Analýza toku dat

Tomáš Havlát, Mojmír Křetínský
Katedra matematické informatiky
PřF MU Brno

1 Předvýpočtová analýza programu

V tomto příspěvku se budeme převážně zabývat metodami předvýpočtové analýzy a příklady aplikací jejích výsledků. Nezůstaneme u některé ze základních, relativně jednoduchých metod, ale budeme prezentovat celou jejich škálu, i když ne zdaleka vyčerpávající. Tento záměr, spolu s rozsahem příspěvku a jeho orientací na programátorskou obec, nás nutí k poněkud méně formálnímu či intuitivnímu vyjadřování i tam, kde se předkládané metody zcela opírají o matematický aparát. Úplný popis jednotlivých metod, včetně důkazu správnosti a analýzy složitosti algoritmů najde čtenář v doporučené literatuře.

Předvýpočtová analýza programu spočívá ve vyšetřování některých vlastností programu, aniž by byl program prováděn. Analýze je podrobena některá ze statických forem zápisu programu (proto je někdy používán termín *statická analýza programu*) a cílem je jejím jednorázovým provedením získat o programu informace globálního charakteru, které často nelze získat ani opakovaným prováděním výpočtu programu pro různá vstupní data.

Do předvýpočtové analýzy patří dvě základní oblasti: verifikace programů a analýza toku dat. Mají do jisté míry podobné cíle, liší se však svými prostředky a metodami, kterými analýzu provádějí. *Verifikace programů* se obvykle prezentuje jako proces nalezení vhodných invariantů umožňujících formálními logickými prostředky dokázat správnost programu ([Flo67], [Man81]), či (z našeho hlediska vhodnější charakterizace) jako proces hledání úplné charakteristiky chování programu popisem vlastností všech možných výpočtů. Tyto cíle jsou vzhledem ke snaze provádět tuto analýzu automaticky příliš ambiciózní a už i z teoretického hlediska jsou předurčeny k selhání (snad s výjimkou těch nejtriviálnějších modelů programů).

Analýza toku dat (ATD) je orientována pragmatičtěji než verifikace programů a lze ji charakterizovat jako přibližnou, aproximující analýzu programu, kdy za cenu ztráty určité informace či přesnosti, ne však za cenu nepravdivé informace, jsme sto nalézt algoritmy, které požadované řešení poskytují. (*Analýzu toku řízení* zde nebudeme považovat za samostatnou oblast, ale za jednu z fází analýzy toku dat.)

Motivaci pro tuto analýzu lze rozdělit opět zhruba do dvou oblastí. První z nich je orientována na odvození informace o programu (jeho objektech a způsobech jejich užití), kterou posléze využívá člověk. Typickými příklady z této oblasti jsou např. automatická dokumentace programu, ladění a testování a další, o nichž se zmíníme později. Druhou typickou oblastí (z historického hlediska první) je optimalizace (či spíše "vylepšení") cílového programu generovaného kompilátorem.

Pragmatičtější orientace ATD spočívá v tom, že si klade za cíl extrahovat z textu programu jen relativně jednoduché, přesně definované vlastnosti (objektů) programu; úkoly nalézt ty nejjednodušší z nich jsou známy jako tzv. *základní problémy ATD*. Jejich důležitost vidíme jednak z hlediska metodologického (snadná formulace i hledání řešení, přičemž nejde o triviální zjednodušení), zejména však v tom, že řešení těchto základních problémů je obvykle částí i složitějších problémů ATD. Analýze toku dat se budeme proto věnovat především.

2 Základní pojmy

Orientovaný graf G je dvojice (N, E), kde N je konečná množina uzlů, E je konečná množina hran; každá hrana e má počáteční uzel $source(e) \in N$ a koncový uzel $target(e) \in N$; hrana e vede z uzlu $source(e)$ do uzlu $target(e)$. Uzel $source(e)$ je (bezprostředním) *předchůdcem* uzlu $target(e)$ a uzel $target(e)$ je (bezprostředním) *následníkem* uzlu $source(e)$. Označme $PRED(v)$, resp. $SUCC(v)$ množinu všech předchůdců, resp. následníků uzlu v.

