



Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

---

# LLVM programok szuperoptimalizálása

*Konzulens:*

Dr. Kozsik Tamás

Egyetemi docens

*Szerző:*

Juhász Dávid

Programtervező informatikus MSc

II. évfolyam

Budapest, 2011

A kutatás az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával  
valósult meg, a támogatási szerződés száma TÁMOP 4.2.1./B-09/KMR-2010-0003.

# Tartalomjegyzék

|                                                                     |           |
|---------------------------------------------------------------------|-----------|
| <b>1. Az optimalizálásról formálisan</b>                            | <b>6</b>  |
| 1.1. Fogalmi alapok . . . . .                                       | 6         |
| 1.2. Néhány egyszerű optimalizáció . . . . .                        | 10        |
| 1.2.1. Konstans továbbterjesztés . . . . .                          | 10        |
| 1.2.2. Elérhetőség-vizsgálat . . . . .                              | 11        |
| 1.2.3. Redundáns adatok törlése . . . . .                           | 12        |
| 1.2.4. Nem használt értékadások törlése . . . . .                   | 13        |
| 1.2.5. Függvény beágyazása . . . . .                                | 13        |
| <b>2. Iteratív optimalizálás és szuperoptimalizációk</b>            | <b>15</b> |
| 2.1. Optimalizációk iteratív végrehajtása . . . . .                 | 15        |
| 2.2. Az iteratív optimalizálás futásának hatékonysága . . . . .     | 18        |
| 2.2.1. Fixpont elérhetősége . . . . .                               | 18        |
| 2.2.2. Közös használt analízisek kérdése . . . . .                  | 18        |
| 2.2.3. Optimalizációk lehetséges sorrendjei . . . . .               | 18        |
| 2.3. Az iteratív optimalizálás eredményének hatékonysága . . . . .  | 20        |
| 2.4. Szuperoptimalizációk . . . . .                                 | 22        |
| 2.4.1. A szuperoptimalizációk összeállítása . . . . .               | 23        |
| <b>3. Adatáram-analízisek</b>                                       | <b>25</b> |
| 3.1. Az adatáram-egyenletek . . . . .                               | 25        |
| 3.1.1. Előre haladó adatáram-analízisek . . . . .                   | 26        |
| 3.1.2. Visszafelé haladó adatáram-analízisek . . . . .              | 28        |
| 3.2. Az adatáram-egyenletek megoldása . . . . .                     | 29        |
| 3.3. Két egyszerű adatáram-analízis . . . . .                       | 36        |
| 3.3.1. Konstans továbbterjesztés . . . . .                          | 37        |
| 3.3.2. Nem használt értékadások törlése . . . . .                   | 37        |
| 3.4. A vezérlés szerinti adatáram-analízis működés közben . . . . . | 38        |

|                                                   |           |
|---------------------------------------------------|-----------|
| <b>4. Szuperoptimalizációs keretrendszer</b>      | <b>42</b> |
| 4.1. Integrált analízisek . . . . .               | 43        |
| 4.2. A kombinált analízis . . . . .               | 45        |
| 4.3. A kombinált analízis megoldása . . . . .     | 49        |
| 4.4. Néhány integrált analízis . . . . .          | 57        |
| 4.4.1. Konstans továbbterjesztés . . . . .        | 57        |
| 4.4.2. Elérhetőség-vizsgálat . . . . .            | 57        |
| 4.4.3. Függvény beágyazása . . . . .              | 58        |
| 4.4.4. Nem használt értékadások törlése . . . . . | 59        |
| 4.5. A keretrendszer működés közben . . . . .     | 59        |
| <b>5. Az LLVM</b>                                 | <b>65</b> |
| 5.1. Az LLVM dióhéjban . . . . .                  | 65        |
| 5.2. Az <i>LLVM assembly</i> . . . . .            | 66        |
| 5.2.1. Az LLVM IR 3 alakja . . . . .              | 66        |
| 5.2.2. Az LLVM programok felépítése . . . . .     | 67        |
| 5.2.3. DU- és UD-láncok . . . . .                 | 67        |
| 5.2.4. Static Single Assignment Form . . . . .    | 67        |
| 5.3. Optimalizálás az LLVM 2.9-ben . . . . .      | 68        |
| <b>6. A keretrendszer megvalósítása LLVM-hez</b>  | <b>70</b> |
| 6.1. Analízisek és információik . . . . .         | 70        |
| 6.2. Az analízisek eredményei . . . . .           | 71        |
| 6.3. A kombinált analízis . . . . .               | 72        |
| 6.4. A kombinált analízis végrehajtása . . . . .  | 73        |
| 6.5. Az implementált analízisek . . . . .         | 75        |
| <b>7. További kutatási lehetőségek</b>            | <b>77</b> |
| <b>8. Kapcsolódó munkák</b>                       | <b>79</b> |
| <b>9. Összegzés</b>                               | <b>81</b> |
| <b>A További fontos optimalizációk röviden</b>    | <b>83</b> |
| A.1. Közös részkifejezések törlése . . . . .      | 83        |
| A.2. Ciklusok optimalizációi . . . . .            | 83        |
| A.3. Részleges redundancia eltávolítása . . . . . | 84        |
| A.4. Mutatók optimalizációi . . . . .             | 85        |

|          |                                                                       |            |
|----------|-----------------------------------------------------------------------|------------|
| <b>B</b> | <b>Előre haladó adatáram-egyenletek megoldó algoritmusai</b>          | <b>86</b>  |
|          | B.1. Jól ismert algoritmusok . . . . .                                | 86         |
|          | B.2. Vezérlés szerinti adatáram-analízis . . . . .                    | 89         |
| <b>C</b> | <b>A vezérlés szerinti adatáram-analízis helyessége és lépésszáma</b> | <b>93</b>  |
|          | C.1. Állítások előre haladó analízisek esetén . . . . .               | 93         |
|          | C.2. Állítások visszafelé haladó analízisek esetén . . . . .          | 96         |
| <b>D</b> | <b>A keretrendszer értékelése</b>                                     | <b>99</b>  |
|          | D.1. A mérés menete . . . . .                                         | 99         |
|          | D.2. A mérési eredmények és értékelésük . . . . .                     | 100        |
|          | <b>Irodalomjegyzék</b>                                                | <b>104</b> |

# Bevezetés

Tudományos diákköri dolgozatomban a programok optimalizációjára szolgáló viszonylag új, de mindenesetre kevésbé ismert módszert mutatok be. Ennek lényege, hogy az egyes optimalizációkat nem a fordítóprogramokban jól megszokott iteratív módon egymás után hajtjuk végre, hanem egy menetben, a transzformációkat rekurzív módon tovább optimalizálva. A módszer ugyanazon optimalizációk segítségével jobb végeredményt képes elérni, valamint ezt a jobb eredményt könnyen karbantartható elemekből építkezve biztosítja. Alapjait a Washington Egyetem kutatói dolgozták ki a *Vortex* nevű fordítóprogramjukhoz. A publikált eredmények ([1, 4, 7, 8, 9]) biztatóak voltak, de a módszerrel kapcsolatos további kutatási tevékenységre utaló információkat nem találtam.

Az 1. fejezetben bevezetem a dolgozatomban használt fogalmakat és jelölésrendszert. Ezután a 2. fejezetben röviden bemutatom az iteratív optimalizálási módszert, külön kiemelve a végrehajtásával és eredményével kapcsolatos problémákat. A 3. fejezetben az adatáram-analízisek megoldására szolgáló hatékony algoritmust ismertetek, mely alapjaiban jól ismert ([12, 13]), azonban implementáció szintű leírását és az általános esetbeli hatékonyságára vonatkozó megállapításokat nem találtam. Ez az algoritmus képezi az általam kidolgozott optimalizáló algoritmus alapját, melyet a szükséges fogalmak bevezetése után a 4. fejezetben ismertetek. A módszert meg is valósítottam az LLVM fordítóprogram-eszköztárhoz. Az LLVM optimalizálás szempontjából fontosabb jellemzőit az 5. fejezetben foglaltam össze, magáról a megvalósításról pedig a 6. fejezetben található részletek. A 7. fejezetben további kutatási lehetőségeket vázolok fel, valamint a 8. fejezetben a dolgozat témájához kapcsolódó munkákat foglalom össze. Végül a 9. fejezetben összegzem a dolgozatomban ismertetett eredményeket. Emellett további példák és magyarázatok a függelékben kaptak helyet.

A dolgozat tartalmának rövid áttekintése után külön kiemelném a kutatásom során elért saját eredményeimet:

- Kidolgoztam az 1. fejezetben ismertetésre kerülő fogalmi alapok felhasználásával egy formalizmust, mely a szuperoptimalizációs keretrendszer implementációja irányába mutató leírására alkalmas.
- A 2.4. fejezetben ismertetett szuperoptimalizációk bázisaira vonatkozó egyszerű állításokat tettem az optimalizációk függési gráfjának felhasználásával.
- Az általam kidolgozott formalizmus felhasználásával a 3. fejezetben definiáltam az adatáram-analíziseket.
- Az adatáram-analízisek megoldására alkalmas ismert algoritmust módosítottam oly módon, hogy az elemzést a program egyszeri bejárásával képes legyen megoldani.
- Bizonyítottam az algoritmus lépésszámára vonatkozó állításokat előre- és visszafelé haladó analízisek esetére.
- A 4. fejezetben az integrált analíziseket az adatáram-analízisek általánosításaként definiáltam, valamint az általam kidolgozott formalizmusnak megfelelően adtam meg az integrált analízisek által meghatározott kombinált analízist.
- A kombinált analízis megoldására alkalmas algoritmust adtam az adatáram-analízisek megoldására ismertetett algoritmusom kiegészítésével.
- A 6. fejezetben ismertetésre kerülő megfontolásokkal implementáltam a szuperoptimalizációs keretrendszert az LLVM fordítóprogram-eszköztárhoz, ami az általános algoritmus további, LLVM-specifikus kiegészítését tette szükségessé. (A keretrendszer letölthető a <http://kp.elte.hu/superoptimization> címről.)

# 1. fejezet

## Az optimalizálásról formálisan

Dolgozatom elején röviden ismertetem a programok optimalizálásával kapcsolatos legfontosabb fogalmakat, valamint bemutatok néhány elterjedt optimalizációt.

Az optimalizációk megadására többféle formalizmus létezik, melyek más-más célra alkalmazhatók jól. Dolgozatomban a [8] cikkben használt formalizmus kicsit módosított változatához igazodom, mivel a szuperoptimalizációs keretrendszer implementációs szempontból hatásos leírására ezt fogom használni.

### 1.1. Fogalmi alapok

Először a programok reprezentálására használt végrehajtási gráfot definiálom, valamint ennek felhasználásával az optimalizálás szempontjából szem előtt tartandó fogalmakat adom meg.

**1.1.1. definíció** (Végrehajtási gráf (*CFG*)). A végrehajtási gráf (*Control Flow Graph*, esetleg vezérlésfolyam-gráf vagy vezérlési gráf) egy programhoz rendelt irányított gráf. A gráf csúcspontjai a program egyes utasításai. Két csúcs között akkor és csak akkor halad irányított él, ha a vezérlés a program végrehajtása során a megelőző csúcsból a rákövetkezőre kerülhet.

**1.1.2. definíció** (A végrehajtási gráf fontosabb elemei). A végrehajtási gráf optimalizálás szempontjából kitüntetett elemei a következők:

**Belépési pont.**<sup>1</sup> Az az utasítás, ahová a program végrehajtásának megkezdésekor kerül a vezérlés. Amennyiben nincsenek el nem érhető utasítások, úgy az egyetlen olyan utasítás, amelynek nincs megelőzője.

<sup>1</sup> Tipikusabb a belépési blokk használata, amennyiben a *CFG* csúcsai nem utasítások, hanem alablokkok. Azonban az optimalizálás szempontjából kényelmesebb a *CFG* 1.1.1. definíciójához igazodó 1.1.2. definícióban szereplő *belépési pont* használata.

**Kilépési pont.** Azok az utasítások, amelyek végrehajtásával befejeződik a program futása, vagyis nincs rákövetkezőjük.

**Ciklus.** A vezérlési gráf egy erősen összefüggő komponense<sup>2</sup>.

**Belépő él.** Olyan él, melynek célja egy cikluson belül van, forrása viszont azon kívül.

**Kilépő él.** Olyan él, melynek forrása egy cikluson belül van, célja viszont azon kívül.

**Ciklus feje.** Olyan ciklusbeli csúcs, melybe befut belépő él.

**Természetes ciklus.** Olyan ciklus, melynek egyetlen feje van.

**Visszaél.** Olyan cikluson belül haladó él, melynek célja a ciklus feje.

**Beágyazott ciklus.** Olyan ciklus, melynek feje egy cikluson belül található.

1.1.1. *megjegyzés* (Természetes ciklushoz tartozó fogalmak alternatív definíciója). A természetes ciklust és a kapcsolódó fogalmakat a *dominálás* segítségével is szokás megfogalmazni. Az alternatív definíció dolgozatom szempontjából csak áttételesen fontos, azonban a ciklusok algoritmikus megtalálásához a dominálási kapcsolatok segítségével megfogalmazott feltételek ellenőrzése a járható út, így ennek rövid ismertetése, úgy érzem, nem maradhat el.<sup>3</sup>

Egy programban az  $A$  utasítás pontosan akkor dominálja a  $B$  utasítást (röviden:  $A \text{ dom } B$ ), ha a belépési pontból a  $B$  utasításhoz vezető minden út érinti az  $A$  utasítást. Ekkor azt mondhatjuk, hogy egy  $T \rightarrow H$  él *visszaél* akkor és csak akkor, ha  $H \text{ dom } T$ .<sup>4</sup> Egy  $T \rightarrow H$  visszaélhez tartozó természetes ciklus pedig azon  $X$  csúcsok összessége, melyekre  $H \text{ dom } X$  és van olyan  $X \rightsquigarrow T$  út, melyen nincs rajta a  $H$  csúcs.

1.1.2. *megjegyzés* (Természetes ciklusok fontossága). Amennyiben egy program csak természetes ciklusokat tartalmaz, úgy végrehajtási gráfjának élei két diszjunkt halmazba sorolhatók. Az előre mutató élek a *CFG* feszítőfáját határozzák meg, melyek segítségével minden utasítás elérhető a belépési pontból. A *visszaélek* halmazában pedig pontosan azok az élek tartoznak, amelyek célja dominálja a forrását. Az ilyen tulajdonságú programokat egyszerűsíthetőknek (*Reducible*) nevezzük.

<sup>2</sup> Komponensen a megfelelő maximális részgráfot értjük.

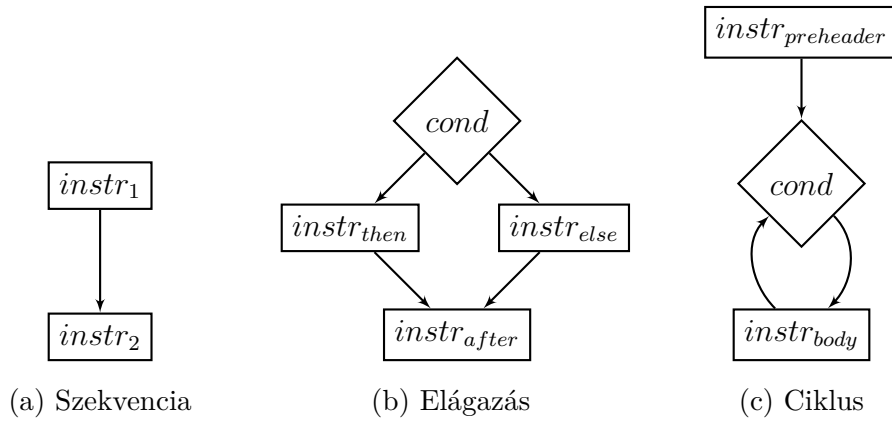
<sup>3</sup> Erről a témakörrel részletesebben [12] 7. fejezetében olvashatunk.

<sup>4</sup> Az utasítások betűjelei az angol terminológiából adódnak: a ciklus feje a *Header Node*, egy visszaél forrása pedig a *Tail Node*.

Az egyszerűsíthető programokban minden ciklusnak egyértelmű a feje, ami nagyban megkönnyíti a program optimalizálását. Valamint az előző megjegyzésben vázolt domináláson alapuló megközelítéssel egy egyszerű algoritmussal megtalálhatók a ciklusok.

A strukturált programok jellemzően egyszerűsíthetőek, azonban a modern programozási nyelvek is tartalmaznak különféle nem-strukturált vezérlési szerkezetet, amik nem-természetes ciklust eredményezhetnek. A nem-természetes ciklusok azonban viszonylag egyszerűen átalakíthatók természetes ciklusokká. Így csak a természetes ciklusokkal kell foglalkoznunk az optimalizálás szempontjából. Az egyszerűség kedvéért ezért a továbbiakban el is hagyom a *természetes* jelzőt a ciklusok előtt, azonban mindig természetes ciklusokra gondolok.

1.1.3. *megjegyzés* (CFG megadása). A végrehajtási gráfokat a dolgozatomban az 1.1. ábrán lévő egyszerű szerkezetekből építkezve adom meg. A belépési pont mindig az ábra tetején lévő utasítás, a vezérlés pedig lefelé haladhat az élek mentén. A feltételes elágazásokat a rombuszok jelölik, melyben a feltétel szerepel. Az elágazás esetén a bal ág az igaz, a jobb ág pedig a feltétel hamis értékéhez tartozik. A ciklusok elején lévő feltételes elágazás igaz érték esetén a ciklusmagba visz, egyébként kilép a ciklusból. A dolgozatomban szereplő egyszerű példák esetén a ciklusból való kilépés a program befejeződését is jelenti, ezért a ciklus feltételéből csak egyetlen kifelé haladó él van, mely a ciklusmagba vezet.



1.1. ábra. Vezérlési gráf

Most pedig áttérek az optimalizáláshoz kapcsolódó alapvető fogalmak bevezetésére.

**1.1.3. definíció** (Analízis). Egy  $\mathcal{A} : \text{Graph} \rightarrow \text{Info}$  függvényt analízisnek nevezünk, ahol  $\text{Graph}$  programok végrehajtási gráfjainak halmaza,  $\text{Info}$  pedig valamilyen információt tartalmazó struktúra elemeinek halmaza. Az információ konkrét formája a megvalósítandó optimalizációtól függően változhat.

**1.1.4. definíció** (Transzformáció). Egy  $\mathcal{T} : \text{Graph} \times \text{Info} \rightarrow \text{Graph}$  függvényt transzformációnak nevezünk, ahol Graph végrehajtási gráfok halmaza, Info pedig a transzformáció elvégzéséhez szükséges információ lehetséges értékeinek halmaza.

1.1.4. *megjegyzés* (Analízisek és transzformációk kapcsolata). Az előbbiek alapján a transzformáció felelős a végrehajtási gráf, ezáltal a program módosításáért. Ehhez azonban felhasznál valamilyen információt, melyet a fentebb definiált analízis képes előállítani egy végrehajtási gráfból. Ahhoz, hogy a transzformáció a megfelelő információk alapján elvégezhesse munkáját, ugyanazt a programot kell módosítania, melyből a felhasznált információ származik. Ezért egy  $G \in \text{Graph}$  végrehajtási gráf optimalizálásának eredménye a  $\mathcal{T}(G, \mathcal{A}(G))$  gráf lesz.

1.1.5. *megjegyzés* (Transzformáció helyessége). Az optimalizálás szempontjából fontos biztosítani, hogy helyes transzformációkat végzünk a programokon. Ez az jelenti, hogy a transzformáció nem változtatja meg a program jelentését, vagyis a program ugyanazt a feladatot oldja meg a transzformáció után, mint előtte. Ennek formális definiálása és még inkább vizsgálata teljesen elkanyarodna dolgozatom fő témájától, ráadásul terjedelmi okokból sem lenne szerencsés belefogni. Így erről bővebben nem értekezem.

1.1.6. *megjegyzés* (Transzformáció végrehajtásához szükséges információk). Természetesen előfordulhat, hogy egy transzformáció elvégzéséhez több egyszerű analízis által gyűjtött információra is szükség van. Ezt az esetet nem szükséges külön kezelni, mivel a szükséges információ felírható  $\text{Info} = \times_{i=0}^n \text{Info}_i$  alakban, ahol a direkt szorzat komponensei között nem szerepel Info.

Ekkor ha az  $\text{Info}_i$  információt az  $\mathcal{A}_i (i = 0, 1, \dots, n)$  analízis határozza meg, akkor  $\forall G \in \text{Graph} : \mathcal{A}(G) = \times_{i=1}^n \mathcal{A}_i(G)$  alak egyszerűen adódik. Tehát bármilyen bonyolult transzformáció esetén a fenti definícióban megadott alakú transzformáció használható.

**1.1.5. definíció** (Optimalizáció). Egy transzformációt és a végrehajtásához szükséges analízist együtt optimalizációnak nevezzük.

Ekkor egy  $\mathcal{T}$  transzformáció által meghatározott optimalizáció egy  $\mathcal{O} : \text{Graph} \rightarrow \text{Graph}$  függvény, amire  $\forall G \in \text{Graph} : \mathcal{O}(G) = \mathcal{T}(G, \mathcal{A}(G))$ , ahol  $\mathcal{A}$  a  $\mathcal{T}$  transzformáció végrehajtásához szükséges információt meghatározó analízis.

1.1.7. *megjegyzés* (Optimalizáció megadása párral). A fenti definíció alapján egy optimalizációt egyértelműen meghatároz egy transzformáció és a hozzá tartozó analízis. Így az optimalizációk megadására használhatjuk az  $\mathcal{O} = (\mathcal{A}, \mathcal{T})$  alakot.

## 1.2. Néhány egyszerű optimalizáció

Ebben az alfejezetben a fentebb megadott formalizmushoz igazodva ismertetek néhány elterjedten használt optimalizációt. Ezek többé-kevésbé formális leírásai a [11] egyetemielőadás-diákon és a [3] könyvbeli fejezetben megtalálhatóak. A hivatkozott szakirodalomban az általam ismertetett optimalizációk megadásának egy része informális, másik része pedig nem illeszkedik az általam bevezetett formalizmushoz. Ezért a pusztán hivatkozás helyett az egyszerűbb optimalizációk újradefiniálása mellett döntöttem.

A jelen alfejezetbe hely hiánya miatt be nem került néhány további fontos optimalizáció informális ismertetése a teljesség kedvéért megtalálható az A. függelékben.

### 1.2.1. Konstans továbbterjesztés

Talán a legtipikusabb optimalizáció a konstans továbbterjesztés (*Constant Propagation*), amely során a program azon változóira történő hivatkozásokat, mely változókban a program adott részén statikusan, fordítási időben ismert konstans található, az adott konstansra cseréljük. Ezzel a transzformációval a memóriahivatkozások számát csökkenthetjük.

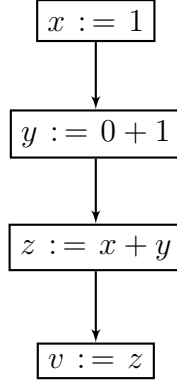
**1.2.1. definíció** ( $\mathcal{O}_{ConstProp}$ ).  $\mathcal{O}_{ConstProp} = (\mathcal{A}_{ConstProp}, \mathcal{T}_{ConstProp})$ , hogy  $\mathcal{A}_{ConstProp} : \text{Graph} \rightarrow \text{Info}_{ConstProp}$  és  $\text{Info}_{ConstProp} : \text{Inst} \times \text{Var} \rightarrow \text{Value} \cup \{\top, \perp\}$ .

Ahol  $\text{Inst}$  a program utasításainak halmaza,  $\text{Var}$  az összes lehetséges változók halmaza,  $\text{Value}$  pedig a változók által tárolható értékek halmaza.

$$\text{Info}_{ConstProp}(i, v) = \begin{cases} \perp & \text{ha } v \text{ nem fordult elő az } i \text{ utasításig} \\ x & \text{ha } v \text{ értéke az } i \text{ utasításban biztosan } x \ (x \in \text{Value}) \\ \top & \text{ha } v \text{ értéke az } i \text{ utasításban nem konstans} \end{cases}$$

Valamint  $\mathcal{T}_{ConstProp} : \text{Graph} \times \text{Info}_{ConstProp} \rightarrow \text{Graph}$  és  $\mathcal{T}_{ConstProp}(\mathbf{G}, \mathbf{A})$  eredménye úgy keletkezik  $\mathbf{G}$ -ből, hogy minden  $i$  utasításának azon  $v$  változó-hivatkozásait, melyekre  $\mathbf{A}(i, v) = v' \in \text{Value}$ ,  $v'$ -re cseréljük.

**1.2.1. megjegyzés** (A konstans továbbterjesztés hatékonysága). A konstans továbbterjesztés hatékonyságának elemzéséhez tekintsük az 1.2. ábrán látható végrehajtási gráfot. Az 1.2.1. definíció nem szól arról, hogy pontosan milyen módon kerül meghatározásra a konstans továbbterjesztéshez használt információ. A legegyszerűbb esetben csak az explicit konstansok kerülnek tárolásra. Ekkor a harmadik értékadásban csak az  $x$  változóra történő hivatkozást cseréljük le 1-re. Azonban egyértelmű, hogy az  $y$  változó is egy fordítási időben ismert értéket tárol, melyet célszerű lenne valóban konstansként kezelni.



1.2. ábra. A hatékony konstans továbbterjesztéshez szükséges a konstansok összevonása.

Ennek egy lehetséges módja a konstansok közötti műveletek elvégzése egy külön optimalizációban, mely csak az egyes utasításokat vizsgálja. Ha egy utasításban csak konstansok szerepelnek, és a köztük lévő művelet fordítási időben elvégezhető, úgy a műveletet a megfelelő konstansra cseréli. Ezt az optimalizációt a konstansok összevonásának (*Constant Folding*) nevezik. Ha ezt elvégezzük, akkor a második értékadás jobb oldalán az 1 érték fog szerepelni, melyet már a legegyszerűbb konstans továbbterjesztés is megfelelően kezel. Azonban a harmadik értékadás még ebben az esetben sem kerül a statikusan megadható legegyszerűbb alakba, hiszen a konstans továbbterjesztés után is jobb oldalán egy összeadás fog szerepelni, melynek operandusai konstansok.

A fenti probléma kiküszöbölésére célszerű a konstans továbbterjesztés analízisébe beépíteni a konstansok összevonását, mely a vizsgált műveletben szereplő operandusokra vonatkozó, már rendelkezésre álló információk figyelembe vételével próbálkozik a művelet statikus elvégzésével. Ekkor a konstans továbbterjesztés elvégzése után a fenti programrészlet mind a négy értékadásának jobb oldalán a megfelelő konstans fog szerepelni.

### 1.2.2. Elérhetőség-vizsgálat

A konstansok behelyettesítése azon kívül, hogy csökkenti a memóriahivatkozások számát, a bináris programkód méretére is jótékony hatást gyakorolhat, ha a programot elérhetőség-vizsgálatnak (*Reachability Analysis*) vetjük alá.

**1.2.2. definíció** ( $\mathcal{O}_{Reachability}$ ).  $\mathcal{O}_{Reachability} = (\mathcal{A}_{Reachability}, \mathcal{T}_{Reachability})$ , hogy  $\text{Info}_{Reachability} = \perp$  és  $\mathcal{T}_{Reachability} : \text{Graph} \times \text{Info}_{Reachability} \rightarrow \text{Graph}$ . Vagyis lényegében  $\mathcal{T}_{Reachability} : \text{Graph} \rightarrow \text{Graph}$ , mivel a transzformáció elvégzéséhez csak az egyes utasításokra van szükség.

$\mathcal{T}_{Reachability}(\mathbf{G})$  eredménye egy olyan gráf, mely  $\mathbf{G}$ -ből a konstans feltétellel rendelkező feltételes elágazások és a konstans érték szerinti elágazások megfelelő ágakra mutató feltétel nélküli ugrásokra történő cseréjével adódik.

1.2.2. *megjegyzés* (Elérhetetlenné váló programrészek). Az elágazások lecserélésével bizonyos utasítások elérhetetlenné válhatnak. Ezek a *CFG*-ben megelőzővel nem rendelkező csúcsokként jelennek meg.<sup>5</sup> Az ilyen utasítások, valamint az ezekből induló láncok<sup>6</sup> mind olyanok, melyekre a program futása során sosem kerülhet a vezérlés. Ezért ezeket az utasításokat nyugodtan törölhetjük, ezzel a program bináris kódjának méretét csökkentve.

Elérhetetlen utasítások egyéb optimalizációk során is keletkezhetnek, így célszerű ezt a rendrakási feladatot külön optimalizációba szervezni, mely az elérhetetlen utasításokat törli a programból. Az angol szaknyelv ezt nevezi *Dead Code Elimination*-nek.

### 1.2.3. Redundáns adatok törlése

Hasznos optimalizáció a redundáns adatok törlése (*Redundancy Elimination*), mellyel a program memóriaigénye csökkenthető, valamint a kevesebb változó miatt az egyes műveletek akár hatékonyabban elvégezhetők, mivel nem kell annyi változót a regiszterekben tartani. Ehhez a program változóit kongruencia osztályokba soroljuk. Az egy osztályba tartozó változók ugyanazt az értéket tárolják, így a rájuk történő hivatkozások az osztály egy reprezentáns elemére történő hivatkozásra cserélhetők. Így az eredményben az egyes kongruencia osztályokból csak egyetlen változó fog operandusként szerepelni a programban, a többi teljesen felesleges, így törölhető lesz.

1.2.3. **definíció** ( $\mathcal{O}_{RedElim}$ ).  $\mathcal{O}_{RedElim} = (\mathcal{A}_{RedElim}, \mathcal{T}_{RedElim})$ , hogy  $\mathcal{A}_{RedElim} : \text{Graph} \rightarrow \text{Info}_{RedElim}$  és  $\text{Info}_{RedElim} : \text{Inst} \times \text{Var} \rightarrow \text{Var}$ .

Ahol  $\text{Inst}$  a program utasításainak halmaza,  $\text{Var}$  az összes lehetséges változók halmaza,  $\text{Info}_{RedElim}(i, v)$  értéke pedig a  $v$  változó  $i$  utasításnál érvényes osztályának reprezentánsa.

Valamint  $\mathcal{T}_{RedElim} : \text{Graph} \times \text{Info}_{RedElim} \rightarrow \text{Graph}$  és  $\mathcal{T}_{RedElim}(\mathbf{G}, \mathbf{A})$  eredménye úgy adódik  $\mathbf{G}$ -ből, hogy minden  $i$  utasításának minden  $v$  változó-hivatkozását az  $\mathbf{A}(i, v)$  reprezentánsra cseréljük.

1.2.3. *megjegyzés* (A kongruencia osztályok meghatározása). A kongruencia osztályok meghatározására a *Value Numbering* technika szolgál. Ennek legegyszerűbb

<sup>5</sup> Természetesen csak a belépési ponttól különböző 0 befokú csúcsok az elérhetetlenek.

<sup>6</sup> Lényegében mindazok az utasítások, melyeket nem dominál a belépési pont.

változata a *Local Value Numbering*, mely nem képes a programbeli elágazások kezelésére, így a gyakorlatban kevésbé alkalmazható, de az elvet hatékonyan képes szemléltetni. A használható változat a *Global Value Numbering*, mely megbízhatóan működik tetszőlegesen bonyolult programok esetén. Ez utóbbi analízisnek két változata is létezik, egy pesszimista és egy optimista. A pesszimista abból indul ki, hogy minden változó más értéket tárol, tehát kezdetben minden változó külön osztályt alkot. Amikor bebizonyosodik, hogy valamely két osztály elemei ugyanazt az értéket tárolják, akkor egyesíti a két osztályt. Az optimista megvalósítás ezzel szemben kezdetben egyetlen osztályt képez az összes változóból, majd iteratíván keres olyan változókat, melyek ellentmondanak annak, hogy az egyes osztályokban lévő változók ugyanazt az értéket tárolják. Ha talál ilyen ellentmondást, akkor megfelelően kettévágja az osztályt. Ez utóbbi lépésszáma nagyobb, azonban pontosabb eredményt ad, vagyis azon változók, melyek végül külön osztályba kerülnek, nagyobb valószínűséggel fognak tényleg különböző értéket tárolni.

#### 1.2.4. Nem használt értékadások törlése

Mindenképp érdemes megemlíteni a nem használt értékadások törlését (*Dead Assignment Elimination*), mely azon értékadásokat törli egy programból, mely értékadások bal oldalán szereplő változót nem használják fel a változónak történő következő értékadásig. Az ilyen helyzetek nem feltétlen a programozó oda nem figyeléséből adódnak, hanem jellemzően más optimalizációk eredményeként állnak elő. Ilyen lehet a korábban említett konstans továbbterjesztés vagy a redundáns adatok törlése, melyek a helyettesítések elvégzésével nem használt értékadásokat hagynak maguk mögött.

**1.2.4. definíció** ( $\mathcal{O}_{DAE}$ ).  $\mathcal{O}_{DAE} = (\mathcal{A}_{DAE}, \mathcal{T}_{DAE})$ , ahol

$\mathcal{A}_{DAE} : \text{Graph} \rightarrow \text{Info}_{DAE}$  és  $\text{Info}_{DAE} : \text{Inst} \rightarrow \mathbb{L}$ . Ahol  $\text{Inst}$  a program utasításainak halmaza,  $\mathbb{L} = \{\uparrow, \downarrow\}$  a logikai értékek halmaza.

$$\text{Info}_{DAE}(i) = \begin{cases} \downarrow & \text{ha } i \text{ nem értékadás vagy az értékadás által tárolt érték szükséges} \\ \uparrow & \text{ha az } i \text{ értékadás által tárolt érték nem kerül felhasználásra} \end{cases}$$

Valamint  $\mathcal{T}_{DAE} : \text{Graph} \times \text{Info}_{DAE} \rightarrow \text{Graph}$  és  $\mathcal{T}_{DAE}(\mathbf{G}, \mathbf{A})$  eredménye úgy adódik  $\mathbf{G}$ -ből, hogy törölünk minden olyan  $i$  utasítást, melyre az  $\mathbf{A}(i)$  érték igaz.

#### 1.2.5. Függvény beágyazása

A programok végrehajtása során egy nem túl olcsó művelet a függvényhívás, melynek kiküszöbölésével csökkenthetjük a futási időt. Ugyanakkor a program bi-

náris mérete növekedni fog, mivel a függvény törzsét bemásoljuk a meghívásának helyére. Ez az optimalizáció az *Inline Expansion* vagy *Inlining*.

**1.2.5. definíció** ( $\mathcal{O}_{\text{Inliner}}$ ).  $\mathcal{O}_{\text{Inliner}} = (\mathcal{A}_{\text{Inliner}}, \mathcal{T}_{\text{Inliner}})$ , ahol

$\mathcal{A}_{\text{Inliner}} : \text{Graph} \rightarrow \text{Info}_{\text{Inliner}}$  és  $\text{Info}_{\text{Inliner}} : \text{Graph} \rightarrow \mathbb{L}$ .

$$\text{Info}_{\text{Inliner}}(\mathbf{G}) = \begin{cases} \downarrow & \text{ha a } \mathbf{G} \text{ által reprezentált függvényt nem kell beilleszteni} \\ \uparrow & \text{ha a } \mathbf{G} \text{ által reprezentált függvényt be kell illeszteni} \end{cases}$$

Valamint  $\mathcal{T}_{\text{Inliner}} : \text{Graph} \times \text{Info}_{\text{Inliner}} \rightarrow \text{Graph}$  és  $\mathcal{T}_{\text{Inliner}}(\mathbf{G}, \mathbf{A})$  eredménye úgy adódik  $\mathbf{G}$ -ből, hogy minden olyan függvényhívás helyére, mely a  $g$  függvényre mutat, beillesztjük a  $g$  függvény törzsét, ha az  $\mathbf{A}(\mathbf{G}')$  érték igaz. Ahol  $\mathbf{G}'$  a  $g$  függvény végrehajtási gráfja.

1.2.4. *megjegyzés* (A szükséges információ). Ez az optimalizáció kicsit kilóg a sorból, mivel a transzformáció elvégzéséhez szükséges információ nem az éppen optimalizált függvényből<sup>7</sup> származik, hanem a meghívott függvényre kell vonatkozzon. Éppen ezért az  $\mathcal{A}_{\text{Inliner}}$  eredménye nem függ az éppen optimalizált program gráfjától.

