Strukturált programozás (részlet 7. fejezet)

Dahl — Dijkstra — Hoare

Műszaki Könyvkiadó, Budapest, 1978

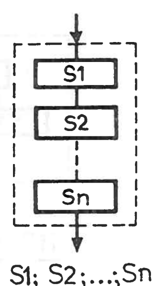
7. A PROGRAMOK MEGÉRTÉSÉRŐL

Életemben sok olyan programozási tanfolyamot láttam, amelyek hasonlatosak voltak az átlagos gépjármű-vezetői tanfolyamokhoz, ahol t.i. az ember nem azt tanulja meg, hogyan érhet el céljához a gépkocsival, hanem azt, hogyan kell a kocsival bánni.

Véleményem szerint a program önmagában sohasem végcél. A program célja, hogy bizonyos számításokat váltson ki, és ezeknek a számításoknak a célja valamely meghatározott eredmény elérése. Bár a programozó által előállított végtermék a program, foglalkozásának igazi tárgyai azok a lehetséges számítások, melyek a program hatására létrejöhetnek, és melyeknek az elvégzése már a gépre hárul. Például ha a programozó azt állítja, hogy programja helyes, akkor állítása valójában a program által kiváltható számítások halmazára vonatkozik.

Az a tény, hogy a teljes tevékenységi lánc végső szakasza, az áttérés a statikus programszövegről a dinamikus programvégrehajtásra, a gép feladata marad, újabb bonyodalmat okoz. Bizonyos értelemben nehezebb programot írni, mint egy matematikai elméletet felállítani. Mind a matematikai elmélet, mind a program strukturált, időtlen objektumok. De míg a matematikai elméletnek önmagában is értelme van, addig a programnak csak a végrehajtása ad értelmet.

Ennek a pontnak hátralevő részében szekvenciális gépre írott programokra szorítkozva azt elemzem, hogy a program megértéséből miként vonhatunk le következtetéseket a végrehajtásával kapott számításokra vonatkozóan. Állítom, bár bizonyítani nem tudom, hogy az ilyen következtetések levonása annál könnyebb és megbízhatóbb, minél egyszerűbb a kapcsolat a program és végrehajtása között. Nem túl pontos megfogalmazásban azt mondhatjuk, kívánatos, hogy a program Struktúrájára tükrözze a számítás struktúráját. Másképpen fogalmazva: "Hogyan csökkenthetjük a távolságot a statikus programszöveg (melynek csak térbeli kiterjedése van), és a program alapján (időben) végrehajtott feldolgozás között?"



1. ábra

A feldolgozás célja valamilyen meghatározott eredmény előállítása. A feldolgozás valamely t_0 időpillanatban kezdődik meg, egy későbbi t_1 időpillanatban ér véget, és feltesszük, hogy a feldolgozás eredményét meg tudjuk határozni a t_0 és a t_1 állapotok összehasonlítása útján. Ha közbülső állapotváltozásokat nem veszünk figyelembe, akkor a feldolgozás hatását egy elemi tevékenység (operáció) eredményének tekintjük.

Ha ezzel szemben figyelembe veszünk közbülső állapotokat is, ez azt jelenti, hogy az eseményeket időben feldaraboljuk. A feldolgozást sorosnak tekintjük, vagyis feltesszük, hogy az részműveletek időbeni

sorozataként valósul meg, és arról kívánunk meggyőződni, hogy e részműveletek eredője éppen a teljes feldolgozás kívánt eredményét adja.

A legegyszerűbb eset, amikor a feldolgozást résztevékenységek lineáris sorozatára bontjuk fel. Folyamatábra formájában ezt az 1. ábra mutatja.