Cesta v grafu G je posloupnost $p = e_1, e_2, \ldots, e_k$ hran takových, že $target(e_i) = source(e_{i+1})$ pro $1 \leq i < k$. Říkáme, že cesta p vede z uzlu $source(e_1)$ do uzlu $target(e_k)$, je tvořena hranami $e_1, e_2, \ldots, e_k$ resp. uzly $source(e_1), source(e_2), \ldots, source(e_k), target(e_k)$; číslo k nazýváme délkou cesty p.

Cykl v grafu je cesta nenulové délky, která vede z uzlu do téhož uzlu ($source(e_1) = target(e_k)$).

Cesta je *k-jednoduchá* ($k \geq 1$), jestliže neobsahuje žádný uzel více než k krát. 1-jednoduchá cesta se nazývá jednoduchá. Cesta p je *k-semijednoduchá*, jestliže $p = r, e, q$, kde r je jednoduchá cesta, e je hrana a q je k-jednoduchá cesta.

Strom je orientovaný graf, v němž existuje uzel s (*kořen*) takový, že z něj vede právě jedna cesta do libovolného uzlu. Uzly stromu, které nemají následníka, se nazývají *listy*.

Graf toku řízení (GTŘ) je trojice (N, E, s), kde (N, E) je orientovaný graf, $s \in N$ počáteční uzel, existuje právě jeden koncový uzel $h \in N$ takový, že z něj nevede žádná hrana, a pro libovolný uzel v existuje cesta z počátečního do koncového uzlu obsahující uzel v.

Uzly grafu toku řízení programu P reprezentují (elementární) příkazy programu, hrany možnost předání řízení mezi jednotlivými příkazy. Protože mnoho metod analýzy toku dat pracuje s grafem toku řízení programu, patří transformace programu na jeho GTŘ mezi základní techniky analýzy programu. Algoritmy ATD opakovaně

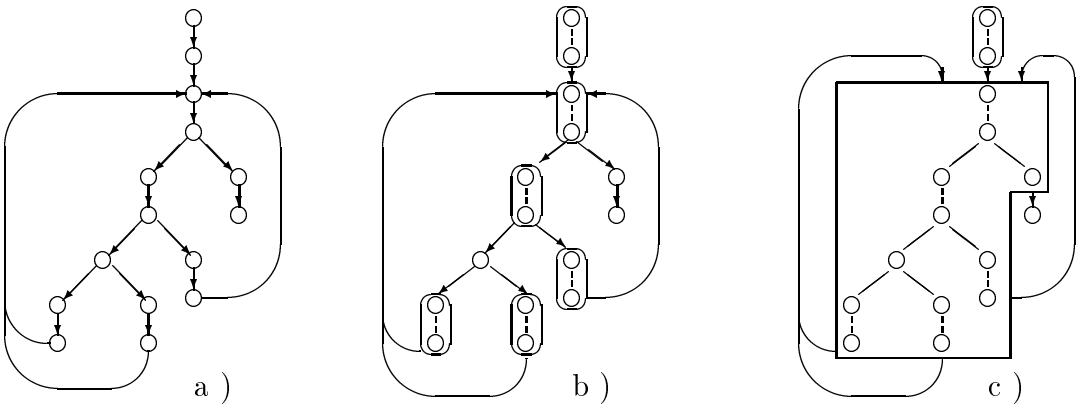
procházejí graf toku řízení a jejich složitost závisí vedle struktury GTŘ i na počtu jeho hran a uzlů. Je tedy snaha do uzlu GTŘ sdružovat více než jeden příkaz (a redukovat tak počet uzlů i hran), ovšem tak, aby analýza toku dat uvnitř uzlu (lokální analýza) se dala snadno provést před analýzou celého programu (globální analýza). Tento požadavek jednoduchosti (obvykle lineární časové složitosti vzhledem k počtu hran) splňují tzv. bloky příkazů.

