C programok dinamikus szeletelése

Diplomamunka

Készítette:
Faragó Csaba
V. éves programtervező matematikus szakos hallgató

Témavezető:
dr. Gyimóthy Tibor
tudományos főmunkatárs

Szegedi Tudományegyetem,
Mesterséges Intelligencia Kutatócsoport

Szeged, 2001. május 16.

Tartalomjegyzék

1. Bevezetés

A programszeletelési módszerek széles körben alkalmazhatóak hibakeresésre (*debugging*), tesztelésre és karbantartásra. Egy szelet tartalmazza az összes olyan utasítást és predikátumot, amely hatással lehet változók egy `v` halmazára a program adott `p` pontján. Egy szelet lehet egy futtatható program vagy a programkód egy részhalmaza. Az első esetben a redukált program viselkedése egy `v` változó szempontjából a program `p` pontján ugyanaz, mint az eredeti programé. A második esetben a szelet azon utasítások halmaza, melyek befolyásolhatják a `v` változó `p` pontbeli értékét. A szeletelő algoritmusok osztályozhatók aszerint, hogy csak statikus információkat használnak (*statikus szeletelés*), vagy egy adott inputra számolják ki, hogy mely utasítások befolyásolják egy változó értékét (*dinamikus szeletelés*).

Sok alkalmazásban (pl. hibakeresés) a dinamikus szeletelés sokkal előnyösebb, hiszen pontosabb eredmének előállítására képes (vagyis a dinamikus szelet kisebb, mint a statikus).

Többféle szeletelési módszer létezik ([5], [1]). Agrawal és Horgan egy pontos szeletelési módszert írt le ([1]), amiben a dinamikus függőségeket gráffal reprezentálták. Ez a Dinamikus Függőségi Gráf (DDG – Dynamic Dependence Graph) egy utasítás minden előfordulásához tartalmaz egy külön csúcsot. A `DDG` alapján egy `v` változóhoz számolt dinamikus szelet azokat az utasításokat tartalmazza, amelyeknek hatása van `v` értékére. Ennek a megközelítésnek a legnagyobb hátránya az, hogy a `DDG` mérete a végrehajtott utasítások számával arányos. Bár Agrawal és Horgan ajánlott egy módszert a `DDG` csökkentésére, még ez a redukált `DDG` is nagyon nagy lehet. Emiatt ez az eljárás nem alkalmazható valódi méretű programokra, ahol adott esetben több millió lépés is végrehajtódhat.

Gyimóthy Tibor és társai kidolgoztak egy módszert a dinamikus szeletek számolására ([4]). Az általuk adott eljárás a program elejétől kezdve folyamatosan számolja a szeleteket, így egy adott lépésben az összes változó aktuális szeletét megadja. Az eljárást azonban csak egy egyszerű programnyelvre dolgozták ki. Ezt a módszert továbbfejlesztettük úgy, hogy alkalmas legyen valódi `C` programok szeletelésére is. A továbbfejlesztett algoritmus részletes leírása megtalálható a [7], [8] és a [9] leírásokban.

A diplomamunkám felépítése a következő: először ismertetem a dinamikus szeletelés alapfogalmait, bemutatom a Gyimóthy Tiborék új algoritmusát és röviden ismertetem az algoritmus kiterjesztését a `C` programozási nyelvre. A nemstruktúrált ugró utasítások kezelésének egy külön fejezetet szenteltem. Ezután részletesen bemutatom a kifejezések statikus `DU` építését és instrumentálását, ami a diplomamunkám fő témája. A végén egy összefoglaló olvasható az eddigi tapasztalatainkról. A végén egy bő irodalomjegyzék és a kifejezések nyelvtana található.

2. Dinamikus szeletelés

2.1. Bevezetés, alapfogalmak

Néhány esetben a statikus szeletek felesleges utasításokat is tartalmaznak. Ilyen eset például a hibakeresés, amikor dinamikus információk is a rendelkezésünkre állnak. A dinamikus szeletelés célja az volt, hogy sokkal pontosabban lehessen meghatározni azokat az utasításokat, melyek a hibát tartalmazhatják, feltéve, hogy a hiba egy adott bemenetre fordult elő.

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
 1.    scanf("%d", &n);
 2.    scanf("%d", &a);
 3.    i = 1;
 4.    s = 1;
 5.    if (a > 0)
 6.      s = 0;
 7.    while (i <= n) {
 8.      if (a > 0)
 9.        s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

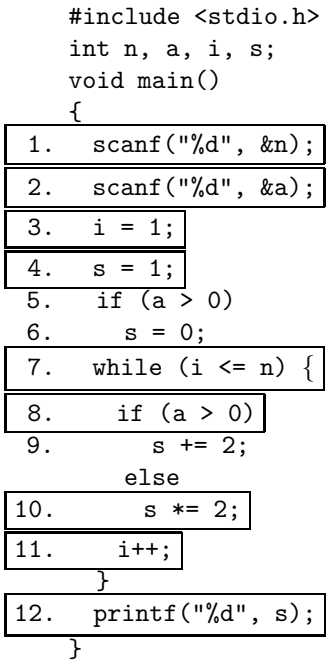
1. ábra. Példaprogram

Mindenekelőtt néhány alapfogalmat definiálok.

A végrehajtott utasítások sorozatát *végrehajtási út*-nak (execution history) nevezzük és `EH`-val jelöljük. Legyen a példánk bemenete `a=0, n=2`. Az `EH` ekkor a következő: $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12\rangle$. Láthatjuk, hogy az `EH` olyan sorrendben tartalmazza az utasításokat, amilyen sorrendben azok végrehajtódtak. Így `EH(`j`)` megadja a j-edik lépésben végrehajtott utasítás sorszámát.

Egy utasítás két előfordulásának megkülönböztetéséhez az *akció* fogalmát használjuk. Ez egy (i, j) pár, amit i^j alakban írunk, ahol i a j-edik lépésben végrehajtott utasítás sorszáma. A példánkban a 12^{15} akció tartozik a kiíratóutasításhoz az előző input esetén.

A *dinamikus szeletelési feltételt* definiálhatjuk egy $(\mathbf{x}, i^j, V)$ hármasként, ahol $\mathbf{x}$ az input, i^j egy akció az `EH`-ban, v pedig változók egy halmaza. Egy szeletelési feltételre a dinamikus szelet definiálható azon utasítások halmazaként, melyek befolyásolhatják a v halmazbeli változók értékét.



2. ábra. A keretezett utasítások alkotják a dinamikus szeletet

2.2. Dinamikus szeletek előrehaladó számítása

A Gyimóthy Tibor és munkatársai által kidolgozott algoritmus előrehaladó, ami azt jelenti, hogy a szükséges információkat (azaz egy adott utasításhoz tartozó dinamikus szeletet) az utasítás lefutásával egyidőben számolja ki. Ennek következtében az eljárás globális, azaz miután az utolsó utasítás is végrehajtódott, az összes végrehajtott utasításhoz tartozó szeletet megkapjuk. A globális szeletelés nem szükséges a hibakereséshez, de hasznos lehet a teszteléshez.

Ez az algoritmus egy olyan programábrázolást alkalmaz, ami csak a változók definícióját (definition, `d`) és használatát (use, `U`), illetve a közvetlen vezérlési függőségeket tudja tárolni. Erre az ábrázolásra mint `DU` *programábrázás* fogok hivatkozni. Az eredeti program egy utasításához a következő `DU` kifejezés tartozik:

$$i.\ d : U,$$

ahol `i` az utasítás sorszáma, d pedig az a változó, amelyik új értéket kap, ha az utasítás értékadó utasítás. Kiíratő utasítás vagy predikátum esetén d egy újonnan létrehozott ,,kimeneti változó"-nevet vagy ,,predikátum-változó"-nevet jelöl (lásd az alábbi példát). Az U halmaz változók egy halmaza úgy, hogy $U = \{u_1, u_2, ..., u_n\}$ esetén minden $u_k \in U$ egy, az i utasításban használt változó vagy egy predikátum-változó, amitől az i utasítás (közvetlenül) kontrol-függ. (Ha az *entry* kezdőutasítást definiáljuk, minden U halmazban pontosan egy

predikátum-változó lesz.)

A példánk DU ábrázolása a következő:

$i.$	$d:$	U
1.	$n:$	$\emptyset$
2.	$a:$	$\emptyset$
3.	$i:$	$\emptyset$
4.	$s:$	$\emptyset$
5.	$p5:$	$\{a\}$
6.	$s:$	$\{p5\}$
7.	$p7:$	$\{i, n\}$
8.	$p8:$	$\{p7, a\}$
9.	$s:$	$\{s, p8\}$
10.	$s:$	$\{s, p8\}$
11.	$i:$	$\{i, p7\}$
12.	$o12:$	$\{s\}$

3. ábra. A példaprogram DU ábrázolása

A $p5$, $p7$ és $p8$ predikátum-változókat jelölnek, az $o12$ pedig kimeneti-változót, aminek az értéke a kiíratott utasításban használt változó(k)tól függ.

Ezek után a program egy adott bemenethez tartozó dinamikus szeletét a bemenethez tartozó végrehajtási út és a program DU ábrázolása alapján a következőképpen számítják ki. A végrehajtási út minden utasítását az elsőtől kezdve sorban feldolgozzák. Egy $i.\ d : U$ utasítás feldolgozása közben kiszámítják a $DynSlice(d)$ halmazt, mely az összes olyan utasítást tartalmazza, ami hatással van a d változó i utasításbeli értékére. A DU programábrázolás használatával az adat- és kontrolfüggőségek azonos módon kezelhetők. Miután egy utasítás végrehajtódott és a megfelelő $DynSlice$ halmazt kiszámolták, meghatározzák az $LS(d)$-t, ami a d változó utolsó definíciójának a helye. Nyilvánvaló, hogy a j-edik lépésben végrehajtott $i.\ d : U$ utasítás után az $LS(d)$ értéke mindaddig i marad, amíg egy következő utasításban a d-t újra nem definiáljuk. Ha p predikátum, $LS(p)$ a predikátum legutolsó kiértékelésének a helyét jelenti. Ha például $EH(10) = 7$ (azaz az aktuális akció 7^{10}), akkor $LS(d) = 7$.

A dinamikus szeleteket a következő módon számíthatjuk ki. Tegyük fel, hogy a t bemeneten futtatunk egy programot. Miután az $i.\ d : U$ utasítás a p lépésben végrehajtódott, a $DynSlice(d)$ halmaz pontosan azokat az utasításokat fogja tartalmazni, amelyek benne vannak a $C = (t, i^p, U)$ szeletelési feltételhez tartozó szeletben. A $DynSlice$ halmazok a következő egyenlőség alapján számíthatók:

$$DynSlice(d) = \bigcup_{u_k \in U} \Big(DynSlice(u_k) \cup \{LS(u_k)\} \Big)$$

Miután a $DynSlice(d)$ halmazt kiszámoltuk, meghatározzuk $LS(d)$ értékét az értékadó utasításokra és predikátumokra:

$$LS(d) = i$$

Megjegyezzük, hogy ez a kiszámítási sorrend kötött, mert a $DynSlice(d)$ számításánál egy előző végrehajtási lépésben kiszámolt $LS(d)$ értékre van szükség, és nem az újonnan kiszámoltra (ez jól látszik például a ciklusbeli `x=x+y` utasítás esetén).