Ennek a felbontásnak a helyességét lépésenkénti elemzéssel igazolhatjuk. Az adott esetben a program és a feldolgozás közötti elvi szakadékot azáltal csökkenthetjük, hogy megköveteljük, a program szövegének lineárisan összefüggő darabjai a résztevékenységek neveit vagy leírásait tartalmazzák, mégpedig olyan sorrendben, ahogyan azokat végre kell hajtani. Korábbi példánkban (a `0<= r< dd` reláció invarianciája) ez a feltétel teljesült:

```
dd := dd/2;
If dd <= r do r := r-dd
```

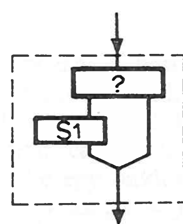
A programot elsődlegesen két résztevékenység időbeli sorozatára bonthatjuk fel, minthogy a program szövegében felismerjük a következő struktúrát:

```
felezd meg dd értékét;
redukáld r-et modulo dd
```

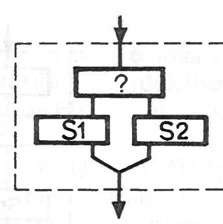
Figyelembe véve minden olyan lehetséges kezdeti állapotot, amely eleget tesz a `0<= r< dd` feltételnek, megállapíthatjuk, hogy ez a felbontás két résztevékenységre a program által kiváltott minden számítás esetére alkalmazható. Eddig tehát minden rendben van.

A programot azonban azon feltételezés mellett írtuk, hogy a "`redukáld r-et modulo dd`" művelet nem elemi tevékenység, ezzel szemben a "`csökkentsd r-et dd-vel`" az. Ha megvizsgáljuk az összes lehetséges hatást a "`redukáld r-et modulo dd`" operáció végrehajtása során, azt állapítjuk meg, hogy ez egyes esetekben a "`csökkentsd r-et dd-vel`" tevékenységgel egyértelmű, más esetekben viszont `r` változatlan marad. Írjuk fel a fenti tevékenységet a következő alakban:

```
if dd <= r do csökkentsd r-et dd-vel
```



2. ábra



3. ábra

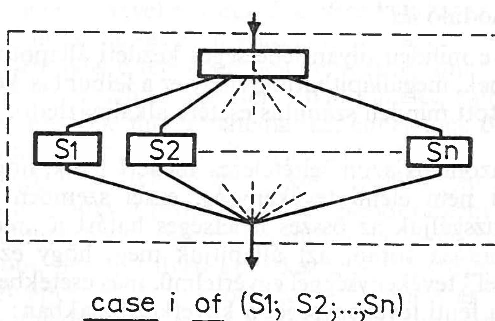
Ezáltal felismerhető, hogy az adott részletezési szinten a "`redukáld r-et moduló dd`" tevékenységnek két egymást kölcsönösen kizáró alakja lehet, és megadtuk azt a szabályt is, amelynek alapján a kettő közötti választás végbemegy. Ha az "`if dd <= r do`" szöveget logikai feltételnek tekintjük, amely a "`csökkentsd r-et dd-vel`" utasításhoz tartozik, akkor kézenfekvővé válik, hogy a feltétel a feltételtől függő utasítás előtt áll. (Ebben az értelemben az

```
if feltétel then utasítás_1
else utasítás_2
```

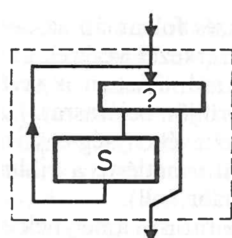
változatban a felírás sorrendje sohasem tükrözi a végrehajtás sorrendjét, mert az `utasítás_1` és az `utasítás_2` egymást kölcsönösen kizárják, ezért nem lehet őket egymás utáni sorrendben végrehajtani.) A választási feltételt C.A.R. Hoare általánosította, létrehozván a **case of** szerkezetet, amelyben kettőnél több választási lehetőség állhat. A 2., 3. és 4. ábrákon ezeket a feltételes utasításokat mutatjuk be folyamatábrán.

Mindezeknek a folyamatábráknak a közös tulajdonságuk, hogy (felül) egyetlen belépési pontjuk van, továbbá (alul) egyetlen kijáratuk. Amint a szaggatott vonal mutatja, ezeket - függetlenül attól, mi van

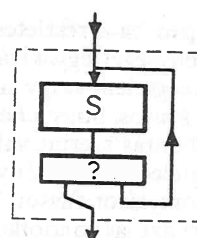
a szaggatott vonalon belül - egy-egy szekvenciális számítás egyetlen résztevékenységének tekinthetjük. Hogy egy kissé szabatosabbak legyünk: nagyszámú lehetséges számítást egyszerre vizsgálva, azokat elsődlegesen ugyanazon résztevékenységek sorozatának tekintjük, és csak a részletesebb vizsgálat - melyben már az egyes szaggatott vonallal határolt blokkok belsejére is kíváncsiak vagyunk - fedi fel, hogy valamely résztevékenység elvégzése során a számítás valójában véges sok különböző lehetséges forma egyikének megfelelően mehet végbe.