Uvažujme program P s GTŘ, jehož uzly reprezentují elementární příkazy. *Blok* s jediným vstupním příkazem (vstupem bloku) a s jedním či více výstupními příkazy (výstupy bloku) je část programu P splňující tyto podmínky:

1. do příkazu bloku, který není vstupním příkazem, vede hrana pouze z příkazu bloku, který není výstupní;
2. z příkazu bloku, který není výstupní, nevede hrana do příkazu mimo blok;
3. graf toku řízení odpovídající bloku je strom, který dostaneme z GTŘ programu P vypuštěním všech příkazů mimo blok a všech hran, které vedou z příkazů mimo blok a z výstupních příkazů bloku. Kořenem tohoto stromu je vstupní příkaz bloku a jeho listy jsou právě výstupní příkazy bloku.

Má-li blok právě jeden výstupní příkaz, nazývá se *základním blokem* (je tvořen sekvencí elementárních příkazů). Elementární příkaz je rovněž (základním) blokem a je současně vstupním i výstupním příkazem.

Předpokládejme, že uzly GTŘ reprezentují bloky příkazů a hrana vede z uzlu (bloku) i do uzlu (bloku) j, je-li možnost předání řízení z výstupu bloku i na vstup do bloku j. Na obr.1 uzly GTŘ reprezentují (elementární) příkazy a), základní bloky b), bloky c).



Obr. 1

S každým blokem b, přesněji řečeno s každým výstupem w bloku b, jsou spjaty některé elementární informace o toku dat na cestě ze vstupu bloku do výstupu w. Uveďme některé z nich. Jednak proto, abychom hned z počátku podpořili tvrzení, že blok byl volen tak, aby šel snadno analyzovat (čtenáři bude jasné, jak tyto informace získat), jednak proto, že se na ně hodláme později odkazovat.

Df(b,w) je množina proměnných, které mají na výstupu w hodnotu přiřazenou jim na cestě blokem b do w.

Rf(b,w) je množina proměnných, nad nimiž je první akcí na cestě blokem b do w referencování jejich hodnoty.

Undf(b,w) je množina proměnných, které mají na výstupu w nedefinovanou hodnotu jako důsledek akce na cestě blokem b do w.
Je-li b základní blok, pak informace nezávisí na výstupu (je právě jeden).

Kostra grafu toku řízení $G = (N, E, s)$ je takový podgraf $G' = (N, E'), E' \subseteq E$, že G' je strom s kořenem s.

rPostorder je úplné uspořádání na množině uzlů N takové, že vždy, když v kostře GTŘ vede hrana z uzlu u do uzlu v, pak $(u, v) \in$rPostorder. Algoritmy pro nalezení kostry GTŘ a uspořádání rPostorder lze nalézt např. v [Hec77]. *Postorder* je reverzí uspořádání rPostorder.

Uzel x grafu toku řízení *dominuje vzhledem k počátku* uzlu y, $x \neq y$, jestliže x leží na každé cestě z počátečního uzlu do uzlu y.

Uzel x grafu toku řízení *dominuje vzhledem ke konci* uzlu y, $x \neq y$, jestliže x leží na každé cestě z uzlu y do koncového uzlu.

Obě dominance jsou ostrá částečná uspořádání na množině uzlů, algoritmy pro jejich nalezení jsou uvedeny např. v [Tar74].

Hrana e je *zpětná hrana* GTŘ, jestliže $target(e)$ dominuje vzhledem k počátku uzlu $source(e)$.

Cyklická souvislost $d(G)$ GTŘ G je rovna maximálnímu počtu zpětných hran na libovolné acyklické cestě v G.

Poznamenejme, že metody analýzy toku dat berou do úvahy všechny cesty grafem toku řízení, tedy i ty, které nejsou částí žádného možného výpočtu podle programu (neproveditelné cesty). To je nutno mít vždy na paměti při posuzování výsledků těchto metod.