Az algoritmus formalizálva a 4. ábrán látható.

program DynamicSlice
begin
  LS és $DynSlice$ halmazok inicializálása
  D/U felépítése
  `EH` kiszámítása
  for $j = 1$ **to** `EH` *elemszáma*
    a `DU` aktuális eleme i^j. $d: U$
    $DynSlice(d) = \bigcup_{u_k \in U} \big(DynSlice(u_k) \cup \{LS(u_k)\}\big)$
    $LS(d) = i$
  endfor
  Output: LS és $DynSlice$ halmazok az összes változó utolsó definíciójához
end

4. ábra. Dinamikus szeletelés algoritmusa

Az `EH` az eredeti kód instrumentálásával, majd az instrumentált program futtatásával kapható meg. Az instrumentálásról még részletesen is szó lesz, valamint leírás található a már említett cikkekben is.

A fenti módszert alkalmazva az 2. ábrán látható példára, az $\langle 1, 2, 3, 4, 5, 7, 8, 10, 11, 7, 8, 10, 11, 7, 12\rangle$ végrehajtási út esetén a következő értékeket kapjuk:

Akció	`d`	U	$DynSlice(d)$	$LS(d)$
1^1	n	$\emptyset$	$\emptyset$	1
2^2	a	$\emptyset$	$\emptyset$	2
3^3	i	$\emptyset$	$\emptyset$	3
4^4	s	$\emptyset$	$\emptyset$	4
5^5	$p5$	$\{a\}$	{2}	5
7^6	$p7$	$\{i, n\}$	{1,3}	7
8^7	$p8$	$\{p7, a\}$	{1,2,3,7}	8
10^8	s	$\{s, p8\}$	{1,2,3,4,7,8}	10
11^9	i	$\{i, p7\}$	{1,3,7}	11
7^{10}	$p7$	$\{i, n\}$	{1,3,7,11}	7
8^{11}	$p8$	$\{p7, a\}$	{1,2,3,7,11}	8
10^{12}	s	$\{s, p8\}$	{1,2,3,4,7,8,10,11}	10
11^{13}	i	$\{i, p7\}$	{1,3,7,11}	11
7^{14}	$p7$	$\{i, n\}$	{1,3,7,11}	7
12^{15}	$o12$	$\{s\}$	{1,2,3,4,7,8,10,11}	12

A végső szelet a $DynSlice(o12)$ és az $\{LS(o12)\}$ uniója.

3. Valódi `C` programok dinamikus szeletelése

Az előző fejezetben bemutattam a témavezetőm és társai által kidolgozott szeletelési módszert. Az eljárást egy egyszerű programnyelvre dolgozták ki, így `C` nyelvű programokra csak jelentős megszorításokkal alkalmazható. Ebben a fejezetben röviden vázolom azokat a problémákat és megoldásukat, amelyek a valódi `C` programok szeletelésénél merültek fel.

Az eredeti algoritmusban megoldatlan volt a mutatók, függvényhívások és ugró utasítások kezelése. Az alapalgoritmus feltételezte, hogy egy adott utasításon belül legfeljebb 1 darab értékadás lehetséges, és mivel a `C` programozási nyelvben ez nem feltétlenül teljesül, ezért ezt a problémát is meg kellett oldanunk. A fenti problémák együttes megoldásához az alapalgoritmust is módosítani kellett.

A fejezetben először vázolom a fent említett problémák megoldását, majd az instrumentálásról írok pár gondolatot.

3.1. Mutatók

A mutatók helyes kezelése miatt egy kicsit változtatnunk kellett a szeletelési feltétel jelentésén. Ez ezt jelenti, hogy ha például egy `*p` pointer dereferencia dinamikus szeletére vagyunk kíváncsiak, akkor egy *memóriacím* dinamikus függőségeit keressük (és nem csupán egy változóét, mint az eredeti definícióban).

A tömbök és a struktúrák mezőinek kezelése visszavezethető memóriacímek szeletelésére (hasonlóan a mutatókhoz).

A `C` programok szeletelésére alkalmas eljárásunk a következő fő lépésekből áll:

- A program statikus függőségei alapján elkészítjük annak `DU` ábrázolását, és instrumentáljuk a programot, hogy a szükséges futás közbeni információkat kinyerhessük belőle.
- Az instrumentált programot lefordítjuk és futtatjuk, így kapunk egy olyan fájlt, ami a szeletelő algoritmushoz szükséges dinamikus információkat tartalmazza. Ennek neve: `TRACE`.
- Az algoritmust végrehajtjuk az előzőleg megkapott `DU` és `TRACE` segítségével.

3.2. Ugró utasítások

Az ugró utasításokhoz egy új változótípust vezettünk be, amit `címke` változónak neveztünk el. Ez a `goto` ugró utasításban definiálódik, és a címkét követő összes utasítás `U` halmazában benne van az aktuális függvény végéig. A `break` és `continue` utasítások visszavezethetők a `goto` utasításra

Az ugró utasítások statikus DU-jának a kezeléséről még részletesen lesz szó.

3.3. Függvények

Az argumentum és return változók a függvényhívások kezeléséhez szükségesek. A függvényhívás helyén az `f` függvény minden paraméterére *definiálunk* egy `arg(f,n)` változót, ahol `n` a paraméter sorszáma. Ezek a változók a függvény nekik megfelelő aktuális paraméterét fogják *használni*. Az `f` függvény definíciójánál ezeket az argumentum változókat fogják használni a függvény „skalár" argumentumai (ez lesz a függvény első „utasítása"). Így amikor a szeleteket számoljuk, nem kell az argumentumok `U` halmazait cserélgetni, hiszen az argumentum változókon keresztül mindig az aktuális függéseket kapjuk meg. Hasonló okok miatt vezetjük be a `return` változókat. A `ret(f)` változót az `f` függvény `return` utasításaiban *definiáljuk*, és a függvényhívás helyén a függvény értékeként *használjuk*.

3.4. Instrumentálás

A dinamikus szelet kiszámításához a program statikus DU ábrázolása mellett szükségünk van dinamikus információkra is. Ezeket az információkat az eredeti program *instrumentálásával* kaphatjuk meg. Az instrumentálást úgy végezzük, hogy fordítás után az instrumentált program futása csak annyiban térjen el az eredetitől, hogy a szükséges dinamikus információkat kiírja a `TRACE` állományba. A `TRACE` tartalmazza magát a végrehajtási utat és különféle „adminisztratív" információkat, mint például a skalár változók címei, függvény és blokk kezdet/vég, stb.

A `TRACE` a következő sorokból áll össze:

- Végrehajtás út sorok (`EH` sorok). Ezen sorok mindegyike egy-egy akciót tartalmaz. Ha a `TRACE` többi sorát töröljük, akkor megkapjuk a program végrehajtási útját.
- Deklarációs sorok. Ezen sorok mindegyike egy-egy skalár változó nevét és címét tartalmazza.
- Pointer sorok. Ezen sorok mindegyike egy-egy dereferencia változó nevét és értékét tartalmazza. Ezen sorok segítségével tudjuk a dereferencia változókat memóriacímekké alakítani.
- Függvény és blokk sorok. Ezek a sorok jelzik egy-egy függvény vagy blokk kezdetét/végét.

3.5. A módosított `DU`

Az előző fejezetben az `i` utasítás `DU` ábrázolását $i.\ d : U$-ként definiáltuk. `C` programokra a `DU` ábrázolás $d:\ U$ *elemek* egy *sorozatát* fogja tartalmazni, azaz:

$$i.\ \langle (d_1 : U_1), (d_2, U_2), \ldots \rangle$$

Ez azért szükséges, mert egy C utasításban (azaz kifejezésben) több memóriahely is új értéket kaphat. Megjegyezzük, hogy a sorozat tagjainak a sorrendje fontos,

hiszen előző DU elemek d értékei használhatók következő DU elemek U halmazaiban. Az elemek sorrendjét a kifejezések ,,végrehajtási" (kiértékelési) sorrendje határozza meg.

A definiált d és a használt $u_k \in U$ változóknak többféle jelentése lehet. Ezek a következők:

- skalár változók
- predikátum változók
- címke változók
- kimeneti változók
- dereferencia változók
- argumentum változók
- return változók

3.6. A módosított algoritmus

Az instrumentált kód lefordítása és futtatása után a rendelkezésrünkre álló információk segítségével számítjuk ki a dinamikus szeleteket. Mint azt már említettem, a szeletelés minden egyes olyan utasításban, ahol változó értéket kap, memóriacím alapján történik. Az egyes változók címei és a pointerek konkrét értékei már a rendelkezésünkre áll. Az algoritmusban egy adott utasításhoz érve a neki megfelelő DU-lista összes elemén végre kell hajtani az eredeti algorimtusban ismertetett műveletet. A függvényhívás esetén ügyelni kell arra, hogy a visszatérési függőséget a visszatérés után használjuk fel.

Az algoritmust megvalósítottuk. Én személy szerint a kifejezések instrumentálásáért voltam felelős, ezért a következőkben erről szólok részletesen.

A kód többi részének az instrumentálását és statikus DU-jának a felépítését Szabó Zsolt Mihály vállalta magára, aki magát az algoritmust is implementálta. A statikus DU-hoz szükséges struktúrákat Gergely Tamás valósította meg. Létrehozta a DefUse osztályt, amely tartalmaz egy definiált változót és a hozzá tartozó használt változók halmazát. A DefUseList DefUse típusú osztályokból álló lista. Minden egyes kifejezéshez egy egyedi DefUseList típusú osztály tartozik. A DefUseStruct nem más, mint DefUseList-eket tartalmazó vektor. Mind a definiált mind a használt változók különböző típusúak lehetnek: skaláris változók, pointer-dereferenciák, címkeváltozók, argumentumváltozók, return változók, predikátumváltozók, input változók, output változók stb. Mindegyik típushoz külön osztály tartozik.

A program egy C++ elemzőn alapul, ami kétségkívül a program legjelentősebb része. Mindezt Beszédes Árpád valósította meg Ferenc Rudolf és Magyar Ferenc segítségével.

4. A nemstruktúrált utasítások kezelése

Az előző fejezetben már néhány mondatban megemlítettem az ugró utasítások kezelésének a problémáját. Ebben a fejezetben részletesen szólok erről. A `C` programozási nyelv ugró utasításait vettem alapul, de a módszer átültethető tetszőleges más programozási nyelv ugró utasításaira. A témából egy különálló cikk is készült ([10]).

A következő alfejezetekben sorra leírom a `goto`, a `break`, a `continue` és végül a `switch` utasítások kezelését.

4.1. A `goto` utasítás

A `goto` utasításhoz tartozó DU struktúra felépítése a következő: a definiált változó - mint azt már említettem - legyen egy ún. *címke* típusú változó. Fontos, hogy minden egyes címkének egyedi neve legyen. Mivel a címkéknek eleve egyedieknek kell lenniük, ezért az ,,eredeti" nevük megfelelő, de ha a kódolás szempontjából egyszerűbb, akkor sorszámozni is lehetne őket. Az egyszerűség kedvéért a példákban az előbbi elnevezést fogom használni.

A használt változók halmaza (az `U` halmaz) nem tartalmaz ,,extra" változót, csak a megfelelő predikátum-változót, és amint azt látni fogjuk, tartalmazhat még néhány *címke* változót is.

Az imént definiált *címke* típusú változó azon utasítások `U` halmazába fog belekerülni, amelyek a címkét követik, egészen az adott függvény végéig. Fontos, hogy minden esetben a függvény végéig kerüljön bele az utasításokba, ne csak az aktuális blokkban található utasításokba.