4. ábra



5. ábra



6. ábra

A fentiek alapján jól tárgyalhatók azok a feldolgozások, melyek elsődlegesen valamilyen jól meghatározott és rögzített számú tevékenységsorozatra bonthatók; nem tárgyalhatók viszont azok, amelyek elsődlegesen változó számú résztevékenység sorozatára vannak felbontva. És itt válik nyilvánvalóvá a ciklusutasítások szerepe. A "**while** feltétel **do** utasítás" és a "**repeat** utasítás **until** feltétel" konstrukciókat az 5. és 6. ábrán látható folyamatábrákkal szemléltetjük. Ezeknek a folyamatábráknak ugyancsak egyetlen belépési pontjuk van (felül), és egyetlen kijáratuk (alul). Az általuk meghatározott tevékenységet a szaggatott vonalon belüli blokk részletesebb vizsgálata alapján úgy jellemezhetjük, mint adott típusú résztevékenységek "alkalmas számú" időbeli sorozatát.

Áttekintettük a felbontás három fajtáját; ezeket nevezhetjük rendre "láncolásnak", "kiválasztásnak" és "ismétlésnek". Az első kettőt a lépésenkénti elemzés segítségével értelmezhetjük, az utóbbit pedig a teljes indukció segítségével.

Azok a programok, amelyeknek vezérlése kizárólag kiválasztások és ismétlések segítségével valósul meg, magától értetődő módon fordíthatók le a szokásos programozási nyelvekre. Ennek az állításnak a fordítottja nem igaz. Továbbá: ha a felbontásnak e három felsorolt típusára szorítkozunk, korlátozott topológiájú folyamatábrákhoz jutunk azokhoz a folyamatábrákhoz képest, amelyekben bármelyik blokktól bármelyikhez húzhatunk nyilakat. Ez a teljes szabadsághoz képest egy bizonyos programtervezési fegyelmet követel.

Miért javaslom én ennek a programtervezési fegyelemnek a betartását? - Ezt többféleképpen is meg lehet indokolni. Ezek közül az indokok közül néhányat felsorolok, abban a reményben, hogy az olvasó legalább egyet meggyőzőnek talál majd.

Céljaink egyike, hogy olyan struktúrájú programokat állítsunk elő, amelyeknek a megértéséhez szükséges szellemi ráfordítás (valamilyen nem szabatosan definiált értelemben) arányos a program hosszával

(melyet most ugyancsak nem definiálunk pontosan). Különösen védekeznünk kell a lépésenkénti elemzés alkalmazásának robbanásszerű növekedése ellen. Ez rákényszerít minket a régi "Oszd meg és uralkodj" bölcsesség egy újfajta alkalmazására, ezért javasoljuk a felbontás fokozatos tovább finomítását.

Egy ilyen felbontási fokozatot a lépésenkénti elemzés módszerével értelmezhetünk. (Ezt megtehetjük, feltéve, hogy azon résztevékenységek száma, melyekre a számítást elsődlegesen felbontottuk, elegendően kicsiny, és az egyes résztevékenységek hatásának leírása elegendően tömör. Ezekre a feltételekre később még visszatérek; a jelen pillanatban felteszem, hogy teljesülnek.) Ebben az esetben a program szövege alapján le tudunk vonni bizonyos következtetéseket a feldolgozásra vonatkozólag, minthogy a feldolgozásnak és a program szövegének az előrehaladása között nyilvánvaló a kapcsolat. Speciálisan ha a résztevékenységek valamelyikéről a részletesebb elemzés azt állapítja meg, hogy azt egy kiválasztási vagy ismétlési szerkezet valósítja meg, ez a körülmény nem nehezíti meg az elsődleges felbontás megértését, mivel ehhez a résztevékenységek csupán az eredő hatása a lényeges.