1.2.5. *megjegyzés* (A beágyazás járulékos hatásai). A függvényhívás költségének el-tüntetése mellett a bemásolt függvénytörzsben egyéb hatékonyságot növelő optima-lizációkat is végezhetünk a befoglaló függvényre vonatkozó információk felhasználá-sával. Ilyen lehet például a konstans paraméter propagálása a bemásolt függvény-törzsön belül.<sup>8</sup>

<sup>7</sup> Eddig következetesen program optimalizációjáról beszéltem, itt csak azért használom a függ-vény szót, mert a programozási gyakorlatban igazából erről van szó.

<sup>8</sup> Ilyen optimalizálást más módon is lehetne végezni, de az már az interprocedurális optima-lizálás világába vezetne. A dolgozatomban ismertetésre kerülő keretrendszer jelen állapotában intraprocedurális optimalizálásra alkalmas, ezért nem térek ki az egyéb technikákra.

## 2. fejezet

# Iteratív optimalizálás és szuperoptimalizációk

Az iteratív optimalizálási technika jelenti a jelenlegi optimalizáló fordítóprogramok körében az optimalizálást. Ebben a megközelítésben pontosan az [1.1.](#) fejezetben ismertetett módon értelmezettek az optimalizációk. Azaz két elkülönülő részből állnak: egy vagy több analízisből, valamint egy transzformációból. Egy transzformáció elvégzéséhez először lefuttatják a szükséges analíziseket, majd a kapott információt átadják a transzformációnak, amely módosítja a programot.

A [2.1.](#) alfejezetben röviden bemutatom ezen iteratív optimalizálási módszer működését, majd a következő két alfejezetben rávilágítok arra, hogy bár ez a módszer egyeduralkodónak tekinthető az optimalizáló fordítóprogramok körében, mégis több probléma felmerül a használatával kapcsolatban. Végül a problémák ismertetése után a [2.4.](#) alfejezetben ismertetem a szuperoptimalizációk elvét, ami a problémák bizonyos részét sikeresen kiküszöböli, azonban más problémákat még égetőbbé tesz.

Az iteratív optimalizálás eredményének problémáit és a szuperoptimalizációk alapjait ismertető részek az [\[1, 8\]](#) cikkekre és a [\[9\]](#) konferencia előadásra épülnek. Az iteratív optimalizálás végrehajtására és karbantartására vonatkozó megállapításaimat a fenti cikkek szuperoptimalizációk hasonló tulajdonságairól szóló részeiből szűrtem le. A szuperoptimalizációk összeállítására vonatkozó állításaimat nem letem fel a szakirodalomban.

### 2.1. Optimalizációk iteratív végrehajtása

Amíg egyetlen optimalizációt szeretnénk végrehajtani, addig nagyon egyszerű a dolgunk: Végrehajtjuk a szükséges analízis(ek)e)t, majd a transzformációt. Azonban amikor több optimalizációnk is van, ami a tipikus eset, nem mindegy, hogy hogyan

hajtjuk végre az optimalizációkat, hiszen azok sorrendje befolyásolhatja az végeredményt.

A legegyszerűbb példa lehet a *Dead Assignment Elimination* elhelyezése az optimalizációk sorában. Ha ezt az optimalizációt elsőként hajtjuk végre, akkor nem sok hasznát látjuk, mivel annak feladata jellemzően az egyéb optimalizációk által hátrahagyott nem használt értékadások eltüntetése lenne. Tehát ezt az optimalizációt célszerű lehet utolsóként végrehajtani a programon.

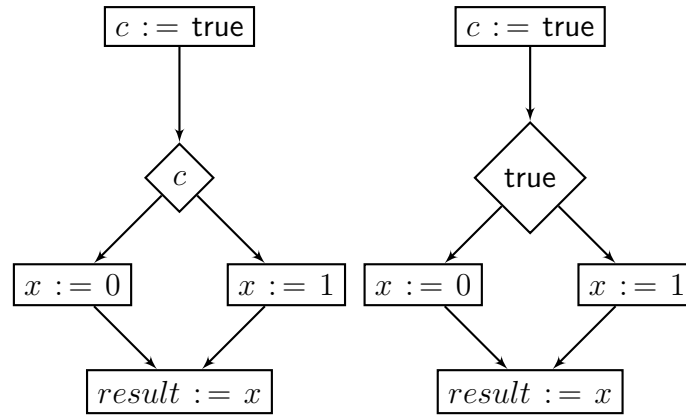
A fenti egyszerűen eldönthető feladaton, a *Dead Assignment Elimination* elhelyezésén kívül azonban általában nem lehet biztosan megmondani, hogy tetszőleges program esetén az optimalizációk milyen sorrendjében kapjuk a legjobb eredményt. Persze ahhoz, hogy ezt legyen esélyünk meghatározni, még a legjobb eredmény fogalmát is tisztázni kellene, hiszen az optimalizációk különböző sorrendje által adott végeredmények lehet, hogy más-más szempontok szerint jók. De ha lenne is egy konkrét szempontunk, ami számunkra a legfontosabb, akkor sem lehetne előre megmondani, hogy tetszőleges program esetén milyen sorrendben hajtsuk végre az optimalizációkat az adott szempont szerinti legjobb eredmény eléréséhez.

Tovább bonyolítja a képet, hogy nem csak egyszerűen az optimalizációk egy permutációját kell keresnünk, hanem az optimalizációk egy olyan sorozatát, amelyben az egyes optimalizációk többször is előfordulhatnak. Sőt bizonyos esetekben egyes optimalizációk igazából feleslegesek, mert nem változtatnak semmit a programon, de ezt aztán végképp nem lehet előre megjósolni.

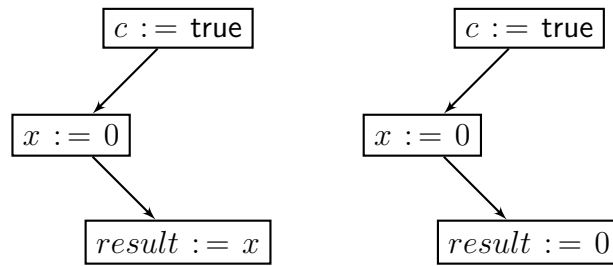
**2.1.1. megjegyzés (Példa).** A fenti bekezdés alátámasztásához nézzünk egy egyszerű programot, mely a **2.1a.** ábrán látható. Ebben a programban a  $c$  változó konstans, így a program eredménye is konstans lesz. Ez a következtetés egészen triviális, azonban ha formálisan szeretnénk bizonyítani, akkor apró lépésekben kell haladnunk. Valami ilyesmi figyelhető meg az optimalizációk esetén is.

A program optimalizációjához lényegében csak konstans továbbterjesztésre és elérhetőség-vizsgálatra van szükségünk. Ezen két optimalizációt pedig a **2.1.** ábrán feltüntetett sorrendben kell végrehajtanunk. Más sorrend esetén lenne olyan menet, amikor az éppen végrehajtott optimalizáció nem tudna semmit sem változtatni a programon. A **2.1d.** ábrán látható alakot már egyik optimalizáció sem képes tovább optimalizálni.

A fenti példából látható, hogy egy egyszerű program optimalizációjához sem annyira magától értetődő az optimalizációk megfelelő sorrendjének meghatározása. Azonban ha már egyik optimalizáció sem képes módosítani a programot, akkor biztosan megállhatunk. Így egy matematikailag jól meghatározott dolgot kell keresnünk: fixpontot.



(a) Optimalizálendő program (b) Konstans továbbterjesztés után



(c) Elérhetőség-vizsgálat után (d) Újabb konstans továbbterjesztés után

2.1. ábra. Egy egyszerű program iteratív optimalizálásának menetei közti állapotok.

**2.1.1. definíció** (Optimalizáció tevékeny végrehajtása). Azt mondjuk, hogy egy optimalizáció egy végrehajtása tevékeny volt, ha az adott végrehajtásban az optimalizáció transzformációja valamilyen módosítást végzett a programon.

**2.1.2. definíció** (Naiv iteratív optimalizálás). Az optimalizációk naiv iteratív végrehajtása során az egyes optimalizációkat tetszőleges sorrendben hajtjuk végre egymás után mindaddig, amíg már egyik végrehajtása sem tevékeny.

**2.1.2. megjegyzés** (Fixpont ellenőrzése). Annak ellenőrzését, hogy egy optimalizáció végrehajtása tevékeny volt-e könnyen megtehetjük, ha az egyes transzformációktól lekérdezhető, hogy legutóbbi végrehajtás során módosították-e a programot.

## 2.2. Az iteratív optimalizálás futásának hatékonysága

### 2.2.1. Fixpont elérhetősége

A legnagyobb probléma a naiv iteratív megközelítéssel, hogy egyáltalán megtalálhatjuk-e a fixpontot. Általánosságban feltehetjük, hogy van két optimalizációnk, az egyik  $A$ -ból  $B$ -t csinál, majd a következő menetben a másik  $B$ -ből  $A$ -t. Így az optimalizációink sosem jutnak nyugvópontra. Ennek kiküszöböléséhez egyrészt megtehetnénk, hogy figyeljük az optimalizációk iterálása során előállt program-állapotokat, és amennyiben olyan állapothoz jutunk, amivel már találkoztunk, akkor leállítjuk az iterációt. Ez elég erőforrásigényes megoldás lenne, ráadásul igazán jól csinálni még annál is bonyolultabb.

2.2.1. *következmény* (Korlátozás bevezetése). Az előbbiek miatt az iteratív végrehajtásra explicit korlát bevezetése szükséges. Ez a korlát vonatkozhat az iterációk számára, vagy lehet időkorlát. A lényeg, hogy a korlát elérésekor mindenképpen leállítjuk az optimalizációk végrehajtását.

2.2.2. *megjegyzés* (Bizonyított terminálás). Természetesen ha külön bizonyítjuk, hogy a használt optimalizációk biztosan jól működnek együtt, és előbb-utóbb elérnek egy fixpontot, akkor elhagyható a korlátozás.

### 2.2.2. Közös használt analízisek kérdése

Egy másik probléma, hogy sok optimalizáció végrehajtása esetén lehetséges, hogy egyes transzformációk érvénytelenítik a már végrehajtott analízisek által összegyűjtött valamely információt. Ugyanakkor lehetnek olyan transzformációk is, amelyek közös analízisektől függnék.

2.2.3. *következmény* (Optimalizációk hatékony ütemezése). Azon optimalizációkat, melyek közös analíziseket használnak, a lehető legközelebb kell végrehajtani egymáshoz úgy, hogy közben a lehető legkevesebb egyéb analízist érvénytelenítsük a végrehajtásukkal.

### 2.2.3. Optimalizációk lehetséges sorrendjei

A gyakorlatban a legegyszerűbb megoldás az optimalizációk iteratív végrehajtására, hogy egy vektorba helyezzük az egyes optimalizációkat, majd ezeket sorban végrehajtjuk egymás után az egész vektort ismételve, amíg a vektor egy bejárása-

kor már egyik optimalizáció sem módosítja a programot. A vektort fel lehet tölteni például az előző pontban ismertetett hatékony ütemezést biztosító elv szerint.

2.2.4. *megjegyzés* (Hatékony ütemezés pro és kontra). A fenti elv szerint ütemezve az optimalizációkat biztosíthatjuk, hogy a közösen használt és időközben biztosan nem érvénytelenített analíziseket ne kelljen újra futtatnunk a később végrehajtott transzformációhoz. Ezzel hatékonyabbá tehetjük az optimalizációk végrehajtását, legalábbis ami az egyes menetek<sup>1</sup> futási idejét illeti. Azonban előfordulhat, hogy ez a hatékony ütemezés a transzformációk végrehajtásának olyan sorrendjét eredményezi, mely mellett több menetre lesz szükség, mint egy lassabb meneteket eredményező sorrend esetén.

Eddig arról volt szó, hogy az optimalizációk iterálását milyen hatékonyan lehet végrehajtani. De még fontosabb kérdés, hogy az optimalizálás végeredménye, az optimalizált program milyen hatékony lesz. Mivel magát az optimalizálást csak egyszer kell végrehajtanunk, míg a programot jó esetben sokszor, ezért ha az optimalizálás kicsit lassú, de nagyon gyors programot eredményez, az sokkal jobb, mint egy gyors optimalizáció, ami nem nagyon javít az eredeti program hatékonyságán.

Az optimalizált program hatékonyságával kapcsolatos kérdések feszegetésének elején visszautalnék a 2.1. ábrán bemutatott egyszerű példára. Abban a példában láthattuk, hogy vannak olyan optimalizációk, melyek eredményessége függ egy másiktól. Az egyik csak akkor tudja módosítani a programot, ha a másik már végrehajtott valamilyen módosítást. Ezek alapján célszerűnek tűnhet a függésben lévő optimalizációkat egy meneten belül a megfelelő sorrendben végrehajtani, azonban ezzel legfeljebb egy menetet takaríthatunk meg az ismételtetésből.

Arról sem feledkezhetünk meg, hogy a példában lévő függés két irányú volt. Először a konstans továbbterjesztés, majd az elérhetőség-vizsgálat, végül ismét a konstans továbbterjesztés tudott javítani a programon. A kölcsönös függés esetén tehát nem egy egyszerű sorbarendezerési problémával<sup>2</sup> állunk szemben.

2.2.5. *következmény* (Optimalizációk kölcsönös függése). Ha  $\mathcal{O}_1$  és  $\mathcal{O}_2$  optimalizációk kölcsönösen függnek egymástól, akkor a lehető legjobb eredmény elérése érdekében az  $\mathcal{O}_1$  minden tevékeny végrehajtása után végre kell hajtani az  $\mathcal{O}_2$ -t, és az  $\mathcal{O}_2$  minden tevékeny végrehajtása után végre kell hajtani az  $\mathcal{O}_1$ -t.

2.2.6. *megjegyzés* (Lehetetlen küldetés). Ha valamilyen módon sikerülne is a konkrét optimalizációk közti függéseknek megfelelően nagyon hatékony végrehajtási sorrendet felállítani az optimalizációk között, akkor az nem lenne több egy szép ered-

<sup>1</sup> Egy menet alatt itt a vektor egyszeri bejárását értem.

<sup>2</sup> A sorbarendezerési problémára az angol szakirodalomban a *Phase-ordering Problem* kifejezéssel hivatkoznak.

ménynél. A gyakorlat azt mutatja, hogy bizonyos esetekben bizonyos optimalizációkat akarunk végrehajtani egy programon, más esetekben, más programokon részben vagy egészben más optimalizációkat. Ez azt jelenti, hogy az optimalizációink elvileg minden, de legalábbis sok kombinációjára rendelkezésünkre kellene álljon egy hatékony sorrend. Tovább rontja ezt a helyzetet, hogy sorban keletkezhettek újabb és újabb optimalizációk, melyeket a meglévő sorrendek mindegyikébe be kellene tudni szűrni egy optimális helyre. Az is előfordulhat, hogy bizonyos optimalizációk úgy módosulnak, hogy az optimalizáció korábbi függései megszűnnek, esetleg újak keletkeznek. Ekkor megint csak felül kellene vizsgálni az összes iterációs sorrendet.

2.2.7. *megjegyzés* (A használt hatékony ütemezés). Az előbbiek figyelembe vételével, jobb híján, az optimalizációk iteratív végrehajtását a gyakorlatban úgy optimalizálják, hogy a közös analíziseket felhasználó transzformációkat próbálják egymáshoz minél közelebb tartani, és a több transzformáció által használt analízisek információinak érvénytelenítését, így azok újraszámolásának szükségességét próbálják elkerülni.

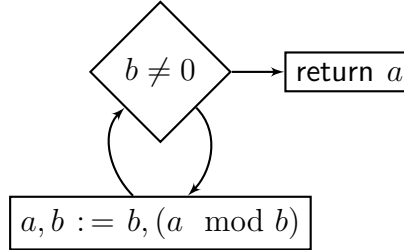
2.2.8. *megjegyzés* (Felső korlát az optimalizáláshoz). Annyit megállapíthatunk, hogy az iteratív megközelítéssel egy tetszőleges program optimalizációja során nem tudjuk előre megmondani, hány menetet kell végrehajtanunk a fixpont eléréséig. Természetesen a gyakorlatban használt iterációs- vagy időkorlátok miatt tudunk felső korlátot adni a futásidőre, de ha a korlát elérése miatt áll le az optimalizálás, akkor az eredmény nem is került fixpontba.

## 2.3. Az iteratív optimalizálás eredményének hatékonysága

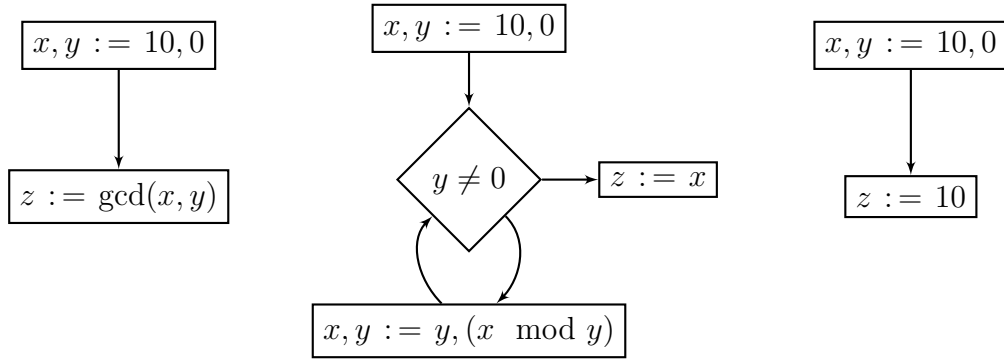
A korábbi példában a kölcsönös függés csak annyit jelentett, hogy egymás után többször végre kellett hajtani az optimalizációkat, és így elértük a lehető legjobb eredményt. Ez általában azonban nem igaz a kölcsönös függésben lévő optimalizációkra. Előfordulhatnak olyan esetek, amikor egy kioptimalizálható kódrész olyan helyen található a programban, ahol az iteratív megközelítés nem képes a legjobbat kihozni a kölcsönösen függő optimalizációkból.

Jellemzően a ciklusmagok azok a helyek, amikbe kölcsönös függésben lévő optimalizációk esetén az iteratív optimalizálásnak beletörhet a bicskája. Ebben az a különösen szomorú, hogy éppen a ciklusmagokat lenne célszerű a lehető legjobban optimalizálni.

2.3.1. *megjegyzés* (Példa). Nézzünk egy egészen egyszerű példát a fenti állításra. Vegyük a 2.2. ábrán látható Euklideszi algoritmust<sup>3</sup> és a 2.3a. ábrán lévő programrészt, melynek két szám legnagyobb közös osztójára van szüksége. Az Euklideszi algoritmust, mivel egy nagyon egyszerű függvényről van szó, beágyazzuk a hívás helyére. Az így kapott, a 2.3b. ábrán látható, programot próbáljuk tovább optimalizálni.



2.2. ábra. Az Euklideszi algoritmus ( $\text{gcd}(a, b)$ ) végrehajtási gráfja.



(a) Az eredeti program    (b) A  $\text{gcd}(x, y)$  beágyazása után    (c) A kívánatos állapot

2.3. ábra. Egyszerű példa, ahol eredménytelen az iteratív megközelítés.

Rövid meggondolás után látható, hogy az eredeti program jelentése megegyezik a 2.3c. ábrán lévő programéval. Hogy eljussunk a kívánatos állapotba nem is lenne szükség semmi másra, csak a konstans továbbterjesztés és az elérhetőség-vizsgálat logikájának összekapcsolására. Azonban sajnos akárhányszor, akármilyen sorrendben hajtjuk is végre a szükségesnek látszó optimalizációkat, azok nem fognak tudni semmilyen módosítást végrehajtani a programon. A konstans továbbterjesztés nem tehet semmit, amíg a feltételben lévő változó szerepel a ciklusmagban lévő értékadás jobb oldalán. Az elérhetőség-vizsgálat szintén nem tehet semmit, amíg egy konstans nem kerül az elágazás feltételébe.

2.3.2. *következmény* (Az iteratív megközelítés korlátai). Az előbbi példában láthattuk, hogy az egymástól kölcsönösen függő optimalizációk esetén az iteratív megközelítés során korlátokba ütközünk a ténylegesen végrehajtható transzformációkat

<sup>3</sup> Itt és a továbbiakban helytakarékosági szempontból szimultán értékadásokat is használunk a gráfokban, még ha a gyakorlatban használt programozási nyelvek ezt nem is támogatják.

illetően. Ez a korlát pedig azért jelentkezik, mert az egyes optimalizációk nem képesek más optimalizációkkal egyszerre működni.

**2.3.1. definíció** (Optimalizációk egyszerre működése). Az optimalizációk egyszerre működnek, ha képesek a programon ténylegesen végrehajtott módosítások nélkül is információt cserélni egymással a program egy adott állapotának vizsgálata során.

2.3.3. *megjegyzés* (Egyszerre működés és együttműködés). Az általam bevezetett *egyszerre működés* fogalma nem egyszerűen együttműködést jelent, hiszen a különálló optimalizációk az iteratív végrehajtás során is együttműködnek, amennyiben az egyik eredménye a következő bemenete. Viszont ez az együttműködés annyiban kevesebb az *egyszerre működés*nél, hogy az iteratív optimalizálás során együttműködő optimalizációk csak a programon ténylegesen végrehajtott módosítások útján képesek kommunikálni, információt a többi optimalizáció tudtára hozni.

## 2.4. Szuperoptimalizációk

A 2.3b. ábrán látható ciklus optimalizálásával kapcsolatban felmerült probléma az egyes transzformációk kölcsönös függéséből származott. Egész pontosan abból, hogy a kölcsönösen függő optimalizációk külön-külön elemezték a programot, és a saját tudásuk birtokában nem tudtak semmilyen transzformációt végrehajtani rajta. Azonban ha bármely optimalizáció elérte volna a másik optimalizáció információját az elemzés közben, sikeresen eltüntethették volna a ciklus fejében lévő elágazást, azaz magát a ciklust. Az információk ilyenét megosztását elérhetjük, ha a kölcsönösen függő optimalizációkból egyetlen egységet képzünk, mely a közös tudás birtokában képes döntéseket hozni.

**2.4.1. definíció** (Szuperoptimalizáció). A több alapvető optimalizáció logikáját egyesítő optimalizációt szuperoptimalizációnak nevezzük.

2.4.1. *megjegyzés* (Példa). Ha a konstans továbbterjesztésből, beleértve a *Constant Folding* optimalizációt is, valamint az elérhetőség-vizsgálatból egyetlen szuperoptimalizációt készítünk, akkor a 2.3b. ábrán látható programrészről ez az egyetlen optimalizáció képes lesz eltüntetni a ciklust. Ezt megteheti, mivel a két alapvető optimalizáció közös tudásának birtokában egyértelművé válik, hogy az  $y$  változóban lévő érték a feltétel első elérésekor 0, vagyis egyszer sem kerül végrehajtásra a ciklusmagban lévő szimultán értékadás, ezért azt nem is kell figyelembe venni az  $y$  változóban lévő érték meghatározásakor. A  $z$  változónak történő értékadás ezután triviálisan adódik. Így egyetlen, bár kétség kívül bonyolultabb optimalizáció végrehajtásával el lehet jutni a 2.3c. ábrán látható alakhoz.

Tehát a kölcsönös függésben lévő egyszerű optimalizációkból készített superoptimalizáció képes megmenteni a helyzetet. Bár ez a megoldás is problémás.

### 2.4.1. A superoptimalizációk összeállítása

Egy superoptimalizáció megalkotásához minden kölcsönös függésben lévő optimalizációt be kell építeni a superoptimalizációba. Lehet több superoptimalizációnk, melyek mindegyike kölcsönösen függő optimalizációkat egyesít, de két superoptimalizáció nem kerülhet kölcsönös függésbe.

**2.4.2. definíció** (Optimalizációk függési gráfja). Legyen az optimalizációink halmaza  $O$ . Ekkor a  $G = (O, E)$  irányított gráf az  $O$  optimalizáció-halmaz függési gráfja, ahol  $(p, q) \in E \Leftrightarrow p$  függ  $q$ -től.

**2.4.3. definíció** (Superoptimalizáció bázisa). Egy superoptimalizáció bázisának azon optimalizációk halmazát nevezzük, melyeket egyesít.

**2.4.4. definíció** (Superoptimalizációk optimális bázisai). Legyen  $O$  az optimalizációk halmaza,  $G$  pedig a hozzátartozó függési gráf. Ekkor  $G$  minden erősen összefüggő komponensének csúcshalmaza egy-egy superoptimalizáció optimális bázisa.

*2.4.2. megjegyzés* (Függések öröklődése). A superoptimalizációk között a bázisukat képező optimalizációk közti függések állnak fenn.

*2.4.3. következmény* (Optimális bázisú superoptimalizációk sorbarendezhetők). A pontosan az optimális bázisok elemeit tartalmazó superoptimalizációk a függések mentén sorbarendezhetők.

*Bizonyítás.* Elég azt belátni, hogy az egyes optimális bázisok között nincsenek kölcsönös függések, hiszen ekkor a superoptimalizációk függési gráfja körmentes lesz, ami azt jelenti, hogy a superoptimalizációk sorbarendezhetők.

Mivel az optimális bázisokat az optimalizációk függési gráfjának erősen összefüggő komponensei adják, ezért tetszőlegesen kiválasztott két bázis között nem haladhat mindkét irányban irányított út, mert akkor egy erősen összefüggő komponens alakult volna.  $\square$

*2.4.4. megjegyzés* (Optimális bázisok és superoptimalizációk karbantartása). Bár az optimális bázisok lehetőséget biztosítanak iteratív végrehajtás mellett is a lehető leghatékonyabb végeredményt produkáló superoptimalizációk készítésére, az optimalizációk halmazának változása és az egyes optimalizációk módosítása az optimális

bázisok módosulásához vezethet. Ekkor pedig a meglévő szuperoptimalizációk módosítása válik szükségessé. Ráadásul az egyes optimalizációk ki-be kapcsolhatóságának igénye bonyolítja a képet.

Egyszerűbb megoldásnak tűnhet, hogy egyetlen szuperoptimalizációt készítsünk az összes optimalizáció egybeépítésével. Így egyrészt csak egyetlen optimalizációt kell végrehajtani a programon, másrészt elkerülhetjük a különböző szuperoptimalizációk és a köztük lévő függések karbantartását.

2.4.5. *megjegyzés* (Szuper-lehetetlen küldetés). Ha egyetlen szuperoptimalizációt készítünk, akkor egy nagyon bonyolult logikára épülő komplex optimalizációnk lesz, amiben a legapróbb változtatás is átláthatatlan módosításokat eredményezhet. Ebben az esetben is problémát jelentene az egyes egyszerű optimalizációk ki-be kapcsolhatóságának igénye, ami ahhoz vezet, hogy minden szükséges optimalizációhalmazhoz külön szuperoptimalizációt kellene készíteni. Ezek után pedig egy egyszerű optimalizáció megváltoztatása vagy egy új optimalizáció beépítése nem csak egy bonyolult szuperoptimalizáció módosítását eredményezné, hanem egyszerre sokét. Érezhető, hogy ez a gyakorlatban nem kivitelezhető feladat, akármilyen jó programozók is állnak a rendelkezésünkre.

2.4.6. *megjegyzés* (Motiváció). A 2.3.1. és 2.4.1. megjegyzésekből látható, hogy az iteratív optimalizálási technika által nyújtott eredmény hatékonyságát növelhetjük szuperoptimalizációk használatával, ugyanakkor a 2.4.5. megjegyzés alapján a monolitikus szuperoptimalizációk használhatatlanok a gyakorlatban. Egy olyan keretrendszerrel, mely szuperoptimalizációk moduláris összeállítását teszi lehetővé, kombinálhatjuk az iteratív optimalizálás modularitását és karbantarthatóságát a szuperoptimalizációk hatékonyságával. Egy ilyen keretrendszer elméleti leírását adja dolgozatom 4. fejezete, valamint az LLVM-hez történő megvalósítást ismerteti a 6. fejezet. Az implementált keretrendszer által optimalizált programok futási idejéről a D. függelék tartalmaz rövid összefoglalót, mely alapján a [8] cikkben ismertetethez hasonló eredmények születtek.

## 3. fejezet

# Adatáram-analízisek

Mielőtt rátérnék a szuperoptimalizációs keretrendszerre, ismertetem az adatáram-analíziseket (*Data Flow Analysis*), melyek általam meghatározott megoldó algoritmusuk képezi a keretrendszer alapját.

Az adatáram-analízisek hagyományos ismertetése megtalálható a [11] diákon és a [3, 12, 13] könyvekben. Dolgozatom 3.1. alfejezetében azoktól eltérően, általánosan definiálom az adatáram-analízisek fogalmát, melyek így egyszerűen felhasználhatók lesznek a szuperoptimalizációs keretrendszer leírásához. A szuperoptimalizációs keretrendszer szempontjából az analízisek definíciójának mikéntjétől sokkal fontosabb az elemzések elvégzése, ezért a 3.2. alfejezetben ismertetek egy ismert megoldásra épülő, általam kidolgozott algoritmust, mely az adatáram-analíziseket hatékonyan oldja meg. Ezenkívül a B. függelékben ismertetek néhány jól ismert megoldó algoritmust az általam használt formalizmushoz igazítva, valamint a jelen fejezetben szereplő algoritmus előre haladó analíziseket megoldó speciális esetét. Ezt abban a reményben teszem, hogy az elsőre talán kissé bonyolultnak tűnő algoritmushoz vezető út állomásainak végignézése segít az algoritmus működési elvének megértésében. Az algoritmus ismertetése mellett annak helyességére és lépésszámára vonatkozó állításokat is megfogalmazok, melyek bizonyítása a C. függelékben található.

A fejezet végén mutatok két konkrét adatáram-analízist, valamint két egyszerű példát az előre haladó analízisek algoritmusom segítségével történő megoldására.

### 3.1. Az adatáram-egyenletek

Az adatáram-analízisek az analízisek egy speciális csoportját képezik, melybe tartozó analízisek a program dinamikus jellemzőiről gyűjtenek információt statikus

időben. Jellemzően a változók, kifejezések futás közbeni értékeivel kapcsolatos információk meghatározását valósítják meg.

**3.1.1. megjegyzés** (Speciális analízisek). Tehát az adatáram-analízisek absztrakt szinten olyanok, mint bármely más analízis, így  $\mathcal{DA} : \text{Graph} \rightarrow \text{Info}_{DA}$  alakban megadhatók. Különbség az általuk gyűjtött információban van, melynek meghatározására különböző megoldások léteznek.

Az adatáram-analízisek által meghatározott információ  $\text{Info}_{DA} : \text{Inst} \rightarrow \text{Info}$ , ahol  $\text{Inst}$  a program utasításainak halmaza. Vagyis a transzformáció minden egyes utasításról lekérdezhet valamilyen információt. Ez a lekérdezhető információ az, ami-  
ben az egyes adatáram-analízisek különböznek.

### 3.1.1. Előre haladó adatáram-analízisek

**3.1.1. definíció** (Előre haladó adatáram-egyenletek). Az előző megjegyzés jelöléseivel, egy program minden  $i$  utasításához tartozzon két érték<sup>1</sup>:  $in_i, out_i \in \text{Info}$ , legyen  $\tau : \text{Info} \times \text{Inst} \rightarrow \text{Info}$  átmenet-függvény, valamint  $\mu : \wp(\text{Info}) \rightarrow \text{Info}$  összefésülő függvény, ahol  $\wp(\text{Info})$  az  $\text{Info}$  halmaz hatványhalmazát jelöli.

Ekkor az adatáram-analízis megoldását a következő egyenletrendszer adja:

$$\begin{aligned} out_i &= \tau(in_i, i) \\ in_i &= \mu(\{out_p | p \in \text{pred}(i)\}) \end{aligned} \quad (i \in \text{Inst})$$

Ahol  $\text{pred} : \text{Inst} \rightarrow \wp(\text{Inst})$  egy utasítás megelőzőinek halmazát határozza meg.

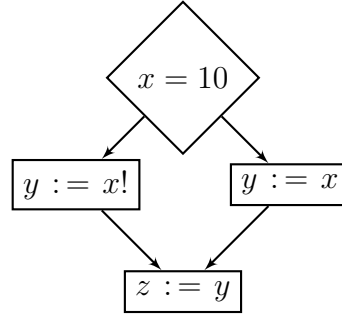
Ezeket az egyenleteket nevezzük adatáram-egyenleteknek (*Data Flow Equations*).

**3.1.2. megjegyzés** (Az egyenletekben használt függvények). Az adatáram-egyenletekben szereplő átmenet- és összefésülő függvények a konkrét analízis által meghatározottak. Az átmenet-függvény feladata, hogy a bejövő adatáram-információt az aktuális utasításnak megfelelően módosítsa. Ez a frissített információ kerül az utasítás kimenő éleire. Az összefésülő függvény felel azért, hogy egy utasítás megelőzőiből származó információk egyetlen bemenő információra képződjenek le, ami átadható az átmenet-függvénynek. Amennyiben egy utasításnak nincs egyetlen megelőzője sem, akkor az összefésülő függvény egy kitüntetett információt ad vissza. Ezt az információt nevezzük *kiindulási információ*nak, mivel a belépési pont ezzel az információval rendelkezik.

**3.1.3. megjegyzés** (Kimenő információk). A fenti definícióban minden utasításnak egyetlen kimenő információja lehet, ami továbbítódik minden rákövetkezőhöz. Ez

<sup>1</sup> Általában alaplombok felett szokták definiálni az adatáram-egyenleteket, azonban dolgozat témájának szempontjából szerencsésebb az utasításonkénti megadása.

megszorítást jelent az előállítható információk pontosságára, mely egyértelműen látható a 3.1. ábrán is. Ha csak egyetlen kimenő információja lehet a feltételes elágazásnak, akkor nem tudjuk a bal oldali ágban a  $10!$  értéket fordítási időben meghatározni. Ezért a különböző kimenő élek mentén különböző adatáram-információ továbbítása szükségeltetik a jó eredmény elérése érdekében.<sup>2</sup>



3.1. ábra. Egy elágazás két ágában különböző információk szükségesek.

**3.1.2. definíció** (Adatáram-egyenletek több kimenő információval). Az adatáram-egyenletek 3.1.1. definíciója a következők szerint módosul:

Minden  $i$  utasításhoz tartozik egy bemenő információ ( $in_i \in \text{Info}$ ), valamint minden rákövetkezője felé vezető kimenő információk:  $\forall j \in \text{succ}(i) : out_{i,j} \in \text{Info}$ , ahol  $\text{succ} : \text{Inst} \rightarrow \wp(\text{Inst})$  egy utasítás rákövetkezőinek halmazát határozza meg. Ekkor az átmenet-függvény alakja a következő:  $\tau : \text{Info} \times \text{Inst} \rightarrow (\text{Inst} \times \text{Info})$ , hogy  $\tau(n, i) = o \in (\text{Inst} \rightarrow \text{Info})$  és  $\mathcal{D}(o) = \text{succ}(i)$ , ahol  $\mathcal{D}(o)$  az  $o$  függvény értelmezési tartománya. Azaz az átmenet-függvény eredménye az utasítás minden rákövetkezőjéhez pontosan egy adatáram-információt rendel. Az összefésülő függvény nem változik, az adatáram-egyenletek alakja pedig a következő lesz:

$$\begin{aligned} out_{i,j} &= \tau(in_i, i)(j) \quad (j \in \text{succ}(i)) \\ in_i &= \mu(\{out_{p,i} | p \in \text{pred}(i)\}) \end{aligned} \quad (i \in \text{Inst})$$

3.1.4. *megjegyzés* (Adatáram-analízis és végrehajtási gráf). A fenti definíció megfogalmazható a végrehajtási gráfok segítségével is. Ekkor azt mondhatjuk, hogy egy  $CFG$  minden csúcsához tartozik egy információ. Ez az  $i$  csúcs esetén az  $in_i$  érték. Valamint minden él mentén továbbítható információ. Az  $(i, j)$  él mentén haladó információ az  $out_{i,j}$  értéknek felel meg. Ezután a  $\tau$  és  $\mu$  függvények használata triviálisan adódik az információk frissítéséhez.

<sup>2</sup> Az adatáram-analízisek formális leírásai túlsiklanak a különböző kimenő információk szükségességén, és csak a 3.1.1. definíciónak megfelelő egyszerű esetet adják meg.

### 3.1.2. Visszafelé haladó adatáram-analízisek

A 3.1.1. definícióban megadott adatáram-analízis egyenletei az információt a vezérlés irányának megfelelően továbbították. Ezért az ilyen típusú analíziseket előre haladó analíziseknek (*Forward Analysis*) is nevezik. Bár a legtöbb adatáram elemzéssel járó információgyűjtés megoldható így, van olyan optimalizáció, melyhez a szükséges információt nem lehet előre haladva meghatározni. Ilyen tipikusan az 1.2.4. részben ismertetett nem használt értékadások törlése.