Ha több címke is található a programban, akkor nyilván mindegyiket ugyanolyan módon kell kezelni. Ha egy `goto` utasítás a neki megfelelő címke után fordul elő, akkor a neki megfelelő használt változók halmaza tartalmazza az adott helyen definiált címke változót. Az algoritmus futása során ez nem okoz gondot, mivel a az utasítások végrehajtási sorrendjében egy, már definiált változóként szerepel, amit vagy egy másik `goto` vagy éppen ő maga definiált. Ez alól kivételt képez az az eset, amikor még nem volt az adott címkére ugrás. Ekkor mind az aktuális szelet-halmaza, mind az utolsó módosítás időpontja üres, illetve definiálatlan, tehát nincs kihatással az aktuális dinamikus szeletre.

Ha egy `goto` utasítás végrehajtódik a program futása során és legalább az egyik olyan utasítás bele kerül a végeredménybe, amely az adott függvényen belül a címke helyét követi, akkor az utoljára végrehajtott `goto` és vele együtt az összes hozzá tartozó predikátum benne lesz az eredményben. Ezenkívül természetesen más, az adott címkére ugró `goto` utasítás is bele kerülhet az eredmény szeletbe a teljes függőségével együtt. Így a sok `goto` utasítást használata jelentősen megnehezíti a program elemzését, mivel szerencsétlen esetben irreálisan nagy lehet a szelet.

Egy egyszerű példa látható a 5, az eredménye pedig a 6 ábrán. Habár látszólag a végeredmény csak az első sortól függ, mivel az `a` változó ott kapja meg a végső értékét, az eredmény mégis helyes, mivel mind a 2., mind a 3. sor megváltoztatása (így a 4. soré is) hatással lehet a végeredményre.

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`  int a,b;`	
1.	`  a=1;`	$a : \emptyset$
2.	`  b=1;`	$b : \emptyset$
3.	`  if (b==1)`	$p3 : \{b\}$
4.	`  goto l;`	$l : \{p3\}$
5.	`  a++;`	$a : \{a, l\}$
	`l:`	
6.	`  printf("%d",a);`	$o6 : \{a, l\}$
	`}`	

5. ábra. Egy egyszerű `C` program a `goto` utasítás bemutatására

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^4	$\{2, 3\}$
6^5	$\{1, 2, 3, 4\}$

6. ábra. A 5 program eredménye

Egy nagyobb példa látható a 7 ábrán, aminek az eredménye a 8 ábrán található. A példa eredményéből jól látszik a `goto` utasítás sokat hangoztatott gyenge ponja. Az eredmény helyes.

4.2. A `break` utasítás

A `break` utasítás egy olyan `goto` utasítás hatásával egyezik meg, amely a megfelelő `while`, `do...while`, `switch` vagy `for` utasítás blokkjából ugrik ki, mégpedig a blokkot követő első utasításra. A konkrét megoldás a következő. A `break` utasítás minden egyes előfordulási helyén egyedi *címke* típusú változó lesz. Szerintem a legegyszerűbb elnevezési konvenció a `break<Nr>`, ahol `<Nr>` a `break` utasítás programon belüli sorszáma, azaz `<Nr>` maximális értéke a programon belül előforduló összes `break` utasítás száma. Mindegyik, a megfelelő blokkot követő utasítás függeni fog a megfelelő `break` *címke* változótól. Vegyük észre, hogy ha a megfelelő blokkot követő utasítás elé beszúrunk egy új címkét, a `break` utasítást pedig kicseréljük egy, a most létrehozott címkére ugró `goto` utasításra, akkor az előzővel ekvivalens programot kapunk.

A `break` utasításra egy példa látható a 9 ábrán, az eredmény pedig a 10 ábrán található.

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`  int i,j,k,l;`	
1.	`  k=0;`	$k : \emptyset$
2.	`  l=0;`	$l : \emptyset$
3.	`  i=0;`	$i : \emptyset$
	`l1:`	
4.	`  j=0;`	$j : \{l1\}$
	`l2:`	
5.	`  k=k+i+j;`	$k : \{k, i, j, l1, l2\}$
6.	`  l++;`	$l : \{l, l1, l2\}$
7.	`  j++;`	$j : \{j, l1, l2\}$
8.	`  if (j<2)`	$p8 : \{j, l1, l2\}$
9.	`  goto l2;`	$l2 : \{p8, l1, l2\}$
10.	`  i++;`	$i : \{i, l1, l2\}$
11.	`  if (i<2)`	$p11 : \{i, l1, l2\}$
12.	`  goto l1;`	$l1 : \{p11, l1, l2\}$
13.	`  printf("%d",k);`	$o13 : \{k, l1, l2\}$
	`}`	

7. ábra. Egy nagyobb, `goto`-t tartalmazó `C` program

4.3. A `continue` utasítás

Amint azt a `break` utasításnál tettük, minden egyes `continue` utasításnál is egy egyedi *címke* változót kell bevezetni. A `break` utasításhoz hasnolóan célszerűen találom a következő konvenciót: a definiált változó neve legyen `continue<Nr>` alakú, ahol `<Nr>` a `continue` utasítás programon belüli sorszáma. Ez a `continue` utasítás helyén definiálódik. Használódni a megfelelő `for`, `while`, `do...while` vagy `switch` utasításblokk első utasításától fog, mégpedig az adott függvény utolsó utasításáig.

A `continue` utasításra egy példa látható a 11 ábrán, az eredmény pedig a 12 ábrán található.

4.4. A `switch` utasítás

Miután megoldottuk a `break` utasítás kezelésétt, a `switch` utasítás nem okozhat nehézséget.

A `switch` utasítás helyén egy predikátum változó definiálódik, és az összes, az adott blokkon belül levő utasítás függeni fog ettől a predikátumtól. Ha a `switch` blokkon belül legalább az egyik utasítás bele kerül az eredménybe, akkor az összes `case` címkét, és ha van, a `default` címkét is célszerű bele tenni az eredménybe. A `break` utasításokat a már említett módon kell kezelni, változtatás nélkül.

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\emptyset$
5^5	$\{1, 3, 4\}$
6^6	$\{2\}$
7^7	$\{4\}$
8^8	$\{4, 7\}$
9^9	$\{4, 7, 8\}$
10^5	$\{1, 3, 4, 5, 7, 8, 9\}$
11^6	$\{2, 4, 6, 7, 8, 9\}$
12^7	$\{4, 7, 8, 9\}$
13^8	$\{4, 7, 8, 9\}$
14^{10}	$\{3, 4, 7, 8, 9\}$
15^{11}	$\{3, 4, 7, 8, 9, 10\}$
16^{12}	$\{3, 4, 7, 8, 9, 10, 11\}$
17^4	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
18^5	$\{1, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$
19^6	$\{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$
20^7	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
21^8	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
22^9	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
23^5	$\{1, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$
24^6	$\{2, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$
25^7	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
26^8	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
27^{10}	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
28^{11}	$\{3, 4, 7, 8, 9, 10, 11, 12\}$
29^{13}	$\{1, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$

8. ábra. A 7 program eredménye

$i.$		$\langle d : U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`  int a,b,i;`	
1.	`  a=1;`	$a : \emptyset$
2.	`  b=1;`	$b : \emptyset$
3.	`  i=2;`	$b : \emptyset$
4.	`  while (i>0) {`	$p4 : \{i\}$
5.	`  b--;`	$b : \{p4, b\}$
6.	`  i--;`	$i : \{p4, i\}$
7.	`  if (b==0)`	$p7 : \{b\}$
8.	`  break;`	$break8 : \{p7\}$
9.	`  a++;`	$a : \{p4, a\}$
	`  }`	
10.	`  printf("%d",a);`	$o10 : \{a, break8\}$
	`}`	

9. ábra. Egy egyszerű `C` program a `break` utasítás bemutatására

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2, 3, 4\}$
6^6	$\{3, 4\}$
7^7	$\{2, 3, 4, 5\}$
8^8	$\{2, 3, 4, 5, 7\}$
9^{10}	$\{1, 2, 3, 4, 5, 7, 8\}$

10. ábra. A 9 program eredménye

$i.$		$\langle d: U \rangle$
	`#include <stdio.h>`	
	`void main() {`	
	`  int a,b,i;`	
1.	`  a=1;`	$a : \emptyset$
2.	`  b=1;`	$b : \emptyset$
3.	`  i=2;`	$b : \emptyset$
4.	`  while (i>0) {`	$p4 : \{i, continue8\}$
5.	`  b--;`	$b : \{p4, b, continue8\}$
6.	`  i--;`	$i : \{p4, i, continue8\}$
7.	`  if (b==0)`	$p7 : \{b, continue8\}$
8.	`  continue;`	$continue8 : \{p7, continue8\}$
9.	`  a++;`	$a : \{p4, a, continue8\}$
	`  }`	
10.	`  printf("%d",a);`	$o10 : \{a, continue8\}$
	`}`	

11. ábra. Egy egyszerű `C` program a `continue` utasítás bemutatására

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\emptyset$
4^4	$\{3\}$
5^5	$\{2, 3, 4\}$
6^6	$\{3, 4\}$
7^7	$\{2, 3, 4, 5\}$
8^8	$\{2, 3, 4, 5, 7\}$
9^4	$\{2, 3, 4, 5, 6, 7, 8\}$
10^5	$\{2, 3, 4, 5, 6, 7, 8\}$
11^6	$\{2, 3, 4, 5, 6, 7, 8\}$
12^7	$\{2, 3, 4, 5, 6, 7, 8\}$
13^9	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
14^4	$\{2, 3, 4, 5, 6, 7, 8\}$
15^{10}	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

12. ábra. A 11 program eredménye

$i.$		$\langle d : U \rangle$	
	`#include <stdio.h>`		
	`void main() {`		
	`  int a,b;`		
1.	`  b=0;`	$b : \emptyset$	
2.	`  a=2;`	$a : \emptyset$	
3.	`  switch (a) {`	$p3 : \{a\}$	
	`  case 1:`		
4.	`  b=5;`	$b : \{p3\}$	
5.	`  break;`	$break5 : \{p3\}$	`case 2:`
6.	`  b=3;`	$b : \{p3\}$	
	`  case 3:`		
7.	`  b++;`	$b : \{p3, b\}$	
8.	`  break;`	$break7 : \{p3\}$	`default:`
9.	`  b=6;`	$b : \{p3\}$	
	`  }`		
10.	`  printf("%d",b);`	$o10 : \{b, break5, break7\}$	
	`}`		

13. ábra. Egy egyszerű `C` program a `switch` utasítás bemutatására

Action (i^j)	DynSlice(i)
1^1	$\emptyset$
2^2	$\emptyset$
3^3	$\{2\}$
4^6	$\{2, 3\}$
5^7	$\{2, 3, 6\}$
6^8	$\{2, 3\}$
7^{10}	$\{2, 3, 6, 7, 8\}$

14. ábra. A 13 program eredménye

A `switch` utasításra egy példa látható a 13 ábrán, az eredmény pedig a 14 ábrán található.

5. A kifejezések statikus `DU`-ja

A kifejezések statikus `DU`-ját már nem a szemantikus akciók segítségével kellett megoldanom, hanem egy elkészített struktúrából fejtettem vissza, ami jelentős könnyítést jelentett. Ezt a struktúrát *Havasi Ferenc* készítette el, és részletesen olvashatunk erről az ő vizsgadolgozatában [11] (Szeged, 1999).

