SLD-trees. In: *Proc. 4th Int. Workshop on Inductive Logic Programming* (ILP-94) (S.Wrobel, ed.), Bad Honnef/Bonn, 1994. GMD-Studien Nr. 237, Gesellschaft für Mathematik und Datenverarbeitung, 1994. 31-47.

[BPM93] Boye J., Paakki J., Małuszyński J.: Synthesis of Directionality Information for Functional Logic Programs, in: *Proceedings of the 3rd International Workshop on Static Analysis*, Lecture Notes in Computer Science 724, Springer-Verlag, 1993. 165–177.

[DeL93] Džeroski S.,Lavrač N.: Inductive Learning in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5. No. 6. 1993.

[DeM93] Deransart P., Małuszyński, J.: *A Grammatical View of Logic Programming*, MIT Press, 1993.

[DJL88] Deransart P., Jourdan M., Lorho B.: *Attribute Grammars - Definitions, Systems and Bibliography*, Lecture Notes in Computer Science 323, Springer-Verlag, 1988.

[FAG90] Fábricz K., Alexin Z., Gyimóthy T., Horváth T.: THALES – a Software Package for Plane Geometry Constructions. *Proc. of COLING'90*,Helsinki, 1990. 44-46.

[FGK91] Fritzson P., Gyimóthy T., Kamkar M., Shahmehri N.: Generalized Algorithmic Debugging and Testing, in: *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices* 26(6):317–326 (1991).(An extended version published in ACM LOPLAS see [FKS92]).

[FKS92] Fritzson P., Kamkar M., Shahmeri N., Gyimóthy T.: Generalized Algorithmic Debugging and Testing. *ACM LOPLAS*, 1992. 1.4. 303-323.

[Fu82] Fu K.S.: Syntactic Patterrn Recognition and Applications. *Prentice-Hall,Englewoood Cliffs*, 1982.

[GAS92] Gyimóthy T.,Alexin Z.,Szűcs R.: Integrated Graphic Environment to Develop Applications Based on Attribute Grammars. *Proc. of CC'92, Paderborn, FRG, LNCS 641* Springer Verlag 1992. 51-59.

[GHK88] Gyimóthy T.,Horváth T.,Kocsis F.,Toczki J.: Incremental algorithms in PROF-LP. *Proc. of CC'88, Berlin, LNCS 371* Springer Verlag 1988. 93-103.

[GSM83] Gyimóthy T., Simon E., Makay Á.: An implementation of the HLP. *Acta Cybernetica*, 6.3. 1983. 315-327.

[GyH94] Gyimóthy T., Horváth T.: Learning Attribute Grammars. *munuscript, will be submitted to ILP'95.*

[Gyi91a] Gyimóthy T.: Natural-Language Interface Construction Based on Attribute Grammars. *Proceedings of the International Summer School on Attribute Grammars, Applications and Systems* (SAGA),LNCS 545, Springer-Verlag, 1991. 460-469.

[Gyi91b] Gyimóthy T.: Software Generation Based on Attribute Grammars. *Periodica Polytechnica Ser.El.Eng.* 35.2. 1991. 147-152.

[GyK88] Gyimóthy T., Kocsis F.: Incremental Generation of the LL(1) Parsing Table. *Proc. of Symp. on Automata,Languages and Programming Systems*,Salgótarján. 1988. 157-161.

[GyP94] Gyimóthy T., Paakki J.: Static Slicing of Logic Programs. *accepted to AADEBUG'95.*

[GyT87] Gyimóthy T., Toczki J.: Syntactic Pattern Recognition in the HLP/Pas system. *Acta Cybernetica*, 8.1. 1987. 79-88.

[HAF90] Horváth T.,Alexin Z.,Fábricz K.,Gyimóthy T.: Towards a Multilingual Natural Language Understanding Interface. *Proc. of the CC'90, Schwerin, FRG, LNCS 477* Springer Verlag 1990. 217-219.

[HGA93] Horváth T., Gyimóthy T., Alexin Z., Kocsis F.: Interactive Diagnosis and Testing of Logic Programs, in: Tombak, M. (ed.), *Proceedings of the 3rd Symposium on Programming Languages and Software Tools*, Estonia 1993. 34–46.

[HoR92] Horwitz S., Reps T.: The Use of Program Dependence Graphs in Software Engineering, in: *Proceedings of the 14th International Conference on Software Engineering*, IEEE Computer Society Press, 1992. 392–410.