Következésképpen ha a részletesebb elemzés folyamán az derül ki, hogy valamely adott résztevékenységet egy kiválasztási szerkezet vezérel, akkor az elsődleges vizsgálat szintjén lényegtelen, hogy a konkrét számításban a vezérlés melyik ágat járta be (egyedül az a fontos, hogy a helyes ág kerüljön bejárásra). Hasonlóan, amennyiben a részletes felbontás szerint valamely résztevékenység egy ismétlési szerkezetből áll, akkor lényegtelen, hogy hányszor kerül ismétlésre a ciklus magja (csak az fontos, hogy éppen annyiszor fusson le, ahányszor kell).

Tehát ismerjük azt a gondolkodási apparátust, amelynek segítségével megérthetjük a kiválasztási szerkezeteket - ez a lépésenkénti elemzés; és ismerjük azt is, amely az ismétlési szerkezetek megértését teszi lehetővé - ez a teljes indukció. Vagyis mindhárom szerkezettípushoz tudunk rendelni egy megfelelő gondolkodási sémát - és ez nagy segítség.

A javasolt tervezési fegyelemnek egy másik előnye is van. Amikor programokat értelmezünk, tulajdonképpen logikai állításokat bizonyítunk be. A lépésenkénti elemzéssel kapcsolatos példánkban azt igazoltuk, hogy a

```
dd := dd/2;
if dd<= r do r := r-dd
```

programrészlet nem változtatja meg a

```
0<= r < dd
```

egyenlőtlenség érvényét. Azonban még ha biztosítani is tudjuk a fenti egyenlőtlenségek teljesülését a programrészlet végrehajtása előtt, akkor sem következik, hogy az mindig fennáll; ti. nem szükségképpen marad érvényes a két utasítás végrehajtása között. Más szavakkal: az ilyen állítások érvénye a feldolgozás végrehajtása közben változhat és ez tipikus a szekvenciális számításokra nézve.

Hasonlóan tulajdoníthatunk értelmet a változóknak. Egy változó számlálhatja bizonyos események bekövetkezését, p1. a pillanatnyilag nyomtatás alatt álló lapra nyomtatott sorok számát. A következő lapra való áttéréskor az említett változó aktuális értéke azonnal 0-ra áll vissza, majd minden sor kinyomtatása után rögtön megnöveli értékét 1-gyel. Azonban a visszaállítást, ill. növelést közvetlenül megelőző pillanatokban a változónak "a pillanatnyi oldalra kinyomtatott sorok száma"-ként való értelmezése nem helyes. Tehát a változónak ilyen értelmezést csak a feldolgozás végrehajtási stádiumától függően adhatunk. Ez az észrevétel azonnal felveti a kérdést: hogyan jellemezzük a feldolgozás előrehaladását?

Röviden, egy olyan koordinátarendszerre van szükségünk, amelynek segítségével azonosíthatók a számítás előrehaladásának különböző diszkrét pontjai, és azt akarjuk, hogy ez a koordinátarendszer független legyen a program felügyelete alatt álló változóktól: hogyha ugyanis a számítás lefolyásának jellemzésére olyan változókat használnánk fel, melyek függnek magától a számítástól, ezzel nem érnénk semmit, minthogy éppen ezeket a változókat kívánjuk értelmezni a feldolgozás előrehaladásának függvényében.

(Egy még nyomósabb ok, amiért nem támaszkodhatunk a program változóira, azonnal nyilvánvalóvá válik, ha a végtelen ciklusokra gondolunk. Végtelen ciklus esetén a program véges sok különböző állapoton való áthaladás után visszakerül egy korábbi állapotba, vagyis a számítás előrehaladásának különböző stádiumában azonos állapot jön létre. Ekkor nyilván nem lehetséges ennek az állapotnak alapján megkülönböztetni a számítás előrehaladásának e két különböző pontját!)

Problémánkat másként is megfogalmazhatjuk. Legyen adva egy végrehajtás alatt álló program és tegyük fel, hogy ez a programvégrehajtás bevégződése előtt egy meghatározott diszkrét ponton félbeszakad.

Hogyan tudjuk azonosítani ezt a pontot, ha p1. a feldolgozást meg kívánjuk ismételni pontosan addig a pontig? Vagy ha a feldolgozás megszakítása valamely dinamikusan jelentkező hiba következménye volt, hogyan azonosíthatjuk a feldolgozás ama pontját, ahol a hiba bekövetkezett, ha nem áll rendelkezésünkre egy teljes memóriakiíratás?

