Részleges kiértékelés

Hajós Gergely

Parciális Evaluáció

A fogalomnak sokféle definíciója létezik, a következőekben azonban mindegyik megegyezik: a parciális evaluátor két paramétert kap: egy programot és egy részben felparaméterezett lekérdezést (query). Egyetlen kimenő paramétere pedig egy program.

Két igen mérvadó megfogalmazás:

D1.: Dan Sahlin (a Mixtus magalkotója) két szigorú megkötést tesz, melyeket további kiegészítések követnek:

1. a kapott program az eredetivel működésében tökéletesen megegyezik
2. a parciális evaluátor minden körülmények között leáll.

D2.: Jocelyn Frechot definíciója alapján, a parciális evaluáció nem más, mint a megadott paraméterek ismeretében a program egyszerűsítésének automatizált módja.

A fent leírt definíciók jól leírják a részleges kiértékelés problémáját, de nem festenek teljes képet a részleges kiértékelők családjáról. A parciális evaluátorok legfőbb célja valóban a program egyszerűsödése, vagyis az, hogy hatékonyabb legyen a programfutás, valamint esetlegesen a program hossza is csökkenjen. Ez azonban nem jelenti azt, hogy akár egy igen hatékony részleges kiértékelő speciális esetekben esetleg ne produkálna az eredetinél lassabb programkódot. Tehát nem állítható, hogy minden esetben egyszerűsödést érünk el.
A parciális evaluátorok egy igen nagy családja nagyfokú felhasználói interakciót igényel a részleges kiértékelés során. Ezek a programok feltételezik, hogy a kiértékelni kívánt program írója, vagy annak jó ismerője használja őket. Ezek a parciális evaluátorok jóval egyszerűbbek lehetnek automatizált társaiknál, kihasználva a felhasználó, programozó esetleges emberi intelligenciáját. Vagyis sok olyan részfeladatra optimalizáló program létezik, melyek valóban parciális evaluátorok, de mégsem teljesítik az automatizmus feltételét.
A részleges kiértékelőket valóban egy, vagy több parametrizált lekérdezéssel futtatják, azonban előfordulhat, hogy felhasználója nem ad meg plusz paramétereket, mivel a kiértékelés során ezek nélkül is érhetők el hatékonyságbeli javulások. Ezeknek több oka is lehet ilyen pl. a program szerkezetének átalakítása, vagy esetleg a program belsejében lezajló behelyettesítések, melyeket a részleges kiértékelők hatékonyan kihasználnak.

Mindezek mellett Sahlin kiköti, hogy a részleges kiértékelő minden esetben megáll. Vagyis a feldolgozandó programban található esetleges végtelen ciklus esetén is. A legtöbb megvalósítás azonban feltételezi, hogy az eredeti program nem tartalmaz ilyen, és ehhez hasonló programozói hibákat. Tehát a részleges kiértékelők esetén ez egy igen erős megkötés, mely általában arra szorítkozik, hogy a kiértékelő algoritmusa álljon meg minden paraméterül kapott helyesen futó program esetén.

Alap fogalmak

- *Parciális dedukció* - valójában nem más, mint tiszta (pure) Prologon történő részleges kiértékelés. Tiszta Prolog esetén a kiértékelés nem jelent mást, mint azt, hogy a parciális evaluátornak adott programból, és paraméterekből következnek az egyéb behelyettesítések, mivel a tiszta Prolog program felfogható, mint logikai implikációk és állítások halmaza. Mellékhatásként pedig egy az előző működésével megegyező programot kapunk. Ebben az esetben, mint majd látni fogjuk a részleges kiértékelés legfőbb lépésének helyessége is magyarázható a matematikai logikai megközelítés segítségével.

- *self-applicable* (ön alkalmazható; mostantól s.-a.) - a részleges kiértékelő azon tulajdonságát írja le, hogy alkalmazható -e önmagára. Vagyis képes -e a parciális evaluátor arra, hogy az egyik bemenő paramétere önmaga legyen, míg a megadott lekérdezés nem más, mint a parciális evaluátor belépési pontja a paraméterül kapott, eredetileg részlegesen kiértékelni kívánt programmal és az ahhoz tartozó lekérdezéssel. Ennek azért van jelentősége, mert sokszor - főleg automatizált, összetettebb kiértékelők esetén - egy program részleges kiértékelése igen hosszú ideig tarthat, míg az adott programra optimalizálva a részleges kiértékelés felgyorsulhat. Ahogy majd a későbbiekben is látni fogjuk a s.-a. tulajdonság már nem cél, mivel már léteznek olyan rendszerek (pl.: LOGEN) amelyek kifejezetten r. kiértékelők r. kiértékelésére lettek kifejlesztve és optimalizálva.