3 Základní problémy analýzy toku dat

Obecně jsou (základní) problémy ATD formulovány následovně. Definuje se vlastnost V objektu (např. proměnné) O v bodě b GTŘ (bod je vstup či výstup bloku). Tuto vlastnost V objekt O nabude v závislosti na tom, co se s ním děje podél (máme na vybranou jednu ze čtyř základních alternativ)

1. alespoň jedné cesty do bodu b (označení: "$\exists$,*dopředu* ")
2. alespoň jedné cesty z bodu b(označení: "$\exists$,*zpět* ")
3. všech cest z počátečního uzlu GTŘ do bodu b (označení: "$\forall$,*dopředu* ")
4. všech cest z bodu b do koncového uzlu GTŘ (označení: "$\forall$,*zpět* ").

Řešením problému ATD je stanovení množiny objektů, které mají v daném bodě vlastnost V, a to pro všechny uzly GTŘ.

Řešení problému ATD zadaného alternativami 1, resp. 2 vypovídá o tom, co se s objektem může stát předtím, resp. potom co výpočet dospěje do, resp. bude

pokračovat z daného bodu. Tyto problémy označujeme jako $\exists$-problémy. Naproti tomu řešení problémů zadaných alternativami 3, resp. 4 vypovídají o tom, co se s objektem stane (musí se stát) předtím, resp. potom co výpočet dospěje do, resp. bude pokračovat z daného bodu. Označujeme je jako $\forall$-problémy.

Při řešení problémů zadaných alternativami 1 nebo 3 se (jak uvidíme později) informace o vlastnostech objektů šíří od předchůdce k následníkovi, tj. ve směru hran GTŘ, tzn. "dopředu". Naproti tomu při řešení problémů zadaných alternativami 2 nebo 4 se informace o vlastnostech objektů šíří proti směru hran GTŘ, tj. od následníka k předchůdci, tzn. "zpět".

Rozlišujeme tedy čtyři základní problémy ATD. Poznamenejme, že řešení kteréhokoliv z nich lze použít k řešení zbývajících: $\exists$-problém je ekvivalentní s $\neg\forall\neg$-problémem, a problémy "dopředu" lze převádět na zpětné prostřednictvím inverze (obrácení orientace hran) GTŘ.

Metody řešení ukažme na následujícím příkladu. Máme zjistit, zda v průběhu dalšího výpočtu existuje (potenciální) možnost použití momentální hodnoty proměnné.

Proměnnou nazveme *živou* v bodě p GTŘ, právě když existuje cesta z p do nějakého uzlu w, podél níž není proměnná definována a první akcí nad touto proměnnou ve w je použití její hodnoty. Problém nalezení živých proměnných se nazývá *LIVE problém* a je typu "$\exists$, zpět ".

Abychom mohli pro libovolnou proměnnou určit zda a kde je živá, musíme vycházet z informací o akcích, které jsou nad touto proměnnou prováděny v jednotlivých uzlech (pro jednoduchost v základních blocích) GTŘ. Bude nás zajímat

1. množina všech proměnných, nad nimiž je první akcí v bloku u redefinování jejich hodnoty; označme je (pro daný uzel u) $KILL(u)$ a jejich doplňky $NOTKILL(u)$.
2. množina $Rf(u)$ všech proměnných, nad nimiž je první akcí v bloku u referencování jejich hodnoty; označme je $GEN(u)$.

Reprezentuje-li blok u např. zdrojový text $x := x + 1; y := x$, pak $GEN(u) = \{x\}$ a $KILL(u) = \{y\}$. Je evidentní, že proměnná x je (lokálně) živá na vstupu bloku u.

Libovolný problém analýzy toku dat, a tedy i hledání živých proměnných lze chápat jako *problém šíření informace*: lokální informaci (GEN) danou uzlem šíříme (zde proti orientaci hran) po grafu toku řízení tak, že zpráva o živosti proměnné je předána dál pokud v momentálně zpracovávaném uzlu není živost proměnné "zabita" ($KILL$), resp. zpráva je obohacena o živost nějaké další proměnné, patří-li tato do GEN uzlu. Z lokální informace se tak stává hledaná globální vlastnost.