A fentebb ismertetett adatáram-analízisekkel teljesen analóg módon definiálhatjuk a visszafelé haladó adatáram-analíziseket (*Backward Analysis*), ha képzeletben a CFG éleinek irányítását megfordítjuk.

**3.1.3. definíció** (Visszafelé haladó adatáram-egyenletek). Egy program minden  $i$  utasításához tartozzon két érték:  $in_i, out_i \in \text{Info}$ , legyen  $\tau : \text{Info} \times \text{Inst} \rightarrow \text{Info}$  átmenet-függvény, valamint  $\mu : \wp(\text{Info}) \rightarrow \text{Info}$  összefésülő függvény.

Ekkor a visszafelé haladó adatáram-analízis megoldását a következő egyenletrendszer adja:

$$\begin{aligned} out_i &= \tau(in_i, i) \\ in_i &= \mu(\{out_p \mid p \in succ(i)\}) \end{aligned} \quad (i \in \text{Inst})$$

Ahol  $succ : \text{Inst} \rightarrow \wp(\text{Inst})$  egy utasítás rákövetkezőinek halmazát határozza meg.

3.1.5. *megjegyzés* (Különbség az előre- és visszafelé haladó egyenletek között). Ha a 3.1.1. és 3.1.3. definíciókat összevetjük, akkor látható, hogy az előre- és visszafelé haladó adatáram-analízisek egyenletei között a különbség formálisan mindössze annyi, hogy az előre haladó esetben egy utasítás bejövő információit a megelőző, míg visszafelé haladó esetben a rákövetkező utasításaitól kapja.

3.1.6. *megjegyzés* (Visszafelé haladó analízisek és több kimenő információ). Míg az előre haladó adatáram-analízisek esetén egyértelmű igény mutatkozott az egyszerű definíció kibővítésére, hogy a különböző rákövetkezők felé különböző információt lehessen továbbítani, itt erre semmi szükség. A visszafelé haladó analízisek egy utasítástól annak megelőzőjéhez arra vonatkozó információt továbbítanak, hogy az utasítás és a program vége között mi történik. Ez az információ pedig egy utasítás bármely megelőzője számára ugyanaz kell legyen. Így nincs értelme a definícióban több kimenő információ megadását lehetővé tenni. Azonban az egyszerűség kedvéért a visszafelé haladó adatáram-analízisek esetén is használhatjuk a kimenő információ  $out_i$  alakja helyett az  $out_{i,j}$  alakot. Ekkor természetesen az  $i$  utasítás minden  $j$  rákövetkezője esetén az  $out_{i,j}$  alatt az  $out_i$  kimenő információt értjük.

3.1.7. *következmény* (Adatáram-analízisek megadása hármassal). A fentiek alapján két tetszőleges adatáram-analízis az információ terjedésének iránya mellett az

átmenet- és összefésülő függvényeikben különbözhet. Tehát ezen három érték egyértelműen meghatároz egy adatáram-analízist, így a továbbiakban az adatáram-analízisekre  $\mathcal{DA} = (\delta, \tau, \mu)$  alakban hivatkozhatunk, ahol  $\delta \in \{\text{Forward}, \text{Backward}\}$ .

3.1.8. *megjegyzés* (Az adatáram-analízisek eredménye). A 3.1.1., 3.1.2. vagy 3.1.3. definíciók bármelyikében megadott egyenletrendszer megoldása után az analízis által szolgáltatott információ a következő lesz:

$$\forall i \in \text{Inst} : \text{Info}_{\mathcal{DA}}(i) = in_i$$

## 3.2. Az adatáram-egyenletek megoldása

A következőkben ismertetett algoritmus elvei megegyeznek a [12] 8.7. fejezetében megadott strukturális elemzéssel. Továbbá az algoritmusom nagyban hasonlít a [13] 6.3. fejezetében bemutatott algoritmusra, azonban attól eltérően a *CFG* bejárési sorrendjét menet közben határozza meg, nem egy előzetesen elvégzett postorder bejárás segítségével.<sup>3</sup>

Az algoritmus egy egyszerűbb, csak előre haladó analízisek megoldására alkalmazható változata, illetve további egyszerűbb és kevésbé hatékony megoldó algoritmusok találhatók a B. függelékben. Az algoritmus helyességére és hatékonyságára vonatkozó állítás bizonyítása a C. függelékben kapott helyet.

3.2.1. *megjegyzés* (Nem elérhető utasítások). Fontos megjegyezni, hogy az adatáram-analízisek akkor szolgáltatnak pontos eredményt, ha a program minden utasítását dominálja a belépési pont. Másként szólva, ha van a programban nem elérhető utasítás, mely tudvalevőleg nincs hatással a program működésére, az ahhoz tartozó adatáram-egyenletek befolyásolják a többi egyenletet is, ami így a dinamikus működést kevésbé pontosan leíró információhoz vezet. Ezért az adatáram-analízisek elvégzése előtt célszerű biztosítani, hogy a programban lévő minden utasítás elérhető. Ezt megtehetjük például egy *Dead Code Elimination* optimalizáció végrehajtásával.

**3.2.1. definíció** (Újraformalizált vezérlés szerinti adatáram-analízis<sup>4</sup>). A 3.2.1. algoritmus egy adott  $\mathcal{DA} = (\delta, \tau, \mu)$  adatáram-analízis esetén a következő definíciók felhasználásával képes megoldani egy  $G = (\text{Inst}, \text{Edges})$  végrehajtási gráffal adott program elemzését:

<sup>3</sup> Az algoritmus eredetileg a szuperoptimalizációs keretrendszer igényeihez készült, azonban menet közben kiderült, hogy az algoritmus magját képező részek az adatáram-analízisek megoldására alkalmas algoritmust adnak.

<sup>4</sup> Az algoritmus a B.2.1. definícióban adott algoritmus újraformalizálásával adódik.

$\text{Edges} = \text{Edges}_\delta \subset \text{Inst} \times \text{Inst}$  a program gráfjában az adott analízis bejárási irányának megfelelő irányítású élek halmaza, azaz

$$\begin{aligned}\text{Edges}_{\text{Forward}} &= \text{Edges} \\ \text{Edges}_{\text{Backward}} &= \{(i, j) \mid (i, j) \in \text{Inst} \times \text{Inst} \wedge (j, i) \in \text{Edges}\}\end{aligned}$$

$\mathcal{P} : \text{Inst} \rightarrow \wp(\text{Inst})$  és  $\mathcal{S} : \text{Inst} \rightarrow \wp(\text{Inst})$  a bejárás iránya szerinti megelőzők és rákövetkezők halmazait adják meg. Ezek az  $\text{Edges}$  halmaz segítségével könnyen definiálhatók:

$$\begin{aligned}\mathcal{P}(j) &= \{i \mid i \in \text{Inst} \wedge (i, j) \in \text{Edges}\} \\ \mathcal{S}(i) &= \{j \mid j \in \text{Inst} \wedge (i, j) \in \text{Edges}\}\end{aligned}$$

$\text{Loops}$  a programban lévő természetes ciklusok halmaza és  $\text{Headers} \subset \text{Inst}$  a ciklusok fejeinek halmaza. A két halmaz közti kapcsolatot a  $\mathcal{H} : \text{Loops} \rightarrow \text{Headers}$  bijekció teremti meg, mely minden ciklushoz a hozzátartozó fejet rendeli.

Az  $\mathcal{L} : \text{Inst} \rightarrow \text{Loops} \cup \{\perp\}$  függvény minden utasításhoz az adott utasítást közvetlenül tartalmazó ciklust rendeli<sup>5</sup>. Ezt a  $\mathcal{B} : \text{Loops} \rightarrow \wp(\text{Inst})$  függvény segítségével lehet megfogalmazni, mely egy ciklushoz a hozzá tartozó utasítások halmazát rendeli.

$$\mathcal{L}(i) = \begin{cases} \perp & \text{ha } \nexists l \in \text{Loops} : i \in \mathcal{B}(l) \\ l & \text{ha } l \in \text{Loops} \wedge i \in \mathcal{B}(l) \wedge \forall l' \in \text{Loops} : (i \in \mathcal{B}(l') \Rightarrow \mathcal{H}(l) \in \mathcal{B}(l')) \end{cases}$$

Az  $\mathcal{I} : \text{Loops} \rightarrow \wp(\text{Edges})$  függvény a bejárás iránya szerint a ciklusokba belépő élek halmazát határozza meg, azaz:

$$\mathcal{I}(l) = \{(i, j) \mid (i, j) \in \text{Edges} \wedge i \notin \mathcal{B}(l) \wedge j \in \mathcal{B}(l)\}$$

Az  $\mathcal{F} : \text{Loops} \rightarrow \wp(\text{Edges})$  függvény azon élek halmazát adja, melyek forrása a bejárás iránya szerint az adott cikluson belül található, azaz:

$$\mathcal{F}(l) = \{(i, j) \mid (i, j) \in \text{Edges} \wedge i \in \mathcal{B}(l)\}$$

Az  $\text{entering} : \text{Inst} \times \text{Inst} \rightarrow \mathbb{L}$  logikai függvény azt adja meg, hogy egy él a bejárás iránya szerint belépő él-e, azaz:

$$\text{entering}(i, j) \Leftrightarrow \mathcal{L}(j) \in \text{Loops} \wedge (i, j) \in \mathcal{I}(\mathcal{L}(j))$$

Az  $\text{exiting} : \text{Inst} \times \text{Inst} \rightarrow \mathbb{L}$  logikai függvény azt adja meg, hogy egy él a bejárás iránya szerint kilépő él-e, azaz:

$$\text{exiting}(i, j) \Leftrightarrow \mathcal{L}(i) \in \text{Loops} \wedge j \notin \mathcal{B}(\mathcal{L}(i))$$

<sup>5</sup> A cikluson kívüli utasításokhoz a  $\perp$  érték tartozik.

Az  $\mathcal{E} : \text{Loops} \rightarrow \wp(\text{Edges})$  függvény adott ciklushoz meghatározza azon élek halmazát, melyek a bejárás iránya szerint legfeljebb egy szinttel kintebb lévő ciklusba lépnek. Ennek precíz definiálásához szükséges az utasítások beágyazottsági mélységének megadása, melyet a  $\text{depth} : \text{Inst} \rightarrow \mathbb{N}$  függvény határoz meg:

$$\text{depth}(i) = |\{l \mid l \in \text{Loops} \wedge i \in \mathcal{B}(l)\}|$$

Vagyis egy utasítás beágyazottsági mélysége azt adja meg, hogy az utasítást közvetlenül befoglaló ciklus a ciklusok egymásba ágyazása mentén milyen mélységben található<sup>6</sup>. Ezek alapján az adott ciklusból legfeljebb egy szintet kilépő élek halmaza a következő lesz:

$$\mathcal{E}(l) = \{(i, j) \mid (i, j) \in \mathcal{F}(l) \wedge \text{exiting}(i, j) \wedge \text{depth}(j) \geq \text{depth}(i) - 1\}$$

Nem lenne elég a  $\text{depth}(j) = \text{depth}(i) - 1$  feltétel, mivel a ciklussal azonos szinten lévő másik ciklusba beugró élek is ide kell tartozzanak, ráadásul visszafelé haladó analízisek esetén tetszőlegesen mélyre is beugorhat egy él.

Végül az **Entries** halmaz a bejárás kezdőcsúcsait tartalmazza:

$$\text{Entries} = \{i \mid i \in \text{Inst} \wedge \mathcal{P}(i) = \emptyset\} \cup \{\mathcal{H}(l) \mid l \in \text{Loops} \wedge \mathcal{I}(l) = \emptyset\}$$

**3.2.2. megjegyzés** (A vezérlés szerinti algoritmus működése). Az újraformalizált vezérlés szerinti algoritmus működését a **B.2.2.** megjegyzésben szereplő elvekből eredetileg a következőképpen fogalmazhatjuk meg:

- A feldolgozandó utasításokat egy listában tároljuk, és egy utasítást csak akkor teszünk a listába, ha minden bemenő információja a rendelkezésünkre áll. Kivéve a ciklusbeli utasítások kívülről történő elérését, mivel a ciklusok a bejárás szempontjából egy egységet alkotnak.
- Tehát egy utasításhoz tartozó egyenletek kiszámítása után csak a bejárhatóvá váló rákövetkezőket tesszük a listába.
  - A cikluson kívüli, vagy a feldolgozott utasítással egy ciklusban lévő rákövetkezőket akkor tesszük a listába, ha minden bemenő élükön elérhető információ.
  - Egy ciklusba belépő élen keresztül elért utasítás esetén a közvetlen befoglaló ciklusának fejét tesszük a listába, ha a ciklus minden bemenő élén elérhető információ.

<sup>6</sup> A ciklusokon kívüli utasításokra **depth** értelemszerűen 0-át ad.

---

**Algorithm 3.2.1** Újraformalizált vezérlés szerinti adatáram-analízis

---

```
1:  $I := \text{Edges} \times \{\downarrow\}$ 
2:  $Q$  üres sor
3: Entries elemeit betesszük  $Q$ -ba
4: while  $Q$  nem üres do
5:   Kivesszük  $Q$  következő elemét  $i$ -be
6:   if  $i \in \text{Headers} \wedge \bigwedge_{j \in \mathcal{P}(i)} I(j, i)$  then
7:     if  $in_i = \mu(\{out_{p,i} | p \in \mathcal{P}(i)\})$  then
8:       for all  $(p, q) \in \mathcal{E}(\mathcal{L}(i))$  do
9:         if  $\text{entering}(p, q) \wedge \bigwedge_{(r,s) \in \mathcal{I}(\mathcal{L}(q))} I(r, s)$  then
10:            $\mathcal{H}(\mathcal{L}(q))$ -t betesszük  $Q$ -ba
11:         else if  $\neg \text{entering}(p, q) \wedge \bigwedge_{k \in \mathcal{P}(q)} I(k, q)$  then
12:            $q$ -t betesszük  $Q$ -ba
13:         end if
14:       end for
15:       continue
16:     else
17:       for all  $(p, q) \in \mathcal{I}(\mathcal{L}(i))$  do
18:          $I(p, q) := \downarrow$ 
19:       end for
20:     end if
21:   end if
22:    $in_i := \mu(\{out_{p,i} | p \in \mathcal{P}(i)\})$ 
23:   for all  $j \in \mathcal{S}(i)$  do
24:      $out_{i,j} := \tau(in_i, i)(j)$ 
25:      $I(i, j) := \uparrow$ 
26:     if  $\neg \text{exiting}(i, j)$  then
27:       if  $\text{entering}(i, j) \wedge \bigwedge_{(r,s) \in \mathcal{I}(\mathcal{L}(j))} I(r, s)$  then
28:          $\mathcal{H}(\mathcal{L}(j))$ -t betesszük  $Q$ -ba
29:       else if  $\neg \text{entering}(i, j) \wedge \bigwedge_{k \in \mathcal{P}(j)} I(k, j)$  then
30:          $j$ -t betesszük  $Q$ -ba
31:       end if
32:     end if
33:   end for
34: end while
```

---

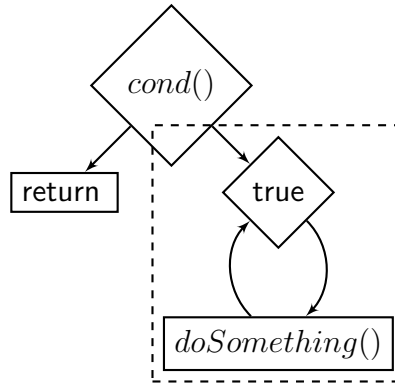
- Cikluson belüli utasítások esetén a külső utasításokhoz nagyon hasonlóan járunk el. Fontos különbség azonban, hogy a ciklus belsejében induló élek mentén megjelenő információk nem feltétlenül vannak fixpontban, ezért a kimenő élek mentén nem ellenőrizzük az utasítások bejárhatóvá válását. Ezt az algoritmus 26. sorában lévő feltétel biztosítja.
- Amikor egy ciklus fejénél az összes visszaélen rendelkezésre áll az információ, akkor a fejet újra betesszük a listába.
- Amikor egy ciklus  $h$  fejét nem először látogatjuk meg, vagyis már a befutó visszaélek mentén is elérhető információ, akkor ellenőrizzük, hogy  $in_h$  megegyezik-e a befutó információk aktuális értékeinek  $\mu$  összefésülő függvény által szolgáltatott eredményével. Ehhez a feltételhez tartozik az algoritmus 7-20. sorai között található elágazás.
  - Ha  $in_h$  értéke stimmel, akkor a ciklusmag egyenletei fixpontban vannak. Ekkor az egy szintnyit kilépő éleken<sup>7</sup> elérhető utasítások bejárhatóságát ellenőrizzük, és a bejárható utasításokat betesszük a listába. A kilépő élek feldolgozása után a lista következő elemét vesszük, és azzal folytatjuk a feldolgozást.
  - Ha  $in_h$  értékét frissíteni kell, akkor az egész ciklust újra be kell járni, ezért a ciklusmagon belül induló minden élen töröljük az információ elérhetőségét jelző értéket. Ebben az esetben a ciklus fejéhez tartozó egyenletek újbóli feldolgozásával folytatódik az algoritmus.

3.2.3. *megjegyzés* (A bejárás kezdőcsúcsai). Az **Entries** halmaz definíciójának első fele ( $\{i | i \in \text{Inst} \wedge \mathcal{P}(i) = \emptyset\}$ ) azokat a csúcsokat választja ki, melyeknek a bejárás iránya szerint nincs megelőzőjük, a második fele ( $\{\mathcal{H}(l) | l \in \text{Loops} \wedge \mathcal{I}(l) = \emptyset\}$ ) pedig azon ciklusok fejét, melyekhez a bejárás iránya szerint egyetlen belépő él sem tartozik.

Mivel az elemzendő programoknak nincsenek nem elérhető utasításai, a definíció előre haladó analízisek esetén ekvivalens az  $\text{Entries}_{\text{Forward}} = \{e\}$  halmazzal, ahol  $e$  a program belépési pontja.

<sup>7</sup> Beágyazott ciklusok esetén fontos, hogy csak az eggyel kintebb lévő szintre irányuló információkat tekintsük fixpontban lévőnek, hiszen ha a befoglaló ciklus még nincs fixpontban, akkor a beágyazott ciklus információi is csak a befoglaló ciklus aktuális állapotára nézve kerültek fixpontba. Azaz a befoglaló ciklus újraszámolásakor a beágyazott ciklus egyenleteit is újra kell számolni.

Visszafelé haladó analízisek esetén azonban valóban szükséges a rákövetkezővel nem rendelkező utasítások<sup>8</sup> mellett a kilépő éllel nem rendelkező<sup>9</sup> ciklusok listába helyezése az algoritmus elején, mivel bejövő információ hiányában az algoritmus során nem kerülnének a listába. Erre példa a 3.2. ábrán látható program, melyben a `return`-t tartalmazó utasítás mellett a szaggatott vonallal jelölt ciklus fejét is a listába kell helyezni, hogy az algoritmus aztán el tudja végezni az elemzést.



3.2. ábra. Kilépő él nélküli ciklus

3.2.4. *megjegyzés* (Belépés a ciklusokba). A 3.2.1. algoritmus egy ciklus bejárhatóvá válása esetén a ciklus fejét behelyezi a listába. Ez a működés magától értetődik előre haladó elemzés esetén. Visszafelé haladó analíziseknél felvetődhet a kérdés, hogy van-e jobb ciklusbeli utasítás, amelynél elkezdhetnénk a bejárást. Az világos, hogy bármely ciklusbeli utasítástól indíthatjuk a ciklus elemzését, mivel úgyis körbeér az információ. Ugyanakkor a ciklusbeli egyenletek fixpontjának ellenőrzését egy olyan utasításnál kell elvégezni, melyet akkor érünk el, amikor egész biztosan a ciklus minden utasításának információit frissítettük. Erre a szerepre a ciklus feje kínálja magát, mivel erről további vizsgálatok nélkül tudható, hogy megfelel a feltételnek. Látszólag a bejárást máshol is kezdhethetnénk, mint a fixpont ellenőrzésének helye. Azonban ha nem az ellenőrzési ponttól indítjuk a bejárást, az első menetben az algoritmus nem érinti a ciklus minden utasítását az ellenőrzés előtt. Így rossz esetben az első, részleges menet után fixpontban lévőknek tűnhet az ellenőrzési ponthoz tartozó be-menő információ, ami egyértelműen teljesen hibás végeredményhez vezethet. Így a bejárást az algoritmus felesleges bonyolításának elkerülése érdekében attól az utasítástól érdemes kezdeni, amelynél a fixpontot ellenőrizzük. Erre a feladatra pedig a korábbiak értelmében a ciklus feje a leghatékonyabb választás.

3.2.5. *megjegyzés* (Nem elérhető utasítások). Előre haladó analízis egyenleteinek megoldása esetén mindenképp biztosítani kell az algoritmus alkalmazása előtt, hogy

<sup>8</sup> Visszafelé haladó esetben  $\mathcal{P} = succ$ .

<sup>9</sup> Visszafelé haladó esetben  $\mathcal{I}$  a programbeli kilépő éleket rendeli egy ciklushoz.

a programban ne legyenek nem elérhető utasítások. Ellenkező esetben az elérhetetlen utasításokból származó információk hiányában az algoritmus megakadhat.<sup>10</sup> Ugyanakkor visszafelé haladó analízisek esetén a nem elérhető utasítások jelenléte nem rontja el a program elemzését, mivel az információk nem azokból jönnek, hanem azok irányába továbbítódnak.

3.2.6. *megjegyzés* (A B.2.1. és 3.2.1. definíciók előre haladó esetben). A két algoritmus között alapvetően a ciklusba történő belépés feltétele különbözik, illetve a 3.2.1. definícióban formálisan megadásra kerültek a használt fogalmak. Könnyen belátható, hogy a 3.2.1. definícióban adott algoritmus előre haladó analízis esetén ekvivalens a B.2.1. definícióban adott algoritmussal.

Ezek alapján az itt bemutatott algoritmust nevezethetjük egyszerűen a vezérlés szerinti adatáram-analízis algoritmusának.

3.2.7. *megjegyzés* (A vezérlés szerinti és a [13]-beli algoritmusok kapcsolata). A [13] 6.3. fejezetében bemutatott algoritmus előre haladó elemzés esetén a *CFG* erősen összefüggő komponenseinek fordított postorder bejárási-sorrend szerinti meglátogatását írja elő. Az általam megfogalmazott algoritmus előre haladó esetben ugyanezt a sorrendet eredményezi. A különbség a sorrend meghatározásának módjában van. Míg én a bejárás során a bemenő információk elérhetőségétől teszem függővé egy utasítás feldolgozását, addig az explicit módon fordított postorder sorrendet használó algoritmus a bejárási sorrendet egy előzetesen elvégzett postorder bejárással határozza meg.

Visszafelé haladó analízisek esetén a hivatkozott könyv nem adja meg az algoritmus pontos leírását, azonban a könyvbeli megjegyzésekből egyértelmű, hogy a postorder bejárás szerinti sorrend a hatékony megközelítés. Az algoritmusom ezt a sorrendet majdnem biztosítja, egyetlen kivétel a ciklusokba való belépés. Ez az eltérés a 3.2.4. megjegyzés értelmében azonban nem rontja el a bejárás eredményét, sőt a lépésszámmra vonatkozó állítás alapján a legrosszabb esetben is csak a ciklusmagon végzett egyetlen iterációval lehet rosszabb az elképzelhető legjobb megoldásnál.

3.2.8. *megjegyzés* (Ciklushoz tartozó egyenletek fixpontja). A fixpont biztos megtalálásához a végrehajtott adatáram-analízis meg kell feleljen bizonyos tulajdonságoknak, melyekre a B.1.1. megjegyzésben hivatkozom. Ha a szükséges feltételt egy analízis nem teljesíti, akkor egy program elemzése során egy ciklus egyenleteinek megoldása az algoritmusban végtelen ciklust eredményezhet. Mivel dolgozatomban nem az egyes analízisek vizsgálata a célom, ezért erre nem térek ki bővebben.

3.2.9. *megjegyzés* (Összefésülés visszaélek esetén). A ciklusok fixpontjának megtalálása az egyenletrendszer megoldásának időigényes része, mivel a konkrét analízistől

<sup>10</sup> Lásd a B.2.3. megjegyzést.

függ, hogy hányszor kell bejárni a ciklust. Ezt azonban nem csak a  $\tau$  átmenet-függvény képes befolyásolni, hanem az összefésülő függvény is komoly segítséget jelenthet a fixpont gyorsabb megtalálásában. Ehhez az összefésülő függvény  $\mu : \wp(\text{Info} \times \mathbb{L}) \rightarrow \text{Info}$  alakú kell legyen, ahol az összefésülendő információk mellett azt is megadjuk, hogy az egyes információk visszaélről származnak-e.<sup>11</sup> Hasznos lehet ez a kiegészítés például intervallum-elemzés során, amikor egy változó értéke a ciklusmagban kis lépésekben monoton változik. Ekkor az egyszerű összefésülés mellett a ciklusmag sokszori bejárása lehet szükséges mire a változóra vonatkozó információ fixpontba jut. Azonban a kibővített összefésülő függvény érzékelve, hogy a megváltozott értékre vonatkozó információ egy visszaélről származik, egyetlen lépésben a megfelelő szélsőértékre állíthatja a változóra vonatkozó információt.

**3.2.2. tétel** (A vezérlés szerinti adatáram-analízis helyessége és lépésszáma<sup>12</sup>). *Tet- szőleges (a B.1.1. megjegyzésben hivatkozott feltételnek megfelelő) adatáram-analízis esetén a vezérlés szerinti algoritmus fixpontba jut. Előre haladó analízis esetén ezt minimális lépésszámmal teszi, visszafelé haladó analízis esetén pedig a legrosszabb esetben is az elképzelhető legjobb algoritmushoz képest legfeljebb eggyel többször járja be a ciklusokat.*

3.2.10. *megjegyzés* (A vezérlés szerinti algoritmus költsége). Láttuk, hogy a vezérlés szerinti algoritmus az egyenletrendszer a lehető legkevesebb lépésben képes megoldani. Ugyanakkor annak eldöntése, hogy egy utasítás mikor áll készen a feldolgozásra, elég költségesen megvalósítható feladat. Így megfontolandó kérdés, hogy használjuk-e a lényegében adminisztráció nélküli eredeti listás algoritmust, ezzel megengedve a lépések számának növekedését, vagy a költséges adminisztrálás ellenére ragaszkodjunk a kevesebb lépésszámhoz. A gyakorlatban a sokkal egyszerűbb listás algoritmust használják az adatáram-analízisek megvalósítására. Azonban a szuperoptimalizációs keretrendszer megvalósításához a vezérlés szerinti algoritmus egy továbbfejlesztett változatára lesz szükségünk.

### 3.3. Két egyszerű adatáram-analízis

Az adatáram-analízisek elméleti ismertetése után mutatok egy-egy egyszerű példát előre- és visszafelé haladó adatáram-analízisre.

<sup>11</sup> A továbbiakban is az eredeti definíció szerinti összefésülő függvényt használom, de az implementáció során célszerű lehetőséget biztosítani az összefésülés során a visszaélről származó információk külön kezelésére.

<sup>12</sup> A tétel bizonyítását lásd a C. függelékben.

### 3.3.1. Konstans továbbterjesztés

A konstans továbbterjesztéshez szükséges információ meghatározása az egyik legegyszerűbb példája az előre haladó adatáram-analíziseknek. A következő definíció az 1.2.1. részben bevezetett jelöléseket követi.

**3.3.1. definíció** ( $\mathcal{DA}_{ConstProp}$ ).  $\mathcal{DA}_{ConstProp} = (\text{Forward}, \tau_{ConstProp}, \mu_{ConstProp})$ , ahol

$$\tau(info, inst) = \begin{cases} info & \text{ha } inst \text{ nem értékadás} \\ info|_{\text{Var} \setminus \{v\}} \cup \{(v, val(info, e))\} & \text{ha } inst \ v := e \text{ alakú értékadás} \end{cases}$$

Ahol  $val(info, e) = c \in \text{Value}$ , ha minden  $e$ -ben szereplő  $x$  változóra  $info(x) \in \text{Value}$ , és ezek behelyettesítésével  $e$  értéke  $c$ . Egyébként  $val(info, e) = \top$ .

A  $\mu_{ConstProp}$  összefésülő függvényre  $\forall v \in \text{Var}$  esetén

$$\mu_{ConstProp}(infos)(v) = \begin{cases} \perp & \text{ha } infos = \emptyset \\ \alpha & \text{ha } infos \neq \emptyset \wedge \forall info \in infos : info(v) = \alpha \\ \top & \text{egyébként} \end{cases}$$

3.3.1. *megjegyzés* (A kiindulási információ). A  $\mu_{ConstProp}$  definíciójának első sora értelmében a kiindulási információ a  $\text{Var} \times \{\perp\}$  függvény, mely azt jelzi, hogy egyetlen változóról sincs információnk.

### 3.3.2. Nem használt értékadások törlése

A második példa a visszafelé haladó adatáram-analízisek szűk körének tipikus képviselője, a nem használt értékadások törléséhez szükséges információt összegyűjtő adatáram-analízis<sup>13</sup>. A következő definíció az 1.2.4. részben leírtakhoz igazodik.

**3.3.2. definíció** ( $\mathcal{DA}_{DAE}$ ).  $\mathcal{DA}_{DAE} = (\text{Backward}, \tau_{DAE}, \mu_{DAE})$ , ahol  $\forall v \in \text{Var} :$

$$\tau_{DAE}(info, inst)(v) = \begin{cases} \downarrow & \text{ha } inst \text{ írja } v\text{-t, de nem olvassa} \\ info(x) \vee info(v) & \text{ha } inst \text{ írja } x\text{-et és olvassa } v\text{-t} \\ info(v) & \text{ha } inst \text{ nem hivatkozik } v\text{-re} \\ \uparrow & \text{egyébként} \end{cases}$$

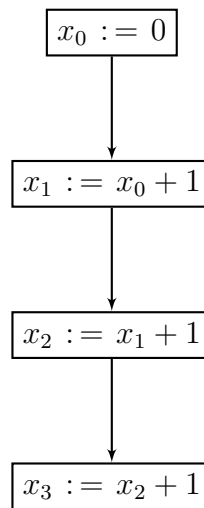
$$\forall v \in \text{Var} : \mu_{DAE}(infos)(v) = \begin{cases} \downarrow & \text{ha } \nexists info \in infos : info(v) \\ \uparrow & \text{egyébként} \end{cases}$$

3.3.2. *megjegyzés* (A kiindulási információ). A nem használt értékadások törlése esetén az információ minden változóra megmondja, hogy a következő értékadásig kiolvashatják-e az értékét. Mivel  $\mathcal{DA}_{DAE}$  visszafelé haladó analízis, így a kiindulási

<sup>13</sup> Nevezik ezt az elemzést az *élő változók analízisének* is.

információnak minden változóra hamis értéket kell adnia. A  $\mu_{DAE}$  definíciójának első sora ezt biztosítja is, hiszen a kiindulási állapot az összefésülő függvény által az üreshalmazra adott eredmény.

**3.3.3. megjegyzés** (A visszafelé haladás szükségessége). Megjegyzendő, hogy ha esetleg az értékadás helyén tudnánk is, hogy az adott változót kiolvassák-e mielőtt legközelebb értéket kap, akkor sem lehetne az optimalizációhoz szükséges információt a vezérlési folyamtnak megfelelő bejárással előállítani. Nézzük például a 3.3. ábrán látható értékadás-láncot. A program eredeti állapotában csak az utolsó értékadás tekinthető ebben az egyszerű értelemben feleslegesnek, azonban annak törlésével az előtte lévő törlendővé válik, majd az azelőtti, és így tovább. Ezt a programban előre haladva nem lehet az adatáram-analízisek keretein belül észrevenni, visszafelé haladva azonban triviálisan adódik.



3.3. ábra. Egy értékadás-lánc

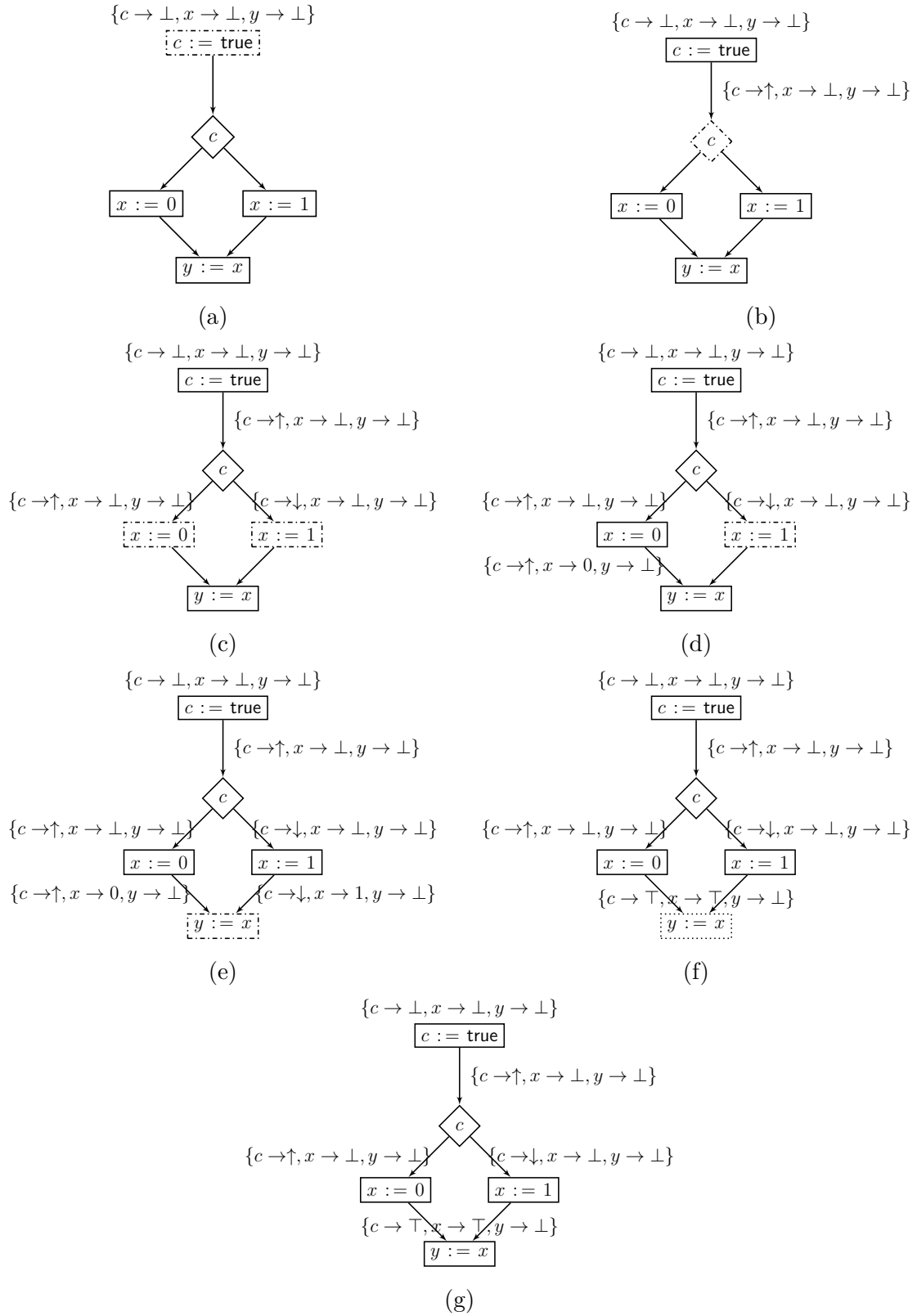
## 3.4. A vezérlés szerinti adatáram-analízis működés közben

A következőkben a 3.3.1. alfejezetben definiált  $\mathcal{DA}_{ConstProp}$  adatáram-analízis segítségével két példán keresztül bemutatom a vezérlés szerinti algoritmus működését. Az első példa a 3.4. ábrán látható, ahol az elemzendő program lényegében egyetlen elágazásból áll. Itt az algoritmus egyszerűen végighalad az utasításokon a vezérlés mentén, így kiszámítva az információkat. A második példa a 2.3b. ábrán megismert egyszerű cikluson prezentálja az algoritmus működését. Ez az elemzés már hosszadalmasabb, mivel a ciklust kétszer kell bejárni az információk fixpontjának eléréséhez. A bejárás lépéseit a 3.5. ábra tartalmazza.