- *online-offline részleges kiértékelés* - Eddig az online r. kiértékelésről volt szó, vagyis olyan kiértékelőkről, melyek a kapott kódból és lekérdezésből azonnal adják a specializált kódot. Az offline r. kiértékelés két lépcsőben hajtja végre a kiértékelést: először elvégez egy úgynevezett "kötés-idejű" analízist (Binding-Time Analysis; BTA), ami nem tesz mást, mint megállapítja, melyek a r. kiértékelni kívánt program dinamikus és statikus részei. Vagyis melyek azok a kódrészletek, melyek nem változnak meg futásidőben, és melyek azok, melyek esetleg függnek egy bemenő paramétertől, vagy működésük esetfüggő. Ilyen a következő c kódrészlet:

```
/*  LEGEND:  STATIC  DYNAMIC
 */
miniprintf(char fmt[], int val[])
{
  int i = 0;
  while( *fmt != '\0' ) {
    if( *fmt != '%' )
       putchar(*fmt);
    else
      switch(*++fmt) {
        case 'd' : putint(val[i++]); break;
        case '%' : putchar('%');     break;
        default  : prterror(*fmt);   break;
      }
    fmt++;
  }
}
```

Ez után a második lépés nem más, mint maga a kiértékelés, felhasználva az eddig kapott adatokat, így gyorsítva az eljárást és hatékonyabbá téve a keletkezett kódot. A Mixtus és az Inliner is használ ilyen, vagy ehhez hasonló

eljárást: mint látni fogjuk a vágók nagyban akadályozzák a Prolog parciális evaluátor végrehajtási mechanizmusát. Ezért a kiértékelő megpróbálja azokat egyszerűen kihagyni (amennyiben az biztosan elhagyható, tehát determinisztikus futás esetén, vagy több vágó esetén, esetleg ';' környékén stb.), vagy például végrehajtani azt fordítási időben. Az utóbbit csak akkor teheti meg, ha biztos, hogy a végrehajtás elér a vágóig, vagyis nem hiúsul meg előtte semmi, és sikeres a klóz fejével történő egyesítés. Az utóbbi biztosan sikerül, ha a fejben található minden paraméter eltérő változó, hiszen akkor akármilyen a hívásban szereplő paraméterek tartalma, mindenképpen illeszkedni fog a Prolog változóra, és abból sem adódhat meghiúsulás, hogy két paraméter azonos változóként eltérő értékeket kapna. Azonban ezt az igen erős feltételt redukálhatjuk, ha tudjuk, hogy egy predikátumnak melyek a bemenő paraméterei, és csak ezektől követeljük meg a fenti feltételt. Ugyan a Mixtus bír ezzel a tulajdonsággal, viszont a paraméterek ilyetén besorolását fordítási időben hajtja végre, tehát nem különül el a két fázis, vagyis nem mondhatjuk, hogy offline kiértékelő volna.

- *Futás idejű optimalizáció* - annyit jelent, hogy a programot fordítási időben bizonyos futási időben szétváló esetekre külön-külön optimalizálja, majd a futási időben a megfelelő esethez tartozó kódrészletet használja fel.

A Prolog részleges kiértékelők általános működése

- *Kibontás (unfolding)* - Annyit tesz, mint egy predikátumot helyettesíteni a törzsével, a fejegyesítésből adódó behelyettesítések elvégzése után:

Ha A *definíciója (klózai):*
A_1 :- B_1.
A_i :- B_i.

A *hívása pedig:*
F :- Before, A, After.
||
|| *unfolding*
\/
F :- Before, \/$_i$(A=A_i,B_i), After.

vagy

F :- A, After.
||
|| *unfolding*
\/
{(F :- B_i, After.)ß} *ahol ß nem jelent mást, mint az A hívással történő fejegyesítésből adódó egyesítések elvégzését a törzsben.*

Ez az eljárás, tehát a következő előnyökkel jár: csökkenti a predikátumhívások számát, valamint segíti az A predikátumban esetlegesen történő egyesítések hamarabbi továbbterjedését (left-, right-propagation), amelyek további egyesítésekhez és egyszerűsítésekhez vezethetnek.
A fenti egyszerű átalakítás kivétel nélkül minden Prolog részleges kiértékelőben előfordul, tehát elmondható, hogy a parciális evaluáció legfontosabb eleme.

- *Definíció (definition)* - Az eljárás tesz mást, mint felvesz egy új predikátumot, melynek törzse a kapott predikátum, feje egyedi, argumentumai pedig az eredetiben található változók:

```
Ha az eredeti klózunk:
p(2,g(A,D),H,H).

Akkor a következő predikátumot veszi fel az
eddigiek mellé:
p2(A,D,H) :- p(2,g(A,D),H,H).
```

  Az eljárás alkalmazásának valódi értelmét a becsomagolás eljárással közös példán mutatom be.

- *Becsomagolás (folding)* - Ha adott egy cél és találunk egy olyan predikátumot, mely egyetlen klózból áll, valamint az adott cél ezen klóz törzsének példánya, akkor az eredeti célt helyettesíthetjük a klóz fejével. Egy cél egy másik cél példánya, ha a funktoruk, aritásuk megegyezik, és a másik cél behelyettesített változói megegyeznek az eredetiével.

```
Ha adott a következő predikátum:
A :- B. (B akármilyen összetett kifejezés lehet)

a részleges kiértékelés során a következő klóz
feldolgozására kerül sor:
F :- Before, A, After.

akkor a feldolgozni kívánt klóz a következőképp
alakítható:
F :- Before, A, After.
```

  Mint a következő példa illusztrálja a becsomagol eljárás erősen épít a definícióra, és nagy szerepük van új, rekurzív eljárások kialakításában.