Než přejdeme k prezentaci algoritmů pro živé proměnné, uveďme představitele zbývajících typů základních problémů ATD.

Problémem typu "$\exists$,*dopředu*" je například *dosažitelnost definic* ($REACHES$ problém): definice d proměnné v v uzlu u dosahuje vstupu do uzlu m, jestliže d je v uzlu u a existuje cesta z u do m na níž není v redefinována.

Problém dostupných výrazů ($AVAIL$ problém) je typu "$\forall$,*dopředu*": výraz v je dostupný v bodě p, právě když na všech cestách z počátečního uzlu do p byla hodnota

výrazu v spočtena a po tomto výpočtu již nebyla hodnota žádného z operandů výrazu v redefinována.

Problém využitelných výrazů (Very Busy Expression problém) je typu $\forall$, *"zpětný "*: výraz v je využitelný v bodě p, právě když na všech cestách z p do koncového uzlu je v referencován, aniž by byla na některé z těchto cest redefinována hodnota některé jeho složky (operandu).

3.1 Řešení problému živých proměnných (*LIVE* problém)

Označme hledanou množinu proměnných, které jsou živé na vstupu, resp. výstupu uzlu u jako $LVTOP(u)$, resp. $LVBOT(u)$. K dispozici máme lokální informace $GEN(u)$ o tom, které proměnné se stávají v uzlu živými a $KILL(u)$, která obsahuje ty proměnné, jež jsou v uzlu "zabity" (definicí jejich hodnoty).

Z obr.2 je patrno, že proměnná je živá za tohoto předpokladu:

1. je živá na vstupu uzlu, je-li živá na výstupu tohoto uzlu a není tímto uzlem zabita, nebo její živost je tímto uzlem generována (obr.2a);
2. je živá na výstupu uzlu, je-li živá na vstupu některého z následníků tohoto uzlu (obr.2b), tj.:

$$(L1) \quad LVTOP(u) = (LVBOT(u) \cap NOTKILL(u)) \cup GEN(u) \quad \forall u \in N$$

$$(L2) \quad LVBOT(u) = \bigcup_{v \in SUCC(u)} LVTOP(v) \quad \forall u \in N$$

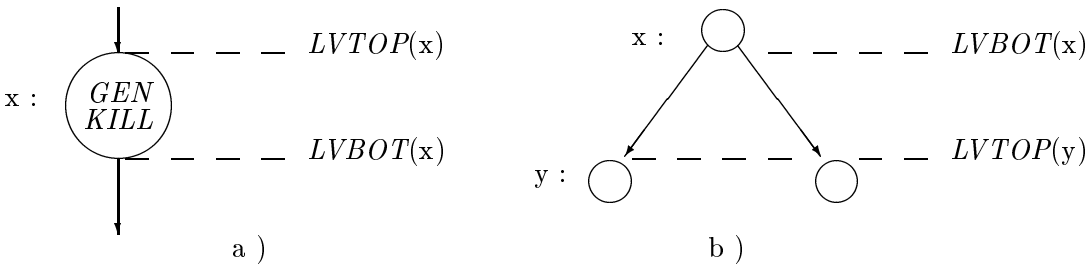
$$je - li \;\; SUCC(u) = \emptyset, \; pak \;\; LVBOT(u) = \emptyset$$

či stručněji (dosadíme-li (L2) do (L1))

$$(LIVE) \quad LVTOP(u) = (\bigcup_{v \in SUCC(u)} LVTOP(v) \cap NOTKILL(u)) \cup GEN(u)$$

a dále

$(L3)$ řešením *LIVE* problému je nejmenší řešení (vzhledem k inklusi) splňující ($LIVE$), resp. (L1) a (L2).