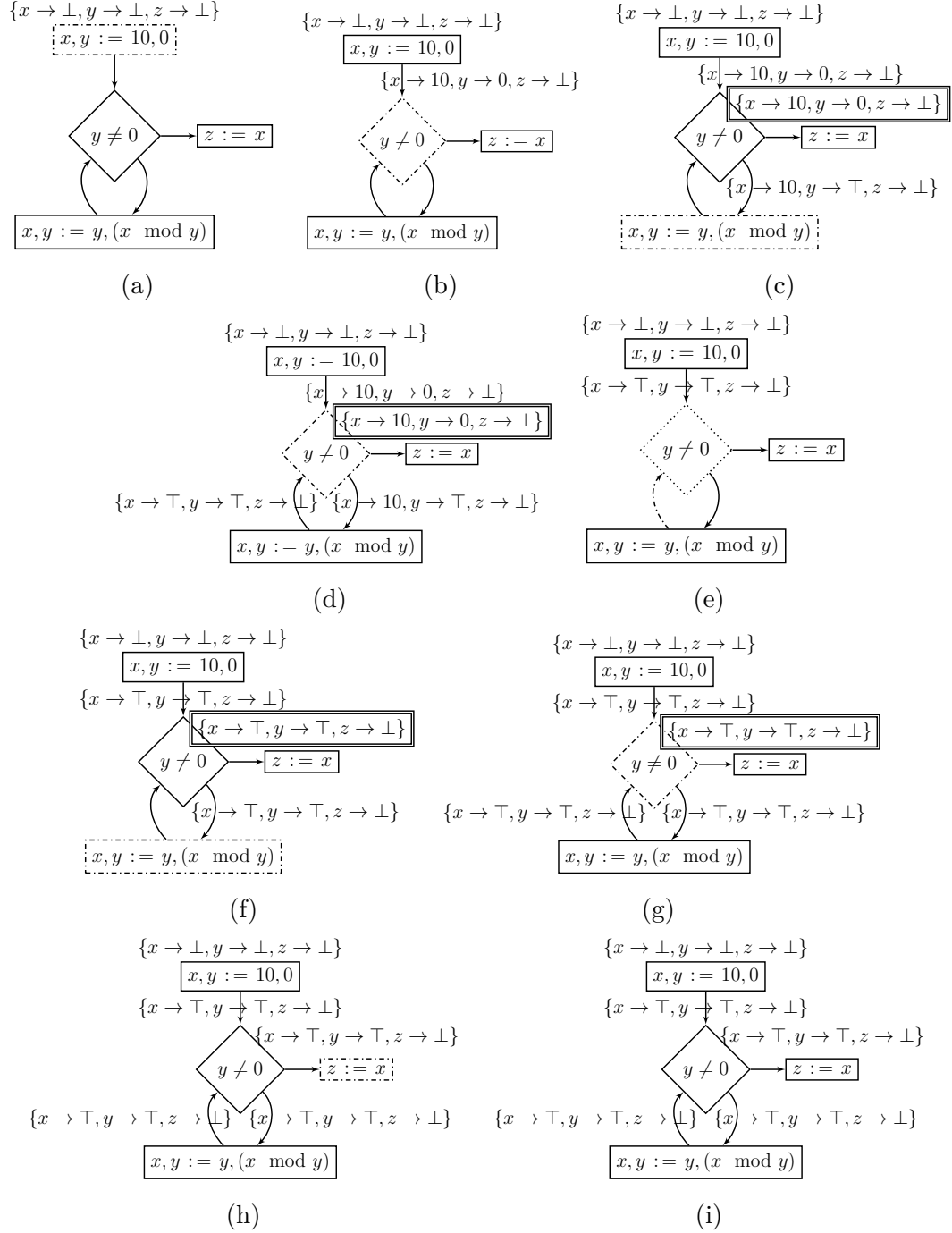
3.4.1. *megjegyzés* (Jelölések). Az algoritmus működését a programok *CFG*-jén mutatom be. A kiindulási információ a belépési pont fölött található, a még ki nem számolt információkat az ábrák nem tartalmazzák. A szaggatott-pöttyözött szegélyű csúcsok jelölik a listában lévő utasításokat. Amikor egy utasítás feldolgozásra kerül, akkor a kimenő információi megjelennek a megfelelő kimenő élei mentén, és természetesen a csúcs szegélye visszaváltozik sima vonallá.

Amennyiben egy utasításnak csak egyetlen megelőzője van, úgy a bemenő információ kiszámítását nem jelöltem külön, mert az megegyezik az egyetlen megelőzője kimenő információjával. Azonban ha egy utasítás több megelőzővel is rendelkezik, akkor a bemenő információ összefésülésének eredményét külön lépésben jelenítem meg. Ebben a lépésben az érintett csúcs szegélye pöttyözött, ezzel jelezve, hogy már nincs a listában, hanem éppen feldolgozás alatt áll. Ezzel egyidőben helytakarékosági szempontból a bejövő élek menti információkat törölöm a gráfról.

A ciklusok bejárása során a fixpont eléréséig feltartott információkat duplaszegélyű téglalapok tartalmazzák. Ha a ciklus fejének elérésekor a fixpont ellenőrzése hamis eredményt ad, akkor szaggatott-pöttyözött vonallal jelölöm azt az élt, mely mentén a fej bemenő információját frissítő információ érkezett. Ezzel egyidőben frissítem a fejhez tartozó információt is. A ciklus újbóli bejárásának kezdetén a cikluson belül induló élek mellől törlődnek az információk.



3.4. ábra. Egy egyszerű program konstans-analízise



3.5. ábra. Egy egyszerű ciklus konstans-analízise.

## 4. fejezet

# Szuperoptimalizációs keretrendszer

A 2. fejezet tanulsága szerint a kívánatos megoldás az lenne, ha a szuperoptimalizációk hatékonyságát különálló egyszerű optimalizációkból tudnánk kihozni. Tehát egy olyan szuperoptimalizációs keretrendszerre van szükségünk, amely lényegi működése könnyen karbantartható, opcionális elemekből áll össze, és az egyszerű optimalizációkba programozott tudást automatikusan képes kombinálni.

Egy ilyen keretrendszert definiáltak a *Diesel* nyelvhez készített *Vortex* fordítóprogram ([4]) megalkotói, mely úgynevezett integrált analízisek (*Integrated Analysis*) segítségével képes egy programot olyan módon optimalizálni, hogy a végeredmény az egyes integrált analízisekben megadott egyszerű optimalizációkból összeállított szuperoptimalizációénak felel meg.

A keretrendszer *Vortex* fordítóbeli megvalósításáról szól az [1] beszámoló, mely minden formalizmust mellőz. Az integrált analízisek formális leírása a [8] cikkben található. Ez a leírás háló-elméleti fogalmakkal definiálja az integrált analízisek és a belőlük összeállítható kombinált analízis működését, valamint vizsgálja a helyesség és terminálás kérdését. Azonban implementációs szempontból ez a leírás legfeljebb a szuperoptimalizációs keretrendszer ötletét, a megvalósításához szükséges elveket ismerteti.

Ebben a fejezetben a [8] cikkben található szuperoptimalizációs keretrendszer egy ponton kibővített változatának olyan formális leírását adom, mely algoritmikus szinten írja le a keretrendszer működését. A kombinált analízis megoldásának algoritmusát és a hozzá vezető megállapításokat, mivel a hivatkozott cikkek csak az analízis egy utasításra vonatkozó pontos hatásáról szólnak, saját eredményeim. Az általam definiált algoritmus helyességét azonban precízen nem tudom bizonyítani, mivel magának a megoldandó feladatnak a megfogalmazása is komolyabb elméle-

ti kutatást igényelne. Azt azonban megmutatom, hogy bár a megoldandó probléma sokban hasonlít az adatáram-analízisek egyenletrendszerének megoldásához, az utasítások bejárását a helyesség biztosítása miatt szigorú szabályokhoz kell kötni. Ezenkívül adok egy megoldó algoritmust, mely az intuitív módon megfogalmazott feladatot képes megoldani. Ez az algoritmus az adatáram-analízisek megoldására adott vezérlés szerinti algoritmusra épül.

A fejezet végén néhány egyszerű példán keresztül szemléltetem az általam ismertett keretrendszer működését.

## 4.1. Integrált analízisek

Ahogy a fejezet bevezetőjében említettem, a szuperoptimalizációs keretrendszer úgynevezett integrált analízisekkel képes dolgozni. Az elnevezés onnan ered, hogy az integrált analízisek az analízist és transzformációt egy egységbe integrálják. Persze ahhoz, hogy a kívánt módon több optimalizációt kombinálni tudjunk, ezt az integrációt speciális módon kell megtenni. Ehhez az adatáram-analízisekre vonatkozó definíciót változtatjuk meg a következő módon:

**4.1.1. definíció** (Integrált analízisek).  $\mathcal{IA} = (\delta, \tau, \mu)$  integrált analízis, ha  $\delta \in \{\text{Forward}, \text{Backward}\}$ ,  $\mu : \wp(\text{Info}) \rightarrow \text{Info}$  összefésülő függvény, a  $\tau = \tau_\delta$  átmenet-függvény a bejárás iránya szerint a következő:

$$\tau_{\text{Forward}} : \text{Info} \times \text{Inst} \rightarrow ((\text{Inst} \rightarrow \text{Info}) \cup \text{Graph} \cup (\text{Graph} \times (\text{Inst} \rightarrow \text{Info})))$$

$$\tau_{\text{Backward}} : \text{Info} \times \text{Inst} \rightarrow (\text{Inst} \cup \text{Graph} \cup (\text{Graph} \times \text{Info}))$$

Ahol  $\text{Graph}$  végrehajtási gráfok halmaza, oly módon, hogy a  $\tau(\text{info}, i)$  eredményében szereplő gráf csúcshalmazában  $\text{succ}(i)$  elemei is szerepelhetnek, de él nem indulhat ki belőlük. A gráf akkor lehet üres, ha  $|\text{succ}(i)| \leq 1$ .

Továbbá előre haladó esetben:

$$\tau_{\text{Forward}}(\text{info}, i) = o \in \text{Inst} \rightarrow \text{Info} \Rightarrow \mathcal{D}(o) = \text{succ}(i)$$

$$\tau_{\text{Forward}}(\text{info}, i) = (g, o) \in \text{Graph} \times (\text{Inst} \rightarrow \text{Info}) \Rightarrow \mathcal{D}(o) = \text{succ}^*(i, g)$$

Ahol  $\text{succ}^*(i, g) = \{j \mid j \in \text{succ}(i) \wedge \exists k \in \text{Inst}' : (k, j) \in \text{Edges}'\}$ , ha  $g = (\text{Inst}', \text{Edges}')$ .

**4.1.1. megjegyzés** (Integrált analízisek és adatáram-analízisek). A 4.1.1. definíció alapján az adatáram-analízisek az integrált analízisek speciális esetét képezik, mivel azok átmenet-függvényei minden esetben információt határoznak meg, míg az integrált analízisek esetén az átmenet-függvény eredményként végrehajtási gráfot is adhat. Megjegyzendő, hogy az adatáram-analízisekre vonatkozó 3.2.9. megjegyzés az integrált analízisekre is érvényes, azaz érdemes az implementációban lehetőséget biztosítani az összefuttatás során a visszaéleken érkező információk külön kezelésére.

4.1.2. *megjegyzés* (Gráfok, mint eredmények). Az integrált analízisek definíciója szerint az átmenet-függvények egy utasítás adott információ melletti elemzése során végrehajtási gráfot is eredményezhetnek. Ekkor az analízis az utasítás adott gráfra történő cseréjét javasolja. Éppen azért megengedettek az elemzett utasítás rákövetkezői a gráf csúcshalmazában, mert így lehetséges a cserét megfelelően elvégezni.<sup>1</sup> Amely rákövetkezőbe az eredmény-gráfban nem fut be él, abba a csere után sem fogunk tudni eljutni, így az átmenet-függvény nem is adhat felé továbbítandó információt. Visszafelé haladó analízis esetén a helyettesítési gráf belépési pontja egy speciális utasítás kell legyen, melynek egyetlen rákövetkezője van, és a helyettesítés végrehajtásakor nem kerül be az eredeti gráfba. Ezen speciális belépési pont feladata pusztán annyi, hogy bemenő információjaként elérhetővé tegye a rákövetkező utasításának kimenő információját, amit továbbítani kell az eredeti gráfban.<sup>2</sup> Az üres gráf az utasítás törlését jelenti, amit csak akkor tehetünk meg, ha az utasításnak legfeljebb egy rákövetkezője van. Ugyanis több rákövetkező esetén az utasítás megelőzőiből jövő éleket nem tudnánk megfelelően átirányítani a törlés során.

4.1.3. *megjegyzés* (Integrált analízisek és adatáram-egyenletek helyettesítéssel). Az integrált analízisekhez felírhatók a megfelelő előre- vagy visszafelé haladó adatáram-egyenletek, melyek által meghatározott egyenletrendszer megoldását kell keresni. Azonban az átmenet-függvények eredményében Graph típusú érték is szerepelhet. Ez azt jelenti, hogy az adott információ mellett az utasítás lecserélését javasolja az átmenet-függvény. Egyszerű lenne a helyzet, ha csak az információk fixpontjában lévő helyettesítési javaslatokat kellene figyelembe venni<sup>3</sup>, mivel ekkor az adatáram-egyenletek megoldása nem változna. Ugyanakkor a csere következtében megszűnő CFG-beli kapcsolatok miatt az egyenletrendszer is változik, tehát az információk meghatározása során figyelembe kell venni a cserék miatti változásokat is. Mivel az integrált analíziseket nem önmagukban, hanem a kombinált analízisen belül használjuk, ezt a kérdéskört a kombinált analízis megoldásával kapcsolatban részletezem.

4.1.4. *megjegyzés* (Az eredeti keretrendszer integrált analízisei). A [8] cikkben definiált integrált analízisek vagy információt, vagy végrehajtási gráfot adhatnak eredményül, egyszerre a kettőt nem. A keretrendszer implementálása során azonban a harmadik lehetőség is hasznosnak bizonyult. Felhasználásával hatékonyabbá tehető

<sup>1</sup> A rákövetkezőkből az eredmény-gráfban azért nem indulhat él, mert azokat az utasításokat nem módosíthatjuk.

<sup>2</sup> A speciális belépési pontot a keretrendszer is beszúrhatja a helyettesítő gráf eredeti belépési pontja elé, vagy az eredeti belépési pont kimenő információját külön tárolhatja, mely azonban a dolgozatban ismertetett algoritmus némi módosítását teszi szükségessé.

<sup>3</sup> Ténylegesen csak az ilyen helyettesítéseket lehet elvégezni, mivel a nem fixpontbeli információ alapján hozott helyettesítési döntés hibás lehet.

például az *Inliner* analízis végrehajtása, amely így egy függvény beágyazása mellett azt az információt is visszaadhatja, hogy az adott függvényt célszerű beágyazni. Így a későbbiekben nem kell a kérdéses függvényt újra költséges vizsgálatnak alávetni annak eldöntéséhez, hogy beágyazzuk-e vagy sem.

4.1.5. *megjegyzés* (A használható optimalizációk köre). Fontos látni, hogy nem minden optimalizáció valósítható meg integrált analízis formájában. Csak olyan lokális transzformáció építhető be egy integrált analízisbe, mely adatáram-analízis által szolgáltatott információk alapján hoz transzformációs döntéseket. Lokális transzformáció alatt pedig azt értem, hogy egy adott utasításra vonatkozó információ alapján csak azt az egy utasítást lehet valami másra cserélni.<sup>4</sup>

## 4.2. A kombinált analízis

A szuperoptimalizációk hatásának eléréséhez az egyes integrált analízisek működését kombinálni kell. Erre szolgál a következőkben definiálandó kombinált analízis, amely működését opcionális integrált analízisek határozzák meg. A kombinált analízis jellemzője, hogy az integrált analízisek által meghatározott helyettesítéseket is optimalizálja, ezért definíciójához fel kell használni magának a kombinált analízisnek a megoldását is.<sup>5</sup>

**4.2.1. definíció** (Kombinált analízis megoldása).  $\text{Process} : \text{Combined} \times \text{Graph} \rightarrow (\text{Graph} \times (\text{Inst} \rightarrow \text{Info}))$ , ahol  $\text{Combined}$  a kombinált analízisek halmaza, a kombinált analíziseket megoldó függvény.  $\text{Process}(\mathcal{CA}, g)$  eredménye a  $g$  gráffal adott program  $\mathcal{CA}$  kombinált analízis által transzformált változata, valamint az eredeti program változatlanul hagyott utasításaihoz tartozó fixpontbeli információkat megoldó függvény.<sup>6</sup>

**4.2.2. definíció** (Kombinált analízis).  $\mathcal{CA} = (\delta, \tau, \mu)$  az  $\mathcal{IA}_l = (\delta_l, \tau_l, \mu_l)$  ( $l = 1, 2, \dots, n$ ) integrált analízisek által meghatározott kombinált analízis, ha  $\forall p, q \in \{1, 2, \dots, n\} : \delta_p = \delta_q$ . Ekkor  $\mathcal{CA}$  egy speciális integrált analízis a következők szerint:

Az analízis iránya  $\delta = \delta_1$ .

Az összefésülő függvény  $\mu : \wp(\text{Info}) \rightarrow \text{Info}$ , ahol  $\text{Info} = \times_{k=1}^n \text{Info}_k$  és  $\text{Info}_k$  az  $\mathcal{IA}_k$

<sup>4</sup> Még csak két utasítás felcserélését sem lehet megoldani integrált analízis felhasználásával.

<sup>5</sup> A helyettesítések optimalizálása a megoldó algoritmus rekurzivitását eredményezi. Ezért a dolgozatomban vázolt iteratív optimalizálási technikával szemben ezt a módszert rekurzív optimalizálásnak is nevezhetjük.

<sup>6</sup> A  $\text{Process}$  függvény megvalósításához szükséges algoritmust a 4.3. fejezetben ismertetem.

analízis által használt információ<sup>7</sup>, valamint  $\mu$  olyan, hogy

$$\mu(\text{infos})_k = \mu_k(\{\text{info}_k | \text{info} \in \text{infos}\})$$

Az átmenetfüggvény  $\delta = \text{Forward}$  esetén  $\tau : \text{Info} \times \text{Inst} \rightarrow ((\text{Graph} \cup \{\perp_R\}) \times (\text{Inst} \rightarrow \text{Info}))$ , hogy

$$\tau(\text{info}, i) = \begin{cases} (\perp_R, \{(j, \text{info}^{(j)}) | j \in \text{succ}(i)\}) & \text{ha } \bigwedge_{k=1}^n (\tau_k(\text{info}_k, i) \in \text{Inst} \rightarrow \text{Info}_k) \\ (g', \{(j, o'(j)) | j \in \text{succ}^*(i, g')\}) & \text{ha } \bigvee_{k=1}^n (\tau_k(\text{info}_k, i) \notin \text{Inst} \rightarrow \text{Info}_k) \end{cases}$$

Ahol  $\perp_R$  extrémális elem jelöli, hogy nem tartozik helyettesítés az adott utasításhoz,  $\text{info}^{(j)} = \text{r}_{k=1}^n \tau_k(\text{info}_k, i)(j)$ ,  $g' = \text{Solve}_G(\text{info}, i) \in \text{Graph}$  és  $o' = \text{Solve}_I^{\text{Forward}}(\text{info}, i) \in (\text{Inst} \rightarrow \text{Info})$ , valamint  $\text{r}_{k=1}^n x_k$  n-est készít az elemekből. Legyen

$$G(\text{info}, i) = \{g \in \text{Graph} | \exists k \in \{1, 2, \dots, n\} : (\tau_k(\text{info}_k, i) = g \vee \tau_k(\text{info}_k, i) = (g, o))\}$$

Továbbá  $\text{Solve}(\text{info}, i) = (G', I') \in \wp(\text{Graph}) \times \wp(\text{Inst} \rightarrow \text{Info})$ , ahol

$$\begin{aligned} G' &= \{g' \in \text{Graph} | \text{Process}(\mathcal{CA}', g)_1 = g' \wedge g \in G\} \\ I' &= \{o \in (\text{Inst} \rightarrow \text{Info}) | \text{Process}(\mathcal{CA}', g)_2 = o \wedge g \in G\} \end{aligned}$$

$\mathcal{CA}' = (\delta, \tau, \mu')$  és

$$\mu'(\text{infos}) = \begin{cases} \text{info} & \text{ha } \text{infos} = \emptyset \\ \mu(\text{infos}) & \text{egyébként} \end{cases}$$

Ekkor  $\text{Solve}_G(\text{info}, i) = \text{Select}(\text{Solve}(\text{info}, i)_1)$ , ahol  $\text{Select} : (\wp(\text{Graph}) \setminus \{\emptyset\}) \rightarrow \text{Graph}$  a paramétere egy elemét választja ki.

$$\begin{aligned} \text{Solve}_I^{\text{Forward}}(\text{info}, i) &= \{(j, \mu(\{o(j) \in \text{Info} | o \in I' \wedge j \in \mathcal{D}(o)\}) \cup \text{Info}^{\text{Forward}}(\text{info}, i)) | j \in \text{succ}(i)\} \end{aligned}$$

Ahol  $\text{Info}^{\text{Forward}}(\text{info}, i) : \text{succ}(i) \rightarrow \text{Info}$ , hogy

$$\begin{aligned} \forall j \in \text{succ}(i) : \forall k \in \{1, 2, \dots, n\} : \text{Info}^{\text{Forward}}(\text{info}, i)(j)_k &= \\ &= \mu_k(\{o(j) \in \text{Info}_k | (\tau_k(\text{info}_k, i) = o \in (\text{Inst} \rightarrow \text{Info}_k) \vee \\ &\quad \vee \tau_k(\text{info}_k, i) = (g, o)) \wedge j \in \mathcal{D}(o)\}) \end{aligned}$$

Az átmenetfüggvény  $\delta = \text{Backward}$  esetén  $\tau : \text{Info} \times \text{Inst} \rightarrow ((\text{Graph} \cup \{\perp_R\}) \times \text{Info})$ , hogy

$$\tau(\text{info}, i) = \begin{cases} (\perp_R, \text{r}_{k=1}^n \tau_k(\text{info}_k, i)) & \text{ha } \bigwedge_{k=1}^n (\tau_k(\text{info}_k, i) \in \text{Info}_k) \\ (g', \text{info}') & \text{ha } \bigvee_{k=1}^n (\tau_k(\text{info}_k, i) \notin \text{Info}_k) \end{cases}$$

<sup>7</sup>  $\text{info} \in \text{Info}$  esetén  $\text{info}_k \in \text{Info}_k$  a direkt szorzat  $k$ . komponensét jelöli.

Ahol  $g' = \text{Solve}_G(\text{info}, i) \in \text{Graph}$  az előre haladó definíciónak megfelelően, és  $\text{info}' = \text{Solve}_I^{\text{Backward}}(\text{info}, i) \in \text{Info}$ . A korábbi definíciók felhasználásával:

$$\begin{aligned} \text{Solve}_I^{\text{Backward}}(\text{info}, i) = & \mu(\{o(e) \in \text{Info} \mid o \in I' \wedge \\ & \wedge e \text{ az } o\text{-hoz tartozó gráf belépési pontja}\} \cup \text{Info}^{\text{Backward}}(\text{info}, i)) \end{aligned}$$

Ahol  $\text{Info}^{\text{Backward}}(\text{info}, i) \in \text{Info}$ , hogy

$$\begin{aligned} \forall k \in \{1, 2, \dots, n\} : \text{Info}^{\text{Backward}}(\text{info}, i)_k = \\ = \mu_k(\{\text{info}'' \in \text{Info}_k \mid (\tau_k(\text{info}_k, i) = \text{info}'' \vee \tau_k(\text{info}_k, i) = (g, \text{info}''))\}) \end{aligned}$$

4.2.1. *megjegyzés* (A definíció jelentése). A kombinált analízis definíciója terjedelme ellenére pusztán a helyettesítések rekurzív optimalizálásának elvét valósítja meg. Ha egy utasítást egyetlen integrált analízis sem akar helyettesíteni, akkor egyszerűen a továbbítandó információk összefésülésével kapott információt eredményezi. Azonban ha van legalább egy helyettesítés, akkor a helyettesítéseket optimalizálja úgy, hogy az optimalizálás során kiindulási információként az aktuális bemenő információt használja.<sup>8</sup> A helyettesítő-gráfok optimalizálása után az eredeti gráfban továbbítandó információt pedig az összes helyettesítés optimalizálása és az eredeti utasítás optimalizálása során keletkezett adott utasítás felé továbbítandó információk összefuttatásával határozza meg.

Az előre- és visszafelé haladó kombinált analízisek átmenet-függvényei eredményének információt tartalmazó részei közti különbség megegyezik az adatáram-analíziseknél megismertekkel.<sup>9</sup>

4.2.2. *megjegyzés* (Kommunikáció az integrált analízisek között). A helyettesítések rekurzív optimalizálása biztosítja az integrált analízisek egyszerre működését<sup>10</sup>, mivel lehetőséget teremt arra, hogy az integrált analízisek a programon ténylegesen elvégzett módosítások nélkül új információt hozzanak egymás tudtára.

4.2.3. *megjegyzés* (Helyettesítés kiválasztása). A definícióban szereplő **Select** függvény felelős az alternatív helyettesítések közül a legjobb kiválasztásáért<sup>11</sup>, amellyel az utasítás ténylegesen helyettesítve lesz. Egy *CFG jóságának* meghatározása az optimalizálás céljától, környezetétől függ. Más eredmény lehet optimális egy beágyazott rendszerben és egy sokmagos környezetben. Így a **Select** függvény pontos meghatározása cél-függő metrika alapján kell történjen. A különböző program-metrikák vizsgálata azonban túlmutat dolgozatom keretein.

<sup>8</sup> A megfelelő kiindulási információt biztosítja  $\mu'$  definíciója.

<sup>9</sup> Lásd a 3.1.3. és 3.1.6. megjegyzéseket.

<sup>10</sup> Lásd a 2.3.1. definíciót.

<sup>11</sup> A **Select** függvény a rekurzívan optimalizált helyettesítő-gráfok közül választ.

4.2.4. *megjegyzés* (A kombinált analízis helyessége és pontossága). Amennyiben az egyes integrált analízisek helyesek, tehát transzformációik megőrzik a program jelentését, úgy az általuk meghatározott kombinált analízis is helyes. Ez a definíció alapján elég nyilvánvalónak tűnik, azonban formális bizonyítása, mely megtalálható a [7] beszámolóban, eléggé hosszadalmas. A kombinált analízis által meghatározott információk pontosságára vonatkozó állítás és bizonyítása szintén a [7] beszámolóban található, mely szerint a kombinált analízis minden esetben legalább annyira pontos információk alapján hoz döntést, mint az azt meghatározó különálló optimalizációk bármely iteratív végrehajtása. Az analízisek által meghatározott információ pontosságának precíz definiálása azonban kívül esik dolgozatom keretein, mivel az általam nem ismertett háló-elméleti definíción alapszik.

4.2.5. *megjegyzés* (A kombinált analízis végrehajtásának terminálása). A kombinált analízis által meghatározott információ dinamikusan változó adatáram-egyenletek<sup>12</sup> megoldásaként áll elő, melyeket a kombinált analízist alkotó integrált analízisek határoznak meg. Így alapvetően a B.1.1. megjegyzésben foglaltak itt is érvényesek az információk fixpontjának meghatározása szempontjából. Ugyanakkor az egyes analízisek jó működése mellett a helyettesítések rekurzív optimalizációja miatt biztosítani kell a megfelelő együttműködésüket is. Például ha adott információ mellett  $\mathcal{IA}_1$  az  $\alpha$  utasítást  $\beta$ -re cseréli, az  $\mathcal{IA}_2$  pedig a  $\beta$ -t  $\alpha$ -ra, akkor ez a kombinált analízisben végtelen rekurziót eredményez. Ennek elkerülésére biztosítani kell, hogy a kombinált analízisben felhasznált integrált analízisek nem okoznak ehhez hasonló végtelen rekurzív helyettesítéseket. Ez egyrészt megtehető a felhasznált analízisekre vonatkozó formális bizonyításokkal, másrészt bizonyos, a [8] cikkben megtalálható, háló-elméleti feltételeknek megfelelő analízisek esetén biztosan nem fordulhat elő a végtelen rekurzió. Formális bizonyítások hiányában az implementációban egy rekurziós limit bevezetése is biztosíthatja a terminálást, bár ennek helyes megválasztása nem feltétlen triviális feladat.

4.2.6. *megjegyzés* (Előre- és visszafelé haladó kombinált analízisek közti függések). A különböző integrált analízisekből összeállított kombinált analízis az optimalizálandó függvény egyszeri bejárásával képes az integrált analízisekből kihozható legjobb eredményt elérni. Ugyanakkor egy kombinált analízisben csak azonos irányba haladó integrált analízisek foghatók össze. Így az előre- és visszafelé haladó integrált analízisek közti kölcsönös függőségek esetén szükséges az előre- és visszafelé haladó kombinált analízisek iterálása, ahogy azt a 2.2.5. következmény mutatja.<sup>13</sup>

<sup>12</sup> Lásd a 4.3. fejezet elején.

<sup>13</sup> A kombinált analízisek közti függések az integrált analízisek által megvalósított optimalizációk közti függésekből a 2.4. fejezetben a szuperoptimalizációkkal kapcsolatban elmondot-

## 4.3. A kombinált analízis megoldása

A fentiek értelmében a kombinált analízis magában foglalja az integrált analíziseket, így az optimalizáláshoz megoldandó feladat a kombinált analízis adatáram-egyenleteinek megoldása. Azonban a kombinált analízis esetén az adatáram-egyenletek mellett megjelennek az egyes utasításokhoz tartozó helyettesítések is. Gondolhatnánk, hogy ezeket a helyettesítéseket figyelmen kívül hagyhatjuk az információk fixpontjának eléréséig, hiszen csak fixpontbeli információ alapján hozott transzformációs döntések lesznek biztosan helyesek. Azonban ezt az elhanyagolást két okból sem tehetjük meg:

- Amikor az eredmény helyettesítő-gráfot is tartalmaz, akkor csak a helyettesítés után is megmaradó rákövetkezők felé továbbítandó információkat adja meg. Így a többi rákövetkező felé nem tudnánk, milyen információt továbbítsunk.
- A helyettesítések következtében megszűnő rákövetkezési kapcsolatok a program további részeire vonatkozó információt is jelentenek, amire mutatok egy példát a 4.3.1. megjegyzésben. Emiatt a lehetségesen megszűnő kapcsolatoknak megfelelően dinamikusán módosítani kell az adatáram-egyenleteket.

Emellett gyakorlati szempontból célszerű lenne az egyes utasításokhoz tartozó egyenleteket a lehető legkevesebbszer frissíteni a fixpont eléréséhez, mivel a kombinált analízis egy utasításra vonatkozó eredményének meghatározásához több helyettesítő-gráf rekurzív optimalizálására is szükség lehet.

4.3.1. *megjegyzés* (Lehetséges helyettesítés távoli hatása). Emlékezzünk vissza az iteratív optimalizálás eredményének korlátait demonstráló, a 2.3a. ábrán látható példára. Tegyük fel, hogy a kombinált analízis az egyenletek megoldásának egy pontján eljut arra a következtetésre, hogy a ciklusban lévő elágazás felesleges, és csak a *skip* utasítást tartalmazó ágat kell megtartani. Ennek jelzésére a feltételes ugrás feltétel nélkülire történő cseréjét javasolja, melyet a fixpont eléréséig ténylegesen nem hajthatunk végre. Azonban ha a másik ágból induló visszaélen ezután információ juthat el a ciklus fejéhez, akkor a hozzá tartozó információ  $x$  változóra vonatkozó részének frissülése miatt a ciklus egyenletei is frissítésre fognak kerülni, és ekkor már a kombinált analízis nem tudja a feltételes ugrás cseréjét javasolni.

4.3.2. *következmény* (Virtuálisan elérhetetlenné váló utasítások figyelmen kívül hagyása). Egy utasítást virtuálisan elérhetetlenné nevezek, ha a kombinált analízis által adott aktuális helyettesítések elvégzése után elérhetetlen lenne. Az előző példából

taknak megfelelően adódnak. Az általam megismert optimalizációk esetén csak a visszafelé haladó kombinált analízis függött az előre haladótól, kölcsönös függés nem alakult ki.

látszik, hogy az ilyen utasításokat, amíg az elérhetetlenségüket okozó helyettesítések tévesnek nem bizonyulnak, az egyenletrendszerben figyelmen kívül kell hagyni.

Ugyanakkor az utasítások illetén figyelmen kívül hagyása az egyenletek frissítésének nem minden sorrendjében vezet helyes eredményre.

**4.3.1. tétel** (Helyes kombinált analízis egyenletei megoldásának sorrendje). *Előre haladó analízis esetén a ciklusban lévő utasítások egyenleteinek – a virtuálisan elérhetetlen utasításokat is figyelembe vevő – megoldásának helyes eredményéhez az utasításokat a vezérlésnek megfelelő sorrendben kell meglátogatni, valamint a ciklusba való belépésnél a visszaéleken lévő információkat figyelmen kívül kell hagyni. Visszafelé haladó analízisek esetén a sorrend nem lényeges.*

*Bizonyítás.* Az állítás két részből áll. Először az előre haladó analízisre vonatkozó részt bizonyítom, majd megmutatom, hogy az arra vonatkozó okfejtés nem érvényes a visszafelé haladó analízisre.

Első lépésként a vezérlés szerinti sorrend fontosságát mutatom meg. Ehhez tegyük fel, hogy egy ciklusban lévő utasítások egyenletei közül a ciklus belsejében lévő utasítást vizsgáljuk elsőnek. Ekkor a bemenő élein lévő lényegében inicializálatlan információk alapján határozzuk meg a bemenő információját. Ennek alapján elképzelhető, hogy a kombinált analízis olyan *téves* helyettesítési javaslatot tesz, melynek hatására bizonyos rákövetkezők virtuálisan elérhetetlenekké válnak. Ennek két következménye van: Egyrészt a ciklusmag kiesett részéből származó információk nem jutnak el a ciklus fejéhez, és nem játszanak szerepet az információ fixpontjának kialakításában. Másrészt olyan részek maradhatnak elérhetőek, melyek más információ alapján hozott helyettesítésnél elérhetlenné válnának. Mivel azonban feltevésünk szerint ez a helyettesítés téves volt, a végső eredmény is pontatlan lesz.<sup>14</sup>

A fentiek alapján a ciklus utasításait a fejtől indulva a vezérlés mentén kell meglátogatnunk úgy, hogy a fejből továbbított információ olyan legyen, mely a program futása során ténylegesen előfordulhat. Ezért a ciklusmag első vizsgálatakor, ami a program végrehajtása során a ciklusba való belépésnek felel meg, a visszaéleken lévő inicializálatlan információkat nem szabad figyelembe venni.<sup>15</sup> Ha a visszaélek inicializálatlan információit figyelembe vesszük, akkor a téves információ miatt hozott rossz helyettesítési döntések a fenti okfejtésnek megfelelően pontatlanná tehetik a végeredményt.

<sup>14</sup> A 4.3.1. megjegyzés példájában ha valamiért a ciklusban lévő elágazás értékadást tartalmazó ágát tartaná meg a kombinált analízis, akkor bár visszakérülne a másik ág is a programba, de az értékadást tartalmazó nem optimalizálódna ki.

<sup>15</sup> Bizonyos értelemben a ciklusok bejárása során a program absztrakt interpretációját kell megvalósítani.

Visszafelé haladó analízis esetén az utasítások bejárásának sorrendje nem számít, mivel egy visszafelé haladó analízis nem javasolhat olyan helyettesítést, mely miatt egy utasítás elérhetetlenné válna. Ugyanis az utasítások elérhetetlenné válását valamilyen elágazás olyan jellegű módosítása okozza, mely egyes rákövetkezési kapcsolatokat eldob. A visszafelé haladó analízisek azonban nem szerezhethetnek az elágazás egyes ágaiból olyan információt, mely alapján eldönthetnék, hogy magában az elágazásban milyen döntés kell szülessen a program végrehajtása során. Így az elágazást okozó utasításokat érdemben nem is módosíthatják.  $\square$

4.3.3. *megjegyzés* (Kívülről jövő információ frissülése). A ciklusba kívülről jövő információ frissülése a ciklus egyenleteinek újbóli megoldását vonja maga után. Az újbóli bejárás előtt fontos a korábbi bejárás helyettesítési döntéseinek törlése, mivel a korábbi bejárás információi pontatlanok voltak, így az azok alapján virtuálisan elérhetetlenné tett utasítások valójában szükségesek lehetnek.