- *Egyszerű példa parciális evaluációra:*

```
Az átalakítandó eljárás:
p([],F,A,X) :- X is F-A.
p([H|R],F,A,X) :- F1 is H*F, p(R,F1,A,X).

A hívás pedig:
p(U,V,5,W).

1. Definíció:
p(U,V,W) :- p(U,V,5,W).

2. Kibontás az előző klózban:
p2([],V,W) :- W is V-5.
ps([H|R],V,W) :- F1 is H*V, p(R,F1,5,W).

3. Becsomagolás (az utolsó klózban):
p2([],V,W) :- W is V-5.
ps([H|R],V,W) :- F1 is H*V, p2(R,F1,W).
```

Tehát jól látszik, hogy az eredeti rekurzív predikátumból egy új rekurzív predikátumot kaptunk, melyben az a paraméter, melyre az eredeti predikátumot optimalizáltuk, már nem megy végig minden híváson a paraméterek között, hanem egyszerűen a behelyettesítés helyére került.

Fellelhető nagyobb projektek Prolog parciális evaluátor témakörben

Mixtus - Dan Sahlin (1991) - Egy automata részleges kiértékelő teljes Prologhoz

A Mixtusról röviden

A Mixtus egy teljesen automatikus, részleges kiértékelő a teljes Prolog (full Prolog) nyelvhez. Mint már láttuk ez annyit tesz, hogy a program kap egy programot és egy lekérdezést paraméterül, futási eredményként pedig egy a paraméterként megadottal megegyező működésű programmal és néhány behelyettesített kimenő paraméterrel tér vissza.
Az hogy a program teljes Prologra íródott nem jelent mást, mint hogy a beépített eljárásokat is képes kezelni. Ez többnyire futás optimalizációt jelent, valamint azt, hogy amennyiben ezek az eljárások kellőképp behelyettesítettek, képes azokat kiértékelni fordítási időben. A beépített eljárások részletes kezelését a dolgozat nem tárgyalja, mert úgy vélem, hogy ezek nem fontos elemek a részleges kiértékelésnek mint mechanizmusnak vizsgálata szempontjából.
Ahogy már említettem, a program írója (Sahlin) kikötötte, hogy a Mixtus minden körülmények között megáll. Erre nagy hangsúlyt fektetett teljes munkája során. Így az elkészült megoldás sokszor nem eredményez olyan hatékony programkódot, mint ha nem nem teljesítené ezt a feltételt, vagy ha éppen nem volna teljesen automatikus.

Főbb segédeljárások, algoritmusok, módszerek

Elsőként röviden tárgyalom az előzőekben ismertetett általános részleges kiértékelő modelltől való eltéréseket, valamint azokat az eljárásokat, melyek a Mixtusban megtalálhatóak a gyakorlatban történő alkalmazhatóságért. A feldolgozás során ugyanis nagyon sok olyan probléma adódik, melyek megoldásához szükséges bizonyos esetek speciális kezelése. Mindemellett remélem, hogy ezen részfeladatok és problémák ismertetése hozzájárul az algoritmus későbbi könnyebb megértéséhez.

- Becsomagolás a Mixtusban

  A becsomagol eljárás a Mixtus esetében messze nem olyan általános, mint az előző definíció, mivel nem fog végrehajtódni összetett célokra. Azaz csak olyan predikátumokat használ fel a becsomagolás során, melyek egyetlen klózból állnak, és törzsük egyetlen eleme egy egyszerű cél.

- Beépített eljárások kezelése

  A beépített eljárásokra a Mixtus egy előre definiált adatbázist használ. Ez olyan tényállítások halmaza, amelyben található kifejezések kiértékelhetőek.

```
Ilyen pl. az egyszerűbbek közül:

evaluable(X=Y).
evaluable(true).
```

  Amennyiben a kifejezés nem volt kiértékelhető, akkor változatlan marad a klózban. Ha viszont kiértékelhető volt akkor hasonlóan a kibontáshoz szétválasztja két estre:

```
Amennyiben a cél a törzsnek nem az első tagja,
akkor a szükséges behelyettesítéseket
elvégezve:

F :- Before, \/(G=Gi)ßi, After.
               i


Ha az első cél volt, akkor több klóz lesz az
eredmény a futási ágaktól függően:

{(F :- Gi, After.)ßi}


Amennyiben az első cél volt a klózban és nem
volt mellékhatása, akkor a cél elhagyható a
megfelelő behelyettesítések elvégzése után:

{(F :- After.)ßi}
```

- Cél-verem (goal-stack)

  A verem arra szolgál, hogy a definíció által létrehozott predikátumokat tartsuk benne. Mint később látni fogjuk, amennyiben a feldolgozás során egy új predikátumhoz érünk, megnézzük, hogy elvégezhető -e a becsomagolás, és ha nem, akkor felvesszük a definícióját a verembe. Vagyis a vermet a definíciós eljárás által kapott predikátumokkal tölti fel a program. Ennek az az oka, hogy a becsomagolás során a Mixtus a definícióval létrehozott predikátumokat fogja felhasználni, így a verem karbantartásával a ezt az eljárást gyorsítja.