5.1. A kiindulási struktúra

Röviden vázolom a struktúra felépítését. A kifejezésből egy olyan lista készül el, amelynek a csúcspontjai ún. 3 címes kódot tartalmaznak. Ez azt jelenti, hogy van egy definiált változó, 2 használt változó és 1 művelet. Egy csúcs ennél egyszerűbb lehet, mint például az `a=b` értékadó kifejezésnél, ahol a definiált változó az `a`, a használt változók halmaza pedig 1 elemű, mégpedig a `b`, művelet pedig nincs, viszont bonyolultabb nem lehet.

Példa a 3 címes kódra: ha a kifejezés `a=b+c`, akkor a definiált változó az `a`, a használt változók a `b` és a `c`, a művelet pedig a `+`.

5.2. Bonyolultabb kifejezések

Bonyolultabb kifejezések esetén ideiglenes változókat kell bevezetni, és a csúcspontokat egy listára kell felfűzni. Például az `a=b+c-d` kifejezés esetén az előre elkészített struktúra egy 2 elemből álló lista. A lista első csúcsában a definiált változó egy egyedi ideiglenes változó lesz, amit nevezzünk `temp1`-nek, a használt változók `b` és `c`, a művelet pedig a `+`. A második csúcsban a definiált változó az `a`, a használt változók pedig a `temp1` és a `d`, a művelet pedig a `-`.

Természetesen a műveletek sorrendjére is ügyelni kell. Például az `a=b+c*d` esetben a lista szintén 2 csúcspontot tartalmaz, viszont nem olyan, mint az előző esetben. Itt az első csúcsban a definiált változó a `temp1`-lesz, viszont a használt változók halmaza a `c` és a `d`, a művelet pedig a `*`, míg a lista második elemében a definiált változó ugyancsak az `a`, a használt változók a `b` és a `temp1` ebben a sorrendben, a művelet pedig a `-`.

Az eddigi példák csak egyszerű skaláris változókat tartalmaztak. Viszont másra is fel kell készülni: a konstansokra, mutatókra, struktúrákra stb. Összesen 8 alap típusú változót különböztetünk meg, amelyekre az összes többi visszavezethető. Ezek a következők:

- konstansok, pl. `1`
- skaláris változók, pl. `a`
- pointer dereferenciák, pl. `*p`
- statikus struktúrák mezői, pl. `s.a`
- dinamikus struktúrák mezői, pl. `q->a`
- statikus változók címe, pl. `&a`
- statikus struktúrák mezőinek a címe, pl. `&(s.a)`
- dinamikus struktúrák mezőinek a címe, pl. `&(q->a)`

Például az `a=*p-7*s.x` kifejezéshez tartozó lista a következő:

- Az első csomópontban a definiált változó a `temp1` egyedi ideiglenes változó, a használt változók a konstans típusú `7` és az `s.x`, ami egy statikus struktúra mezője, a művelet pedig a `*`.
- A második csomópont definiált változója a skaláris típusú `a`, a használt változók a pointer dereferencia típusú `*p` és a `temp1`, ami ebben az esetben skaláris típusú változóként viselkedik.

Látható, hogy a nyolc esetből háromban nem 1, hanem 2 változónevet kell megadni: az előtag és az utótag nevét. A konstansok konkrét értékére a statikus `DU` szempontjából nincs szükség.

5.3. Összetett struktúrák

Felmerül a kérdés, hogy ezek alapján hogyan oldjuk meg a struktúrák mélyebb egymásba ágyazását. Például hogyan kerül bele a struktúrába az `s.t.x` kifejezés, hiszen látszólag a fenti 8 típus nem támogatja ezt az estet. Ezt a problémát a temporális változók segítségével lehet megoldani. A kifejezéssel egyenértékű az `(&(s.t))->x` kifejezés. Tekintsük a `b=s.t.x` értékadást. A lista 2 elemet fog tartalmazni:

- Az első csúcsnak a definiált változója a `temp1`, a használt változók halmaza pedig 1 elemű: a `&(s.t)`-t fogja tartalmazni, ami a fenti változó-típus listában az utolsó előtti. Művelet nincs.
- A második csúcs definiált változója a `b`, a használt változók halmaza pedig szintén 1 elemű, ami `q->a` típusú. Itt a `q` változó szerepét a `temp1` játsza, az `a` szerepét pedig az `x`. Művelet itt sincs definiálva.

5.4. Tömbök

A fenti 8 eset látszólag a tömböket sem támogatja, viszont egy egyszerű ekvivalens módosítással már a tömbelem-hivatkozást tartalmazó kifejezést is kezelni tudjuk. A `t[i]` és a `*(t+i)` ugyanis ekvivalensek egymással, ahol a `t` ebben az esetben a tömb első elemének a címét jelenti.

5.5. A statikus `DU` előállítása

A vázolt struktúrából kellett viszafejtenem a kifejezés teljes statikus `DU` listáját. Adott két `DefUseList` típusú változó: `defUseList` és `tempDefUseList`. A `defUseList` lesz a végeredmény, a `tempDefUseList` pedig az ideiglenes változók `DU`-jának a listája, amely a végén törlésre kerül. A listát sorban be kell járni. Minden egyes csomóponton a definiált változó maga a csomóponton definiált változó lesz, a használt változók pedig az ottani használt változókból számítódnak ki. Amennyiben az egyik használt változó ideiglenes, akkor a `tempDefUseList` listából meg kell keresni a hozzá tartozó halmazt, és annak minden egyes elemét bele kell tenni a használt változók halmazába.

Ha a változó nem ideiglenes, akkor egyszerűen a használt változók halmazába kerül. Így nem fordulhat elő az, hogy a használt változók halmazában ideiglenes változó legyen. Amennyiben a felépített struktúra helyes, akkor ideiglenes változó a csúcspont használt változók halmazában csak úgy lehet, ha már előtte definiálva volt. Miután felépult a `DU` elem, akkor annak függvényében, hogy a definiált változó ideiglenes-e vagy sem, a `tempDefUseList` vagy a `defUseList`-be kerül bele.

A `DU` építésénél ügyelni kell arra, hogy nem mindegyik változó kerül bele statikusként a halmazba. A konstansokat figyelmen kívül kell hagyni. Azok a változókat, amelyek előtt `&` van, szintén ki kell hagyni. A pointer-dereferenciákat és a dinamikus struktúra elemére való hivatkozásokat viszont pointerként kell kezelni, tehát egy egyedi pointer-dereferencia típusú változót kell létrehozni, és azt kell beletenni a halmazba. A pointer-dereferencia majd az instrumentált kód futtatásakor lesz feloldva. Például az `a=*p` kifejezés statikus `DU`-ja a következő: a definiált változó az `a`, ami skaláris típusú, a használt változók halmaza pedig 1 elemű, és egy egyedi pointer-dereferencia típusú változót tartalmat (`PTR1`). Ezt mindkét esetben így kell létrehozni, tehát a temporális esetben is.

5.6. A függvényhívás kezelése

A függvényhívás helyén minden egyes paraméterhez külön `DU` tartozik, a definiált változó pedig egy egyedi argumentum változó lesz. Például ha a függvény neve `f`, akkor a 2. paraméterének a neve `arg_f_2`. A visszatérési érték függvényenként egyedi ún. `return` változó lesz, pl. `ret_f`. Ez benne lesz annak a változónak az `U` halmazában, amely a függvény visszatérési értékét megkapja. Ez a változó nyilván lehet ideiglenes is lehet. Vegyünk egy bonyolultabb példát: `a=b+f(c,d=e+f*g)+h`.
A statikus `DU` felépítése a következő lesz:

- Az `arg_f_1` definiált változó `U` halmaza 1 elemű lesz, mégpedig a `c` változó lesz benne. A `defUseList` listába kerül bele.
- A következő lista elején egy `temp1` változó van, `f` és `g` használt változókkal, $*$ művelettel. Ez bele kerül a `tempDefUseList`-be.
- A következő csomópontban a definiált változó a `d` lesz, a használt változók pedig az `e` és a `temp1`. A `DU` definiált változója tehát a `d` változó lesz, a használt változók halmaza pedig tartalmazni fogja az `e` változót, valamint a `temp1` változó aktuális `U` halmazát, amelyet a `tempDefUseList`-ből tudunk kiolvasni. Így az `U` halmaz elemei a `d`, az `e` és az `f` lesznek. Ez belekerül a `defUseList`-be.
- Az `arg_f_2` `U`-ja a `d` lesz. Ez fog a `defUseList`-be kerülni.
- Egy újabb csomópont definiált változója egy ideiglenes változó lesz: `temp2`, a használt változók halmaza pedig a `ret_f` és a `h` lesz. Ez a `tempDefUseList`-be kerül.

- Végül pedig a definiált változó `a` lesz, a használt változók pedig a `b` és a `temp2`, azaz kifejtve `ret_f` és `h`. Ez belekerül a `defUseList`-be. A legvégén töröljük a `tempDefUseList` tartalmát.

5.7. A `?:` operátort tartalmazó kifejezés statikus `DU`-ja

A `?:` jelentős technikai akadályt jelentett a statikus `DU` építésénél. Ennek az oka a kiindulási struktúra bonyolultsága. A `?:` „kezdetén" a lista kettéválik egy `falseNext` és egy `trueNext` ágra, ami a végén egyesül. Az egyesülés pontján mind a `falseNext` mind a `trueNext` ág eredményét bele kell tenni a `USE` halmazba. Viszont az egyes csomópontok alapértelmezett állapotban nem tartalmazzák ezt az információt. Ezért a `DU` építésének a megkezdése előtt be kell járni a teljes listát annak érdekében, hogy megtaláljam az egybefolyási csomópontokat. Ezt úgy végeztem el, hogy minden egyes csomóponton rekurzívan meghívtam a bejárást a `trueNext` ágra, majd a `falseNext`-re is. Ha egy olyan csúcshoz értem, amelyet már egyszer bejártam, az azt jelenti, hogy egy `falseNext` ág végén vagyok, tehát itt bejelölöm azt, hogy ez egy olyan csomópont, ahol a `falseNext` és a `trueNext` ág találkozik. Az osztályon belül 9 függvényre és 1 új változóra, a csomóponton belül pedig 4 új attribútumra volt szükség ahhoz, hogy a problémát megoldjam. A probléma bonyolultságának az oka az volt, hogy a `?:` egymásba ágyazható elvileg akármilyen mélységig.

6. A kifejezések instrumentálása

A programról a dinamikus információkat úgy kapjuk, hogy a vizsgálandó kódot módosítjuk, és ezt a módosított forrást lefordítva és futtatva kigyűjtjük a dinamikus információkat. A kódot természetesen csak úgy szabad módosítani, hogy az eredeti program ugyanúgy fusson, mint előtte. A kód ilyen jellegű módosítását hívjuk instrumentálásnak.

Mivel a mutatók miatt kénytelenek voltunk módosítani az algoritmust úgy, hogy nem változónevekre, hanem memóriacímekre szeletelünk, szükséges információ az egyes változók címei. A globális változók címeit a főprogram (`main` függvény) elején íratjuk ki, az egyes lokális változókat pedig a definiálásuk helyén. Ezt úgy valósítottuk meg, hogy közvetlenül a definiálásuk után a módosított kódban beillesztünk egy függvényhívást, aminek a paraméterei a változó neve és az aktuális címe.

A mutatók aktuális értékét is ki kell íratni az egyes előfordulási helyeiken. Ez egy igen összetett probléma, amiről később részletesen szó lesz.