4.3.4. *megjegyzés* (Sorrend ciklusokon kívül). A ciklusokon kívüli utasítások bejárásának sorrendje nem befolyásolja a fixpontbeli információk minőségét, mivel a meghozott helytelen döntések az információk újraszámolásával kijavíthatók. Ennek oka, hogy a ciklusokon kívül a helytelen döntés utáni utasítások információi nincsenek hatással a döntés előtti utasítások információira.

**4.3.2. definíció** (Realizálható információ<sup>16</sup>). Adott információt realizálhatónak nevezünk, ha a program tényleges végrehajtása során a program adott pontján előfordulhat olyan állapot, melyet az információ leír.

4.3.5. *megjegyzés* (Realizálható információk és helyes fixpont). A 4.3.1. tétel bizonyításából látható, hogy egy kombinált analízis által meghatározott egyenletrendszernek több fixpontja is lehet, melyek a 4.3.4. megjegyzés alapján csak a cikluson belüli utasításokhoz tartozó információikban, illetve helyettesítéseikben különbözhetnek. A ciklusokon belül hozott téves helyettesítési döntések nem-realizálható információk miatt fordulhatnak elő. Hiszen ha az információ realizálható lenne, akkor az az alapján hozott döntés sem lehetne téves. Ugyanakkor ha az információk frissítésénél az átmenet-függvény paramétere realizálható információ, akkor az eredmény helyes kell legyen.

A realizálható információ intuitív, de formális bizonyításhoz pontatlan definíciója miatt nem tudom pontosan megfogalmazni, hogy mely fixpontot milyen módon kell keresnünk. Igazából már a helyettesítések miatt dinamikusán változó egyenletrendszer pontos definiálása is a dolgozatom keretein jóval túlmutató kérdések vizsgálatát

<sup>16</sup> Hasonlóan a [14] cikkbeli realizálható út fogalmához.

igényelné. A formális leíráshoz szükséges lenne vizsgálni a programok szemantikáját, az egyenletrendszer viselkedését valamilyen absztrakt interpretáció mellett, illetve érdemes lehet a szemantika vezérelt elemzések között az esetleges hasonlóságok után kutatni.

**4.3.3. sejtés** (A vezérlés szerinti bejárás elégségessége). *A realizálható információk intuitív definíciójából kiindulva úgy tűnik, hogy a 4.3.1. tételben megfogalmazott feltételek nem csak szükségesek, de elégségesek is egy helyes fixpont megtalálásához.*<sup>17</sup>

**4.3.6. következmény** (A vezérlés szerinti algoritmus használata). A megfogalmazott sorrendi feltételeknek az adatáram-analízisek megoldására ismertetett vezérlés szerinti algoritmus<sup>18</sup> megfelel. Ráadásul a vezérlés szerinti algoritmus minimális lépésszámmal képes az előre haladó analízis esetén a fixpont megtalálására, így a lehető legkevesebbszer kell a rekurzív optimalizációkat elvégezni. Ezenkívül visszafelé haladó esetben sem sokkal rosszabb a leghatékonyabb algoritmusnál.<sup>19</sup> Mindezek fényében a kombinált analízis megoldására a virtuálisan elérhetetlen utasítások kezelésével kibővített vezérlés szerinti algoritmust használhatjuk.

**4.3.4. definíció** (Virtuálisan elérhetetlen utasításokat kezelő vezérlés szerinti algoritmus). A 4.3.1. algoritmus, mely a vezérlés szerinti adatáram-analízis algoritmusára épül, végrehajtja egy program optimalizálását adott  $\mathcal{CA} = (\delta, \tau, \mu)$  kombinált analízis mellett. Az algoritmus jobb átláthatósága miatt bizonyos részeit kiemeltem, ezeket a részeket a 4.3.2.-4.3.7. algoritmusok tartalmazzák. Az algoritmusban a 3.2.1. definícióban használt jelölések mellett a következők fordulnak elő:

Az  $I : \text{Inst} \times \text{Inst} \rightarrow \{\downarrow, \uparrow, \perp_I\}$  változó inicializálására használt  $\text{initl}$  érték biztosítja, hogy előre haladó analízis esetén kezdetben a ciklusok visszaéleiről származó információk figyelmen kívül maradjanak. Így  $\text{initl} = \text{initl}_\delta$  és

$$\forall (i, j) \in \text{Edges} : \text{initl}_{\text{Forward}}(i, j) = \begin{cases} \perp_I & \text{ha } \exists L \in \text{Loops} : (i \in \mathcal{B}(L) \wedge j = \mathcal{H}(L)) \\ \downarrow & \text{egyébként} \end{cases},$$

valamint  $\text{initl}_{\text{Backward}} = \text{Edges} \times \{\downarrow\}$ .

Az  $I$  változóban  $\perp_I$  jelöli, hogy az adott élt nem kell figyelembe venni a célja bemenő információjának kiszámításához. Az  $R$  változóban, mely az egyes utasításokhoz tartozó helyettesítéseket tárolja,  $\perp_R$  jelöli, hogy egy adott utasításhoz nem tartozik helyettesítés.

<sup>17</sup> Az implementált megoldás futtatásából származó eredmények megerősítik ezt a sejtést.

<sup>18</sup> Lásd a 3.2.1. definíciót.

<sup>19</sup> A 3.2.2. tétel értelmében visszafelé haladó analízisek megoldásánál a legjobb algoritmusnál legfeljebb eggyel többször járja be a ciklusokat a legrosszabb esetben.

Az átmenet-függvény eredményében található információ szükséges alakra<sup>20</sup> hozzását végzi a **completeInfo** függvény:

$$\begin{aligned} \text{completeInfo} : ((\{\text{Forward}\} \times (\text{Inst} \rightarrow \text{Info})) \cup \\ \cup (\{\text{Backward}\} \times \text{Info})) \times \text{Inst} \rightarrow \\ \rightarrow (\text{Inst} \rightarrow (\text{Info} \cup \{\perp_I\})) \end{aligned}$$

$$\begin{aligned} \text{completeInfo}(\text{Forward}, o, i) = \{(j, o(j)) \mid j \in \mathcal{S}(i) \wedge j \in \mathcal{D}(o)\} \cup \\ \cup \{(j, \perp_I) \mid j \in \mathcal{S}(i) \wedge j \notin \mathcal{D}(o)\} \end{aligned}$$

és

$$\text{completeInfo}(\text{Backward}, info, i) = \{(j, info) \mid j \in \mathcal{S}(i)\}$$

4.3.7. *megjegyzés* (Az algoritmus működése). A fenti algoritmus működése alapvetően megegyezik a vezérlés szerinti adatáram-analízis 3.2.1. algoritmusáéval, azt a helyettesítések és ezáltal a virtuálisan elérhetetlen utasítások kezelésével egészíti ki. Az algoritmus az utasítások feldolgozása során a  $\perp_I$  jelzésű éleken keresztüli információkat nem veszi figyelembe. Ha a feldolgozás során egy utasítás minden megelőzője felől érkező információra  $\perp_I$  jelzés található, vagyis az utasítás virtuálisan elérhetetlen, akkor egyszerűen a  $\perp_I$  jelzést továbbítja minden rákövetkezője felé. (4-6. sorok) A ciklusok bejárásának újrakezdésénél a ciklus utasításaihoz tartozó helyettesítéseket törli, mivel az újbóli bejárás során a helyettesítéseket újra meghatározza a kombinált analízis. A 12. és 16. sorokban lévő értékadások mindegyikére szükség van, mivel az algoritmus **resetLoop** része a cikluson belüli  $\perp_I$  jelzéseket törli. A 19. sorban egyszerűen eltároljuk az utasításhoz tartozó helyettesítést, vagy a  $\perp_R$  elemet, a 20. sor pedig biztosítja, hogy az **updateOuts** részben a kimenő információkat kényelmes módon tudjuk kinyerni az *infos* változóból. A bejárás végeztével a **performReplacements** részben az utasításokhoz eltárolt helyettesítések elvégzése történik.

4.3.5. **tétel** (A kombinált analízist megoldó algoritmus helyessége). *Ha a 4.3.3. sejtés igaz, akkor a 4.3.4. definícióban ismertetett algoritmus tetszőleges integrált analízisek által meghatározott előre haladó kombinált analízissel minimális lépésszám mellett, visszafelé haladó kombinált analízissel pedig a legjobb algoritmusához képest legfeljebb a ciklusok eggyel többszöri bejárásával képes egy program optimalizálására.*

<sup>20</sup> A bejárás irányának megfelelő minden rákövetkezőhöz lekérdezhető a továbbítandó információ. Ha előre haladó analízis esetén egy rákövetkező felé irányuló él a helyettesítés miatt megszűnne, úgy ahhoz az utasításhoz  $\perp_I$  érték tartozik.

---

**Algorithm 4.3.1**  $\text{Process}(\mathcal{CA}, g)$ , ahol  $\mathcal{CA} = (\delta, \tau, \mu)$ 

---

```
1:  $I := \text{initl}$ ,  $R := \text{Inst} \times \perp_R$ ,  $Q$  az Entries elemeit tartalmazó sor
2: while  $Q$  nem üres do
3:   Kivesszük  $Q$  következő elemét  $i$ -be
4:   if  $\forall j \in \mathcal{P}(i) : I(i, j) = \perp_I$  then
5:     processVirtuallyDead( $i$ )
6:     continue
7:   else if  $i \in \text{Headers}$  then
8:     if  $\text{in}_i = \mu(\{\text{out}_{p,i} \mid p \in \mathcal{P}(i) \wedge I(p, i) \neq \perp_I\})$  then
9:       finalizeLoop( $\mathcal{L}(i)$ )
10:      continue
11:    else
12:       $\text{in}_i := \mu(\{\text{out}_{p,i} \mid p \in \mathcal{P}(i) \wedge I(p, i) \neq \perp_I\})$ 
13:      resetLoop( $\mathcal{L}(i)$ )
14:    end if
15:  else
16:     $\text{in}_i := \mu(\{\text{out}_{p,i} \mid p \in \mathcal{P}(i) \wedge I(p, i) \neq \perp_I\})$ 
17:  end if
18:   $(g, \text{infos}') := \tau(\text{in}_i, i)$ 
19:   $R(i) := g$ 
20:   $\text{infos} := \text{completeInfo}(\delta, \text{infos}', i)$ 
21:  updateOuts( $i, \text{infos}$ )
22: end while
23: performReplacements
```

---

---

**Algorithm 4.3.2** *processVirtuallyDead*( $i$ )

---

```
for all  $j \in \mathcal{S}(i)$  do
   $I(i, j) := \perp_I$ 
  if  $\neg \text{exiting}(i, j)$  then
    checkReadiness( $i, j$ )
  end if
end for
```

---

---

**Algorithm 4.3.3** *finalizeLoop*( $l$ )

---

```
for all  $(p, q) \in \mathcal{E}(l)$  do
  checkReadiness( $p, q$ )
end for
```

---

---

**Algorithm 4.3.4** *resetLoop*( $l$ )

---

**for all**  $(p, q) \in \mathcal{I}(l)$  **do**  
     $I(p, q) := \downarrow$   
**end for**  
**for all**  $j \in \mathcal{B}(l)$  **do**  
     $R(j) := \perp_R$   
**end for**

---

---

**Algorithm 4.3.5** *updateOuts*( $i, infos$ )

---

**for all**  $j \in \mathcal{S}(i)$  **do**  
    **if**  $infos(j) = \perp_I$  **then**  
         $I(i, j) := \perp_I$   
    **else**  
         $out_{i,j} := infos(j)$   
         $I(i, j) := \uparrow$   
    **end if**  
    **if**  $\neg \text{exiting}(i, j)$  **then**  
        *checkReadiness*( $i, j$ )  
    **end if**  
**end for**

---

---

**Algorithm 4.3.6** *checkReadiness*( $u, v$ )

---

**if**  $\text{entering}(u, v) \wedge \bigwedge_{(r,s) \in \mathcal{I}(\mathcal{L}(v))} (I(r, s) \neq \perp_I \Rightarrow I(r, s) = \uparrow)$  **then**  
     $\mathcal{H}(\mathcal{L}(v))$ -t betesszük  $Q$ -ba  
**else if**  $\neg \text{entering}(u, v) \wedge \bigwedge_{r \in \mathcal{P}(v)} (I(r, v) \neq \perp_I \Rightarrow I(r, v) = \uparrow)$  **then**  
     $v$ -t betesszük  $Q$ -ba  
**end if**

---

---

**Algorithm 4.3.7** *performReplacements*

---

**for all**  $i \in \text{Inst}$ , amire  $R(i) \neq \perp_R$  **do**  
     $i$  helyettesítése  $R(i)$ -re  
**end for**

---

*Bizonyítás.* Tetszőleges kombinált analízis egy speciális integrált analízis. Így az integrált- és adatáram-analízisek kapcsolatára vonatkozó 4.1.1. megjegyzést és a kombinált analízisek esetén a helyettesítések miatt jelentkező virtuálisan elérhetetlen utasítások miatti bejárási sorrendre vonatkozó 4.3.1. tétel állítását figyelembe véve, valamint a kombinált analízist megoldó és a vezérlés szerinti algoritmus kapcsolatára alapozva, felhasználhatjuk a vezérlés szerinti algoritmus helyességére és lépésszámára vonatkozó 3.2.2. tételt. Mindezek alapján az algoritmus tetszőleges programot helyesen optimalizál tetszőleges kombinált analízis felhasználásával, továbbá ezt előre haladó analízis esetén minimális lépésszámmal, visszafelé haladó analízis esetén a legjobb algoritmushoz képest legfeljebb a ciklusok eggyel többszöri bejárásával teszi.  $\square$

4.3.8. *megjegyzés* (A helyettesítések elvégzése). Mivel a vezérlés szerinti algoritmus, így a kombinált analízist megoldó algoritmus is, egy cikluson kívüli utasítás elérésakor fixpontbeli információt határoz meg, a cikluson kívüli utasításokra vonatkozó helyettesítéseket egyből el lehetne végezni. A ciklusokban lévő utasításokra vonatkozó helyettesítéseket csak akkor lehet elvégezni, ha a legkülső ciklus is fixpontba jutott, mivel ekkor a bejárás tovább halad, és nem fog visszatérni a ciklusra.<sup>21</sup>

4.3.9. *megjegyzés* (A  $\text{Process}(\mathcal{CA}, g)$  függvény). A 4.2.1. definícióban adott, a kombinált analízisek 4.2.2. definíciójában felhasznált megoldó függvény a fenti algoritmus segítségével definiálható. A  $\text{Process}(\mathcal{CA}, g) \in \text{Graph} \times (\text{Inst} \rightarrow \text{Info})$  érték első komponense az algoritmus által optimalizált végrehajtási gráf, második komponense pedig a következő halmazzal adott függvény:  $\{(i, in_i) | i \in \text{Inst} \wedge R(i) = \perp_R\}$ , ahol  $in_i$  és  $R$  az algoritmus változóinak végső, azaz fixpontbeli értékei.

4.3.10. *megjegyzés* (A  $\text{Process}$  függvény által adott információk). Az eddigiekből világos, hogy igazából csak rekurzív optimalizálás esetén van szükség a  $\text{Process}$  függvény eredményének második részeként kapott információkra, és akkor is csak a következő utasításokra vonatkozóan: Előre haladó esetben a helyettesítendő utasítás azon rákövetkezőihez tartozó információkra, melyek elérhetők a helyettesítésben is. Visszafelé haladó analízisek esetén pedig a helyettesítő gráf belépési pontjának bemenő információjára<sup>22</sup>.

4.3.11. *megjegyzés* (Szuperoptimalizációs keretrendszer). A kombinált analízis tetszőleges integrált analízisekből összeállítható, így a 4.2.2. definícióban megadott

<sup>21</sup> A megvalósítás során memória-hatékonysági szempontból érdemes a helyettesítéseket minél előbb elvégezni. Az ismertetett algoritmusba a lehető legegyszerűbb alak megtartása miatt nem került be ez a gondolat.

<sup>22</sup> Visszafelé haladó esetben a helyettesítő gráf belépési pontjának speciális szerepéről lásd a 4.1.2. megjegyzést.

kombinált analízis a 4.3.4. definícióban adott megoldó algoritmusával egy keretrendszert alkot az integrált analízisek végrehajtásához. Ráadásul ez a keretrendszer képes tetszőleges integrált analízisek között az információ terjedését a helyettesítések rekurzív optimalizálásán keresztül biztosítani, így a lehető legpontosabb információt biztosítva a transzformációs döntések meghozatalához. Ez pedig a különálló analízisek logikáját egyesítő szuperoptimalizációk jellemzője. Ezért nevezem ezt a keretrendszert *szuperoptimalizációs keretrendszernek*.

## 4.4. Néhány integrált analízis

A szuperoptimalizációs keretrendszer ismertetése után megadom néhány egyszerű integrált analízis formális definícióját.

### 4.4.1. Konstans továbbterjesztés

Dolgozatomban egyik alappéldája a konstans továbbterjesztés, melynek integrált analízis alakja a 3.3.1. definícióban megadott adatáram-analízisére épül.

**4.4.1. definíció** ( $\mathcal{IA}_{ConstProp}$ ).  $\mathcal{IA}_{ConstProp} = (\text{Forward}, \tau'_{ConstProp}, \mu_{ConstProp})$ , ahol  $\mu_{ConstProp}$  összefésülő függvény azonos a 3.3.1. definícióban megadottal, valamint

$$\tau'_{ConstProp}(info, inst) = \begin{cases} info' & \text{ha } inst \ v := e \text{ alakú értékadás} \\ G & \text{ha } \exists v \in \text{Var} : v \in V(inst) \wedge info(v) \in \text{Value} \\ info & \text{egyébként} \end{cases}$$

Ahol

$$info' = info|_{\text{Var} \setminus \{v\}} \cup \{(v, val(info, e))\}$$

valamint  $val : \text{Inst} \times \text{Expr} \rightarrow \text{Value} \cup \{\top\}$  függvény a 3.3.1. definícióban adott és  $V : \text{Inst} \rightarrow \wp(\text{Var})$  megadja egy utasítás által olvasott változók halmazát.  $G$  pedig egyetlen utasításból álló  $CFG$ , melynek utasítása  $inst$ -ből úgy keletkezik, hogy minden olyan  $v$  változó-hivatkozást, melyre  $info(v) \in \text{Value}$  lecserélünk  $info(v)$ -re.

### 4.4.2. Elérhetőség-vizsgálat

Másik alappéldám az elérhetőség-vizsgálat, mely integrált analízisének definíciója a következő:

**4.4.2. definíció** ( $\mathcal{IA}_{Reachability}$ ).  $\mathcal{IA}_{Reachability} = (\text{Forward}, \tau_{Reachability}, \mu_{Reachability})$ , ahol  $\mu_{Reachability}(infos) = \perp$  és

$$\tau_{Reachability}(info, inst) = \begin{cases} G & \text{ha } inst \text{ konstans feltétellel rendelkező elágazás} \\ info & \text{egyébként} \end{cases}$$

$G$  olyan  $CFG$ , melyben egy feltétel nélküli ugrás található az  $inst$  konstans feltétele alapján megfelelő rákövetkezőjére.

4.4.1. *megjegyzés* ( $Info_{Reachability}$ ). Ahogy a fenti definícióból látható, az elérhetőség-vizsgálatnak igazából nincs szüksége külön információra, a vizsgált utasítás alapján dönt annak helyettesítéséről.<sup>23</sup> Tehát ez egy olyan optimalizáció, melynek semmi köze az adatáram-analízisekhez, de integrált analízis alakjában történő megfogalmazása lehetővé teszi, hogy a szuperoptimalizációs keretrendszerben más optimalizációkkal kombinálni lehessen.

4.4.2. *megjegyzés* (Az analízis iránya). Az elérhetőség-vizsgálat fenti definíciója előre haladó integrált analízist ad meg. Mivel azonban információ gyűjtésének hiányában a bejárás iránya lényegtelen, visszafelé haladó is lehetne.<sup>24</sup>

### 4.4.3. Függvény beágyazása

A függvények beágyazása annyiban hasonlít az elérhetőség-vizsgálatra, hogy nem gyűjt információt az éppen optimalizált gráfból. A beágyazásra vonatkozó döntéseit a függvényhívás helyén a meghívott függvény alapján hozza meg.<sup>25</sup> Így kézenfekvőnek tűnhet az üres információ, azaz  $Info_{Inliner} = \perp$  használata. Azonban implementációs szempontból célszerűbb az egyes függvényekről szerzett információt elérhetővé tenni a későbbi függvényhívások elemzése számára, mivel így elkerülhető, hogy a többször meghívott függvényeket minden hívási helyen újra kelljen ellenőrizni.

4.4.3. **definíció** ( $\mathcal{IA}_{Inliner}$ ).  $\mathcal{IA}_{Inliner} = (\text{Forward}, \tau_{Inliner}, \mu_{Inliner})$ , ahol  $\forall f \in \text{Fnc} :$

$$\mu_{Inliner}(infos)(f) = \begin{cases} \perp & \text{ha } infos = \emptyset \vee \forall info \in infos : info(f) = \perp \\ \alpha & \text{ha } \exists info \in infos : info(f) = \alpha \end{cases}$$

Ahol  $\text{Fnc}$  a függvények halmaza, valamint  $Info_{Inliner} : \text{Fnc} \rightarrow (\mathbb{L} \cup \{\perp\})$ .

$$\tau_{Inliner}(info, inst) = \begin{cases} info & \text{ha } inst \text{ nem függvényhívás} \\ info & \text{ha } inst \text{ hívja } f\text{-et és } info(f) = \downarrow \\ G & \text{ha } inst \text{ hívja } f\text{-et és } info(f) = \uparrow \\ \Gamma & \text{ha } inst \text{ hívja } f\text{-et és } info(f) = \perp \end{cases}$$

Ahol

$$\Gamma = \begin{cases} info_{(f, \downarrow)} & \text{ha } f\text{-et nem kell beágyazni} \\ (G, info_{(f, \uparrow)}) & \text{ha } f\text{-et be kell ágyazni} \end{cases}$$

<sup>23</sup> Lásd még az 1.2.2. definíciót.

<sup>24</sup> A definícióban megadott irány oka, hogy az elérhetőség-vizsgálat által végzett transzformáció előre haladó analízisek között tud hatékony eredményt nyújtani.

<sup>25</sup> Lásd az 1.2.4. megjegyzést.

Továbbá  $info_{(f,\downarrow)} = info|_{\text{Fnc} \setminus \{f\}} \cup \{(f, \downarrow)\}$  és  $info_{(f,\uparrow)} = info|_{\text{Fnc} \setminus \{f\}} \cup \{(f, \uparrow)\}$ .

$G$  pedig  $f$  végrehajtási gráfjának módosított változata úgy, hogy a formális paraméterek hivatkozásait a hívás helyén lévő aktuális paraméterekre cseréljük.

4.4.3. *megjegyzés* (Az analízis iránya). Mivel az analízis nem az optimalizált programról gyűjt információt, ezért az elérhetőség-vizsgálathoz hasonlóan, visszafelé haladó analízis is lehetne.

4.4.4. *megjegyzés* (Az analízis információja). A fenti definícióban adott formalizmus azon a feltételezésen alapszik, hogy egy függvény beágyazása nem függ a hívás helyétől, vagyis a beágyazásról mindig ugyanaz a döntés születik. Így a  $\mu_{Inliner}$  összefésülő függvény definíciójának második sorában lévő  $\alpha$  mindig egyértelmű.

A gyakorlatban a beágyazási döntést a hívás helye, az aktuális paraméterek is befolyásolják. A megadott definíció alapvetően ebben az esetben is használható. A továbbított információ a hívott függvény alapján hozott döntést reprezentálja, és amennyiben ez nem tiltja meg a beágyazást, úgy a konkrét hívási helyen további feltételek alapján dönthetünk a tényleges beágyazásról.

#### 4.4.4. Nem használt értékadások törlése

Egy egyszerű visszafelé haladó integrált analízis a nem használt értékadások törlése, melynek definíciója a 3.3.2. definícióban adott adatáram-analízisre épül.

4.4.4. **definíció** ( $\mathcal{IA}_{DAE}$ ).  $\mathcal{IA}_{DAE} = (\text{Backward}, \tau'_{DAE}, \mu_{DAE})$ , ahol  $\mu_{DAE}$  összefésülő függvény a 3.3.2. definícióban adott, valamint

$$\tau'_{DAE}(info, inst) = \begin{cases} G_0 & \text{ha } inst \ v := e \text{ alakú és } \neg info(v) \\ \tau_{DAE}(info, inst) & \text{egyébként} \end{cases}$$

Ahol  $G_0$  üres CFG,  $\tau_{DAE}$  pedig a 3.3.2. definícióban adott átmenet-függvény.

### 4.5. A keretrendszer működés közben

A 2.1. ábrán iteratíván optimalizált program keretrendszer általi szuperoptimalizációjának lépéseit mutatják a 4.1(1).-4.1(2). ábrák. Az optimalizáláshoz használt integrált analízisek az  $\mathcal{IA}_{ConstProp}$  és  $\mathcal{IA}_{Reachability}$ . Az iteratív optimalizálás eredményének korlátaira hozott 2.3b. ábrán látható program ugyanazon kombinált analízis általi optimalizációjának lépései a 4.2(1). és 4.2(2). ábrákon láthatók, melynek végén eljutunk a 2.3c. ábrán lévő, az iteratív optimalizálással elérhetetlen formához.<sup>26</sup>

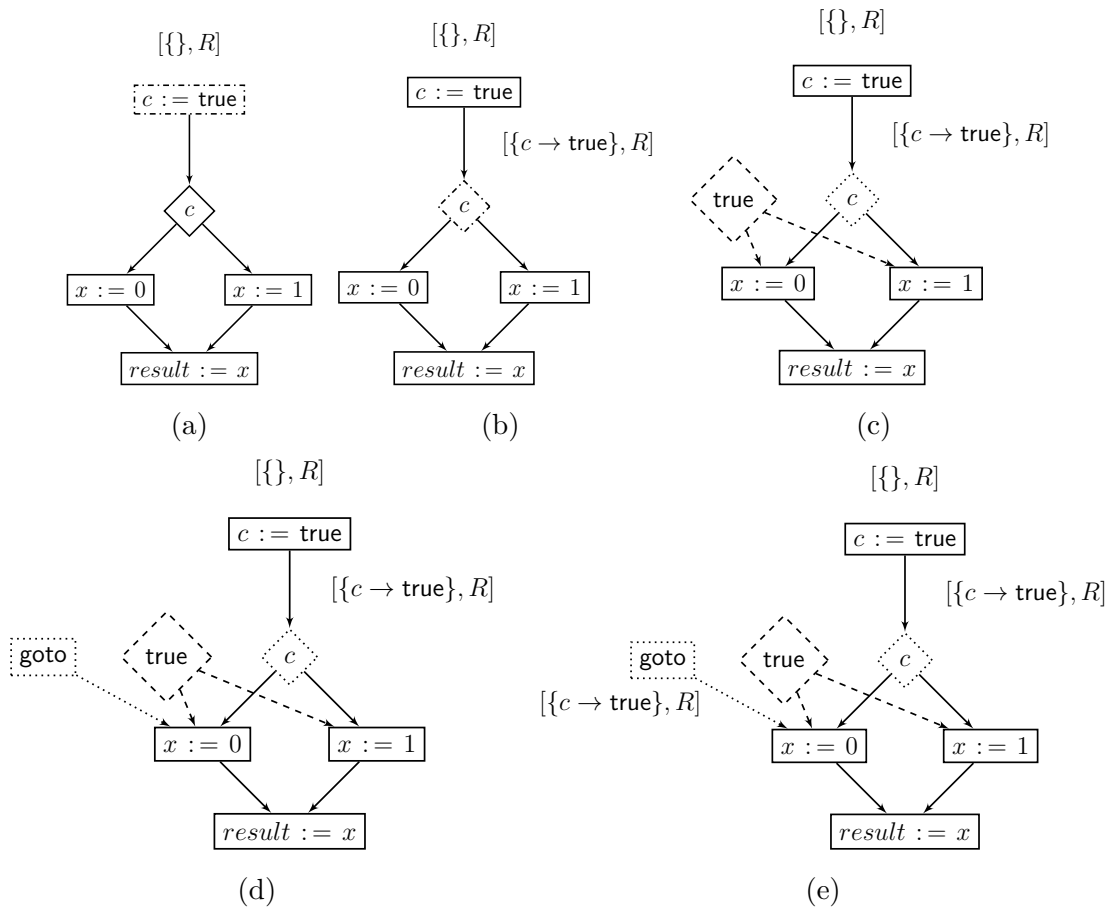
<sup>26</sup> Ha a kombinált analízis meghatározásához az  $\mathcal{IA}_{Inliner}$  integrált analízist is felhasználnánk, akkor a 2.3a. ábrán lévő alakból is eljutnánk a 2.3c. ábrán lévőhöz, de az egyszerűség kedvéért itt a függvény beágyazása utáni állapotból indulunk ki.

4.5.1. *megjegyzés* (Jelölések). A szuperoptimalizációs keretrendszer működését szemléltető ábrák az adatáram-analízisek megoldásánál használt 3.4.1. megjegyzésben ismertetett jelöléseken túl a következőket alkalmazzák.

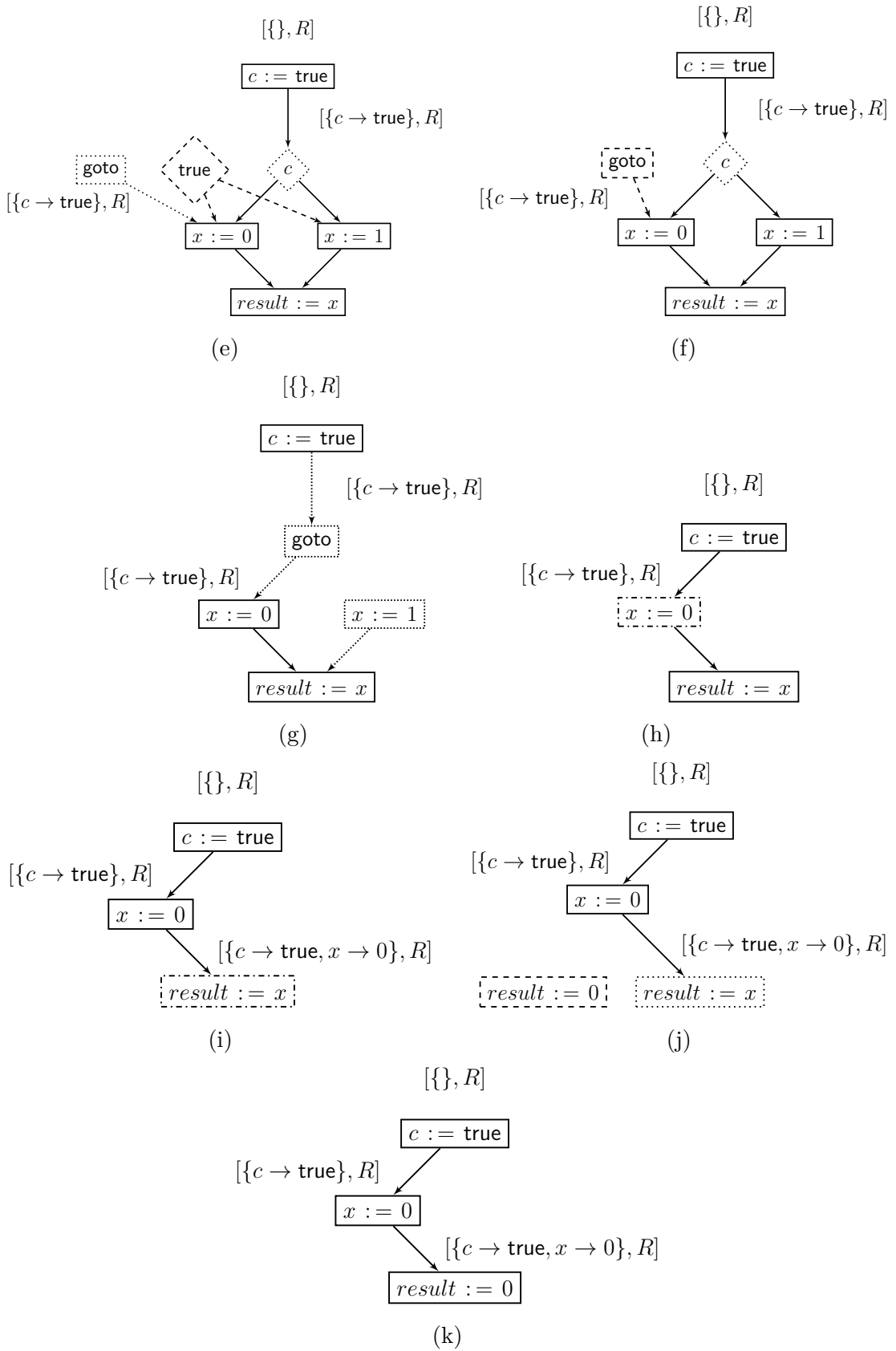
A kombinált analízis információjában szögletes zárójelek között adom meg az egyes integrált analízisek információit. A konstans továbbterjesztésnél csak azokat a változókat jelzem, melyekhez nem  $\perp$  érték tartozik. Az elérhetőség-vizsgálat üres információját nem jelölöm. Az  $(i, j)$  él mentén haladó információ végén lévő  $R$  jelzés  $I(i, j) = \uparrow$ , az  $U$  jelzés  $I(i, j) = \perp_I$  megoldó algoritmusbeli állaputra utal.

Egy integrált analízis által adott helyettesítést a vizsgált utasítás mellett szaggatott szegéllyel jelzek, a helyettesítés rekurzív optimalizálásakor keletkező helyettesítést már pöttyözött szegély jelöli. Egy helyettesítés optimalizálásának kimenő információját a helyettesítés alatt jelenítem meg, majd végül csak az összefuttatott információt írom az elemzett utasítás kimenő éle mellé. Az eltárolt helyettesítéseket dupla szegély jelöli.

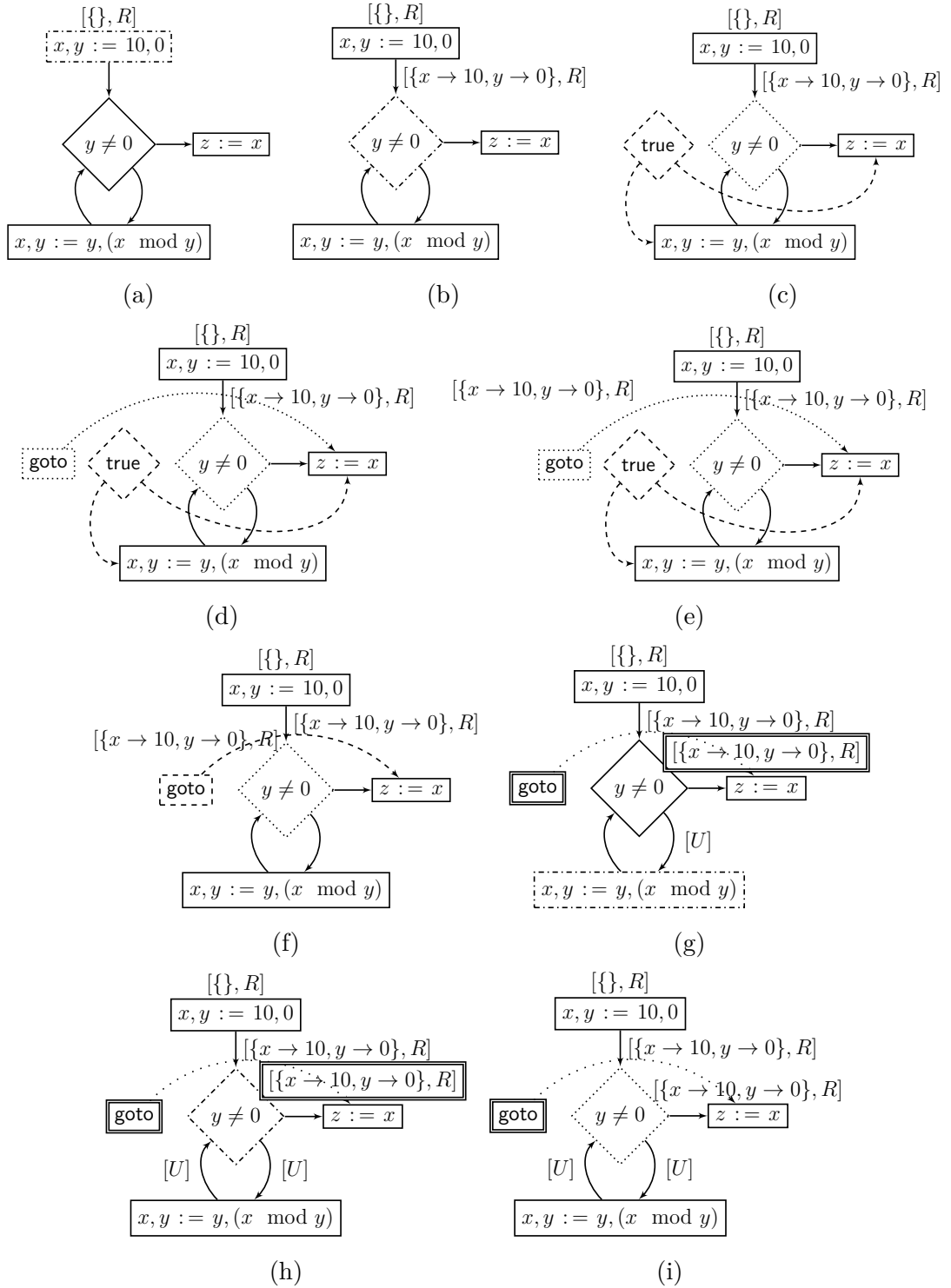
A helyettesítéseket a 4.3.8. megjegyzés alapján ciklusokon kívül egyből elvégzem, a ciklusokban lévő utasításokhoz tartozó helyettesítéseket csak a legkülső ciklus fixpontba jutása után. A behelyettesített feltétel nélküli ugrások miatti összevonásokat és az elérhetetlenné váló részek törlését külön lépésbe kiemelve szerepeltetem.