- Anti-egyesítés (Anti-unification)

  Néhány szükséges fogalom:

  D: t1 **általánosabb** mint t2, ha t1 példánya t1-nek.

  D: t2 **példánya** t1-nek, ha létezik behelyettesítés, hogy azokat t1-en végrehajtva t2-t kapjuk: t2 =< t1.

D: t1 és t2 **egyesítése** t, ha t1 >= t és t2 >= t, valamint bármely s-re ha t1 >= s és t2 >= s, akkor t >= s.

D: t1 és t2 **anti-egyesítése** t, ha t1 =< t és t2 =< t, valamint bármely s-re ha t1 =< s és t2 =< s, akkor t =< s.

- Általánosított újrakezdés (Generalized restart)

  A becsomagolás kapcsán a következő probléma állhat fenn: tételezzük fel, hogy a veremben van egy klóz amely a következőképpen néz ki:

  ```
  Anew :- A'.
  ```

  Eddig feltételeztük, hogy ha találkozunk egy ilyen klózzal:

  ```
  P :- Before, A, After.
  ```

  akkor az A cél példánya az A' -nek. És akkor a megfelelő behelyettesítésekkel a következőt írhattuk:

  ```
  P :- Before, (Anew)ß, After.
  ```

  Csakhogy egyáltalán nem biztos, hogy A' >= A, hiszen az az A funktorával megegyező predikátummal való első találkozáskor keletkezett, ami nem biztosítja ezt nekünk.

  Így Sahlin bevezetett a Mixtusban egy eljárást erre az esetre: amennyiben fennáll a fenti helyzet, a Mixtus veszi az A és az A' anti-egyesítését, ezzel elvégez egy új definíciót, majd újrakezdi erre a klózra a részleges kiértékelést. Ez az általánosított újrakezdés.

  Az általánosított újrakezdés során létrejött predikátummal már biztosan végre tudjuk hajtani a becsomagolást, hiszen a keletkező A" biztosan általánosabb lesz A célnál. Az, hogy A" általánosabb legyen A' -nél is azért szükséges, hogy az előzőleg végrehajtott definíciók jogossága továbbra se sérüljön.

- Trimming

  Eddig a definíció során, minden az eredeti predikátum fejében szereplő változó bekerült a definíció által létrehozott klóz fejének paraméterei közé. Azonban lehetséges olyan eset is, hogy a programozó felesleges változókat ad meg a paraméterek között. Így az új klóz paraméterei közé azok a változók sem fognak bekerülni, amelyek nem szerepelnek az eredeti predikátum hívási helyei környékén. Tegyük fel a következőt:

  ```
  Adott a következő klóz:
  F :- Before, A, After.
  ```

  Amennyiben az A paraméterei közül valamelyik nem szerepel sem a Before, sem az After törzsrészben, sem az F fejben, mint paraméter, akkor azt a definíciónál el lehet hagyni.

```
Vagyis, ha az eredeti predikátumunk:
A(X,Y,V,Z) :- ... .


De pl.: az X változó nem szerepel a fent
felsoroltak közül sehol, akkor lehetséges a
következő definíció elvégzése:
A2(Y,V,Z) :- A(X,Y,V,Z).
```

- A végtelen ciklus megelőzése (Loop prevention)

  Mivel Sahlin kikötötte, hogy a parciális evaluátor minden körülmények között leáll, ezért olyan algoritmusokat készített, amelyek kiszűrik az esetleges végtelen ciklusokat. A megállási probléma megoldhatatlan volta miatt, ezért olyan egyszerű és könnyen számolható eljárásokat használ, amelyek az ismétlődések bizonyos fajtáit figyelik.
  Ezek közül az egyszerűbbek a következőek:

  - Az algoritmus a kibontás során figyeli a predikátum neveket, és ha egy predikátumnév előfordulása elér egy adott számot, akkor az adott ciklust végtelen ciklusnak nyilvánítja.
  - Ha egy bizonyos számnál többször jelenik meg adott funktorú klóz adott paraméterként.
  - Teljes struktúrák figyelése az előző két módszerhez hasonlóan. Az argumentumban szereplő változók csökkenésének figyelése. A feltételezés az, hogy amíg csökken az argumentumban szereplő változók száma a hívások során, addig nem alakulhat ki végtelen ciklus. Így a program azt számolja, hogy hányadik olyan rekurzív hívás kerül sorra, ahol nem csökken az argumentumszám. Ha ez egy adott határt átlép, akkor nem folytatja tovább az algoritmust.