Mindezeken kívül még több helyen kellett módosítani a kódon, ami már inkább technikai, mint elvi kérdés. Ilyen például a függvény elejének és végének a kiírása, az egyes blokkok elejének és végének a jelzése stb.

6.1. Egy konkrét példa

A `C++` elemző az elemzés során ún. szemantikus akciókat hoz létre. Ezek segítségével kellett rekonstruálnom az eredeti kifejezést, majd ennek segítségével a módosítottat. A nyelvtan releváns része megtalálható a függelékben. A kifejezést a `beginTopLevelExpr()` és az `endTopLevelExpr()` ,,határolják". Ezen kívül minden egyes logikai egység után egy `token(...)` szemantikus függvény hívódik meg, aminek a legfontosabb paramétere maga a logikai egység szöveges formátumban. Ahol csak lehetett, ennek a használatát elkerültem.

Először is ,,elrettentésül" hadd álljon itt egy példa, ami az `a=b+c` egyszerű kifejezéshez tartozó szemantikus akciókat tartalmazza:

```
beginTopLevelExpr()
  beginAssignmentExpr(1)
   beginConditionalExpr(1)
    beginLogicalOrExpr(1)
     beginLogicalAndExpr(1)
      beginInclusiveOrExpr(1)
       beginExclusiveOrExpr(1)
        beginAndExpr(1)
         beginEqualityExpr(1)
          beginRelationalExpr(1)
           beginShiftExpr(1)
            beginAdditiveExpr(1)
             beginMultiplicativeExpr(1)
              beginSegmentExpr(1)
               beginPmExpr(1)
                beginPrimaryExpr(1)
Token: a
                endPrimaryExprIdexpr(134,132,3,a)
               medialPmExpr()
               endPmExpr()
              medialSegmentExpr()
              endSegmentExpr()
             medialMultiplicativeExpr()
             endMultiplicativeExpr()
            medialAdditiveExpr()
            endAdditiveExpr()
           medialShiftExpr()
           endShiftExpr()
          medialRelationalExpr()
          endRelationalExpr()
         medialEqualityExpr()
         endEqualityExpr()
        medialAndExpr()
        endAndExpr()
       medialExcluseiveOrExpr()
```

```
       endExclusiveOrExpr()
      medialInclusiveOrExpr()
      endInclusiveOrExpr()
     medialLogicalAndExpr()
     endLogicalAndExpr()
    medialLogicalOrExprExpr()
    endLogicalOrExprExpr()
   endConditionalExpr()
  medialAssignemntExpr()
Token: =
   beginAssignmentExpr(3)
    beginConditionalExpr(3)
     beginLogicalOrExpr(3)
      beginLogicalAndExpr(1)
       beginInclusiveOrExpr(1)
        beginExclusiveOrExpr(1)
         beginAndExpr(1)
          beginEqualityExpr(1)
           beginRelationalExpr(1)
            beginShiftExpr(1)
             beginAdditiveExpr(1)
              beginMultiplicativeExpr(1)
               beginSegmentExpr(1)
                beginPmExpr(1)
                 beginPrimaryExpr(1)
Token: b
                  endPrimaryExprIdexpr(135,132,3,b)
                 medialPmExpr()
                 endPmExpr()
                medialSegmentExpr()
                endSegmentExpr()
               medialMultiplicativeExpr()
               endMultiplicativeExpr()
              medialAdditiveExpr()
Token: +
               beginAdditiveExpr(27)
                beginMultiplicativeExpr(1)
                 beginSegmentExpr(1)
                  beginPmExpr(1)
                   beginPrimaryExpr(1)
Token: c
                    endPrimaryExprIdexpr(136,132,3,c)
                  medialPmExpr()
                  endPmExpr()
                 medialSegmentExpr()
                 endSegmentExpr()
```

```
                medialMultiplicativeExpr()
                endMultiplicativeExpr()
               medialAdditiveExpr()
               endAdditiveExpr()
              endAdditiveExpr()
             medialShiftExpr()
             endShiftExpr()
            medialRelationalExpr()
            endRelationalExpr()
           medialEqualityExpr()
           endEqualityExpr()
          medialAndExpr()
          endAndExpr()
         medialExcluseiveOrExpr()
         endExclusiveOrExpr()
        medialInclusiveOrExpr()
        endInclusiveOrExpr()
       medialLogicalAndExpr()
       endLogicalAndExpr()
      medialLogicalOrExprExpr()
      endLogicalOrExprExpr()
     endConditionalExpr()
    medialAssignemntExpr()
    endAssignemntExpr()
   endAssignemntExpr()
  endTopLevelExpr()
```

Tehát a fenti függvénysorozat képezte számomra az inputot. A `token(...)` akciók segítségével követhető nyomon, hogy ,,hol tartanak" az akciók. Az elvi problémák és ezek megoldásának leírása előtt egy pár mondatban leírom, hogy pontosan hogyan kellett mindezek segítségével instrumentálni a fenti kifejezést.

Induljunk ki az elvárt eredményből. Ez ugyanaz, mint az eredeti, tehát `a=b+c`. A `beginAssignmentExpr(...)` függvénytől a `beginPrimaryExpr(...)` függvényig, mint az látható, a paraméter minden esetben 1. Ez a paraméter típust jelöl. Az 1 az `optNone`, azaz a ,,semmilyen" típusú paraméter. Idáig tehát semmit sem kell csinálni.

Most következik a `endPrimaryExprIdexpr(...)` függvény. Itt az `a` paramétert emelném ki, ami nem más, mint az `a` változó az `a=b+c` kifejezésben. Közvetlenül előtte található az `a` token, ami szintén a fenti állításunkat támasztja alá. Tehát az eredménybe beletesszük az `a` karaktert.

A következő ,,lényeges" függvény a `beginAssignmentExpr(...)`. A többit jelen esetben figyelmen kívül kell hagyni. A jelenlegi függvény paramétere 3, ami az `optAssignequal` típus, tehát az értékadás. Ez azt jelenti, hogy az adott helyen valóban egy értékadás következik. (A `beginAssignmentExpr(...)` akció meghívása nem vonja automatikusan maga után az `=` jel létét. Pl. rögtön a 2. függvény is ez, ott mégsincs egyenlőségjel.) Tehát az eredménybe most bele

kerül az = jel is. A közvetlenül mellette levő = token akció a feltételezésünket támasztja alá, de a program nem használja ki, ez csak ellenőrző információ.

Ezután az `endPrimaryExprIdexpr(135,132,3,b)` függvényig hasonló függvények hívódnak meg, mint az előző (`a` változó) esetben, tehát az addigi függvényeket figyelmen kívül kell hagyni, majd a `b` változót az eredmény sztring végére kell tenni.

A `beginAdditiveExpr(27)` függvényig szintén át kell lépni a többi függvényt. A 27-es paraméter az `optPlus`, tehát összeadás kezdődik. A + jelet az eredmény sztring végére kell tenni.

Az `endPrimaryExprIdexpr(136,132,3,c)` akcióval a `c` változó is a helyére kerül. A hátralevő függvényeket figyelmen kívül hagyva készen vagyunk. Az eredményt a `getInstrumentedExpr()` függvényen keresztül kapja meg a főprogram.

Egy pár megjegyzést szeretnék tenni a fenti példával kapcsolatban. Egy-két érdekesség megfigyelhető az akciókban. Például az `endAdditiveExpr()` függvényhívás kétszer is előfordul, ráadásul közvetlenül egymás után. Egy `beginX` és `endX` függvények által határolt blokk tetszőleges mélyégig tartalmazhat további `beginX ... endX` blokkokat. Ez különösen igaz magára a teljes kifejezésre. Egy kifejezésen belüli zárójel (`(`, `)`) újabb beágyazott ,,teljes értékű" kifejezést ,,nyit". Ez természetesen zárójelek nélkül is előfordulhat, pl. a `?:` kifejezés `?` és `:` közötti része.

Maga az elemző eredetileg egyébként nem mindig ezeket a paramétereket küldi. Az 1-esek helyén döntő többségében más van (általában az előző nem 1, azaz `optNone` paraméter értéke), ami igen zavaró. Szerencsére egy másik kód ezt már átalakítja, így nekem ezzel nem kell foglalkoznom. Viszont ez az említett másik kód az instrumentálástól teljesen független. Sajnos a `C++` elemző módosításakor ezek a paraméterek ,,visszaváltozhatnak" vagy ,,eltűnhetnek". Az elemző minden egyes módosításának súlyok következményei voltak az instrumentálásra. A fenti példában ez nem jön elő, de megfigyelhető, hogy a `beginConditionalExpr(3)` és `beginLogicalOrExpr(3)` paramétere `3` (azaz `optAssignequal`), viszont ez a fenti környezetben ,,nem logikus": vagy 1-nek kellene lennie, vagy a rákövetkező függvényeknek a paramétere is 3 kellene, hogy legyen.

6.2. Mutatók

A kifejezések instrumentálásával a mutatók aktuális értékét szeretnénk kinyerni. Ha a kódban találunk egy ún. pointer-dereferenciát, akkor a statikus `DU`-ban ezt jelezzük, és csak a program futtatásakor dől el, hogy konkrétam melyik memóriacímre mutat a mutató.

Példa: legyen `a` egy `int` típusú változó, `p` pedig egy `int`-re mutató pointer. A kifejezés legyen `a=*p`. Ebben az esetben a statikus `DU`-t a következőképpen építjük fel: a definiált változó az `a`, a használt változók halmazában pedig jelezzük, hogy egy pointer-dereferenciával állunk szemben. Magát a `p` változót is beletesszük ebbe a halmazba. Tehát a statikus `DU a:{p,PTR1}` alakú lesz. Itt

a PTR1 azt jelenti, hogy a tényleges címet a dinamikus információ birtokában fogjuk megkapni.

Az U halmaz tartalmazhat még más elemeket (pl. predikátumok, címke változók stb.) is.

A fentiek alapján tehát az instrumentált kódot úgy kell elkészíteni, hogy az értékadás helyes megtartása mellett a p változó aktuális értékét is megkapjuk. Ezt a következőképpen tehetjük meg: a=*D_P(p). A D_P egy olyan függvény, amely a megfelelő állományba kiírja a paraméterül kapott értéket (jelen esetben a p változó értéke), és a visszatérési értéke pedig maga a paraméter. Azaz a p és a D_P(p) egyenértékű kifejezésekké válnak, az értékadás tehát helyes marad, és sikerült kinyernünk a dinamikus információt, amit az algoritmus futtatása során a fenti PTR1 helyére írunk.

6.3. Tömbök

Egy tömbelemre való hivatkozást is a statikus DU-ban a fenti módon jelölünk. Ennek megfelelően dinamikusan meg kell határozni a hivatkozott tömbelem címét.

A tömbhivatkozás lehet egészen egyszerű is, pl. t[5]. Ebben az esetben talán még statikusan is megoldható lenne a dolog. De az ugyancsak nem túl bonyolult t[i]-t már statikusan kezelni eléggé reménytelen. i helyett pedig akármilyen bonyolult kifejezés is állhat. Az egyszerűség kedvéért példának a t[i]-t veszem.

Legyen a kifejezés a=t[i]. Ekkor a kifejezés helyes instrumentálása a következő: a=*D_P(&t[i]). Statikusan csak annyit tudtunk, hogy az a változó értéke az i változótól és egy pointertől függ. Ez utóbbit a már említett D_P függvénnyel oldjuk fel. A t[i] és a *D_P(&t[i]) egyenértékűek.