Az egyszerűség kedvéért tegyük fel, hogy a program szövege valamely (lineáris) szövegtérben adott, és hogy rendelkezésünkre áll valamely mechanizmus, amelynek segítségével azonosíthatók a számítás lefolyásának különböző diszkrét pontjai. Nevezzük ezt az azonosítási mechanizmust "szövegmutatónak". (Ha a számítás lefolyásának diszkrét pontjait az egymást követő utasítások között helyezzük cl, akkor a szövegmutató p1. az utasítások közti pontosvesszők alapján azonosíthat.) A szövegmutató tehát egy általánosított utasításszámláló, amely a forrásszöveg bizonyos helyeire mutat.

Ha a vezérlési struktúrában csupán a láncolásra és a kiválasztásra szorítkozunk, akkor a számítás lefolyásának jellemzésére elegendő egyetlen szövegmutató. Ha ismétlések is vannak, akkor már nem elegendő a szövegmutató a számítás lefolyásának jellemzésére. Ciklusba való belépéskor bevezethető azonban egy "dinamikus index", amely mindig az aktuális ismétlés sorszámát mutatja, és amely a ciklus végetérésekor törölhető. Minthogy a ciklusok egymásba skatulyázottan is előfordulhatnak, ennek a megoldásához egy ún. veremtárra ("last-in-first-out" memória) van szükség. Kezdetben a verem üres, majd minden ciklusba való belépéskor egy új dinamikus index kerül a verem addigi tartalma fölé, 0 vagy 1 kezdőértékkel. Valahányszor a "ciklus vége" vizsgálat azt eredményezi, hogy a ciklusnak nincs vége, a verem legfelső indexének értékét eggyel megnöveli; valahányszor pedig azt eredményezi, hogy a ciklusnak vége van, törli a legfelső indexet. (Ez a mechanizmus nagyon jól tükrözi azt, hogy ha már a ciklus alkalmas számú ismétlés után véget ért, akkor a továbbiakban nem érdekes, mennyi volt az ismétlések száma.)

Ha a programozási nyelv eljárásokat is megenged, akkor már nem elegendő egyetlen szövegmutató. Amikor a szövegmutató valamely eljárás törzsének belsejébe mutat, a számítás lefolyásának jellemzéséhez azt is tudnunk kell, hogy az eljárás melyik hívása következtében jutott a vezérlés az adott helyre; ehhez egy másik szövegmutatóra van szükség, amely a hívás helyére mutat. Az eljárások bevezetése esetén tehát az egyszerű szövegmutatót egy veremtárban elhelyezett szövegmutató-rendszerrel kell helyettesítenünk, melynek addigi tartalma fölé minden egyes eljárás híváskor egy újabb elem kerül, minden egyes visszatéréskor pedig törlődik a veremben levő legfelső elem.

A dolog lényege az, hogy ezeknek a mutatóknak az értéke kívül esik a programozó hatáskörén; azt a programozó akaratától függetlenül, a program leírása, illetve a számítás előrehaladása határozza meg. Ennélfogva a mutatók olyan koordinátarendszert határoznak meg, amely független a program változóitól, és így alkalmas arra, hogy segítségével jellemezzük a számítás előrehaladását, és az általa szolgáltatott keretben értelmezzük a program változóinak jelentését.

Természetesen akkor is létezik egy programozótól független koordinátarendszer, amelyben a szekvenciális számítás diszkrét pontjai azonosíthatók, ha a vezérlésátadásokra semmiféle megszorítást nem teszünk; ez tekinthető bizonyos értelemben egy normalizált órának, amely a számítás kezdetétől fogva számlálja a számítás előrehaladásának diszkrét pontjait. Ez a koordinátarendszer egyértelmű ugyan, de teljességgel hasznavehetetlen, minthogy a szövegmutató nem képezi részét.

A fentiek tanulsága az, hogyha elfogadjuk annak szükségességét, hogy a feldolgozásokat az őket kiváltó program szövege alapján vezéreljük, akkor szigorúan tartanunk kell magunkat azokhoz az áttekinthető vezérlési struktúrákhoz, amelyek biztosítják, hogy a feldolgozás előrehaladását egyszerű módon megfeleltethessük a program szövegében való előrehaladásnak.