- A kibontásra vonatkozó megszorítások

  - Nem lehet a kibontást végrehajtani, ha a predikátum vágót tartalmaz, és a vágót nem lehet elhagyni, vagy fordítási időben végrehajtani. Ennek az az igen egyszerű oka, hogy a kibontással a vágó egy hívási szinttel feljebb kerülne, és így megváltozna a hatóköre.
  - Nem lehet a kibontást végrehajtani, ha az adott predikátum rekurzív. Ennek pl. az az oka, hogy mivel a Mixtus minden végrehajtási ágra elvégzi a részleges kiértékelést, ezért elkezdhet olyan predikátumokat is kiértékelni, amelyeknek fordítási időben nincs meg a leállási feltétele. Ezt a hiányosságot a Mixtus úgy oldja fel, hogy új rekurzív predikátumokat hoz létre a becsomagolás és a definíciós eljárások segítségével.
  - Nem hajtja végre a kibontást, ha a predikátum az általános újrakezdés során jött létre. Az általánosított újrakezdés lényege, hogy a definíció által létrehozott predikátum törzsét egy a rekurzióban felmerülő céloknál általánosabb célok közül a legspecifikusabbra cseréli. Ezek után erre a predikátumra végzi el a kibontást. Az így kapott predikátumokra azonban a Mixtus már nem végez unfoldingot, mert az további parciális evaluációhoz vezethetne, ami viszont hasonló általánosított újrakezdéshez, és így az algoritmus nem állna le.

Az algoritmus

1. *Clauses eljárás* - paraméterül kapja a programot és a lekérdezést. A lekérdezésben szereplő minden klózra: a klózra és az n-1. állapotban lévő programra meghívja a clause eljárást, majd az így n. állapotba kerülő programra és a következő klózra meghívja a clause eljárást. A program állapota annyit jelent, hogy hányszor hajtódott végre rajta a clause eljárás, mivel a clause eljárás átalakítja a paraméterül kapott programot az adott klóz függvényében.
2. *Clause eljárás* - a paraméterül kap egy programot és egy klózt. A klóz minden céljára és a klózból feldolgozás során alakuló klózok halmazára, valamint a folyamatosan alakuló programkódra meghívja a goals eljárást. Vagyis a goals eljárás egy cél függvényében alakítja a kapott klózt (esetleg több klózzá), valamint a programkódot. Így végül klózok halmazával, valamint egy új programkóddal tér vissza.
3. *Goals eljárás* - paraméterül kap egy célt, egy klóz listát, valamint egy programot. (A klóz lista eredetileg abból a klózból származik, melynek célja volt a kapott cél.) Ezek után a klóz lista minden elemére, valamint az adott célra és programra meghívja a goal eljárást, a programkódra inkrementálisan. (Vagyis az így alakuló programkódot viszi tovább az eljárásban).
4. *Goal eljárás* - paraméterül kap egy célt, egy klózt, melynek törzsében a cél található, és egy programot. A következőeket teszi:
    - Ha a cél beépített eljárás
        - Ha végrehajtható, akkor végrehajtja a beépített eljárásra vonatkozó predikátumokat
        - Ha nem végrehajtható, akkor nem csinál semmit
    - Ha nem beépített
        - Ha lehetséges elvégzi a becsomagolást. Ha ez közvetlenül nem megy, de a cél-veremben van egy az adott cél törzsében található céllal megegyező funktorú és aritású törzzsel rendelkező predikátum, akkor végrehajtja az általánosított újrakezdést, és ez után végzi el a becsomagolást.
        - Ha ezt nem tudja megtenni
            - akkor ellenőrzi, hogy van -e benne feltételezett végtelen ciklus. Ha van akkor nem csinál vele semmit.
            - Ha nincs, akkor
                - Elvégzi a definíciót erre a célra
                - A definíció által létrehozott klózokra (predikátumra) meghívja a clauses eljárást, tehát, az újonnan létrejött predikátumra, és az eddig kifejtett programra meghíja újra a teljes eljárást.
                - Ha a visszakapott program a paraméterül adott predikátumra rekurzív, akkor elvégzi a célra a becsomagolást. Ha ez nem megy, akkor általános újrakezdést csinál a program. Ha ezek után a keletkező predikátumokra mégsem menne a becsomagolás, akkor a kikötés szerint nem lehetséges kibontás, tehát változatlanul adja vissza a klózt.
                - Ha a fentiek közül egyik sem teljesül, ellenőrzi, hogy a predikátum tartalmaz -e vágót, ha igen,

akkor nem tesz semmit.
- Ha nem tartalmaz vágót, akkor elvégzi az újonnan képzett klózokkal (melyeket a clauses eljárás adott vissza) a kibontást.

A program hiányosságai

- Nincs kibontás becsomagolás, esetén, tehát a jobbra kötés sem megy végbe, amivel a részleges kiértékelő leghatékonyabb eszközeit veszti el. Ennek az az oka, hogy a Mixtus szabályként alkalmazza, hogy ha lehet, akkor végrehajtja a becsomagolást, és a kibontást csak a becsomagolás végrehajtásának hiányában alkalmazza. Ez természetesen kiküszöbölhető más stratégiák alkalmazásával.
- Lehetséges, hogy a jobbra kötés nem hatékony, a következő esetben:

```
X=t,{e[X]}  ahol {e[X]} X változót
tartalmazó kifejezések halmaza
átalakítás után a következő alakú lesz:
{e[t]}
```

az átalakítás után tehát a memóriában a t kifejezés annyiszor fog szerepelni, ahányszor az egyesítés után szerepel, míg ha a jobbra kötés nem ment volna végbe, akkor csak egyszer. Ez az eset azonban nem gyakori, vagy nem jelent nagy hatékonyságbeli veszteséget.