6.4. Struktúrák, mélyebb egymásba ágyazás

A struktúrákat hasonlóan kezeljük, mint a pointer-dereferenciákat. Tehát a statikus DU-ban jelezzük, hogy mutatóról van szó (PTR), az instrumentált kódban pedig kiíratjuk az adott mező címét, mégpedig a következőképpen: *D_P(&(a->b)). A többi esetet is hasonlóan oldjuk meg.

Itt igen komoly techniaki akadályokba ütköztem. Ennek az oka az, hogy a szemantikus akciókból nem derült ki előre, hogy az egybeágyazás milyen mély lesz. Pontosabban elég lett volna az az információ is, hogy az adott változónak lesz-e szuffixe vagy sem. Az instrExpr nevű sztring lesz maga az instrumentált kifejezés. Ha jön egy egyszerű változó, amelynek nincs semmiféle mezője (az endPrimaryExprIdExpr(a) akcióval), akkor azt a nyugodtan a kifejezés végére lehetne tenni. Viszont ha a változót mezőnév is követi (pl. az a->b-ben a b), akkor már az a elé kellene valamit beírni. Viszont az a->b változó esetében is az endPrimaryExprIdExpr(a) akcióig teljesen azonos akciók jönnek, mint a „síma” esetben, ezért a fenti akció végrehajtásakor nem tudom, hogy az instrExpr változó végére tehetem-e az a változót. A fenti szemantikus akciót követheti egy beginPrimarySuffExpr(type) szemantikus akció.

A probléma megoldását a következőkben vázolom. Egy stringekből álló listát használok. A lista elemei egy adott szinten definiált atomi egységek. Ilyen atomi egység példáaul egy egyszerű változó (pl. `a`), összetett változó (pl. `a->b`), egy tömbelemre való hivatkozás (pl. `t[i]`) műveleti jel (pl. `=`), egy zárójelben levő teljes kifejezés (pl. `a+b`) stb. Ez utóbbi a maga szintjén van ,,kifejtve". Például a `t[a+b]` egy atomi egységet képez. A szögletes zárójeleken belül egy újabb ,,teljes értékű" kifejezés van, ami az ő szintjén szintés atomi egységekre szét van bontva, mégpedig 3 atomi egységre: `a`, `+` és `b`. Az atomi egységeket a `list<string>` típusú `sList` változóban tárolom. Ha a kifejezésen belül egy újabb kifejezés található (pl. `t[a+b]` esetben az `a+b`), akkor nem használhatom ugyanazt a listát, mivel végül ennek az új kifejezésnek 1 atomi egységek kell, hogy képezzen. Így létrehoztam egy olyan vermet, amelyben `list<string>` típusú elemek vannak (`oStack`). A kifejezés végén az adott listát egy sztringgé alakítom, és a neki megfelelő szint listájának a végére helyezem, immáron mint egy atomi egységet.

Visszatérve az eredeti problémára: megfigyelhető, hogy a `beginPrimarySuffixExpr(...)` szemantikus akció meghívásakor az `sList` utolsó eleme pont az adott változó prefixe. Például ha az `sList` utolsó eleme `a`, és jön egy `->b`, akkor látszólag kivehetjük a lista végéről az utolsó elemet, elébe tehetjük a kiíratót függvény nevét (esetünkben ez a `D_P`), így nyugodtan át tudjuk alakítani a kívánt `(*D_P(&(a->b)))` alakúra, amit aztán visszahelyezhetünk a lista végére.

A megoldáshoz már közel járunk, de ez még nem teljes. Mi van akkor, ha jön egy újabb szuffix, pl `->c`? A fenti módszer követve már az eredmény `(*D_P(&((*D_P(&(a->b)))->c)))` lenne, ami viszont már nem jó, ugyanis ezen a helyen ténylegesen nem 2, hanem csak 1 darab mutató van. Jó volna valahogy ismét ,,előbányászni" az `a->b` hivathozást, majd hozzátenni a `->c`-t, a lista végéről letörölni a `(*D_P(&(a->b)))`-t, és belehelyezni a `(*D_P(&(a->b->c)))`-t. E célból hoztam létre a `string` típusú `expBackup` változót. A módszer tehát a következő. Alapból az `expBackup` változó üres. (Az egyébként ismét egy komoly technikai kérdés, hogy mely akciókban, és milyen esetekben kell üressé tenni. Valahol a műveleti jel vége és a változó kezdete között, de ez korántsem triviális.) Amikor jön az `a` változó az `endPrimaryExpr()` akcióval, belehelyezem az `sList` végére, és az `expBackup` változó is felveszi az `a` értéket. A `->b` szuffix hatása a következő: az utolsó elemet töröljük a listából, az `expBackup`-hoz hozzávesszük a `->b`-t (így az `a->b` lesz), a lista végére pedig az `expBackup` változó címének a kiírása lesz: `(*D_P(&(a->b)))`. A `->c` hatására a lista végéről a `(*D_P(&(a->b)))` törlődik, az `expBackup` végére kerül a `->c` (tehát most már `a->b->c` az értéke), és a lista végére belekerül a kívánt `(*D_P(&(a->b->c)))` érték. Ha ezután műveleti jel jön, akkor az `expBackup` változó értékét üresre kell változtatni. Ezt a következő változó elején (`beginPrimaryExpr()` akció) kell megtenni.

Látszólag megoldottuk a problémát, viszont még mindig van egy kis gond. Mi van például az `a[c->d]->b` kifejezés esetén? Itt nyilván 2 mutató szerepel, ezért a helyes instrumentálása `(*D_P(&(a[(*D_P(&(c->d)))]->b)))`. A fenti módszert alkalmazva módosítás nélkül a `]` után az `expBackup` értéke helytelen lenne, ugyanis vagy üres lenne, vagypedig a `c->d`-t tartalmazná attól függően,

hogy mely akciók során állítottuk üresre (nyilván nem csak 1 helyen lehetséges). Pedig a helyes értéke `a[(*D_P(&(c->d)))]`. Ez a probléma úgy oldható meg, hogy az `expBackup` változót is minden szinten egyedivé kell tenni, tehát a kifejezés elején verembe kell tenni a régit, és üresíteni kell, majd a végén a régihez hozzá kell benni az újat is. Az `expBackup` változók vermét a `stack<string>` típusú `expBckpStack` változó tartalmazza.

Itt még felmerült néhány kisebb-nagyobb technikai akadály, amit most nem részletezek.

6.5. A `?:` operátor

A `?:` operátor kezelése nemcsak a statikus `DU` felépítésénél, hanem az instrumentálásnál is nagyon komoly technikai akadályt jelentett. Ha lett volna egy olyan akció, amely a `:` helyén hívódik meg, akkor elég egyszrű lett volna a probléma. Viszont a nehézséget pont ennek az akciónak a hiánya okozta. Az akciók sorrendjéből kellett kitalálni, hogy melyik ponton kell a `:`-ot a lista végére helyezni. A megoldás a következő: a `?` és a `:` között mindig teljes kifejezés található `beginTopLevelExpr()` és `endTopLevelExpr()` akciókkal határolva. Tehát az `endTopLevelExpr()` helyen kell megállapítani azt, hogy ez a `?` és a `:` közötti kifejezés végét jelenti-e. Ehhez létrehoztam egy logikai típusú elemekből álló vermet. A vermet úgy módosítom, hogy a fenti akció meghívásakor `igaz` érték legyen a tetején akkor, ha a fenti eset áll fenn, `hamis` egyébként. Az inicializáló részen (konstruktor, destruktor) a `beginTopLevelExpr()`, az `endTopLevelExpr()`, a `beginQuestionExpr()` és az `endQuestionExpr()` akciókban kellett módosítanom.

6.6. Függvények

A függvények kezelése nem jelentett komoly akadályt a jól használható szemantikus akciók miatt. A `beginPrimarySuffExpr()` és az `endPrimarySuffExpr()` akciók az `optFuncCall` paraméterrel a függvényhívás paramétereinek az elejét (`(`) és végét (`)`) jelölik. Így az adott helyen gond nélkül behelyezhető a megfelelő síma zárójel. Arra viszont ügyeltem, hogy a teljes függvényhívás az adott szinten 1 atomi egységként jelenjen meg.

6.7. Típuskonverzió, a `sizeof` operátor

A kódot igyekeztem úgy megírni, hogy lehetőleg teljesen a szemantikus akciókra támaszkodjak és abból próbáljam meg rekonstruálni az eredeti kifejezést, és ne kelljen a `token()` függvényt használnom, de sajnos a típuskonverziók és a `sizeof` operátor esetében ezt nem tudtam tartani. A jelenlegi szemantikus akciókból a `token()` kivételével ugyanis sajnos ez nem megoldható. Ezért létrehoztam 2 új string típusú változót: a `typeCastString`-et és az `unaryString`-et. A `token()` akciónál az aktuális tokent mind a `typeCastString` mind az `unaryString`-hez hozzávettem, és a megfelelő helyen beszúrtam az instrumentált kifejezésbe. Itt sok apró technikai probléma előbukkant (például mikor kell

üresre állítani a megfelelő változót), de ezeknek nincs elméleti jelentőségük, ezért nem részletezem őket.

6.8. Érdekességek a megvalósítás során

A kódolás során sok kisebb-nagyobb technikai probléma merült fel. Mindegyiket nem fogom felsorolni, mert túl hosszadalmas én unalmas lenne a dolgozat, viszont egyet-kettőt kiemelek az érdekesebbek közül.

Ki gondolná, hogy az ártalmatlannak tűnő `getchar` és `putchar` függvények komoly galibát okozhatnak? Ezek a függvények voltak az instrumentálás szakítópróbái. Látszólag ez teljesen logikátlan. Viszont az instrumentálás nem az eredeti kódon hajtódik végre, hanem az előfeldolgozott változatára (aminek a kiterjesztése általában `.i`). A fent említett függvények tulajdonképpen makrók, amelyek a perprocesszálás során kifejtődnek. A `getchar()` kifejtésének az eredménye:

```
(--((&_iob[0]))->_cnt >= 0
? 0xff & *((&_iob[0]))->_ptr++
: _filbuf((&_iob[0])));
```

míg a `putchar(1)` makró eredménye a következő:

```
(--((&_iob[1]))->_cnt >= 0
? 0xff & (*((&_iob[1]))->_ptr++ = (char)((1)))
: _flsbuf(((1)),((&_iob[1])));
```

Ezekben aztán minden van! Külön-külön is több mint 1000 szemantikus akció írja le a fenti kifejezést! Ha előjött valahol egy pici hiba, és azt kijavítottam, akkor fennállt a komoly veszélye annak, hogy a fenti kifejezések instrumentálását elrontom, tehát nagyon körültekintőnek kellett lennem. A bonyolult zárójelezés miatt különösen nehéz volt a kifejezés értelmezése is. A program által generált helyesesen instrumentált kifejezés eléggé elrettentő:

```
   (( --(*D_P(&((&(*D_P(&_iob[(0)])))->_cnt))>=0
?(0xff&*(*D_P(&((&(*D_P(&_iob[(0)])))->_ptr))++)
:_filbuf((&(*D_P(&_iob[(0)])))))
```

illetve

```
(( --(*D_P(&((&(*D_P(&_iob[(1)])))->_cnt))>=0
?(0xff&(*(*D_P(&((&(*D_P(&_iob[(1)])))->_ptr))++ =
( char ) ((1))):_flsbuf(((1)) , ((&(*D_P(&_iob[(1)])))))))
```

A bonyolultabb kifejezésekben, mint pl. a fenti, a `++` és `--` szuffixek kezelése is hosszú ideig megoldatlan volt.