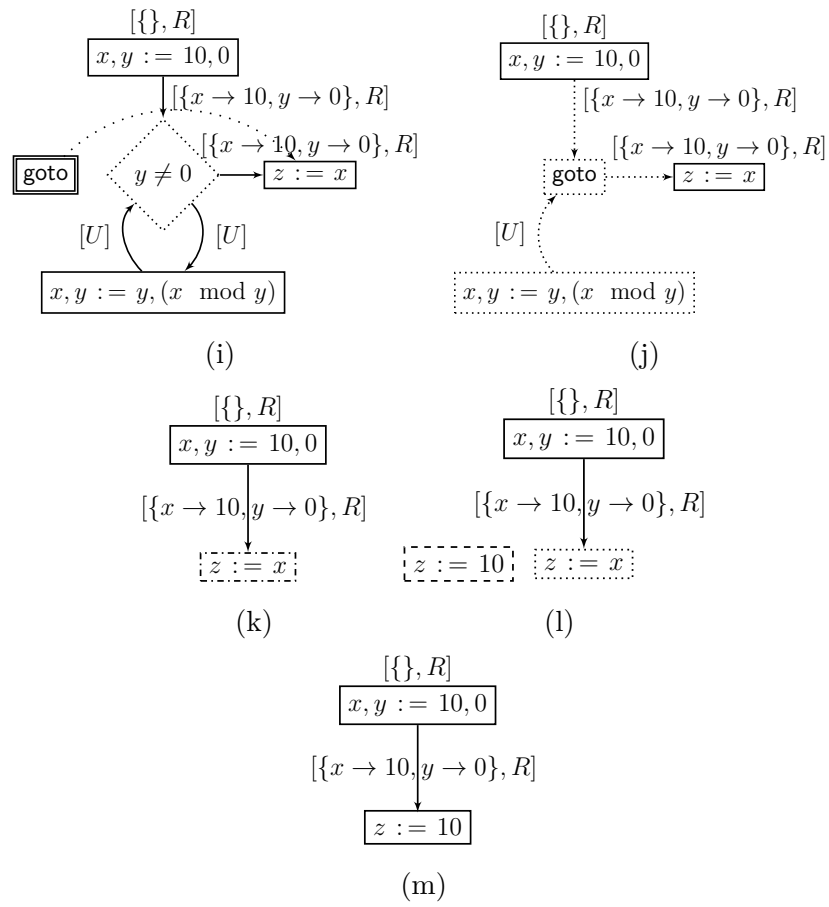
4.1. ábra. Egy egyszerű program szuperoptimalizálásának állapotai. (1)



4.1. ábra. Egy egyszerű program szuperoptimalizálásának állapotai. (2)



4.2. ábra. A problémás ciklus superoptimalizálásának állapotai. (1)



4.2. ábra. A problémás ciklus szuperoptimalizálásának állapotai. (2)

## 5. fejezet

# Az LLVM

Az előző fejezetben ismertetett szuperoptimalizációs keretrendszert az LLVM (*Low Level Virtual Machine*) nevű fordítóprogram-eszköztár aktuális, 2.9-es verziójához meg is valósítottam. Mielőtt rátérnék az implementáció során felmerült néhány érdekesebb probléma ismertetésére, ebben a fejezetben röviden bemutatom magát az LLVM-et, legalábbis a keretrendszer megvalósítása szempontjából fontos részeit.

Egy rövid bevezetőt nyújt az LLVM-hez a [6] prezentáció, ezenkívül az LLVM használatával és megvalósításával kapcsolatos bővebb információk találhatók a [18, 19, 20, 21] online anyagokban.

### 5.1. Az LLVM dióhéjban

Az LLVM projektet 2000-ben indították az Illinois Egyetemen azzal a céllal, hogy különböző dinamikus fordítási technikákat kutassanak statikus és dinamikus programozási nyelvek esetén. Mivel a cél tetszőleges programozási nyelvek fordításának vizsgálata volt, egy saját alacsony szintű nyelvet, az *LLVM assembly*-t definiálták ([19]), melyre a különböző programozási nyelveken készített programok lefordíthatók. Ebből a köztes reprezentációból aztán különböző technikákkal gépi kód generálható az LLVM segítségével.

A kutatási projekt annyira sikeresnek bizonyult, hogy 2003-ban megjelent az LLVM első nyilvános kiadása, mely folyamatos aktív fejlesztés mellett már a 2.9-es verziónál tart. Jól mutatja az LLVM sikerét és széleskörű használhatóságát, hogy a kezdeti C/C++ mellett már a programozási nyelvek széles köréhez létezik front-end, mely adott nyelvű programot képes az LLVM belső reprezentációjára alakítani. Ilyen nyelvek többek között az Ada, a Haskell, a Python, a Fortran, az Objective-C, sőt még Java bájt kódhoz is létezik LLVM front-end. Emellett egy ideje az Apple fejlesztői eszköztárának is szerves részét képezi.

Az LLVM sikerének egyik titka az általános, nyelv-független belső reprezentáció, melyből ma már lényegében tetszőleges architektúrára készíthető alacsony szinten optimalizált gépi kód. Ez lehetővé teszi a magasabb szintű nyelvek készítőinek, hogy a magas szintű problémákra koncentrálhassanak. A magas szinten optimalizált programokat egyszerűen az LLVM belső reprezentációjára kell alakítani, majd az *LLVM IR*-ben<sup>1</sup> lévő programokat a meglévő eszközök segítségével lehet optimalizálni és gépi kódra fordítani.

## 5.2. Az *LLVM assembly*

Az optimalizáció és az egész fordítási folyamat szempontjából fontos jellemző a programok reprezentációja. Ennek megfelelően az LLVM megtervezésekor olyan megoldást választottak, mely képes a fordítás minden fázisában megfelelő alapot biztosítani az elvégzendő feladatok kényelmes kivitelezéséhez. Emellett a programok felépítését is az elemzések és transzformációk könnyű megvalósíthatóságát szem előtt tartva alakították ki.

### 5.2.1. Az LLVM IR 3 alakja

A LLVM programok, ahogy azt már korábban említettem, az *LLVM assembly* utasításaiból állnak. Azonban ezeket három különféle alakban reprezentálhatjuk a felhasználás céljától függően.

**Szöveges reprezentáció.** A programok olvasható alakja, ebben a formában kényelmes a programozó számára írni és olvasni a programokat. Esetleg ilyen reprezentációjú programot generálhatunk magasabb nyelvekből.

**Objektum-gráf.** Az LLVM programok feldolgozásakor egy objektum-gráf épül fel a program utasításaiból. Ebben az alakban egyszerű a programok elemzése és módosítása.<sup>2</sup> Ezt használják például optimalizálásnál.

**Bájtkód.** Az LLVM programok legkompaktabb alakja az *LLVM bájtkód*, mely a programok tárolására alkalmas.

<sup>1</sup> Intermediate Representation, azaz az LLVM belső reprezentációja, ami lényegében az *LLVM assembly*-t jelenti.

<sup>2</sup> Maga az LLVM *C++*-ban készül, ezért reprezentálják objektumok a programokban lévő entitásokat.

### 5.2.2. Az LLVM programok felépítése

Az *LLVM assembly*-vel megadható programok fordítási egysége a modul, mely globális változókat tartalmazhat. A globális változók egy speciális fajtája a függvény, mely címkével ellátott alablokkokból<sup>3</sup> épül fel. Minden függvénynek tartalmaznia kell egy **entry** címkével rendelkező alablokkot, amely a függvény belépési pontja lesz. Az alablokkokban szerepelhetnek az *LLVM assembly* utasításai.

Az utasítások egyik csoportját a terminátor-utasítások jelentik, melyek közül egynek szerepelnie kell minden alablokk végén.<sup>4</sup> Az alablokkokon belül a vezérlés szekvenciálisan halad a blokk első utasításától az utolsóig, és a blokk végén lévő terminátor-utasítás határozza meg a vezérlés további útját.

Az LLVM programok objektum-gráffal történő reprezentációja esetén az utasítások operandusai közvetlenül az adott entitást reprezentáló objektumra hivatkoznak. Így az alablokkok a terminátor-utasításokban lévő mutatók mentén kapcsolódnak egymáshoz. Ilyen formán a programok objektum-gráfos reprezentációja esetén a *CFG* explicit módon adott, ami nagyon könnyűvé teszi a programok bejárását.

### 5.2.3. DU- és UD-láncok

A programok elemzése és módosítása során hasznos információ az egyes entitások definíciójának és felhasználási helyeinek pontos ismerete. A DU-, UD-láncok és hálók definíciója megtalálható [12] 8.10. fejezetében.

Az egyik irányú kapcsolatot az UD-láncok<sup>5</sup> reprezentálják, melyek megadják az utasításokban felhasznált entitások definícióját, illetve annak helyét. Ez objektum-gráfos reprezentáció esetén adott az LLVM programokban, hiszen az operandusok mutatók az entitást reprezentáló objektumra.

A másik irányú kapcsolatot a DU-láncok<sup>6</sup> reprezentálják, amelyek egy entitás definíciójának helyén megmondják, hol kerül az adott entitás felhasználásra. Az LLVM objektum-gráf felépítése során az objektumok ezen információkat összegyűjtik, így a DU-láncok is elérhetők a programok feldolgozása során.

### 5.2.4. Static Single Assignment Form

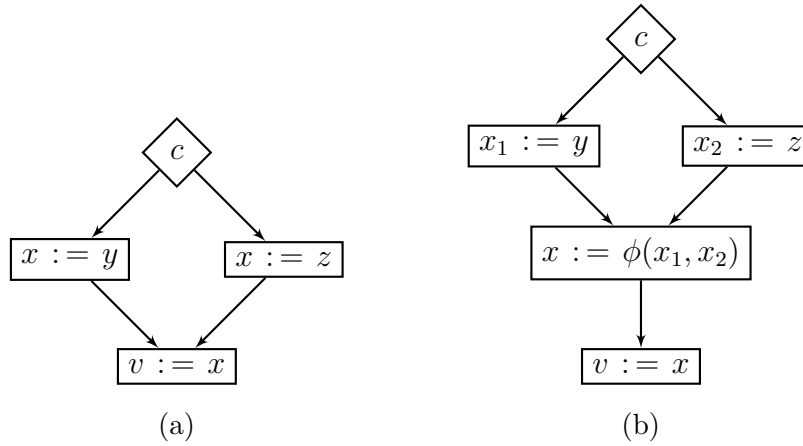
Egy adatáram-analízisek szempontjából hasznos köztes reprezentáció a *Static Single Assignment Form* (*SSA*). Egy program *SSA*-ban van pontosan akkor, ha

<sup>3</sup> Az alablokk szekvenciálisan végrehajtódó utasítások sorozata. Lásd [3].

<sup>4</sup> Az alablokkok végétől különböző helyen nem is szerepelhet terminátor-utasítás.

<sup>5</sup> *UD-chains*, ami a *use-def chains* rövidítése.

<sup>6</sup> *DU-chains*, ami a *def-use chains* rövidítése.



5.1. ábra. Elágazás hagyományos és *SSA*-ban lévő alakja.

minden változó csak egy értékadás bal oldalán szerepel. Az *SSA* alakról és felhasználásának pontos lehetőségeiről [12] 8.11. fejezetében olvashatunk.

Az *SSA* forma egyszerűbbé teszi az adatáram-analízisek elvégzését, mivel minden változó csak a program egy pontján kaphat értéket, így az UD-láncok könnyen meghatározhatók. Ugyanakkor ha egy változó különböző végrehajtási utakon különböző értéket kapna (5.1a. ábra), akkor azt át kell alakítani az *SSA* forma feltételének megfelelően (5.1b. ábra). A különböző utak mentén különböző értéket kapó változók beállítására szolgál a  $\phi$  függvény, mely paraméterei közül annak a változónak az értékét adja, amelyhez tartozó értékadás a tényleges végrehajtás során legutóbb végrehajtott.<sup>7</sup>

Mivel az LLVM programok *SSA* alakban vannak, ezért bizonyos optimalizációk végrehajtása egyszerűsödik. Például konstans továbbterjesztés esetén amikor egy értékadásról kiderül, hogy a jobb oldalán szereplő kifejezés értéke konstans, akkor a bal oldali változó minden előfordulása további vizsgálatok nélkül lecserélhető az adott konstansra. Ezt pedig a definícióhoz tartozó DU-lánc elérhetősége miatt azonnal meg lehet tenni.

### 5.3. Optimalizálás az LLVM 2.9-ben

Mielőtt belekezdtem a szuperoptimalizációs keretrendszer megvalósításába, szerettem volna informálódni az LLVM által biztosított optimalizálási módról. Bár sok hasznos dokumentum érhető el, melyek segítséget nyújtanak az LLVM egyes komponensei felhasználásához, illetve bizonyos irányban történő bővítéséhez, az LLVM

<sup>7</sup> Egyszerű elágazás esetén a  $\phi$  függvény azon értékadás eredményét adja, amely egyáltalán végrehajtott. Ciklusok esetén lehetséges, hogy a felsoroltak közül több értékadás is végrehajtott, ekkor a legutoljára végrehajtott által beállított értéket adja.

magjának megvalósításáról nem találtam átfogó dokumentációt. Így kénytelen voltam az optimalizálás végrehajtásának szervezésére vonatkozó kíváncsiságomat az aktuális forráskódok tanulmányozásával kielégíteni.

A forráskódok tanulmányozása során kiderült, hogy az LLVM az optimalizációkat alapvetően a 2.1. fejezetben ismertetett iteratív módon hajtja végre. Az analízisek és transzformációk nem válnak élesen el egymástól, azokat menetekként kell definiálni. Minden menetnek explicit módon meg kell adnia, hogy mely más menetek információjára van szüksége és mely más menetek információját nem rontja el az általa végzett transzformáció. Ezen információk birtokában az LLVM a 2.2.3. következőben megfogalmazott hatékony módon ütemezi a meneteket.

Tovább árnyalja a képet az optimalizálási szintek bevezetése. Ez azt jelenti, hogy egy adott szintű menetet a program minden megfelelő típusú részére végrehajt az LLVM. A menetek a különböző végrehajtásaik között nem őrizhetnek meg információt, illetve nem okozhatnak változást az éppen optimalizált egységen kívül eső programrészekben. A legkülső szintű menet a modul-menet, mely példányai egy egész modul optimalizálását végezhetik, tehát akár függvényeket is törölhetnek vagy hozzávehetnek egy modulhoz. A következő szint a függvények szintje, mely szinthez tartozó menetek egy függvényen belül gyűjthetnek információt és végezhetnek módosításokat. A legalacsonyabb szint az alapelblokkokhoz tartozó meneteké, melyek pusztán egyetlen alapelblokkon belül végezhetnek módosításokat. Létezik a menetek két szintje, melyek a függvény- és alapelblokk-menetek között találhatók. Ezek a ciklus- és régió<sup>8</sup> szintű menetek, melyek esetén azon kívül, hogy minden ciklusra, illetve régióra végrehajtnak, az egymásba ágyazott ciklusok és régiók esetén a végrehajtás a legbelső előfordulástól halad a legkülső felé.

Az optimalizációk végrehajtása az előzőekben megadott módon sorbarendezett menetek egyszeri végrehajtását jelenti. Ugyanakkor minden menet visszatérési értékében megadja, hogy végrehajtása tevékeny volt-e, így az optimalizálást indító programrész értesül a ténylegesen elvégzett transzformáció tényéről. Ezen információ birtokában pedig újra elindíthatja az optimalizálást a fixpont elérése érdekében.

Mivel a szuperoptimalizációs keretrendszer egy függvény optimalizálását végzi, ezért egy függvény-menet formájában illesztettem be a meglévő optimalizációk közé.

<sup>8</sup> A régiók olyan, akár több alapelblokkból álló programrészek, melyekbe a vezérlés egyetlen ponton kerülhet be, és egyetlen irányban távozhat.

## 6. fejezet

# A keretrendszer megvalósítása LLVM-hez

A 3.2.1. definícióban adott algoritmust a következőkben röviden vázolt implementációs megfontolások figyelembe vételével készítettem el. A megvalósítás az LLVM meglévő részeihez igazodva C++-ban történt, és a keretrendszert egy optimalizációs menet formájában tettem elérhetővé.

### 6.1. Analízisek és információik

Az analízisek és hozzá tartozó információk implementációja az [1] beszámolóból vett ötletek alapján készült el. Ez a dolgozatomban ismertetett formális leírástól a következő pontokban tér el.

#### Az `AnalysisInfo` és `IterativeAnalysisInfo` interfészek

A 4.4. fejezetben láthattuk, hogy nem csak adatáram-analízisek és a rájuk épülő transzformációk alkalmasak integrált analízis definiálására. Ilyen volt az elérhetőség-vizsgálat és a függvény beágyazása integrált analízis is. Ez azt mutatja, hogy egyes integrált analízisek nem igénylik ciklusokban az információ fixpontjának keresését, hanem minden utasítást pontosan egyszer akarnak meglátogatni.

Ilyen integrált analízisek megvalósítására nyújt lehetőséget az `AnalysisInfo` interfész, melyet megvalósító információk nem továbbítódnak a visszaélek mentén. A ciklusokban fixpontot kereső információk interfésze az `IterativeAnalysisInfo`, mely kiegészíti az `AnalysisInfo` interfészt a visszaélek mentén történő továbbításhoz szükséges két művelettel. Ezek egyike a fixpont elérésének vizsgálatára szolgál, mely a ciklus fejének aktuális információját és az éppen továbbítandó információt hasonlítja össze. A másik művelet a `generalizingMerge`.

## A merge és generalizingMerge műveletek

Az `AnalysisInfo` interfész művelete a `merge` művelet, mely a formális leírás  $\mu$  összefésülő függvényének megvalósítása. Figyelembe véve, hogy az összefésülés igazából egy kommutatív és asszociatív művelet, a `merge` egy adott információt belefésül egy meglévő információba. Így inkrementálisan előállítva az összes információ összefésülésének eredményét.

Az `IterativeAnalysisInfo` interfész `generalizingMerge` művelete a 4.1.1. megjegyzésben leírtak alapján biztosítja az információk speciális összefésülésének lehetőségét a visszaélék mentén. Mivel a legtöbb analízis esetén erre nincs szükség, az alapértelmezett tevékenysége a `merge` művelet meghívása.

## Az `IntegratedAnalysis` interfész

Az integrált analízisek definiálásához az `IntegratedAnalysis` absztrakt osztályt kell megvalósítani, mely két tagfüggvényt tartalmaz.

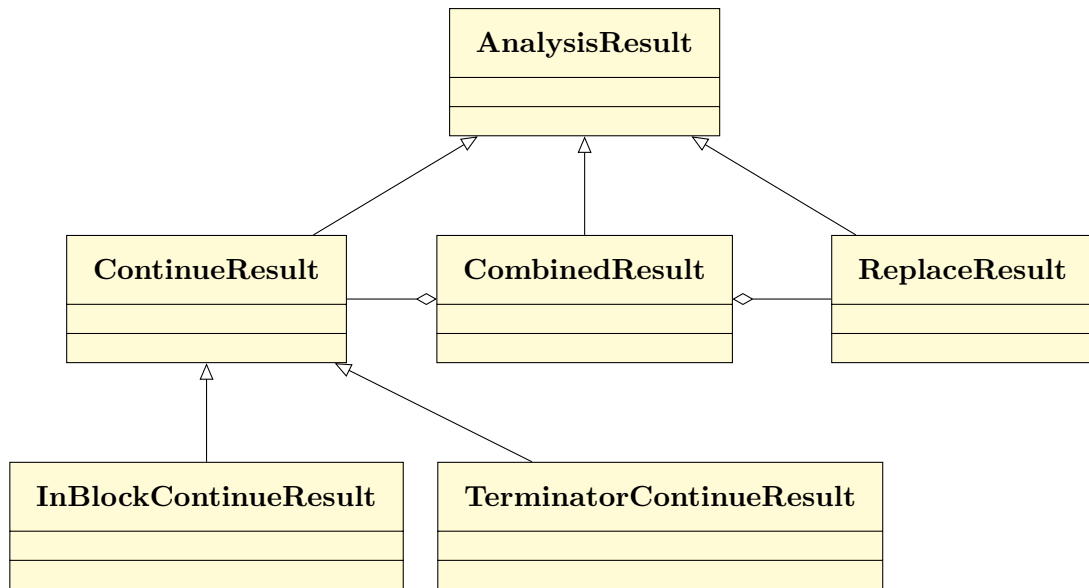
Mivel az információk `merge` művelete egy meglévő információba képes további információkat belefésülni, ezért az analízis kiindulási információját magának az analízisnek kell megadnia. Erre szolgál az `initInfo` paraméter nélküli függvény.

Az `IntegratedAnalysis` interfész `process` művelete a  $\tau$  átmenet-függvény megvalósítása, mely megkap egy utasítást és a hozzá tartozó bemenő információt. Annak jelzésére, hogy az analízis nem változtathatja meg az utasítást, azt konstans paraméterként kaphatná meg. Ugyanakkor ez egyrészt nem biztosítaná, hogy az analízisből nem végezhető módosítás a programon, másrészt a keretrendszer szempontjából is legális műveletek elvégzését lehetetlenítheti el<sup>1</sup>. Így kénytelenek vagyunk bízni az integrált analízisek készítőiben, hogy nem végeznek módosítást közvetlenül az analízisből.

## 6.2. Az analízisek eredményei

A `process` függvény visszatérési típusa az `AnalysisResult` osztály, mely a formális definíciónak megfelelően alapvetően háromféle konkrét eredményt adhat, ahogy az a 6.1. ábrán látható. Az ábra egyetlen magyarázatra szoruló részét a `ContinueResult` osztály leszármazottai adják. A `TerminatorContinueResult` osztály felel a több rákövetkezővel rendelkező utasítások egyes rákövetkezőihez tartozó információk meghatározásáért. Ezt az előre haladó analízisek az alapblokkok végén

<sup>1</sup> Egy tipikus legális művelet az elemzett utasítás lemásolása, azonban másolatot csak nem-konstans utasításokról lehet készíteni.



6.1. ábra. Az **AnalysisResult** osztály leszármazottai

lévő termináló-utasítások eredményeként adják. Az **InBlockContinueResult** osztály egyetlen információt tartalmaz, mely megfelelő az alapblokkok belsejében lévő utasítások eredményének, mivel a rákövetkező ekkor egyértelmű. Ezt az információ-továbbító eredményt használják a visszafelé haladó integrált analízisek minden esetben, mivel ez a 3.1.6. megjegyzés miatt elégséges.

A **ReplaceResult** osztály a helyettesítések összeállításához az aktuálisan vizsgált utasítás rákövetkezőihez proxy objektumokat biztosít, melyeket a helyettesítések elvégzésekor visszacserél az eredeti objektumokra. A proxy objektumok nyilvántartásával pedig meg tudja adni, hogy mely rákövetkezők felé haladó kapcsolat szűnne meg a helyettesítés elvégzésével.

### 6.3. A kombinált analízis

A kombinált analízis a megvalósításban nem válik élesen el a bejárást végző programrészeztől. Azonban a kombinált analízishez tartozó kombinált információ külön osztályként jelenik meg. Ez egy speciális **IterativeAnalysisInfo**, melyben a műveletek a formális definíciónak megfelelően, egyszerűen az egyes komponensek megfelelő műveleteire vannak visszavezetve.

A kombinált analízis átmenet-függvényének működése sem sokkal bonyolultabb. A definíciónak megfelelően az egyes integrált analízisek **process** műveleteit kell meghívni a kombinált információ megfelelő komponensével. Ha az integrált analízisek valamelyike helyettesítést javasol, akkor azt rekurzívan optimalizálni kell a keret-

rendszerrel.<sup>2</sup> A lehetséges sok információ összefuttatását a `ContinueResultBuilder` osztály végzi, melytől a rekurzív optimalizációk végén lekérdezhető a kialakult továbbítandó információ.

Több helyettesítés esetén a legjobb kiválasztását az LLVM-ben elérhető kódmetrika alapján végzem.

## 6.4. A kombinált analízis végrehajtása

A keretrendszer lelkét a bejárást végző részek jelentik, melyek a bejárás végrehajtásához a *CFG*-re vonatkozó információkat használnak fel. Mivel az LLVM programok explicit módon alapblokkokra oszlanak, elegendő az alapblokkok szintjén tárolni a szükséges információkat. A keretrendszerben minden alapblokkhoz tartozik egy leíró, mely egy listában tartalmazza a bejárás iránya szerint megelőző blokkokat, valamint azt, hogy az adott megelőző felől érkezett-e már információ. Egy másik listában a bejárás szerint rákövetkező blokkok találhatók azzal az információval, hogy a feléjük vezető él milyen típusú.<sup>3</sup> Emellett ezek a leírók tárolják a blokkhoz tartozó aktuális bemenő információt, lehetőséget adnak a blokk utasításaihoz tartozó helyettesítések tárolására, és annak jelzésére, hogy a blokk virtuálisan elérhetetlen-e. A helyettesítések megfelelő elvégzéséhez szükséges a *CFG*-ben megelező blokkok tárolása is. Továbbá az alapblokkok leírójához tartozik egy mutató a blokkot közvetlenül befoglaló ciklus leírójához.

A ciklusok leírói a belépő- és visszaélekre vonatkozó információk mellett tárolják a ciklusban lévő blokkok és a befoglalt ciklusok leíróinak mutatóit. A ciklusok leírói ezenkívül kapcsolódnak a közvetlen befoglaló ciklusuk leírójához is, így meghatározható a ciklus beágyazottsági mélysége. A mélységet első kiszámítása után eltároljuk a leíróban, mivel sokszor szükség van rá. A ciklusok fejénél lévő információ fixpontjának ellenőrzése a visszaéleken történő összefuttatásokkal egyszerre, inkrementálisan történik, így a ciklus leírója tartalmaz egy logikai változót, mely minden visszaélen történő információ-továbbítás alkalmával frissül. Valamint a leíró tárolja a kilépő élek mentén átmenetileg feltartott információkat is.

A blokk- és ciklusleírók bejárás előtti megfelelő beállítása az LLVM-ben elérhető ciklus-információk segítségével történik. Mivel a *CFG*-re vonatkozó szükséges információk a függvény objektum-gráfjaként állnak rendelkezésünkre, a függvényt

<sup>2</sup> Minden helyettesítés egy függvény, így egyszerűen meghívható rájuk az optimalizálás.

<sup>3</sup> Az éleknek hat típusa különböztethető meg. Az egyszerű típusok a normál-, belépő-, kilépő- és visszaélek. Ezeken kívül vannak olyan élek melyek kilépnek a közvetlen befoglaló ciklusukból, és egy befoglaló ciklusban visszaélek vagy egy másik külsőbb szintű ciklus belépő élei.

a leírók beállításához egyszer be kell járni. Ez a bejárás a leírók beállítása mellett biztosítja a nem elérhető utasítások törlését is.<sup>4</sup>

A leírók birtokában a bejárás lényegében a 4.3.4. definícióban adott algoritmust követi. Az egyik eltérés a helyettesítések elvégzésében van, mivel azokat nem a bejárás után, hanem a 4.3.8. megjegyzésnek megfelelően menet közben hajtja végre.<sup>5</sup> A keretrendszer a helyettesítések elvégzése során a megszűnő rákövetkezési kapcsolatoknak megfelelően módosítja a leírókat, és az elérhetetlenné váló blokkokat törli. Így a ciklusokon kívül nem szükséges a megszűnő kapcsolatokkal a bejárás során külön foglalkozni. Azonban a ciklusok belsejében, mivel a helyettesítéseket csak a legkülső ciklus fixpontjának elérése után hajtjuk végre, szükséges a virtuálisan megszünt kapcsolatok kezelése. Ezt a blokk-leírók bejövő információkhoz tartozó listájával könnyen megtehetjük. Továbbá ha egy blokk minden megelőzőjétől azt a jelzést kapja, hogy virtuálisan megszünt a kapcsolat, akkor az adott blokk virtuálisan elérhetetlen. Így a megadott algoritmus megoldja a bejárást.

A másik eltérés az algoritmushoz képest a kilépő éleken továbbítandó információk kezelése. Mivel a megvalósításban a bemenő információk inkrementálisan állnak elő, csak a fixpontban lévő információkat szabad propagálni az élek mentén. Ezért a ciklusok bejárása során a kilépő éleken továbbítandó információkat el kell tárolni, és csak akkor kerülhetnek ténylegesen összefuttatásra a megfelelő bemenő információval, ha a ciklus bejárásának végén bebizonyosodik, hogy az információ fixpontban van. Ha a ciklus bejárását újra kell kezdeni, akkor a tárolt kilépő információk törlődnek.<sup>6</sup>

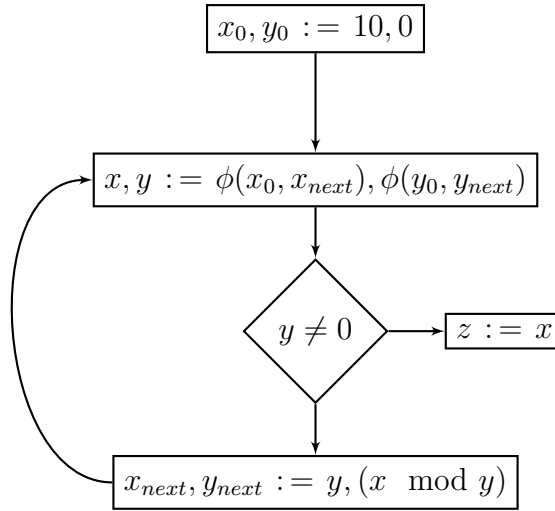
Mivel az LLVM programok *SSA* alakban vannak, a blokkok elején lévő  $\phi$ -utasítások külön kezelése szükséges a virtuálisan elérhetetlen blokkok miatt.<sup>7</sup> Ez természetesen nem azt jelenti, hogy a programban lévő utasításokat ténylegesen módosítjuk, hiszen a virtuálisan elérhetetlen blokkok nem biztos, hogy ténylegesen elérhetetlenné válnak. Ehelyett azt tesszük, hogy minden  $\phi$ -utasításról egy másolatot készítünk, amelyből töröljük a virtuálisan elérhetetlen blokkokra vonatkozó részeket. Az analízisek **process** függvényeit ezzel a módosított  $\phi$ -utasítással hívjuk meg, majd a módosított utasítást töröljük.

<sup>4</sup> Ennek fontosságát lásd a 3.2.5. megjegyzésben.

<sup>5</sup> A helyettesítések elvégzése során biztosítani kell a névütközések feloldását, illetve a lecserélt utasításra vonatkozó hivatkozások megfelelő cseréjét a helyettesítésen belüli megfelelő azonosítóval rendelkező utasításra.

<sup>6</sup> A kilépő éleken továbbítandó információk helyzete alapvetően a ciklusokon belüli helyettesítésekéhez hasonlít.

<sup>7</sup> Ha ténylegesen törölünk egy blokkot, akkor a  $\phi$ -utasítások rá hivatkozó részei automatikusan törlődnek, ezért csak a virtuálisan elérhetetlen blokkokra hivatkozó  $\phi$ -utasításokkal kell foglalkozni.



6.2. ábra. A ciklusos példa *SSA* alakban

A módosított  $\phi$ -utasítások fontosságát demonstrálandó, nézzük a 6.2. ábrát, mely a jól ismert ciklusos példa *SSA* alakban. Az eredeti program szuperoptimalizációs keretrendszer általi optimalizálása megtalálható a 4.5. fejezetben. Könnyen végig-gondolható, hogy ha egy  $\phi$ -utasítás operandusai között szerepel ismeretlen értékű változó-hivatkozás, akkor a konstans továbbterjesztés azt találja, hogy az eredmény is ismeretlen. Tehát az eredeti  $\phi$ -utasítás megtartásával nem tudnánk kioptimalizálni az elágazást. Így nem csak a virtuálisan elérhetetlen utasításokra, hanem kezdetben még a visszaéleken elérhető utasításokra vonatkozó hivatkozásokat is törölni kell a  $\phi$ -utasításokból.<sup>8</sup> Így a ciklus első és második bejárásakor is az  $x := \phi(x_0)$  és  $y := \phi(y_0)$  utasításokat kell a **process** függvénynek átadni az  $x := \phi(x_0, x_{next})$  és  $y := \phi(y_0, y_{next})$  utasítások helyett.

Sajnos az integrált analízisek megvalósítását kissé elbonyolítja a virtuális  $\phi$ -utasítások bevezetése, de a jó eredmény elérése érdekében ehhez alkalmazkodni kell.

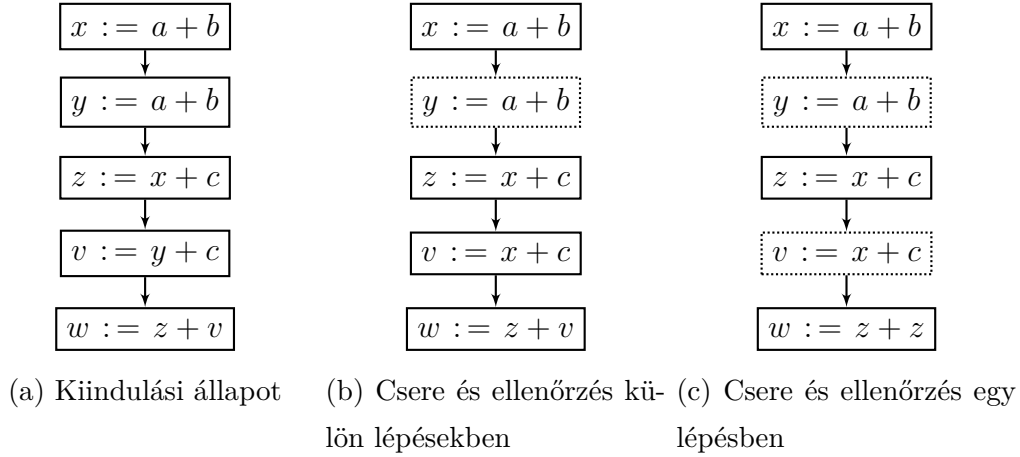
## 6.5. Az implementált analízisek

A keretrendszerhez öt integrált analízist valósítottam meg. Ezek közül az elérhetőség-vizsgálat (4.4.2. fejezet), a függvény beágyazása (4.4.3. fejezet) és az egyetlen megvalósított visszafelé haladó integrált analízis, a nem használt értékadások törlése (4.4.4. fejezet) triviálisan megvalósítható a formális definíció alapján.

A konstans továbbterjesztés integrált analízisével kapcsolatban megjegyzendő, hogy nem képes kihasználni az *SSA* alakból következő lehetőségeket, mivel a szuper-

<sup>8</sup> Ez összecseng az algoritmus definiálásánál ismertett azon elvvel, hogy a ciklus bejárásának kezdetén a visszaélekről származó információkat figyelmen kívül kell hagyni.

optimalizációs keretrendszerben a virtuálisan elérhetetlen blokkok miatt nem biztos, hogy minden olyan változó konstans, ami először annak tűnik. Ugyanakkor a virtuálisan elérhetetlen blokkok következtében jobb eredményre juthat, mint egy különálló konstans továbbterjesztés. Az integrált analízis megvalósítása a 4.4.1. fejezetbeli definícióhoz igazodik.