Inliner - Szeredi Péter (1999)

A programról általában

Ez a program felfogásában teljesen eltér az előzőtől, és az általános részleges kiértékelő modelltől. Nem automatikus, és azt sem tűzi ki célul, hogy minden körülmények között terminál jon, mint a Mixtus tette. Paraméterként sem az eddig megszokott program-lekérdezés párost várja, hanem egyetlen klózt, melyet aztán parciálisan evaluál. Tehát visszatérési értéke is egy klóz, mely természetesen az eredetivel megegyező működésű. Az Inliner a megismert három általános részleges kiértékelő eljárása közül kizárólag a kibontást alkalmazza. Nem hoz létre új rekurzív eljárásokat. A program egyik alapelve az, hogy az esetlegesen megadott paraméterekkel sem determinisztikus futású predikátumokat nem értékeli ki, hiszen azokból csak vagy kapcsolatokat tudna építeni, melyek csupán formai átalakítások, nem járulnak hozzá hatékonyabb programfutáshoz. Ebben az esetben természetesen felmerülhet, hogy így jelentős kiértékelési lehetőségektől esik el az algoritmus, hiszen nem kerül sor az ezekben a predikátumokban szereplő predikátumok részleges kiértékelésére. Ez csak abban az esetben volna igaz, ha a kiértékelő automatikus lenne, azonban most a felhasználónak lehetősége van minden egyes predikátum részleges kiértékelésének explicit megadására.

A felhasználó és a program lehetőségei

A program egyszerűbb megértése érdekében először a felhasználói lehetőségeket, valamint a program paraméterezhetőségét ismertetem. Az Inliner nagy előnye, hogy használója predikátum szintig befolyásolhatja a feldolgozás menetét. Erre több opció és az opciókhoz feltételként megadható predikátumok alkalmasak:

- minden predikátumról kijelenthetjük a következőeket:
    - *inline* - ez az alaptulajdonság. Ha egy predikátum nem rendelkezik semmilyen tulajdonsággal, akkor a program automatikusan ezt rendeli hozzá. Annyit jelent, hogy a program végrehajtja rajta a részleges kiértékelést.
    - *distribute* - amennyiben egy predikátum a kiértékelés előtt megkapja a 'distribute' tulajdonságot és futása a feldolgozás során kapott paraméterekkel nem determinisztikus, akkor az Inliner minden futási ágra elvégzi a részleges kiértékelést. Erre az opcióra azért lehet szükség, mert - mint az általános ismertetőben már említettem - a nem determinisztikus predikátumokat az Inliner az alapbeállítások mellett nem értékel ki, viszont a felhasználó dönthet úgy, hogy erre mégis szükség van. A bevezető szakaszban azt állítottam, hogy az Inliner paramétere és visszatérési értéke is egy klóz. Ez annyiban módosul, hogy a visszatérési érték lehet több klóz, amennyiben a paraméter predikátum, rendelkezik a 'distribute' tulajdonsággal.

- *keep* - ha egy predikátum rendelkezik a 'keep' tulajdonsággal, akkor az eredeti predikátum fog szerepelni a programban abban az esetben is, ha esetleg a predikátumot sikeresen kifejtette az Inliner. Természetesen ez a tulajdonság rekurzívan igaz minden olyan predikátumra is, melynek őse az amelyről eredetileg állítottuk ezt.

- *exclude* - ha egy predikátum 'exclude' tulajdonságú, akkor semmilyen körülmények között nem marad meg a definíciója a programkódban. Ez igen erős tulajdonság, hiszen lehetséges, hogy nem sikerül a predikátum feldolgozása, és így a predikátum nem szerepel semmilyen formában a programkódban, ezért ajánlott feltétellel használni.

- Minden tulajdonsághoz adhatunk meg feltételeket: predikátumokat, melyek értelemszerűen akkor igazak, ha betöltéskor kiértékelhetőek. Mivel akármennyi predikátum megadható, igen bonyolult tulajdonság feltételek fogalmazhatóak meg.

Segédeljárások

A segédeljárások olyan predikátumokat takarnak, melyek az Inliner programban alapvető fontosságúak, azok a kódban több helyen felbukkannak és nagyban hozzájárulnak az absztraktabb predikátumok működésének részletesebb megértéséhez.

- *failure_goal(L1, L2)* Igaz, ha az L1 lista üres és az L2 lista első eleme a fail.

  Az első paramétere mindig egy úgynevezett Substs lista, mely azon egyesítések listája, melyeket nem lehet végrehajtani, az egyesítés mindkét oldala olyan változó, melyek azonos klóz fejében szerepeltek. A második paramétere pedig mindig azon klózzal azonos működést produkáló célok halmaza, melynek fejében szerepeltek a Subts lista elemi. Így tehát a predikátum működése úgy fogható fel, hogy abban az esetben igaz, ha az adott klózból adódó célok meghiusúlnak anélkül, hogy bármiféle mellékhatásuk lenne, hiszen a célok közt az első a 'fail', és nincs be nem helyettesített egyesítés.