7. Kísérleti tapasztalataink

A megvalósított programmal több tesztet hajtottunk végre. Ezek alátámasztották az algoritmushoz fűzött reményeket. A végrehajtott lépések száma általában jóval nagyobb volt, mint a használt memóriahelyek száma, a szelet mérete pedig általában az eredeti program méretének csak töredéke volt.

A legnagyobb állomány, amelyen komoly teszteléseket végeztünk, egy 922 utasításból álló, egyébként 2865 soros program volt. 3 különböző inputon történő

futtatás során a végrehajtási útvonal mérete száma 7481, 64201 és 71858 volt. A második esetben a szelet 13 sort tartalmazott, míg a harmadik esetben átlagosan 41-et. Tehát az eredmény a teljes utasításszámnak kb. 4.4% és 1.4%-a.

Egy másik, kb. 200 utasításból álló programon 3 tesztesetet hajtottam végre. Mindhárom esetben a végrehajtott lépések száma kb. 24400 volt, a szeletbe pedig az utasítások harmada került bele.

Egy 60 utasításból álló programon a végrehajtott lépések száma 356 volt minden esetben, mivel inputtól független, determinisztikus programról van szó. 6 tesztet figyelmbe véve a teljes szelet a programban található utasításoknak az egyötöde-egyharmada volt.

Az ennél kisebb, pár tucatnyi sort tartalmazó programokat tekintve az esetek többségében a programszelet az utasítások egyharmadát tartalmazta.

Néhány esetben feljegyeztük a memóriahelyek és a végrehajtott utasítások számának az arányát is. Ezek közül most hármat emelmék ki. Az egyik egy 68 soros program volt, amelyben a végrehajtott lépések száma 13524 volt, míg a felhasznált memóriahelyek száma csak 175, ami mindössze 1.3%-a a végrehajtott lépések számának. A másik program 133 sort tartalmazott, a végrehajtott lépések száma 3673 volt, a használt memóriahelyek száma pedig 232. Itt a fent említett arány 6.3%. A harmadik tesztpélda 246 sort tartalmazott, a végrehajtott lépések száma 9322, a memóriahelyek száma 433, ami 4.6%-os arányt eredményezett. Nagyobb programoknál, ahol különböző méretű inputok lehetségesek, és a futásidő jelentősen megnőhet, az a tapasztalat, hogy a használt memóriahelyek száma nagyon lassan növekszik, szinte konstans, így már közepes méretű input estén is a végrehajtott lépések száma a különböző memóriahelyek számának több, mint a százszorosát elérik.

A fent említett teszteredmények joggal bátorítanak fel minket a továbblépésre.

8. Összefoglalás

Különféle szeletelési módszereket alkalmaznak hibakeresésre, tesztelésre és karbantartásra. A szeletelő algoritmusok lehetnek statikus vagy dinamikus szeletelő eljárások. Bizonyos alkalmazásokban, mint például a hibakeresés, a dinamikus szeletelés sokkal eredményesebb mint a statikus. A [6] cikkben leírt tapasztalatok azt mutatják, hogy a dinamikus szelet a végrehajtott utasítások kevesebb, mint a felét tartalmazza, illetve nagy valószínűséggel az egész program méretének a 20%-a alatt marad.

Sokféle dinamikus szeletelési módszert publikáltak már, de a legtöbbjükben a dinamikus függőségi gráfot (`DDG`) használták a program futásának belső ábrázolására. Ennek a legnagyobb hátránya az, hogy a `DDG` méretére nincs korlát, hiszen minden végrehajtott utasítás külön csomópontba kerül.

Dr. Gyimóthy Tibor és munkatársai kidolgoztak egy algoritmust, ami a program `D/U` ábrázolását használja a `DDG` helyett, így alkalmas nagyobb méretű programok szeletelésére is. Az eljárás a program futásával párhuzamosan számolja ki a hozzá tartozó szeletet. Az algoritmus azonban csak egy egyszerű programnyelvre lett kidolgozva.

Az algoritmust kiterjesztettük a `C` programozási nyelvre, amihez meg kellett oldanunk egy pár problémát. Ebben a diplomamunkában a nemstruktúrált ugró utasítások használatát, a `C` kifejezések instrumentálását és statikus `DU`-jának a felépítését mutattam be.

Hivatkozások

[1] Agrawal, H., and Horgan, J.: *Dynamic program slicing. SIGPLAN Notices*, No. 6, 1990, pp. 246-256.

[2] Agrawal, H., DeMillo, R.A., and Spafford, E.H.: *Dynamic slicing in the presence of unconstrained pointers.* In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)* (1991), pp. 60-73.

[3] Agrawal, H., DeMillo, R. A., and Spafford, E.H.: *Debugging with dynamic slicing and backtracking. Software—Practice And Experience*, 23(6):589-616, June 1993.

[4] Gyimóthy, T., Beszédes, Á, and Forgács, I.: *An Efficient Relevant Slicing Method for Debugging.* In *Proc. 7th European Software Engineering Conference (ESEC)*, Toulouse, France, Sept. 1999. LNCS 1687, pages 303-321.

[5] Korel, B., and Laski, J.: *Dynamic slicing in computer programs. The Journal of Systems and Software* vol. 13, No. 3, 1990, pp. 187-195.

[6] *Venkatesh, G. A.: Experimental Results from Dynamic Slicing of C Programs.* In *ACM Transactions on Programming Languages and Systems*, Vol 17. No 2, March 1995, Pages 197-216.

[7] Beszédes Árpád, Gergely Tamás, Szabó Zsolt Mihály, Csirik János, Gyimóthy Tibor: *Dynamic Slicing Method for Maintenance of Large C Programs.* 5^{th} European Conference on Software Maintenance and Reengereering (CSMR 2001), Lisszabon, Portugália, 2001.

[8] Beszédes Árpád, Gergely Tamás, Szabó Zsolt Mihály, Faragó Csaba and Gyimóthy Tibor: *Forward Computation of Dynamic Slices of C Programs.* Műszaki leírás, JMLC2000-re elküldve.

[9] Faragó Csaba, Gergely Tamás: `C` *programok dinamikus szeletelése.* Országos Tudományos Diákköri Konferencia, 2001, Eger.

[10] Faragó Csaba, Gergely Tamás: *Handling the Unstructured Statements in the Forward Dynamic Slice Algorithm.* A *Seventh Symposium of Programming Language and Software Tools* (Szeged, 2001) konferenciára elfogadva.