6.3. ábra. Redundáns értékadások kezelése

Az ötödik integrált analízis a közös részkifejezések törlése, amely az LLVM adta keretek között<sup>9</sup> lényegében megegyezik a redundáns adatok törlése optimalizációval (1.2.3. fejezet). Ennél az analízisnél egy értékadás feldolgozása alapvetően két lépésből áll: a már megtalált redundáns értéket tároló változók cseréje az adott értéket tároló első változóra, valamint annak ellenőrzése, hogy az így kapott kifejezés redundáns-e. A rekurzív optimalizálás tudatában felmerülhet, hogy ezeket a lépéseket egymástól függetlenül végezzük, mivel az esetleges cserék után kapott helyettesítést úgyis optimalizálja a keretrendszer. Azonban nem szabad elfelejtenünk, hogy a helyettesítések csak lehetőségek, ezért ha egy helyettesítésben találunk redundáns műveletet, abban nem lehetünk biztosak, hogy az a művelet valóban bekerül a programba.<sup>10</sup> Ezért a pontosabb eredmény érdekében a cserét és az ellenőrzést egy lépésben kell elvégezni. Egy egyszerű példa kiindulási állapota és a kétféle megközelítés eredménye látható a 6.3. ábrán. A 6.3b. és 6.3c. ábrákon pöttyözött szegéllyel jeleztem azokat az utasításokat, melyek a redundancia eltávolítása miatt feleslegessé váltak.

<sup>9</sup> Az LLVM-ben az értékadások jobb oldalán mindig egyetlen művelet áll, így nincsenek bonyolultabb részkifejezések.

<sup>10</sup> Az információk összefésülése során az egyes helyeken redundánsnak talált változók metszetében lévő elemek tekinthetők valóban redundánsnak. Azaz ebben az esetben az összefésülést egy metszetképzés valósítja meg.

## 7. fejezet

# További kutatási lehetőségek

Ebben a fejezetben röviden felvázolok néhány olyan, további kutatást igénylő irányt, amely mentén a keretrendszert tovább lehetne fejleszteni.

Természetesen az implementált megoldás ereje tovább növelhető újabb optimalizációk integrált analízis formájában történő megvalósításával. A különböző optimalizációkkal, köztük funkcionális programokra vonatkozó speciális esetekkel (lásd a [D. függelék](#)t) történő kísérletezés akár doktori kutatás témáját is adhatná.

A [\[8\]](#) cikk két továbbfejlesztési lehetőséget említ részletesebb ismertetés nélkül. Az egyik lehetőség a *Snooping* bevezetése, mely lehetővé tenné, hogy az integrált analízisek közvetlenül belenézhessenek egymás információjába. Ez hasznos lehetőség lehet olyan *tiszta analízisek*<sup>1</sup> esetén, melyek által gyűjtött információt több más analízis is felhasználná.

A másik egy sokkal érdekesebb lehetőség, amit *Tunneling*nek neveztek el. Ez az integrált analízisek számára lehetővé tenné virtuális élek bevezetését a *CFG*-ben, oly módon, hogy az adott utasításba bemenő virtuális élek az utasításra vonatkozó helyettesítések esetén szerepet játszanának a helyettesítés optimalizálása során a belépési pont bemenő információjának meghatározásában. Ezzel csökkenthető lenne az integrált analízisek lokalitása<sup>2</sup>, így lehetővé téve akár kódrészek mozgatását is a programon belül<sup>3</sup>.

A lokalitási feltételek enyhítésére egy másik lehetőség az integrált analízisek által kezelt egység módosítása, vagyis megengedni egy utasításnál nagyobb egységek kezelését. Érdekes feladat lenne vizsgálni ennek az elképzelésnek az életképességét, hasznosságát és megvalósíthatóságát. Már önmagában az egységek meghatározása sem egy magától értetődő probléma.

<sup>1</sup> Olyan integrált analízisek, amelyek sosem adnak eredményül helyettesítést.

<sup>2</sup> Az ily módon dinamikusan kiegészített adatáram-egyenletekben a program távoli részei közvetlen kapcsolatba kerülhetnek.

<sup>3</sup> Vesd össze a [4.1.5.](#) megjegyzéssel.

Egy másik teljesen természetesen adódó irány az interprocedurális optimalizálás támogatása. Ezt az [1] beszámoló tanúsága szerint a *Vortex* fordítóprogramban megvalósították, de részletes dokumentáció és elméleti leírás nem áll rendelkezésre erről a kiegészítésről.

Az adatáram-analízisek esetén léteznek megoldások az elemzések inkrementális elvégzésére, azaz annak biztosítására, hogy egy program változtatása esetén csak a program azon részeit kelljen újraelemezni, melyek információira a változtatás hatással volt. Egy hasonló lehetőség kidolgozása a kombinált analízis megoldásának esetére szintén érdekes kutatási téma lehet.

Az LLVM-ben adott *SSA* forma éppen az adatáram-analízisek egyszerűsítésére szolgál, ahogy a szintén elérhető *DU*- és *UD*-láncok is. Azonban a virtuálisan megszűnő élek és az emiatt virtuálisan elérhetetlen utasítások az általam vizsgált optimalizációkban kioltották a speciális forma adta lehetőségeket. Fontos kérdés lenne megvizsgálni, hogy pontosan milyen módon fogalmazható meg ezen lehetőségek kioltódása, valamint léteznek-e olyan feltételek, melyek mellett az integrált analízisek kihasználhatják ezeket a lehetőségeket az optimalizáció logikájának egyszerűsítésére. Emellett azt is vizsgálni kellene, hogy a keretrendszer milyen kiterjesztései adhatnak jó alapot ezen lehetőségek kihasználására.<sup>4</sup>

A szuperoptimalizációs keretrendszer egyik előnye az önálló transzformációkkal szemben, hogy a legjobb eredmény eléréséhez csak egyszer kell bejárni a programot. Azonban az előre- és visszafelé haladó analízisek külön kezelése ezt a tulajdonságot bizonyos mértékben elrontja.<sup>5</sup> A keretrendszertől kicsit elrugaszkodó, általánosabb kutatási kérdés lehetne az előre- és visszafelé haladó adatáram-analízisek, és ezáltal az előre- és visszafelé haladó integrált analízisek egységesítése, amely lehetővé tenné a jelenleg különböző csoportba tartozó integrált analízisek kombinálását egyetlen kombinált analízisben.

<sup>4</sup> Például a *DU*-láncok közvetlen kihasználásához az integrált analíziseknek át kellene nyúlni a vizsgált utasítástól közvetlen adatfüggésben lévő utasításokba.

<sup>5</sup> Lásd a 4.2.6. megjegyzést.

## 8. fejezet

# Kapcsolódó munkák

A fejezetek elején összefoglaltam az adott fejezethez szorosabban kapcsolódó hivatkozott munkákat, melyeket a teljesség kedvéért itt újra ismertetek. Ezeken kívül néhány további lazábban kapcsolódó munkát is bemutatok.

A fordítóprogramokkal kapcsolatos alapvető ismeretek tömör magyar nyelvű összefoglalását nyújtja Csörnyei Zoltán [3] egyetemi tankönyve, melyből a dolgozatomban használt adatáram-analízis kifejezés is származik. A fordítóprogramokkal kapcsolatos angol nyelvű összefoglalást ad a [11] egyetemi előadáshoz tartozó diasor, mely főleg az optimalizációkra összpontosít.

Az adatáram-analízisek egy régóta vizsgált részét képezik a programok optimalizációjának. Így a rájuk vonatkozó alapvető ismereteket több könyvben is összefoglalták már, mint például a [12, 13] könyvekben. Dolgozatomban azonban az adatáram-analízisek általában használt háló-elméleti definíciója helyett egy általánosabb definíciót használtam, mely lehetővé tette, hogy az integrált analíziseket egyszerűen az adatáram-analízisek általánosításaként értelmezhessem, valamint a megoldás implementációját jól tükröző leírást adjak. A dolgozatomban ismertetett, adatáram-analíziseket hatékonyan megoldó algoritmus megtalálható a hivatkozott könyvekben, azonban az általam megadottnál absztraktabb szinten definiálva. Ezenkívül a megoldó algoritmus futási idejét a hivatkozott könyvekben csak bizonyos konkrét adatáram-analízisek esetére vizsgálták, míg én egy általános adatáram-analízis algoritmus általi megoldásának hatékonyságát bizonyítottam.

Szintén erre az adatáram-analíziseket hatékonyan megoldó algoritmusra építkezett például Forgács István az [5] cikkében, ahol interprocedurális adatáram-analízisek polinomális időben történő megoldására használta fel. Az interprocedurális adatáram-analízisek egy másik hatékony megoldását ismerteti a [14] cikk.

Az adatáram-egyenletek inkrementális megoldása, vagyis annak biztosítása, hogy egy program változtatása után csak azon programrészeket kelljen újraelemezni, me-

lyekre vonatkozó információkat érint a változás, szintén egy többek által vizsgált terület. Egy inkrementális megoldó algoritmust ismertet a [15] cikk. Annak megállapítására, hogy mely programrészeket érint egy változtatás, a hatáselemzés (*Impact Analysis*) különböző módszereit használhatjuk. Egy egész programra kiterjedő hatáselemzés elvégzése nem triviális feladat, éppen ezért különböző módszerek léteznek a megoldására. A létező hatáselemzési módszerekről és azok hatékonyságának összehasonlításáról szól a [16] cikk.

Az optimalizációk iteratív végrehajtása során jelentkező sorbarendeziési probléma feloldásával foglalkozik a [2] cikk, mely egy formális eredményt ad három konkrét optimalizáció kombinálására, azonban nem nyújt általános megoldást.

Ugyancsak a sorbarendeziési probléma megoldásával foglalkozik a [17] diplomamunka, mely a dolgozatomban ismeretetteltől eltérően, nem az iteratív megközelítési módon változtat, hanem azon belül az egyes optimalizációk megfelelő sorrendjét dinamikusan határozza meg. Ehhez egy olyan keretrendszert biztosít, melyben minden optimalizációhoz bizonyos metrikák segítségével meg kell adni, hogy milyen jellegű programokon végez hatékony transzformációt. A keretrendszer a menetek között meghatározza a metrikák értékét a program aktuális állapotára vonatkozóan, és az optimalizációk megadott igényei alapján eldönti, hogy a következő menetben melyiket futtassa a programon.

A dolgozatom fő témáját jelentő szuperoptimalizációs keretrendszerrel foglalkoznak az [1, 7] beszámolók, valamint a [8] cikk és a hozzá tartozó [9] konferencia előadás. Ezekből megtudhatók az iteratív optimalizálás problémái, az integrált analízisek és az általuk meghatározott kombinált analízis definíciói, valamint a segítségével történő optimalizálás megvalósításának néhány részlete. Azonban a kombinált analízis megoldásának pontos definíciója és a megoldás módja nem található meg ezekben a forrásokban. Az optimalizációk kombinálását megvalósító *Vortex* fordítóprogramról szól a [4] cikk, mely pusztán átfogó képet ad a *Vortex* felépítéséről.

Az egyes optimalizációk helyességének bizonyítása egy olyan fontos kérdés, amit általában kézzel szoktak megtenni. Egy megoldást mutat ennek automatizálására a [10] cikk, melyben egy tárgyköri nyelvet<sup>1</sup> adnak az optimalizációk definiálásához. Az ezen nyelven megadott optimalizációkból különböző állításokat generálnak, melyek helyességét egy egyszerű tételbizonyító segítségével ellenőrizni tudják.

<sup>1</sup> A tárgyköri nyelv kifejezést a *Domain Specific Language* angol szakkifejezés magyar megfelelőjeként használom.

## 9. fejezet

### Összegzés

Dolgozatomban bevezettem egy olyan formalizmust, melyben kényelmesen megadható az általam kidolgozott algoritmus implementációját tükröző leírás. A használt jelölésrendszer és fogalmi háttér megadása után a 2. fejezetben ismertettem a fordítóprogramokban általánosan használt optimalizálási módszert, az optimalizációk egymás utáni iteratív végrehajtását. Felvázoltam az optimalizációk ilyen végrehajtásával kapcsolatos problémákat, melyek egyrészt magával a végrehajtással kapcsolatosak, de ami fontosabb, korlátot szabnak az optimalizált program hatékonyságára is. Ez a korlát az egyes optimalizációk szeparáltságából adódik, melyet az optimalizációk *egyszerre működésének* hiánya okoz.

A fejezet végén bemutattam a több különálló optimalizáció logikáját egyesítő szuperoptimalizációk elvét, melyek lehetővé teszik a lehető leghatékonyabb eredmény elérését, azonban kézi összeállításuk gyakorlatilag kivitelezhetetlen feladat. Ugyanakkor a hatékony összeállításukra vonatkozó olyan elméleti megállapításokat tettem, melyeket nem találtam a szakirodalomban.

Ezután a 3. fejezetben a szakirodalomban szokásostól eltérő módon, általánosabban definiáltam az adatáram-analíziseket, melyeket így az integrált analízisek speciális eseteként tudtam értelmezni. Az adatáram-analízis során megoldandó feladat ismertetése után megadtam egy hatékony megoldó algoritmust, mely lényegét tekintve jól ismert az adatáram-analízisek témakörében. Azonban a szokásos absztrakt szinten történő megadással szemben a dolgozatomban egy implementációközelí formalizmussal definiáltam, mely így könnyen átültethető konkrét programozási nyelvekre. Az algoritmusról megmutattam, hogy előre haladó analízisek esetén minimális lépésszámmal működik, és visszafelé haladó analízisek esetén a legjobb algoritmusához képest legrosszabb esetben legfeljebb eggyel többször járja be a ciklusokat.

A 4. fejezetben a [8] cikk alapján, a dolgozatomban használt formalizmushoz igazodva bevezettem az integrált analízisek fogalmát, mint az adatáram-analízisek egy általánosítását. Az integrált analízisek oly módon egyesítik a klasszikusan külön kezelt analíziseket és transzformációkat, hogy a rendelkezésre álló információ alapján az integrált analízis helyettesítési javaslatot tehet egy utasításra. Definiáltam az integrált analízisek által meghatározott kombinált analízist, mely lehetővé teszi az integrált analízisek egyszerre működését, ezzel biztosítva az integrált analízisek formájában megadott optimalizációk olyan automatikus kombinálását, melynek eredménye a szuperoptimalizációkéval azonos hatékonyságú.

A kombinált analízis megoldásának pontos definícióját nem tudtam megadni, mivel a dinamikusan változó adatáram-egyenletek rendszerének formális megadása túlmutat egy tudományos diákköri dolgozat keretein. Azonban a megoldás intuitív definíciójára alapozva felhasználtam az adatáram-analízisek megoldására adott algoritmust, melyet a virtuálisan megszűnő élek, valamint az ezek miatt virtuálisan elérhetetlenné váló utasítások kezelésével egészítettem ki. Ezután megfogalmaztam és indokoltam azt a sejtésemet, hogy az általam kidolgozott algoritmus jól oldja meg a kombinált analízist.

A formális ismertetés után a keretrendszer megvalósításának, mely az LLVM fordítóprogram-eszköztárhoz C++-ban készült, fontosabb jellemzőit a 6. fejezetben ismerttettem. A különböző implementációs megfontolások mellett az LLVM-ben használt *SSA* forma miatti  $\phi$ -utasítások megfelelő kezelése a 4. fejezetben adott algoritmus kiegészítését igényelte.

Az implementált keretrendszer felhasználásával végzett, a D. függelékben ismertett mérési eredmények tanúsága szerint, a *Vortex* fordítóprogramban elért eredményeket sikerült megismételni az LLVM-ben. Így további kutatásoknak nyílt meg a lehetőség a szuperoptimalizálás témakörében egy fejlődő és egyre terjedő platformon.

# A függelék

## További fontos optimalizációk röviden

### A.1. Közös részkifejezések törlése

Nagyon hasznos optimalizáció a közös részkifejezések törlése (*Common Subexpression Elimination*), mellyel biztosítható, hogy a program futása során egy dolog csak egyszer kerüljön kiszámításra. Ehhez meg kell határoznunk a program kifejezéseinek részkifejezéseit, valamint azt, hogy ezen részkifejezésekben található változók mikor kaptak utoljára értéket. Ugyanis ha két kifejezésben található közös részkifejezés, akkor az elsőben lévő részkifejezést kiemelve egy új változóba, az újra felhasználható a második kifejezésben, de csak akkor, ha az adott részkifejezésben szereplő változók által tárolt értékek nem változtak meg a két kifejezés között.

A.1.1. *megjegyzés* (Közös részkifejezések és redundáns adatok törlésére). A közös részkifejezések törlése sokban hasonlít a redundáns adatok törlésére, azonban míg utóbbi az egyes értékekkel foglalkozik, a közös részkifejezések törlése esetén kifejezések azonosságát kell vizsgálni. Ráadásul az optimalizáció transzformációs része nem csak bizonyos utasításokat módosít, hanem kifejezésekből részkifejezéseket vághat ki, melyeket változóhoz kötve a program korábbi részeiben helyez el.

### A.2. Ciklusok optimalizációi

A programokban jellemzően a ciklusokban lévő programrészek kerülnek sokszor végrehajtásra, így célszerű ezeket a lehető leghatékonyabban megvalósítani. Egy viszonylag egyszerű optimalizáció a ciklusmag méretének csökkentésére a ciklusmag invariáns utasításainak kimozgatása a ciklus elé. Ez az optimalizáció a *Loop Invariant Code Motion*. Ehhez meg kell állapítani minden ciklusra, hogy a magjában lévő

utasítások közül melyek azok, amelyek a ciklusmag bármely lefutásakor ugyanazt tesszik, és a ciklusmag lefutására sincsenek hatással. Ilyen lehet tipikusan egy invariáns érték eltárolása egy változóba. Az ilyen utasításokat a ciklus elé lehet helyezni, ezzel a program jelentését megőrizve csökkenthető a ciklusmag mérete, így a program futásának ideje.

A ciklusok másik jellemző optimalizációja, amikor a futási időt a program bináris méretének rováására szeretnénk javítani úgy, hogy a ciklusmagot többször egymás után másoljuk, ezzel csökkentve a programban lévő ugrások számát. Ezt az optimalizációt nevezik *Loop Unrolling*-nek vagy *Loop Unwinding*-nek is.

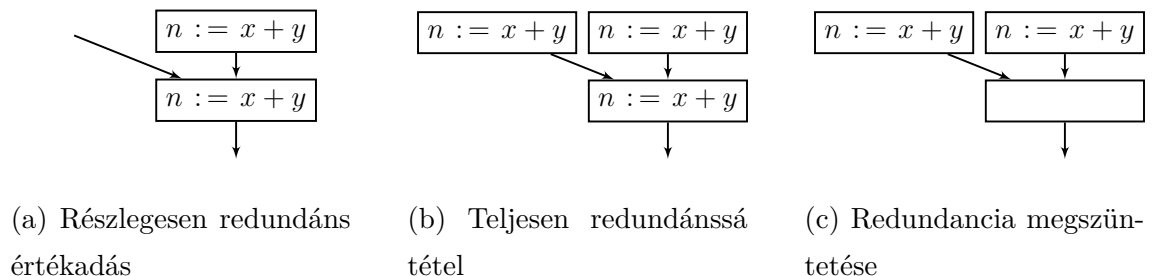
### A.3. Részleges redundancia eltávolítása

Létezik egy optimalizáció, a *Partial Redundancy Elimination*, mely egyszemélyben kiváltja a *Common Subexpression Elimination* és a *Loop Invariant Code Motion* optimalizációkat.

**A.3.1. definíció** (Részleges redundancia). Részlegesen redundánsnak nevezzük azokat az utasításokat, melyek a vezérlés egyik ágán redundánsak, de egy másikon nem.

Az optimalizáció lényege, hogy a részlegesen redundáns utasítást teljesen redundánssá tesszük, tehát bemásoljuk abba az ágba is, amelyben eredetileg nem volt jelen, majd elimináljuk a redundanciát. Ezzel a kifejezés értékét biztosan elérhetővé tesszük a programban a vezérlési utak összefutása után is, ugyanakkor a program bármely végrehajtása során csak egyszer kerül kiértékelésre a kifejezés.

**A.3.1. megjegyzés** (Példa). A részleges redundancia eltávolítását szemlélteti az [A.1.](#) ábra. Egy ciklus invariáns utasításainak kiemelése teljesen analóg módon történik, mivel ciklusok invariáns utasításai részlegesen redundánsak.



A.1. ábra. Részleges redundancia eltávolítása

## A.4. Mutatók optimalizációi

Végül megemlítem az optimalizációk azon körét, melyek a mutatók használatát optimalizálják egy programban. Legtipikusabb példája ennek a csoportnak az *Alias Analysis*, mely mutatók, referenciák esetén képes meghatározni, hogy melyek mutatnak ugyanarra a memóriabeli entitásra. Egész pontosan minden egyes mutatóra meghatározza, hogy a program egyes pontjain hova mutat, majd ezen információ alapján egyszerűen eldönthető, hogy a program adott pontján két mutató ugyanarra az entitásra hivatkozik-e.

Ennek az analízisnek legalább két változata ismert. Az egyik a *May Points-To Analysis*, ami a mutatókhoz egy halmazt rendel, melybe azok az entitások kerülnek, melyekre hivatkozhat a mutató a program adott helyén. Tehát a program végrehajtása során a mutató a halmaz egyik elemére fog ténylegesen hivatkozni, és a különböző végrehajtások során a halmaz más-más elemére hivatkozhat. A másik a *Must Points-To Analysis*, mely csak azokat az eseteket sorolja fel, melyek teljes bizonyossággal fennállnak a program bármely lefutásakor.

A fenti információk felhasználhatók többek között a konstans továbbterjesztés hatékonyabbá tételére, valamint annak vizsgálatára, hogy egy változó értéke megváltozhat-e a program két pontja között.<sup>1</sup>

<sup>1</sup> Ez utóbbi vizsgálat akkor válik igazán érdekessé, amikor lehetőség van paraméterátadás miatti alias-képződésre. Ez azonban az interprocedurális optimalizálás irányába vinne el.

## B függelék

# Előre haladó adatáram-egyenletek megoldó algoritmusai

Ha adott egy adatáram-analízis, akkor az általa meghatározott egyenletrendszer megoldásához különböző módszerek léteznek. Itt előre haladó adatáram-analízisek esetére mutatok négy megoldást a legtriviálisabbtól haladva a leghatékonyabb felé.

A jelen függelékben ismertetett algoritmusok, a vezérlés szerinti algoritmus kivételével, megtalálhatók a [3, 11, 13] hivatkozott irodalmakban. A B.2.1. definícióban megadott vezérlés szerinti algoritmus működését tekintve ekvivalens a [12] 8.7. fejezetében ismertetett megoldással, valamint a [13] 6.3. fejezetében szereplő algoritmussal. Ezek a megoldások azonban kétszer járnak be az utasításokat. Előbbi az első menetben határozza meg a kiszámítandó egyenleteket, mivel az egyenletrendszert alapblokkok szintjén definiálja. Utóbbi pedig az első menetet használja az utasítások bejárásának azon sorrendjének meghatározására, mely az egyenletrendszer hatékony megoldásához szükséges. Az általam kidolgozott változat azonban az adatáram-analízisek 3. fejezetben megadott utasításonkénti definíciója következtében nem igényli az egyenletrendszer külön menet általi pontos meghatározását, továbbá a megfelelő bejárési sorrendet az információk elérhetőségére alapozva határozza meg, így nem igényli a bejárési sorrend előzetes meghatározását sem. Ezen tulajdonságoknak a szuperoptimalizációs keretrendszer implementációjánál lesz jelentősége, ahol a *CFG* a bejárás közben változhat.

## B.1. Jól ismert algoritmusok

**B.1.1. definíció** (Iteratív adatáram-analízis). A legegyszerűbb megoldás során véletlenszerűen kiválasztjuk az egyenletrendszer egy egyenletét és a jobb oldal kiszá-

mításával frissítjük a bal oldalon lévő értéket. Ezt addig folytatjuk, amíg valamely egyenlet jobb oldalának eredménye nem egyezik meg a bal oldal aktuális értékével.

**B.1.1. megjegyzés** (Fixpont keresése). A fenti algoritmus iteratívan pontosítja a kiszámított értékeket mindaddig, amíg el nem ér egy fixpontot. Az algoritmus terminálásához biztosítani kell, hogy a megoldandó egyenletrendszer rendelkezzen bizonyos tulajdonságokkal, melyek biztosítják a fixpont elérését. Ezeket a feltételeket az adatáram-analízisek háló-elméleti definiálásával lehet megfogalmazni. Ez megtalálható a [11] egyetemi kurzus 6. előadásának diáin, valamint a [13] könyvben is. Amennyiben egy adatáram-analízis nem tesz eleget ezeknek a jól ismert feltételeknek, akkor a következőkben ismertetendő algoritmusok nem feltétlen terminálnak. Természetesen a dolgozatomban szereplő példák mind megfelelnek a fixpont eléréséhez szükséges feltételnek.

**B.1.2. megjegyzés** (Adatáram-keretrendszer). A fenti algoritmus léte azt jelenti, hogy tetszőleges adatáram-analízis megoldható egy általános algoritmussal, melybe csak az egyes adatáram-analízisek átmenet- és összefésülő függvényeit kell behelyettesíteni. Egy ilyen általános algoritmus implementációját nevezik *Data Flow Framework*nek.<sup>1</sup>

**B.1.3. megjegyzés** (Fixpont ellenőrzése). A **B.1.1.** definícióban ismertetett algoritmus közvetlen implementációja nem igazán kivitelezhető, mivel a fixpont elérésének ellenőrzéséhez az egymás után véletlenszerűen választott egyenletek nem adnak jó lehetőséget.

**B.1.2. definíció** (Javított iteratív adatáram-analízis). A fixpont elérésének ellenőrzését egyértelműen megvalósító megoldás a **B.1.1.** algoritmus, melyben az  $Eqs$  halmaz az egyenletrendszer egyenleteit tartalmazza.

**B.1.4. megjegyzés** (A javított algoritmus hatékonysága). A fenti algoritmusban már egyszerű módon észleljük az algoritmus terminálását, azonban ez az algoritmus az egyenletek ellenőrzését elég pazarlóan végzi. Amennyiben egyetlen érték változik, úgy az összes egyenletet újra ellenőrzi. Azonban egy  $i$  utasításhoz tartozó  $in_i$  és  $out_{i,j}$  értékek frissítése esetén csak a  $succ(i)$  rákövetkezőkhöz tartozó egyenletek ellenőrzésére lenne szükség.

**B.1.3. definíció** (Listás adatáram-analízis). Az előző megjegyzést figyelembe véve a **B.1.2.** algoritmus adódik.

<sup>1</sup> Bizonyos helyeken magukat az analíziseket, melyeket egy ilyen általános algoritmussal meg lehet oldani, nevezik *Data Flow Framework*nek.

---

**Algorithm B.1.1** Javított iteratív adatáram-analízis

---

```
1:  $Check := Eqs$ 
2: while  $Check$  nem üres do
3:    $Eq \in Check$ 
4:    $Check := Check \setminus \{Eq\}$ 
5:   if  $Eq$  bal oldala nem egyezik meg a jobb oldali kifejezésének értékével then
6:     Az  $Eq$  bal oldalán lévő értéket frissítjük
7:      $Check := Eqs$  // Minden egyenletet újra fogunk ellenőrizni
8:   end if
9: end while
```

---

---

**Algorithm B.1.2** Listás adatáram-analízis

---

```
1:  $Q$  üres sor
2: A belépési pontot betesszük  $Q$ -ba
3: while  $Q$  nem üres do
4:   Kivesszük  $Q$  következő elemét  $i$ -be
5:    $in_i := \mu(\{out_{p,i} | p \in pred(i)\})$ 
6:   for all  $j \in succ(i)$ , amelyre  $out_{i,j} \neq \tau(in_i, i)(j)$  do
7:      $out_{i,j} := \tau(in_i, i)(j)$ 
8:      $j$ -t betesszük  $Q$ -ba
9:   end for
10: end while
```

---

B.1.5. *megjegyzés* (A listás algoritmus hatékonysága). A fenti listás algoritmus jóval hatékonyabb az előző algoritmusnál, azonban az egyes utasításokhoz tartozó egyenleteket feleslegesen sokszor kiszámíthatja. Például a 3.1. ábra utolsó utasítása kétszer is bekerül a listába. Általánosságban azt mondhatjuk, hogy egy utasítás legalább annyszor bekerül a listába, ahány megelőzője van, de legfeljebb annyszor, ahány út vezet a belépési pontból az utasításhoz. Természetesen elég lenne ha a hasonló elhelyezkedésű utasítások csak egyszer kerülnének a listába úgy, hogy a hozzájuk tartozó egyenletek kiszámításakor az adott utasítás összes megelőzőjének egyenlete már ki lenne számítva.

B.1.6. *megjegyzés* (Halmaz használata az algoritmusban). Ha a listás algoritmusban a sor helyett az előző algoritmushoz hasonlóan halmazban tárolnánk az ellenőrizendő utasításokat, akkor az egyes utasítások felesleges ellenőrzéseinek számát csökkenthetnénk, amennyiben már bent lévő utasítást akar az algoritmus a listába helyezni. Azonban az alapvető problémát, hogy egy utasítás egyenleteinek frissítése után minden rákövetkezőt a listába teszünk, nem oldaná meg. Így ha az ellenőrizendő

utasításokat a halmazból szerencsétlen sorrendben vesszük ki, akkor az utasításokat ugyanannyiszor látogatnánk meg, mint a sor használata esetén.

**B.1.7. megjegyzés** (Ciklusok a programokban). Az előző megjegyzésből közvetlenül adódó megoldás, mely csak akkor tesz a listába egy utasítást, amikor minden megelőzőjéből származó információ rendelkezésre áll, nem működne ciklusok esetén, hiszen a visszaélek mentén a ciklusba történő belépéskor nem állhat rendelkezésre az információ. A másik megoldandó probléma a ciklusmagban lévő utasítások egyenletei fixponjának meghatározása. Ugyanis míg a fenti megközelítés a cikluson kívüli utasítások egyenletei esetén biztosítja, hogy azokat csak egyszer kelljen kiszámolni, egy ciklusban a visszaélek mentén továbbított információ hatására a ciklus fejéhez, majd közvetve a ciklusmag minden utasításához tartozó egyenletek újraszámolása válhat szükségessé.

## B.2. Vezérlés szerinti adatáram-analízis

**B.2.1. következmény** (Ciklusok kezelése). Az előbbieket alapján a ciklusokkal kapcsolatban a következőkre kell figyelemmel lenni:

- A ciklus fejének első elérésekor csak a belépő élek információinak meglétét kell ellenőrizni.
- A ciklus fejének a ciklusmag bejárása végén történő elérésekor ellenőrizni kell, hogy a fejhez tartozó egyenlet kiszámításával frissül-e a fej bemenő információja. Ha igen, akkor a ciklusmagot újra be kell járni.
- Mindennek beágyazott ciklusok esetén is működnie kell.

**B.2.1. definíció** (Vezérlés szerinti adatáram-analízis). A fentiek alapján a **B.2.1.** algoritmus adódik, ahol **Headers** a programban lévő fejek halmaza, **Edges** a program *CFG*-jében lévő élek halmaza,  $\text{backedge} : \text{Inst} \times \text{Inst} \rightarrow \mathbb{L}$  logikai függvény, melyre  $\text{backedge}(i, j) \Leftrightarrow (i, j)$  visszaél,  $\text{exitedge} : \text{Inst} \times \text{Inst} \rightarrow \mathbb{L}$  logikai függvény, melyre  $\text{exitedge}(i, j) \Leftrightarrow (i, j)$  kilépő él, valamint  $\text{pred}^* : \text{Inst} \rightarrow \wp(\text{Inst})$  egy utasítás azon megelőzőit adja meg, melyek nem visszaéllel kapcsolódnak az utasításhoz.

**B.2.2. megjegyzés** (A vezérlés szerinti algoritmus működése). A vezérlés szerinti algoritmus bár első ránézésre kissé bonyolultnak tűnik, az utasítások feldolgozására vonatkozó néhány elvvel könnyen megfogalmazható a működése:

- A **B.1.2.** algoritmushoz hasonlóan egy listában tároljuk a feldolgozandó utasításokat.

---

**Algorithm B.2.1** Vezérlés szerinti adatáram-analízis

---

```
1:  $I := \text{Edges} \times \{\downarrow\}$  // Kezdetben egyetlen él mentén sem érhető el információ
2:  $Q$  üres sor
3: A belépési pontot betesszük  $Q$ -ba
4: while  $Q$  nem üres do
5:   Kivesszük  $Q$  következő elemét  $i$ -be
6:   if  $i \in \text{Headers} \wedge \bigwedge_{j \in \text{pred}(i)} I(j, i)$  then // Ciklus fejének nem első elérése
7:     if  $\text{in}_i = \mu(\{\text{out}_{p,i} \mid p \in \text{pred}(i)\})$  then // A ciklus fixpontban van
8:       for all  $(p, q)$  az  $i$ -hez tartozó ciklusból egy szintet kilépő él do
9:         if  $\bigwedge_{r \in \text{pred}(q)} I(r, q) \vee (q \in \text{Headers} \wedge \bigwedge_{r \in \text{pred}^*(q)} I(k, q))$  then
10:           $q$ -t betesszük  $Q$ -ba
11:        end if
12:      end for
13:      continue // Folytassuk a bejárást  $Q$  következő elemével
14:    else // A ciklust újra be kell járni
15:      for all  $(p, q)$  az  $i$ -hez tartozó ciklusmagon belülről induló él do
16:         $I(p, q) := \downarrow$  // Ezek az információk újra kiszámításra kerülnek
17:      end for
18:    end if
19:  end if // Következik az utasítás egyenleteinek feldolgozása:
20:   $\text{in}_i := \mu(\{\text{out}_{p,i} \mid p \in \text{pred}(i)\})$ 
21:  for all  $j \in \text{succ}(i)$  do
22:     $\text{out}_{i,j} := \tau(\text{in}_i, i)(j)$ 
23:     $I(i, j) := \uparrow$ 
24:    if  $\neg \text{exitedge}(i, j) \wedge (\bigwedge_{k \in \text{pred}(j)} I(k, j) \vee (j \in \text{Headers} \wedge \neg \text{backedge}(i, j) \wedge \bigwedge_{k \in \text{pred}^*(j)} I(k, j)))$  then // Kilépő élen csak fixpontban propagálunk
25:       $j$ -t betesszük  $Q$ -ba
26:    end if
27:  end for
28: end while
```

---

- Egy utasításhoz tartozó egyenletek kiszámítása után a bejárhatóvá váló rákövetkezőket betesszük a listába.
  - A cikluson kívüli, vagy a feldolgozott utasítással egy ciklusban lévő rákövetkezőket akkor tesszük a listába, ha minden bemenő élükön elérhető információ.
  - Egy ciklus fejét, annak a cikluson kívülről történő elérése esetén, akkor tesszük a listába, ha minden nem-visszaél bemenő élén rendelkezésre áll az információ.
- Cikluson belüli utasítások esetén a külső utasításokhoz nagyon hasonlóan járunk el. Fontos különbség azonban, hogy a ciklus belsejében induló élek mentén megjelenő információk nem feltétlenül vannak fixpontban, ezért a kimenő élek mentén nem ellenőrizzük az utasítások bejárhatóvá válását. Ezt az algoritmus 24. sorában lévő feltétel első tagja biztosítja.
- Amikor egy ciklus fejénél az összes visszaélen rendelkezésre áll az információ, akkor a fejet újra betesszük a listába.
- Amikor egy ciklus  $h$  fejét nem először látogatjuk meg, vagyis már a befutó visszaélek mentén is elérhető információ, akkor ellenőrizzük, hogy  $in_h$  megegyezik-e a befutó információk aktuális értékeinek  $\mu$  összefésülő függvény által szolgáltatott eredményével. Ehhez a feltételhez tartozik az algoritmus 6-19. sorai között található elágazás.
  - Ha  $in_h$  értéke stimmel, akkor a ciklusmag egyenletei fixpontban vannak. Ekkor az egy szintnyit kilépő éleken<sup>2</sup> elérhető utasítások bejárhatóságát ellenőrizzük, és a bejárható utasításokat betesszük a listába. A kilépő élek feldolgozása után a lista következő elemét vesszük, és azzal folytatjuk a feldolgozást.
  - Ha  $in_h$  értékét frissíteni kell, akkor az egész ciklust újra be kell járni, ezért a ciklusmagon belül induló minden élen töröljük az információ elérhetőségét jelző értéket. Ebben az esetben a ciklus fejéhez tartozó egyenletek újbóli feldolgozásával folytatódik az algoritmus.