- *cut_goal(L, Goals0, Goals)* Igaz, ha az L lista üres, a Goals0 első eleme a '!'. A Goals0 '!' utáni része a Goals

  Hasonlóan ez előző predikátumhoz ennek jelentése is főként a hívási környezetéből adódik. Első paramétere ennek is a Substs lista, míg a második eleme egy céllista, pont úgy, mint a fenti predikátumnak. Ez a predikátum tehát annyit tesz, hogy ha a célok listájában az első egy vágó, és nincs be nem helyettesített egyesítés. Tehát a vágó elhagyható.

- *expansion(Goal, Goals)* Ahol minden a programban megtalálható expansion állítás assert-el lett felvéve és a Goals a Goal célhoz, mint fejhez tartozó, a

törzsben megtalálható - előzőleg konjunktív - célok listája.

A program azokat a predikátumokat, melyeket részlegesen ki kell értékelni az expansion/2 predikátumba csomagolja, hogy aztán feldolgozás során hozzáférhessen annak törzséhez.

- *findall/3*

- *make_substs(CVs, BVs, Substs)* A Cvs és a Bvs listában található kifejezéseket rendre Cv=Bv alakra hozva visszaadja a Substs listában. Ahol Cv a Cvs, míg Bv a Bvs lista elemei.

  A make_substs/3 predikátumot mindig egy findall/3 hívás által egy változólistára adott megoldások és a változólista megfelelő elemeinek egyesítéssé alakítására alkalmazza a program.

- *filter_substs(RawSubst, AllCVs, Substs)*: Substs azon egyesítések listája, melyek RawSubsts egyesítések listájából maradtak, miután abból eltávolítottuk, és végrehajtottuk azokat, melyek legalább egyik oldalán található elem nem egy AllCvs változólistában található változó volt.

  Az AllCvs listában található változók úgynevezett tiltott változók. Ezek mindegyike megtalálható azon adott predikátum fejében melynek a részleges kiértékelése folyik. Ezt a predikátumot a program tehát arra használja, hogy kiszűrje azon egyesítéseket, melyekben mindkét oldalon olyan változó található, mely megtalálható a kiértékelni kívánt predikátum fejében.

- *expand_meta_preds(G, EG)* Amennyiben G cél metapredikátumot tartalmaz, részekre bontja azt és végrehajtja a kibontását (ha az lehetséges), majd visszaadja az EG változóban.

  Ilyen fontosabb metapredikátumok lehetnek:
  - ',' - és
  - ';' - vagy
  - '->' - feltételes szerkezet
  - if/3 - feltételes szerkezet
  - '\+' - tagadás

- *add_term_vars(Term, Vars1, Vars)* - A Term egy kifejezésben található változókat a Vars1 változóban találhatókkal együtt a Vars változóban adja vissza rendezetten.

  Mivel a program nagyon sokszor dolgozik változókkal, és gyűjti a fent ismertetett úgynevezett tiltott változókat, ezért igen gyakran használt predikátum.

- *safe_bip(A)* - igaz, ha a predikátum végrehajtható

  Ennek a predikátumnak az a feladata, hogy eldöntse, mely egyszerű szerkezetek azok, melyek a feldolgozás ideje alatt már kiértékelhetőek, végrehajthatóak. Ez az egyik legfontosabb alappredikátum, hiszen a

részleges kiértékelők egyik legfőbb eszköze, hogy minél több változó behelyettesítést hajt végre, még a részleges kiértékelés alatt ezzel egyszerűsítve a programot.

- *save_clause_for_expansion(Head, Goals)* - Ha a Head fejű predikátum inline tulajdonságú, akkor hozzáad egy expansion(Head, Goals) klózt a programhoz.

  Ez a predikátum felelős azért, hogy a feldolgozni kívánt predikátumok klózainak megfelelő expansion/2 predikátumok létrejöjjenek.

Főbb eljárások, a program működése

A program három főbb rekurzív ciklusra bontható. Ezek egyben a program három absztrakciós szintjét is jelentik. Egy ciklusban a predikátumok egymást felülről lefele irányuló sorrendben hívják, a legutolsó pedig a legelsőt. A predikátumok mellett zárójelben jelöltem, hogy esetlegesen mely más predikátumokat hívnak az azonos, vagy eltérő szintű ciklusból. Ezen ciklusok segítségével fogom bemutatni az Inliner részletesebb, működését.

- Belső ciklus (egyetlen predikátum {vagyis a hozzá tartozó céllista} feldolgozása)
    - inline_expandable/4
    - expand_single_goal/3 (expands_to/2;)
    - simplify_matches/3
    - expand_sgoals/5
    - expand_to_sgoals/4 (inline_expandable/4;)