[11] Havasi Ferenc vizsgadolgozata, Szeged, 1999

A. A kifejezések nyelvtana

A CAN `C++` elemzőnek a C kifejezéseire vonatkozó nyelvtana és szemantikus akciói a következők:

```
#token P_LCURLYBRACE "\{"
#token P_RCURLYBRACE "\}"
#token P_LSQUAREBRACKET "\["
#token P_RSQUAREBRACKET "\]"
#token P_LPARENTHESIS "\("
#token P_RPARENTHESIS "\)"
#token P_SEMICOLON ";"
#token P_COLON ":"
#token P_QUESTIONMARK "?"
#token P_SCOPE "::"
#token P_DOT "."
#token P_DOTSTAR ".\*"
#token P_PLUS "\+"
#token P_MINUS "\-"
#token P_STAR "\*"
#token P_DIVIDE "/"
#token P_MOD "%"
#token P_BITWISEXOR "^"
#token P_AMPERSAND "&"
#token P_BITWISEOR "\|"
#token P_TILDE "\~"
#token P_NOT "!"
#token P_ASSIGNEQUAL "="
#token P_LESSTHAN "<"
#token P_GREATERTHAN ">"
#token P_PLUSEQUAL "\+="
#token P_MINUSEQUAL "\-="
#token P_TIMESEQUAL "\*="
#token P_DIVIDEEQUAL "/="
#token P_MODEQUAL "%="
#token P_BITWISEXOREQUAL "^="
#token P_BITWISEANDEQUAL "&="
#token P_BITWISEOREQUAL "\|="
#token P_SHIFTLEFTEQUAL "\<\<="
#token P_SHIFTRIGHTEQUAL "\>\>="
#token P_SHIFTLEFT "\<\<"
#token P_SHIFTRIGHT "\>\>"
#token P_EQUAL "=="
#token P_NOTEQUAL "!=|not_eq"
#token P_LESSTHANOREQUALTO "<="
#token P_GREATERTHANOREQUALTO ">="
#token P_AND "&&|and"
#token P_OR "\|\||or"
#token P_PLUSPLUS "\+\+"
#token P_MINUSMINUS "\-\-"
#token P_COMMA ","
#token P_POINTERTO "\->"
#token P_POINTERTOSTAR "\->\*"
#token P_ELLIPSIS "..."
#token P_SEGMENT ":>"
```

```
optor :
      K_NEW { P_LSQUAREBRACKET P_RSQUAREBRACKET }
    | K_DELETE { P_LSQUAREBRACKET P_RSQUAREBRACKET }
    | P_PLUS
    | P_MINUS
    | P_STAR
    | P_DIVIDE
    | P_MOD
    | P_BITWISEXOR
    | P_AMPERSAND
    | P_BITWISEOR
    | P_TILDE
    | P_NOT
    | P_ASSIGNEQUAL
    | P_LESSTHAN
    | P_GREATERTHAN
    | P_PLUSEQUAL
    | P_MINUSEQUAL
    | P_TIMESEQUAL
    | P_DIVIDEEQUAL
    | P_MODEQUAL
    | P_BITWISEXOREQUAL
    | P_BITWISEANDEQUAL
    | P_BITWISEOREQUAL
    | P_SHIFTLEFT
    | P_SHIFTRIGHT
    | P_SHIFTRIGHTEQUAL
    | P_SHIFTLEFTEQUAL
    | P_EQUAL
    | P_NOTEQUAL
    | P_LESSTHANOREQUALTO
    | P_GREATERTHANOREQUALTO
    | P_AND
    | P_OR
    | P_PLUSPLUS
    | P_MINUSMINUS
    | P_COMMA
    | P_POINTERTOSTAR
    | P_POINTERTO
    | P_LPARENTHESIS P_RPARENTHESIS
    | P_LSQUAREBRACKET P_RSQUAREBRACKET
    ;

literal :
      L_OCTALINT
    | L_DECIMALINT
    | L_HEXADECIMALINT
    | { "L" } L_CHARACTER
    | L_FLOATONE
    | L_FLOATTWO
    | { "L" } ( L_STRING )+
    | K_TRUE
    | K_FALSE
    ;

primary_expression :
      << ac->beginPrimaryExpr(leftoperator); >>
```

```
      literal
      << ac->endPrimaryExprLiteral(literaltype, literalstring); >>
    | << ac->beginPrimaryExpr(leftoperator); >>
      K_THIS
      << ac->endPrimaryExprLiteral(ltThis, "this"); >>
    | << ac->beginPrimaryExpr(leftoperator); >>
      P_SCOPE
      K_OPERATOR optor
      << ac->endPrimaryExprOperator(); >>
    | << ac->beginPrimaryExpr(leftoperator); >>
      P_SCOPE ID
      << ac->endPrimaryExprScopeid(idstring); >>
    | << ac->beginPrimaryExpr(leftoperator); >>
      P_SCOPE
      qualified_id
      << ac->endPrimaryExprScopequalified(); >>
    | << ac->beginPrimaryExpr(leftoperator); >>
      P_LPARENTHESIS expression P_RPARENTHESIS
      << ac->endPrimaryExprParen(); >>
    | << ac->beginPrimaryExpr(leftoperator); >>
      id_expression
      << ac->endPrimaryExprIdexpr(unqualifiedidtype, unqualifiedidstring); >>
    ;

id_expression :
      { ms_modifier_list }
      ( ( class_templ_namespc )+ { K_TEMPLATE } unqualified_id
      | unqualified_id
      )
    ;

unqualified_id :
      K_OPERATOR optor
    | K_OPERATOR ( type_specifier | primitive_type_specifier | cv_qualifier )+
      ( ptr_operator )*
    | { P_TILDE }
      ( class_templ_namespc | ID )
    ;

qualified_id :
      { ms_modifier_list } ( class_templ_namespc P_SCOPE )+
      { K_TEMPLATE } unqualified_id
    ;

casted_primary_expression :
        << ac->beginCastedPrimaryExpr(leftoperator, optExplTypeConv); >>
        ( qualified_type_specifier | primitive_type_specifier )
        P_LPARENTHESIS { expression_list } P_RPARENTHESIS
        << ac->endCastedPrimaryExpr(); >>
      | primary_expression
      | ( << ac->beginCastedPrimaryExpr(leftoperator, optDynamiccast); >>
          K_DYNAMIC_CAST P_LESSTHAN type_id P_GREATERTHAN
          P_LPARENTHESIS expression P_RPARENTHESIS
        | << ac->beginCastedPrimaryExpr(leftoperator, optStaticcast); >>
          K_STATIC_CAST P_LESSTHAN type_id P_GREATERTHAN
          P_LPARENTHESIS expression P_RPARENTHESIS
        | << ac->beginCastedPrimaryExpr(leftoperator, optReinterpretcast); >>
```

```
          K_REINTERPRET_CAST P_LESSTHAN type_id P_GREATERTHAN
          P_LPARENTHESIS expression P_RPARENTHESIS
        | << ac->beginCastedPrimaryExpr(leftoperator, optConstcast); >>
          K_CONST_CAST P_LESSTHAN type_id P_GREATERTHAN
          P_LPARENTHESIS expression P_RPARENTHESIS
        )
        << ac->endCastedPrimaryExpr(); >>
      | << ac->beginCastedPrimaryExpr(leftoperator, optTypeid_type); >>
        K_TYPEID P_LPARENTHESIS type_id P_RPARENTHESIS
        << ac->endCastedPrimaryExpr(); >>
      | << ac->beginCastedPrimaryExpr(leftoperator, optTypeid_value); >>
        K_TYPEID P_LPARENTHESIS expression P_RPARENTHESIS
        << ac->endCastedPrimaryExpr(); >>
    ;

primary_suffixes :
      (   << ac->beginPrimarySuffExpr(optArray); >>
          P_LSQUAREBRACKET expression P_RSQUAREBRACKET
          << ac->endPrimarySuffExpr(); >>
        | << ac->beginPrimarySuffExpr(optFuncCall); >>
          P_LPARENTHESIS { expression_list } P_RPARENTHESIS
          << ac->endPrimarySuffExpr(); >>
        | << ac->beginPrimarySuffExpr(optDot); >>
          P_DOT { K_TEMPLATE }
          { P_SCOPE } id_expression
          << ac->endPrimarySuffExpr(unqualifiedidtype, unqualifiedidstring); >>
        | << ac->beginPrimarySuffExpr(optPointerto); >>
          P_POINTERTO { K_TEMPLATE }
          { P_SCOPE } id_expression
          << ac->endPrimarySuffExpr(unqualifiedidtype, unqualifiedidstring); >>
        | << ac->beginPrimarySuffExpr(optPlusplus); >>
          P_PLUSPLUS
          << ac->endPrimarySuffExpr(); >>
        | << ac->beginPrimarySuffExpr(optMinusminus); >>
          P_MINUSMINUS
          << ac->endPrimarySuffExpr(); >>
      )*
    ;

expression_list :
      assignment_expression ( P_COMMA assignment_expression )*
    ;

unary_expression :
      << ac->beginUnaryExpr(leftopt); >>
      P_PLUSPLUS cast_expression
      << ac->endUnaryExpr(optPlusplus); >>
    | << ac->beginUnaryExpr(leftopt); >>
      P_MINUSMINUS cast_expression
      << ac->endUnaryExpr(optMinusminus); >>
    | << ac->beginUnaryExpr(leftopt); >>
      K_SIZEOF
      ( P_LPARENTHESIS type_id P_RPARENTHESIS
        << ac->endUnaryExpr(optSizeof_type); >>
      | unary_expression[optSizeof_expr]
        << ac->endUnaryExpr(optSizeof_expr); >>
      )
```

```
    | new_expression
    | delete_expression
    | casted_primary_expression primary_suffixes
    | << ac->beginUnaryExpr(leftopt); >>
      unary_operator cast_expression
      << ac->endUnaryExpr(thisopt); >>
    ;


unary_operator :
      P_STAR
    | P_AMPERSAND
    | P_PLUS
    | P_MINUS
    | P_NOT
    | P_TILDE
    ;

new_expression :
      { P_SCOPE } K_NEW { nmodel }
      ( P_LPARENTHESIS expression_list P_RPARENTHESIS
      | /* empty */
      )
      ( new_type_id
      | P_LPARENTHESIS type_id P_RPARENTHESIS
      )
      { new_initializer }
    ;
new_type_id :
       ( type_specifier | primitive_type_specifier | cv_qualifier )+
       { new_declarator }
    ;

new_declarator :
      ptr_operator { new_declarator }
    | direct_new_declarator
    ;

direct_new_declarator :
      P_LSQUAREBRACKET expression P_RSQUAREBRACKET
      ( P_LSQUAREBRACKET constant_expression P_RSQUAREBRACKET )*
    ;

new_initializer :
      P_LPARENTHESIS { expression_list } P_RPARENTHESIS
    ;

delete_expression :
      { P_SCOPE } K_DELETE
      ( P_LSQUAREBRACKET P_RSQUAREBRACKET cast_expression
      | cast_expression
      )
    ;

cast_expression :
      << ac->beginCastExpr(leftoperator); >>
      P_LPARENTHESIS type_id P_RPARENTHESIS
```

```
      << ac->endCastType(); >> cast_expression
      << ac->endCastExpr(); >>
      | unary_expression
    ;

parameter_declaration :
      declaration_specifiers
      { declarator | abstract_declarator }
      { << ac->beginParameterDefault(); >>
        P_ASSIGNEQUAL assignment_expression
      }
    ;

pm_expression :
      << ac->beginPmExpr(leftoperator); >>
      cast_expression
      << ac->medialPmExpr(); >>
      { P_DOTSTAR pm_expression | P_POINTERTOSTAR pm_expression }
      << ac->endPmExpr(); >>
    ;

multiplicative_expression :
      << ac->beginMultiplicativeExpr(leftoperator); >>
      segment_expression
      << ac->medialMultiplicativeExpr(); >>
      { P_STAR multiplicative_expression
      | P_DIVIDE multiplicative_expression
      | P_MOD multiplicative_expression
      }
      << ac->endMultiplicativeExpr(); >>
    ;

segment_expression :
      << ac->beginSegmentExpr(leftoperator); >>
      pm_expression
      << ac->medialSegmentExpr(); >>
      { P_SEGMENT segment_expression }
      << ac->endSegmentExpr(); >>
    ;

additive_expression :
      << ac->beginAdditiveExpr(leftoperator); >>
      multiplicative_expression
      << ac->medialAdditiveExpr(); >>
      { P_PLUS additive_expression | P_MINUS additive_expression }
      << ac->endAdditiveExpr(); >>
    ;

shift_expression :
      << ac->beginShiftExpr(leftoperator); >>
      additive_expression
      << ac->medialShiftExpr(); >>
      { P_SHIFTLEFT shift_expression | P_SHIFTRIGHT shift_expression }
      << ac->endShiftExpr(); >>
    ;

relational_expression :
```

```
      << ac->beginRelationalExpr(leftoperator); >>
      shift_expression
      << ac->medialRelationalExpr(); >>
      { P_LESSTHAN relational_expression
      | P_GREATERTHAN relational_expression
      | P_LESSTHANOREQUALTO relational_expression
      | P_GREATERTHANOREQUALTO relational_expression
      }
      << ac->endRelationalExpr(); >>
    ;

equality_expression :
      << ac->beginEqualityExpr(leftoperator); >>
      relational_expression
      << ac->medialEqualityExpr(); >>
      { P_NOTEQUAL equality_expression | P_EQUAL equality_expression }
      << ac->endEqualityExpr(); >>
    ;

and_expression :
      << ac->beginAndExpr(leftoperator); >>
      equality_expression
      << ac->medialAndExpr(); >>
      { P_AMPERSAND and_expression }
      << ac->endAndExpr(); >>
    ;

exclusive_or_expression :
      << ac->beginExclusiveOrExpr(leftoperator); >>
      and_expression
      << ac->medialExclusiveOrExpr(); >>
      { P_BITWISEXOR exclusive_or_expression }
      << ac->endExclusiveOrExpr(); >>
    ;

inclusive_or_expression :
      << ac->beginInclusiveOrExpr(leftoperator); >>
      exclusive_or_expression
      << ac->medialInclusiveOrExpr(); >>
      { P_BITWISEOR inclusive_or_expression }
      << ac->endInclusiveOrExpr(); >>
    ;

logical_and_expression :
      << ac->beginLogicalAndExpr(leftoperator); >>
      inclusive_or_expression
      << ac->medialLogicalAndExpr(); >>
      { P_AND logical_and_expression }
      << ac->endLogicalAndExpr(); >>
    ;

logical_or_expression :
      << ac->beginLogicalOrExpr(leftoperator); >>
      logical_and_expression
      << ac->medialLogicalOrExpr(); >>
      { P_OR logical_or_expression }
      << ac->endLogicalOrExpr(); >>
```

```
    ;

conditional_expression :
      << ac->beginConditionalExpr(leftoperator); >>
      logical_or_expression
      { << ac->beginQuestionExpr(); >>
        P_QUESTIONMARK expression P_COLON assignment_expression
        << ac->endQuestionExpr(); >>
      }
      << ac->endConditionalExpr(); >>
    ;

assignment_expression :
      throw_expression
    | << ac->beginAssignmentExpr(leftoperator); >>
      conditional_expression
      << ac->medialAssignmentExpr(); >>
      { assignment_operator assignment_expression }
      << ac->endAssignmentExpr(); >>
    ;

assignment_operator :
      P_ASSIGNEQUAL
    | P_TIMESEQUAL
    | P_DIVIDEEQUAL
    | P_MODEQUAL
    | P_PLUSEQUAL
    | P_MINUSEQUAL
    | P_SHIFTRIGHTEQUAL
    | P_SHIFTLEFTEQUAL
    | P_BITWISEANDEQUAL
    | P_BITWISEXOREQUAL
    | P_BITWISEOREQUAL
    ;

expression :
      << ac->beginTopLevelExpr(); >>
      assignment_expression ( << expressionComma(); >>
      P_COMMA assignment_expression )*
      << ac->endTopLevelExpr(); >>
    ;

constant_expression :
      conditional_expression
    ;
```