<sup>2</sup> Beágyazott ciklusok esetén fontos, hogy csak az eggyel kintebb lévő szintre irányuló információkat tekintsük fixpontban lévőnek, hiszen ha a befoglaló ciklus még nincs fixpontban, akkor a beágyazott ciklus információi is csak a befoglaló ciklus aktuális állapotára nézve kerültek fixpontba. Azaz a befoglaló ciklus újraszámolásakor a beágyazott ciklus egyenleteit is újra kell számolni.

B.2.3. *megjegyzés* (Nem elérhető utasítások a vezérlés szerinti algoritmus esetén). Ha a vezérlés szerinti algoritmust használjuk olyan program elemzésére, melyben nem elérhető utasítások is vannak, akkor a 3.2.1. megjegyzésben jelzett pontatlanságnál is rosszabb a helyzet. Ugyanis ha egy nem elérhető utasítás rákövetkezője egy elérhető, akkor az az elérhető utasítás sosem fog bekerülni a listába, mivel a nem elérhető utasításból érkező élén sosem lesz elérhető információ. Így az algoritmus anélkül megáll, hogy minden elérhető utasítás egyenletét megoldaná. Emiatt dolgozatomban további részében külön jelzés nélkül is program alatt olyan programot értek, melyben minden utasítás elérhető.

B.2.4. *megjegyzés* (A vezérlés szerinti algoritmus helyessége és lépésszáma). A B.2.1. definícióban megadott algoritmus helyességére és lépésszámára vonatkozó megállapítások és azok bizonyításai a C. függelékben találhatók meg.

B.2.5. *megjegyzés* (A vezérlés szerinti algoritmus problémái visszafelé haladó analízisek esetén). Egyrészt a visszafelé haladó analízisek egyenleteinek megoldásához fel kell cserélni a *pred* és *succ* függvényeket a gráf éleinek képzeletbeli megfordítása miatt. Másodszor a ciklusok kezelésének kérdését kell megoldani, amihez csak a már használt elvek pontosabb megfogalmazása szükséges.

Az előre haladó elemzés során a ciklusokba mindig azok fején keresztül juthatunk be, így elég egyszerűen csak a ciklusok fejeire figyelni, hogy ellenőrizhessük egy ciklus bejárhatóságát. Azonban az élek irányának megfordításával az eredetileg kilépő élek mentén jutnak információk a ciklusba, vagyis nem elégséges a ciklusok fejét figyelni. Ehelyett egy belépő élen elérhetővé vált információ esetén annak eldöntéséhez, hogy a ciklus bejárhatóvá vált-e, azaz minden kívülről jövő információja elérhető, a ciklusba a megfordított irányítás szerint belépő éleket kell ellenőrizni. Ha a ciklus bejárhatónak bizonyul, akkor a fejét betehetjük a listába, hogy onnan induljon a megfordított élek mentén az utasításainak bejárása.

A fenti megfontolások figyelembe vételével adódik az itt ismertetett vezérlés szerinti algoritmus újraformalizálásával a 3.2.1. definícióban adott algoritmus.

## C függelék

# A vezérlés szerinti adatáram-analízis helyessége és lépésszáma

Ez a függelék tartalmazza a 3.2.2. tétel bizonyítását, melyhez több lemmát használok fel. Először az előre haladó analízisek esetére bizonyítom az állítást, majd megmutatom, hogy az állítások visszafelé haladó esetben is megállják a helyüket.

A [13] könyv tartalmaz az ott ismertetett algoritmus helyességére és lépésszámára vonatkozó megállapításokat. Az általam kimondott állítás azonban két részben is eltér az ott ismertetettektől:

- Az előre haladó analízisek mellett visszafelé haladó analízisek esetére is bizonyítom a helyességet,
- nem konkrét analízisek esetére vonatkozó lépésszámot bizonyítok, hanem tetszőleges analízis megoldásának hatékonyságát mondom ki.

### C.1. Állítások előre haladó analízisek esetén

**C.1.1. lemma** (A vezérlés szerinti algoritmus helyessége ciklusmentes programokra). *A vezérlés szerinti adatáram-analízis előre haladva minimális lépésszámmal fixpontba jut olyan programok esetén, melyek nem tartalmaznak ciklust.*

*Bizonyítás.* Egy ciklust nem tartalmazó programban az algoritmus egy utasítást csak egyszer helyez a listába, és az egyenletek kiszámolásakor fixpontban lévő információt határoz meg. Ezt a belépési ponttól az utasításhoz vezető leghosszabb út hosszára vonatkozó indukcióval lehet belátni:

$n = 0$  : Kezdetben a belépési pont ( $e$ ) a listába kerül. Erre  $\text{pred}(e) = \emptyset \Rightarrow \text{in}_e = \mu(\emptyset)$ , vagyis  $\text{in}_e$  értéke konstans. Tehát a segítségével meghatározott  $\text{out}_{e,i}$  értékek sem változhatnak. Továbbá mivel  $\text{pred}(e) = \emptyset$ , a belépési pont nem is kerülhet többet a listába.

$n \rightarrow n + 1$  : Amikor egy, a belépési ponttól  $n$  távolságra lévő  $i$  utasítás egyenleteinek fixpontbeli értékét az egyenletek egyszeri kiszámításával meghatároztuk, akkor minden  $j$  rákövetkezőjéhez beállítjuk  $\text{out}_{i,j}$  értékét, mely ezekszerint már nem fog változni. Emellett ha ezzel egy  $j$  rákövetkező  $\text{in}_j$  értékének meghatározásához szükséges összes  $\text{out}_{k,j}$  érték rendelkezésünkre áll, akkor  $j$ -t a listába tesszük. Ez csak úgy történhet meg, hogy  $\forall k \in \text{pred}(j)$  utasítás egyenleteit már meghatároztuk, melyek az indukciós feltevés alapján már fixpontban lévő információt szolgáltatnak. Ezért amikor a  $j$  utasítást majd kivesszük a listából, akkor biztos, hogy az  $\text{in}_j$  értéket meghatározó egyenlet kiszámításával fixpontban lévő információt kapunk. Tehát ha egy utasítást a listába helyezünk, akkor a hozzá tartozó egyenletek egyszeri kiszámításával fixpontban lévő információhoz jutunk. Ugyanakkor az utasítás nem is kerülhet többször a listába, mivel az indukciós feltevés miatt a megelőző utasításai is csak egyszer kerülhetnek bele.

Továbbá az információk meghatározásához minden egyenletet legalább egyszer ki kell számolni, így az előző induktív bizonyítással beláttuk az állítást.  $\square$

**C.1.2. lemma** (Ciklusok kiemelhetősége a vezérlés szerinti algoritmus mellett). *Egy program külső szintű ciklusai egyetlen csúccsal, mely öröklí a ciklus be- és kilépő éleit, helyettesíthetők a végrehajtási gráfban úgy, hogy a vezérlés szerinti algoritmus eredménye helyes marad.*

*Bizonyítás.* Legyen  $h$  egy külső szintű ciklus feje. Amikor az algoritmus szerint  $h$  először bekerül a listába, akkor minden belépő éléhez tartozó információ rendelkezésre áll. Mivel egy ciklusba csak a fején keresztül kerülhet a vezérlés, ezért ez azt is jelenti, hogy a ciklus minden belépő élén rendelkezésre áll az információ. Tehát ha egy ciklus feje bekerült a listába, akkor a ciklushoz tartozó egyenletek fixpontjának meghatározásához csak a ciklusmag egyenleteinek kiszámítása szükséges.

Végül az algoritmus a kilépő élek mentén csak akkor teszi elérhetővé az információkat, amikor azok fixpontba kerültek.

Tehát a ciklus egyenletei fixpontjának megtalálását egy egységként kezelhetjük, és a ciklust a CFG-ből kiemelve, azt egyetlen csúccsal, mely a ciklus be- és kilépő éleit öröklí, helyettesítve<sup>1</sup> ciklusmentes programot kapunk. Az ilyen programok esetén pedig a C.1.1. lemma értelmében az algoritmus minimális lépésszámmal képes megoldani az adatáram-analízist.  $\square$

<sup>1</sup> A hozzátartozó összes adatáram-egyenlet megoldását virtuálisan egy egységként kezelve.

C.1.1. *következmény* (Beágyazott ciklust nem tartalmazó ciklusok kezelése elégséges). Egy kiemelt ciklus tartalmazhat beágyazott ciklusokat, azonban az előző lemma értelmében ezek a ciklusok is kiemelhetők, tehát a ciklusbeli egyenletek megoldásának vizsgálatához elegendő a beágyazott ciklust nem tartalmazó ciklusokat figyelembe venni.

**C.1.3. lemma** (Kiemelt ciklus egyenleteinek megoldása). *Egy beágyazott ciklust nem tartalmazó kiemelt ciklushoz tartozó előre haladó egyenletrendszert a fejhez tartozó adott bemenő információval az algoritmus minimális lépésszámmal megold.*

*Bizonyítás.* Egy kiemelt ciklusból a visszaéleket elhagyva új CFG-t kapunk, melynek belépési pontja a ciklus feje lesz, a kilépési pontokhoz tartozó kimenő információk pedig igazából a belépési pont bemenő információi. A C.1.1. lemma bizonyításának gondolatmenetét követve triviális, hogy a kiemelt ciklus módosított gráfját a vezérlés szerinti algoritmus adott belépési ponthoz tartozó bemenő információ esetén minimális lépésszámmal képes megoldani. Amikor ez megtörtént, akkor a kilépési pontokhoz tartozó  $out_q$  információk rendelkezésre állnak.

Mivel az eredeti ciklusban ezek az információk a visszaélek mentén a ciklus fejének bemenő információjának részét képezik, ezeket továbbítani kell a kiemelt CFG  $h$  belépési pontjának  $in_h$  információja felé a hozzá tartozó adatáram-egyenlet újbóli megoldásával. Ha az egyenlet újbóli megoldása nem változtatja meg  $in_h$  értékét, akkor a ciklus egyenleteinek ismételt megoldása az egyenletek ugyanazon megoldását eredményezné<sup>2</sup>, ezért ekkor az információk fixpontba kerültek. Ha azonban  $in_h$  megváltozik, akkor a kiemelt gráf egyenleteinek újraszámolása megváltoztathatja az információkat. Így  $in_h$  frissítése után a belépési ponthoz tartozó frissített bemenő információkkal újra meg kell oldani a ciklushoz tartozó egyenletrendszert.

A vezérlés szerinti adatáram-analízis algoritmus a fenti gondolatmenetet követi, így az valóban meg fog találni egy fixpontot<sup>3</sup>. Már csak a ciklus megoldásának minimális lépésszáma kérdéses. A korábbi eredmények alapján a kiemelt CFG egyszeri megoldását az algoritmus minimális lépésszámmal megoldja. Az, hogy a ciklusmag kiemelt gráfjának  $in_h$  információját hányszor kell frissíteni, és ezáltal hányszor kell a kiemelt gráfhoz tartozó egyenleteken ismételten végigmenni, a konkrét adatáram-analízistől függ. Mivel az algoritmus a ciklus bejárásának minden menetében frissíti az összes információt, egy menet alatt az eredmény a lehető leginkább közelíti a fixpontot.<sup>4</sup> Valamint a gráf egy megoldása során nem csak az információk finomítása

<sup>2</sup> Szintén a C.1.1. lemma bizonyításának gondolatmenetéből következően.

<sup>3</sup> Lásd a 3.2.8. megjegyzést.

<sup>4</sup> Ennek formális bizonyításához az adatáram-analízisek háló-elméleti definíciójára lenne szükség, amire dolgozatomban nem térek ki.

történik, hanem a fixpont elérésének ellenőrzése is, mivel a megoldás végén az  $in_h$  információ ellenőrzésre kerül. Mivel ez az ellenőrzés minden menet végén megtörténik, az algoritmus egyből észreveszi, ha az információk fixpontba kerültek, így nem végez felesleges menetet.  $\square$

**C.1.4. tétel** (A vezérlés szerinti algoritmus helyessége és lépésszáma előre haladó analízisekre). *A vezérlés szerinti adatáram-analízis algoritmusára előre haladó analízis esetén minimális lépésszámmal fixpontba jutó iteratív algoritmus.*

*Bizonyítás.* Amennyiben egy program nem tartalmaz egyetlen ciklust sem, úgy a C.1.1. lemma értelmében az algoritmus minimális lépésszámmal megoldja a programhoz tartozó egyenletrendszert.

Ha egy program tartalmaz ciklusokat, akkor a külső szintről ezek a C.1.2. lemma értelmében kiemelhetők, és az így redukált CFG-t az algoritmus képes minimális lépésszámmal megoldani. A ciklusokhoz tartozó egyenletek megoldásánál a C.1.1. következmény miatt elég az olyan kiemelt ciklusokat vizsgálni, melyek nem tartalmaznak beágyazott ciklust.

Amikor az algoritmus egy ciklus megoldásába kezd<sup>5</sup>, akkor rendelkezésére áll a ciklus belépő éleinek fixpontbeli információja. A C.1.3. lemma szerint ezzel az információval a kiemelt ciklust az algoritmus minimális lépésszám mellett megoldja.

A fentiek alapján a vezérlés szerinti adatáram-analízis algoritmusra tetszőleges programot képes minimális lépésszámmal elemezni.  $\square$

## C.2. Állítások visszafelé haladó analízisek esetén

C.2.1. *megjegyzés* (A korábbi bizonyítások visszafelé haladó analízisek esetén).

A C.1.4. tétel bizonyításához felhasznált állítások a C.1.3. lemma kivételével<sup>6</sup> a vezérlés szerinti algoritmusra visszafelé haladó analízisek esetén is igazak. Ehhez csak a következőket kell figyelembe venni:

- A C.1.1. lemma bizonyításában a belépési ponttól vett távolságra vonatkozó indukció helyett a leghosszabb úton elérhető kilépési ponttól vett távolság szerinti indukciót kell alkalmazni.
- A C.1.2. lemma bizonyításában a ciklus fejébe a cikluson kívülről bemenő élek helyett a ciklusba belépő éleken lévő információkról kell beszélni.

<sup>5</sup> A ciklus fejét először helyezi a listába.

<sup>6</sup> A lemma átfogalmazott állítását a C.2.1. lemma mondja ki.

- A C.1.1. következmény állítása a C.1.2. lemma előbbiek szerint pontosított bizonyítása miatt fennáll visszafelé haladó analízisek esetén is.

**C.2.1. lemma** (Kiemelt ciklus egyenleteinek megoldása visszafelé haladva). *Egy beágyazott ciklust nem tartalmazó kiemelt ciklushoz tartozó egyenletrendszert a vezérlés szerinti algoritmus visszafelé haladó adatáram-analízis esetén adott belépő információk mellett legrosszabb esetben is a legjobb algoritmusnál legfeljebb a kiemelt gráffal eggyel többszöri bejárásával megold.*

*Bizonyítás.* Az eredeti C.1.3. lemma bizonyításának helyességre vonatkozó részei alkalmazhatók ebben az esetben is. Ehhez csak azt kell figyelembe venni, hogy a ciklusba való belépés után a ciklus fejében lévő aktuális információ a kiemelt gráf-ból törölt visszaélek mentén továbbítódik a kiemelt gráf kilépési pontjaihoz, így elkezdődhet a ciklusmentes CFG visszafelé haladó elemzése. Ezután a ciklus fejéhez visszaérve a fixpont ellenőrzése és szükség esetén a ciklus újbóli bejárása magától értetődően adódik.

A C.1.2. lemma bizonyításának minimális lépésszámmra vonatkozó részében azt használtuk ki, hogy az előre haladó analízisek esetén az információ a ciklus fejtől, mint a ciklus egyértelmű belépési pontjától terjed a ciklus utasításain egészen a visszaélekig. Így a ciklus megoldásának egy-egy menetének minden lépése az éppen rendelkezésre álló legpontosabb információval számolja újra az utasítások egyenleteit. Visszafelé haladó analízisek esetén viszont a belépő élek a ciklus több utasításának egyenletéhez is szolgáltathatnak bejövő információt. Tehát a ciklus bejárásának első menetében a fej és az első belépő éllel rendelkező utasítás között az egyenletek üres információkkal kerülnek kiszámításra. Vagyis ezek a lépések lényegében feleslegesek a fixpont közelítése szempontjából.

Tehát az algoritmus az első menetben tehet felesleges lépéseket. Egy jobb algoritmus ezeket a lépéseket nem tenné meg, vagyis az első olyan utasítástól indítaná a visszafelé haladó bejárást, amely rendelkezik belépő információval. Ugyanakkor a fixpont ellenőrzésének helyességéhez a 3.2.4. megjegyzés alapján a fejtől kell indítani a bejárást. Ha mégsem a fejtől indítjuk a bejárást, akkor biztosítani kell egy teljes menetet a helyességhez. Azaz ez a jobb algoritmus kénytelen egy második, immár teljes menetet is tenni a kiemelt gráfon.

Ekkor két eset lehetséges: a második menetre szükség volt az információk pontosításához, vagy az információk már fixpontban voltak a részleges menet után is. Előbbi esetben a vezérlés szerinti algoritmus is ugyanazt teszi a második menettől kezdődően, tehát csak az első menetbeli felesleges lépésekkel dolgozik többet. Ez azt jelenti, hogy legfeljebb eggyel többször járja be a kiemelt gráfot. A második esetben a vezérlés szerinti algoritmus már az első menet után megáll, mivel az egy

teljes menet, így biztos döntést hozhat a fixpont eléréséről. Ebben az esetben tehát a jobbnak tervezett algoritmus a teljes második menetet feleslegesen teszi meg, vagyis a vezérlés szerinti algoritmus igazából kevesebb lépéssel találja meg a fixpontot.  $\square$

**Tétel** (A vezérlés szerinti adatáram-analízis helyessége és lépésszáma (3.2.2. tétel)). *Tetszőleges (a B.1.1. megjegyzésben hivatkozott feltételnek megfelelő) adatáram-analízis esetén a vezérlés szerinti algoritmus fixpontba jut. Előre haladó analízis esetén ezt minimális lépésszámmal teszi, visszafelé haladó analízis esetén pedig a legrosszabb esetben is az elképzelhető legjobb algoritmushoz képest legfeljebb eggyel többször járja be a ciklusokat.*

*Bizonyítás.* A C.1.4. tétel értelmében az állítás előre haladó analízisekre vonatkozó része igaz.

A visszafelé haladó analízisek esetén követhető a C.1.4. tétel bizonyításának gondolatmenete, mivel az abban felhasznált állítások a C.2.1. megjegyzés értelmében a visszafelé haladó analízisek esetén is használhatók. A gondolatmenet ciklusok megoldására vonatkozó részében a C.1.3. lemma helyett a C.2.1. lemma használandó. A lépésszámmra vonatkozó állítást a C.2.1. lemma biztosítja.  $\square$

C.2.2. *megjegyzés* (Erősebb állítás visszafelé haladó analízisek megoldására). A visszafelé haladó analízisek megoldása lépésszámának átlagos esetére az algoritmus hatékonyságát statisztikai eszközökkel lehetne bizonyítani. A bizonyításhoz az elképzelhető ciklusok és az elképzelhető analízisek tulajdonságai alapján kellene dolgozni. Annyi viszonylag könnyen megállapítható, hogy az algoritmus átlagos esetben a ciklusok várható utasításszámának felényi lépést tesz meg feleslegesen.<sup>7</sup> A C.2.1. lemma bizonyításában vázolt alternatív kezdőpontú algoritmus viszont láthattuk, hogy a szükséges második menetet teheti meg teljesen feleslegesen. A kérdés, hogy mekkora valószínűséggel lesz a második menet ténylegesen felesleges. Ez a valószínűség pedig nem csak az elképzelhető ciklusoktól, de az elképzelhető analízisektől is függ.

Ezt a valószínűséget nem tudtam kiszámolni, de ha 50% lenne, akkor a két algoritmus különbsége várhatóan pont kioltaná egymást. Azaz nem létezne olyan algoritmus, mely egy véletlenszerű programot egy véletlenül választott analízissel várhatóan kevesebb lépésben elemezne.

<sup>7</sup> A ciklus feje és az első belépő éllel rendelkező utasítás közti rész az első bejárás során felesleges, csak az üres információval dolgozik. Az érintett utasításokról könnyen belátható, hogy mennyiségük a várható utasításszám fele.

# D függelék

## A keretrendszer értékelése

Ebben a függelékben alapvetően az eredeti LLVM optimalizációk és az általam megvalósított keretrendszer hatékonyságát kívánom összehasonlítani. Azonban viszonyítási alapként az optimalizálatlan programok futási idejét is megmértem. Ráadásul, ahogy az a következőkből kiderül, az iteratív optimalizálás kétféle összeállításának mérésére is szükség volt.

### D.1. A mérés menete

A mérésekhez a következő programokat használtam fel:

**queens.** N-királynő probléma megoldása  $n \times n$  méretű sakktáblán.

**msort.** Összefésüléses rendezés  $n$  véletlenszerűen választott elemből álló tömbön.

**qsort.** Gyorsrendezés  $n$  véletlenszerűen választott elemből álló tömbön.

**fft.** Gyors Fourier-transzformáció  $n$  véletlenszerűen választott alappont felett.

**life.** A Conway-féle életjáték  $n \times m$  méretű játéktéren  $k$  lépésen keresztül.

**crc32.** 32-bites CRC ellenőrző összeg számolása egy szöveges fájlra.

**huffman.** Egy szöveges fájl Huffman-kódjának kiszámítása.

A fenti programokat C nyelven a *Rosetta Code* [22] nevű forráskód-gyűjteményből töltöttem le, majd az LLVM-hez készült C-fordító, a clang segítségével fordítottam.

Első lépésben a C-forrásból clang -O0 parancs segítségével LLVM-kódot készítettem, majd négy különböző mérést hajtottam végre. A méréseket a programon belül, a *Performance Application Programming Interface* [23] felhasználásával végeztem,

amely a minél pontosabb mérés érdekében a processzorban lévő számlálókat használja az eltelő idő meghatározására. A mérések az optimalizálatlanul<sup>1</sup> generált LLVM-kód továbbfordításával előállt futtatható állományokkal készültek. A következő négy összeállítást mértem minden program esetén:

**optimalizálatlan.** Az LLVM-kód továbbfordítása clang -O0 paranccsal futtatható állománnyá.

**iteratív.** Az LLVM-kód optimalizálása az opt -inline -early-cse -constprop -simplifycfg -dce paranccsal, majd futtatható állomány készítése clang -O0 utasítással.

**iteratív II..** Az LLVM-kód optimalizálása az opt -inline -constprop -simplifycfg -dce paranccsal, majd futtatható állomány készítése clang -O0 utasítással.

**keretrendszer.** Az LLVM-kód optimalizálása az opt -so paranccsal<sup>2</sup>, majd futtatható állomány készítése clang -O0 utasítással.

A generált programokat egy Intel Xeon E7530 (1,86 GHz) processzorokat tartalmazó szerveren futtattam külön-külön. Minden program esetén 128 futási eredmény alapján került megállapításra a futási idők átlaga és szórásnégyzete.

## D.2. A mérési eredmények és értékelésük

A pontos mérési eredmények a D.1. táblázatban láthatók, valamint a D.1. ábra diagramja mutatja minden program futási idejét az optimalizálatlan futási időkhöz viszonyítva, az egyszerűbb összehasonlíthatóság kedvéért.

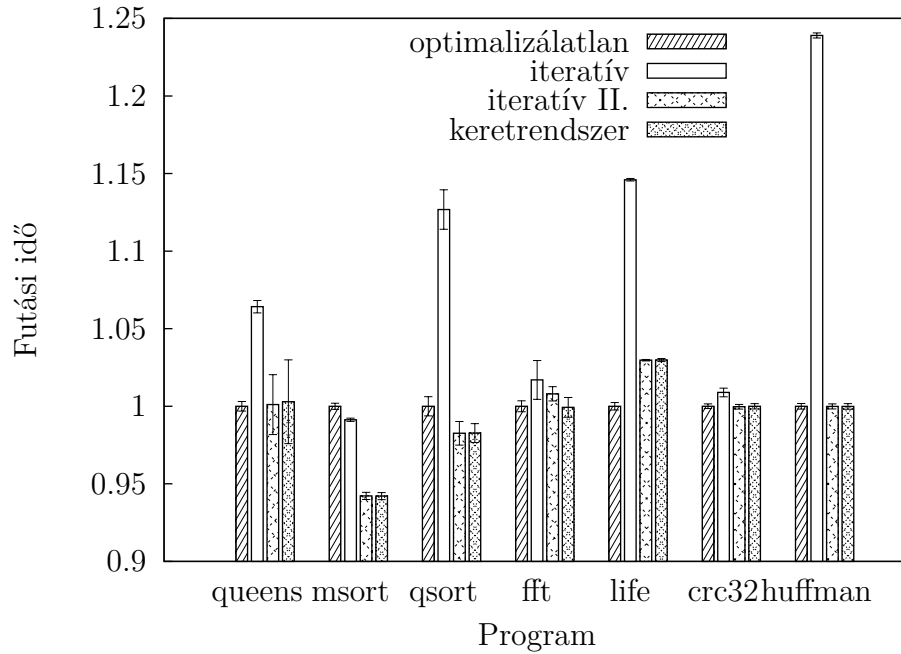
Az eredményekből látható, hogy az optimalizálatlan eredményhez képest az iteratív eredmény, az msort kivételével, mindig rosszabb. Ráadásul ehhez képest a szuperoptimalizációs keretrendszer – mely ugyanazokat az optimalizációkat alkalmazza – által optimalizált program valóban sokkal jobb eredményt produkál. Azonban vegyük figyelembe, hogy az *iteratív II.* esetben már egyből eltűnik a keretrendszer nagy előnye. Tehát felmerül a gyanú, hogy talán az LLVM early-cse optimalizációjának megvalósításával van probléma<sup>3</sup>. Mindenesetre az iteratív<sup>4</sup> eset lassulásának oka,

<sup>1</sup> Legalábbis az -O0 kapcsoló miatt a legkevésbé optimalizált formában generált.

<sup>2</sup> Az opt utasítás -so kapcsolóján keresztül elérhető az általam készített keretrendszer az összes implementált integrált analízist használva. (Konstans továbbterjesztés, elérhetőség-vizsgálat, redundáns adatok törlése, függvény beágyazása, nem használt értékadások törlése.)

<sup>3</sup> Ennek a felvetésnek a részletes vizsgálatára már nem volt időm.

<sup>4</sup> Tehát az early-cse optimalizációt is alkalmazó eset.



D.1. ábra. Mérési eredmények az optimalizálatlan futási időkhöz normálva

hogy a generált assembly kódban a függésben lévő utasítások egymás mellé kerültek, így a processzor nem tudta kihasználni a futószalagja nyújtotta lehetőségeket, vagyis nem tudta az utasításokat párhuzamosan végrehajtani.<sup>5</sup>

A fentiek miatt az *iteratív II.* esetet hasonlítom össze a keretrendszer által produkált eredményekkel. Ekkor megállapítható, hogy általános esetben a szuperoptimalizációs keretrendszer és az LLVM iteratív optimalizációja lényegében ugyanazt az eredményt nyújtja. Néhány százalékos eltérések kimutathatók az egyes esetekben hol az egyik, hol a másik megoldás javára, de ezek összességében kioltják egymást. Ugyanakkor fontos látni, hogy ezek a mérési eredmények teljes mértékben összeeszenegnek a [8] cikkben leírtakkal.

Annak ellenére, hogy a jelenleg elérhető eredmények alapján ez a megközelítés nem tud szignifikánsan jobb lenni a hagyományos megközelítésnél, mégis érdemes elgondolkodni a használatán, hiszen míg az iteratív megközelítés esetén bizonyos optimalizációk ronthatnak a generált program futási teljesítményén, addig a szuperoptimalizációs keretrendszer mindig a legjobb alternatívát választja ki az optimalizációk által felkínált lehetőségek közül. Mindemellett az optimalizációk integrált analízisként történő megvalósítása sem jelent túlzott extra költséget a hagyományos megoldáshoz képest.

<sup>5</sup> A generált assembly-kódra vonatkozó információ forrása: Antypin Artyom Szergejevics, személyes kommunikáció, 2011. november 30.

Például a GHC fordítóprogram készítői is a [8] cikkben ismertetett megoldás megvalósításán gondolkoznak a [24] weboldal tanúsága szerint, amely hivatkozást is tartalmaz egy funkcionális megvalósításra. Bár ott konkrét mérési eredmények nem érhetőek el, de a lelkesedés alapján feltételezhető, hogy más optimalizációkkal, esetleg funkcionális programok esetén valóban lényegesen jobb eredmény érhető el a szuperoptimalizációs módszer használatával.

| Program | Paraméterek            | Optimalizáció    | Átlag ( $\mu s$ ) | Szórás ( $\mu s$ ) |
|---------|------------------------|------------------|-------------------|--------------------|
| queens  | 14                     | optimalizálatlan | 2361030           | 7402               |
|         |                        | iteratív         | 2512520           | 9529               |
|         |                        | iteratív II.     | 2363540           | 45491              |
|         |                        | keretrendszer    | 2367980           | 64201              |
| msort   | 2000000                | optimalizálatlan | 785198            | 1646               |
|         |                        | iteratív         | 778342            | 633                |
|         |                        | iteratív II.     | 739827            | 1808               |
|         |                        | keretrendszer    | 739746            | 1808               |
| qsort   | 2000000                | optimalizálatlan | 524407            | 3251               |
|         |                        | iteratív         | 590905            | 6670               |
|         |                        | iteratív II.     | 515275            | 3969               |
|         |                        | keretrendszer    | 515322            | 3213               |
| fft     | 131072                 | optimalizálatlan | 161564            | 567                |
|         |                        | iteratív         | 164332            | 2021               |
|         |                        | iteratív II.     | 162861            | 744                |
|         |                        | keretrendszer    | 161448            | 1030               |
| life    | 512 512 1000           | optimalizálatlan | 34431300          | 81023              |
|         |                        | iteratív         | 39465100          | 26635              |
|         |                        | iteratív II.     | 35454900          | 17502              |
|         |                        | keretrendszer    | 35458800          | 35491              |
| crc32   | 302.4 MB-os szövegfájl | optimalizálatlan | 2713310           | 4238               |
|         |                        | iteratív         | 2737650           | 7503               |
|         |                        | iteratív II.     | 2712600           | 4008               |
|         |                        | keretrendszer    | 2713360           | 4593               |
| huffman | 302.4 MB-os szövegfájl | optimalizálatlan | 1524270           | 2565               |
|         |                        | iteratív         | 1888700           | 2475               |
|         |                        | iteratív II.     | 1524210           | 2487               |
|         |                        | keretrendszer    | 1524130           | 2788               |

D.1. táblázat. Mérési eredmények 128 futásból számítva

# Irodalomjegyzék

- [1] C. Chambers, J. Dean, D. Grove: *Frameworks for Intra- and Interprocedural Dataflow Analysis*, Technical Report, University of Washington Computer Science & Engineering, 1996.  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.100.5032>
- [2] C. Click, K. D. Cooper: *Combining analyses, combining optimizations*, In: ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 17 Issue 2, March 1995, Pages 181-196, ACM New York, USA, ISSN: 0164-0925, EISSN: 1558-4593. DOI [10.1145/201059.201061](https://doi.org/10.1145/201059.201061)
- [3] Z. Csörnyei: *Fordítóprogramok*, Typotex Kiadó, Budapest, 2006, ISBN: 963 9548 839.
- [4] J. Dean, G. DeFouw, D. Grove, V. Litvinov, C. Chambers: *Vortex: an optimizing compiler for object-oriented languages*, In: OOPSLA '96 Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Pages 83-100, ACM New York, USA, 1996, ISBN: 0-89791-788-X. DOI [10.1145/236337.236344](https://doi.org/10.1145/236337.236344)
- [5] I. Forgács: *Double Iterative Framework for Flow-sensitive Interprocedural Data Flow Analysis*, In: ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 3 Issue 1, Jan. 1994, Pages 29-55, ACM New York, USA, ISSN: 1049-331X, EISSN: 1557-7392. DOI [10.1145/174634.174635](https://doi.org/10.1145/174634.174635)
- [6] C. Lattner: *Introduction to the LLVM Compiler System*, Presentation for ACAT 2008: Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy, Nov 2008.  
<http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.html>
- [7] S. Lerner, D. Grove, C. Chambers: *Composing Dataflow Analyses and Transformations*, Technical Report, University of Washington Computer Science & Engineering, 2001.  
<http://cseweb.ucsd.edu/~lerner/UW-CSE-01-11-01.pdf>

- [8] S. Lerner, D. Grove, C. Chambers: *Composing Dataflow Analyses and Transformations*, In: POPL '02 Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 270-282, ACM New York, USA, 2002, ISBN: 1-58113-450-9. DOI [10.1145/503272.503298](https://doi.org/10.1145/503272.503298)
- [9] S. Lerner, D. Grove, C. Chambers: *Composing Dataflow Analyses and Transformations*, Presentation for POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, January 16-18, 2002.  
<http://www.cs.umd.edu/class/fall2002/cmsc631/lectures/jjorge-combine.pdf>
- [10] S. Lerner, T. Millstein, C. Chambers: *Automatically Proving the Correctness of Compiler Optimizations*, In: PLDI '03 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, Pages 220-231, ACM New York, USA, 2003, ISBN: 1-58113-662-5. DOI [10.1145/781131.781156](https://doi.org/10.1145/781131.781156)
- [11] C. E. Lewis: *Modern Programming Language Implementation*, Lecture Notes, Course at Computer and Information Science of Penn University, Autumn 2006.  
<http://www.cis.upenn.edu/~cis570/>  
<http://www.cis.upenn.edu/~cis570/schedule.html>
- [12] S. S. Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, California, 1997, ISBN: 1-55860-320-4.
- [13] F. Nielsen, H. R. Nielsen, C. Hankin: *Principles of Program Analysis*, Springer-Verlag Berlin Heidelberg, Corrected 2nd printing 2005, ISBN: 3-540-65410-0.
- [14] T. Reps, S. Horwitz, M. Sagiv: *Precise Interprocedural Dataflow Analysis via Graph Reachability*, In: POPL '95 Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages 49-61, ACM New York, USA, 1995, ISBN: 0-89791-692-1. DOI [10.1145/199448.199462](https://doi.org/10.1145/199448.199462)
- [15] B. G. Ryder, M. C. Paull: *Incremental Data-Flow Analysis Algorithms*, In: ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 10 Issue 1, Jan. 1988, Pages 1-50, ACM New York, USA, ISSN: 0164-0925, EISSN: 1558-4593. DOI [10.1145/42192.42193](https://doi.org/10.1145/42192.42193)
- [16] G. Tóth, P. Hegedűs, Á. Beszédes, T. Gyimóthy, J. Jász: *Comparison of different impact analysis methods and programmer's opinion: an empirical study*,

- In: PPPJ '10 Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, Pages 109-118, ACM New York, USA, 2010, ISBN: 978-1-4503-0269-2. DOI [10.1145/1852761.1852777](https://doi.org/10.1145/1852761.1852777)
- [17] M. Queva: *Phase-ordering in optimizing compilers*, Master's thesis, Technical University of Denmark, 2007.  
<http://www2.imm.dtu.dk/pubdb/p.php?5406>
  - [18] *LLVM Tutorial*, Last modified on 2011-04-22 19:30:22, Online, Accessed July 2011.  
<http://llvm.org/docs/tutorial/>
  - [19] *LLVM Language Reference Manual*, Last modified on 2011-10-13 18:04:49, Online, Accessed October 2011.  
<http://llvm.org/docs/LangRef.html>
  - [20] *LLVM Programmer's Manual*, Last modified on 2011-10-11 01:33:56, Online, Accessed October 2011.  
<http://llvm.org/docs/ProgrammersManual.html>
  - [21] *Writing an LLVM Pass*, Last modified on 2011-10-11 02:03:52, Online, Accessed October 2011.  
<http://llvm.org/docs/WritingAnLLVMPass.html>
  - [22] *Rosetta Code*, Online, Accessed November 2011.  
<http://rosettacode.org>
  - [23] *Performance Application Programming Interface*, Online, Accessed November 2011.  
<http://icl.cs.utk.edu/papi/>
  - [24] *The Glasgow Haskell Compiler – Commentary*, An Integrated Code Generator for GHC, Last modified on 2008-08-26 05:58:43, Online, Accessed June 2011.  
<http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/IntegratedCodeGen>