[HRB90] Horwitz S., Reps T., Binkley D.: Interprocedural Slicing Using Dependence Graphs, *ACM TOPLAS*, 12(1):26–61 (1990).

[JAK88] JAKE: The Application-Independent Natural Language Interface. *English Knowledge Systems,Inc* Scotts Valley, California, 1988.

[JaW75] Jazayeri M., Walter K.G.:Alternating Semantic Evaluator. *Proc. of ACM. Annual Conference* 1975. 230-234.

[Kas80] Kastens U.: Ordered Attribute Grammars. Acta Informatica 13 (1980), 229-256.

[Knu68] Knuth, D.E.: Semantics of Context-Free Languages, *Mathematical Systems Theory* 2:127–145 (1968).

[MKM89] Makay Á., Kuba A., Máté E.: Software system for Nuclear Medicine Processing. In:*Proc. Symposium on Programming Languages and Software Tools* (T.Gyimóthy ed.), Szeged 1989. 104-108.

[Mug92] Muggleton S. (ed.): *Inductive Logic Programming*, Academic Press, 1992.

[MuR94] Muggleton S., De Raedt L.: Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 12, 1994.

[OsB88] Ostrand T.J., Balker, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests, *Comm. ACM* 31(6):676–686 (1988).

[PGH94a] Paakki J., Gyimóthy T., Horváth T.: Effective Algorithmic Debugging for Inductive Logic Programming. In: *Proc. 4th Int. Workshop on Inductive Logic Programming* (ILP-94) (S.Wrobel, ed.), Bad Honnef/Bonn, 1994. GMD-Studien Nr. 237, Gesellschaft für Mathematik und Datenverarbeitung, 1994. 175–194.

[PGH94b] Paakki J., Gyimóthy T., Horváth T.: An Integrated Method for Algorithmic Debugging of Logic Programs. *submitted to the Journal of Logic Programming.*

[RSS78] Räihä K.J., Saarinen M., Siosalon-Soinnen E., Tienari M.: The compiler writing system HLP. *Technical reports of University of Helsinki* 1978. 2.

[Sch88] Schröder M.: Evaluating User Utterances in Natural Language Interfaces to Databases. *Computers and Artifical Intelligence*, 7.4. 1988. 317-337.

[SFH89] Somos E., Forgács I., Horváth A.: MICROTEST – a New Testing System on PC. In:*Proc. Symposium on Programming Languages and Software Tools* (T.Gyimóthy ed.), Szeged 1989 318-331.

[Sha83] Shapiro E.: *Algorithmic Program Debugging*, MIT Press, 1983.

[SKF90] Shahmeri N., Kamkar M., Fritzson P.: Semi-automatic Bug Localization in Software Maintanance. *Proc. of IEEE Conf. on Soft. Maintenance.* San Diego 1990.

[SzG91] Szücs R., Gyimóthy T.: T-GEN: An Extended Version of the Category Partition Testing Method, in: Koskimies, K. and Räihä, K.-J. (eds.), *Proceedings of the 2nd Symposium on Programming Languages and Software Tools*, 1991. 70–77.

[TsF80] Tsai W.H., Fu K.S.: Attributed grammar – a tool for combining syntactic and statistical approeches to pattern recognition. In:*IEEE Trans.SMC 10.*1983. 843-855.

[TGH88] Toczki J., Gyimóthy T., Horváth T., Kocsis F.: Automatic Software Generation in Practice- New Features in the HLP/Pas System. *Proc. of System and Implementation Workshop*, Tallin. 1988. 103-120.

[TGJ88] Toczki J., Gyimóthy T., Jáhni G.: Implementation of a LOTOS precompiler in PROF-LP. *Proc. of Symp. on Programming Design*, Budapest,1988. 31-36.

[TGK90] Toczki J., Gyimóthy T., Kocsis F., Dányi G., Kókai G.: SYS/3 – a Software Development Tool. *Proc. of Compiler Construction* Schwerin, FRG, LNCS 477. Springer-Verlag 1990. 313-327.

[TSG86] Toczki J., Simon E.,Garai T., Kocsis F.,Gyimóthy T.: Automatic Compiler Genaration. *Proc. of Symp. on Automata,Languages and Programming Systems*,Salgótarján. 1986. 289-297.

[Wei84] Weiser M.: Program Slicing, *IEEE Transactions on Software Engineering* SE-10(4):352–357 (1984).