A ciklus részletesebb tárgyalását az inline_expandable/4 predikátum mint belepési pont felől közelítem. (Mint ahogy feljebb is látszik nem feltétlenül szükséges, hogy ebbe a ciklusba itt lépjünk be.) Ez a predikátum - a végrehajtható esetektől eltekintve pl.:safe_bip/1 - a következő műveleteket hajtja végre: ha már létezik az adott célra expansion/2 predikátum, akkor kigyűjti a cél változóit, majd megkeresi az összes, a célhoz tartozó predikátum törzséből létrejött célok listáját, melyekhez eredetileg tartozó klóz fejre expansion/2 predikátum alapján illeszkedik a kifejtendő cél. Az így kapott céllistákból tehát több is lehet, ha az adott cél a kapott, vagy esetleg nem létező paramétereivel nem determinisztikus. A megoldások keresése során létrejött behelyettesítéseket A=B alakúra alakítja és elmenti a Susbts listában. Ezek közül végrehajtja azokat, amelyeket lehetséges. Ezek után két lehetőség maradt:

1. ha már nem maradt végre nem hajtott behelyettesítés, akkor folytatja a létrejött célok kifejtését: a kapott céllista elemeire ismét meghívja ezt a predikátumot (inline_expanable/4), és végrehajtja a lehetséges egyesítéseket, valamint az ezzel kapott célokat az eredetiekhez fűzi, addig, amíg nincs behelyettesítetlen egyesítés.

2. amennyiben maradt még behelyettesítetlen egyesítés, vagy meghiúsult a predikátum újrahívása abban az esetben: ha egy céllista további mellékhatás nélkül hiúsulna meg, azt elhagyjuk, valamint, ha nincs behelyettesítés és a vágó előtt nem áll más cél, akkor a vágót kihagyjuk a céllistából, és anélkül folytatjuk az adott céllistában található célok feldolgozását.

Ha az utóbbi két feltétel nem áll fenn (nincs vágó, és meghiúsulás), akkor az eljárás 'eltárolja' a cél - "be nem helyettesített egyesítés párokat", és az így létrejött megoldás halmazok közül csak azokkal foglalkozik, amelyek egy eleműek, vagy üresek, azaz az eredeti cél meghiúsult.

A ciklus leállási feltételei, a predikátumok meghiúsulási okai

A ciklus több okból állhat le: a kiértékelt céllista valamelyik tagja, nem determinisztikus, vagy a feldolgozás során maradt be nem helyettesített egyesítés. Az inline_expand/4 predikátum meghiúsul, ha eredményként nem egyetlen céllistát kap vissza (az lehet üres is, ez azt jelenti, hogy a célok egyike meghiúsult, így a célok konjunkciója is). Az inline_expand/4 predikátummal szemben az expand_to_sgoals/4 predikátum csak abban az esetben hiúsul meg, ha az első cél kifejtése során előálló céllisták listája nem egy számosságú.

- Középső ciklus (Amennyiben elakad a kiterjesztés (a belő ciklus szintjén), akkor feldolgozza a metapredikátumokat és úgy próbálkozik újra)
    - inline_expand/4
    - expand_goals/4 (expand_to_sgoals/4)
    - expand_meta_preds/2
    - expand_goal1/2 (inline_expandable/4; inline_expand/4)

  A ciklus elemei közül az expand_goals/4 predikátumot mutatom be:a predikátum a kapott célok listáját megpróbálja az expand_to_sgoals/4 predikátummal kifejteni:

  1. Ha ez sikerül, akkor végrehajtja a lehetséges egyesítéseket, amelyek nem végrehajthatóak, azok változóit a tiltottak közé teszi, és az egyesítések nélkül folytatja a kiértékelést.

  2. Ha nem sikerül, akkor a céllista első elemét a metapredikátumok mentén szétszedi és ezek után hívja meg rá az inline_expandab/4-et, végrehajtja a lehetséges egyesítéseket, amelyek nem végrehajthatóak, azok változóit a tiltottak közé teszi, és az egyesítések nélkül meghívja a célokra önmagát. A céllista maradék elemére is meghívja önmagát.

- Külső ciklus (az expand_clause/3 a legabsztraktabb eljárás, ez már képes egy teljes klóz feldolgozására. Ezen felül alkalmas a 'distribute' tulajdonsággal rendelkező predikátumok kezelésére.)
    - expand_clause/3 (expand_goals/4)
    - expand_clause/5
    - distribute_clause/5 (expand_single_goal/3; findall/3 pred.-en belül az alatta levő)
    - distribute_elem/3 (inline_expand/4; expand_clause/4)

  Ennek a ciklusnak egyetlen tagját hívja predikátum, - amely mellesleg a program belépési pontja - ez az expand_clause/3: A megfelelő előkészületek után (pl.: save_clause_for_expansion/2 hívása) a kapott klóz törzsében található célok listájával meghívja az expand_goals/4-t, majd a célok között distribute tulajdonságú célt keres:

1. amennyiben talál ilyet, meghívja rá az expand_single_goal/3-t, majd minden futási ág - minden az expand_single_goal/4 által visszaadott struktúra - esetében a következőt teszi:az eredeti céllistában, amelyben a distribute tulajdonságú cél megtalálható, helyettesítjük azt rendre az összes céllistájával külön-külön és az így kapott céllistákra folytatjuk a kiértékelést (expan_clause/5).Ezt minden distribute céllal megtesszük. Így eredményül a predikátum egy klóz listát (predikátumot) fog adni, melyek a distribute elemek futási ágainak felelnek meg.

2. amennyiben nem talál ilyet visszatér az expand_goals/4 által adott eredménnyel.