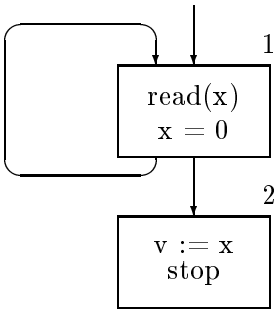
Obr. 2

Požadavek (L3) si zasluhuje několik poznámek. Označme $LIVE(u, p)$ množinu těch živých proměnných v uzlu u, jejichž hodnoty budou použity na cestě p vychá-

zející z uzlu u. Množinu všech cest z uzlu u označme $PATH(u)$ - tato je obecně nekonečná. *LIVE* problém lze pak psát ve tvaru "nekonečného sjednocení" takto:

$$(*) \qquad LIVE(u) = \bigcup_{p \in PATH(u)} LIVE(u,p) \ ,$$

který jsme reprezentovali (konečným způsobem) jako systém (*LIVE*) vzájemně rekurzívních rovnic majících obecně více řešení (viz následující příklad). Sjednocení nekonečného systému množin (*) je ovšem definováno jako nejmenší množina ($LIVE(u)$) obsahující všechny sčítance, což je (na intuitivní úrovni) důvod pro podmínku (L3). Správnost řešení (*LIVE*) nemůžeme ověřovat vůči (*) kontrolou jednotlivých sčítanců - je jich nekonečně mnoho.
Příklad:



$GEN(1) = \emptyset, \quad NOTKILL(1) = \{\mathrm{v}\}, \quad SUCC(1) = \{1,2\}$

$SUCC(2) = \emptyset \quad \Rightarrow \quad LVBOT(2) = \emptyset$
$GEN(2) = \{\mathrm{x}\}, \ NOTKILL(2) = \{\mathrm{x}\} \ \Rightarrow$
$\Rightarrow \quad LVTOP(2) = \{\mathrm{x}\}$

a) $LVTOP(1) = \{v\}, \quad LVBOT(1) = LVTOP(1) \cup LVTOP(2) = \{v, x\}$
b) $LVTOP(1) = \emptyset, \qquad LVBOT(1) = LVTOP(1) \cup LVTOP(2) = \{x\}$
Jak a) tak i b) splňují rovnice (*LIVE*), řešení *LIVE* problému je ad b).

Zbývá objasnit, jak systém (*LIVE*) řešit. Jedna z možných ideí již byla naznačena: na každý uzel (resp. rovnici pro tento uzel) můžeme nahlížet jako na proces, který čeká až mu každý jeho následník v zašle zprávu $LVTOP(v)$. Poté proces zprávu zpracuje tak, že vyhodnotí rovnici (jako přiřazení) a výsledek $LVTOP(u)$ posílá všem svým předchůdcům. Po přijetí nových vstupních zpráv a výpočtu nové hodnoty *LVTOP* proces zkontroluje, zda nově vypočítaná *LVTOP* zpráva není totožná s posledně zasílanou zprávou. Jestliže ano, pak zprávu, která nepřináší žádnou novou informaci, dále nešíří a převede se do stavu "ukončení schopný". V opačném případě zprávu s novým (bohatším) informačním obsahem posílá dál. Výpočet končí, jsou-li všechny procesy ve stavu "ukončení schopný", což, jak ukážeme později při přesnější (‚sekvenční) realizaci myšlenky, musí po jistém čase vskutku platit. Poznamenejme, že (pomineme-li otázku časové efektivity), nezáleží na pořadí v jakém jsou procesy (např. na monoprocesorovém systému) aktivovány. Jediné co musí být splněno je spravedlivé provádění, tj. provedení žádného z procesů nesmí být do nekonečna odkládáno.

Korektnost tohoto principu (formálně ji dokážeme později) souvisí mj. s inicializací informace šířené procesy. K jejímu splnění (včetně požadavku minimality řešení) stačí, abychom na začátku udělali "nejhorší" předpoklad, a to, že žádná proměnná není živá na vstupu žádného uzlu: $LVTOP(u) = \emptyset$. Označíme-li totiž jako

$PATH^i(u)$ cesty z uzlu u délky nejvýše i a

$$LIVE^i(u) = \bigcup_{p \in PATH^i(u)} LIVE(u,p) \quad ,$$

pak můžeme zapsat i nerekurzívní specifikací pro *LIVE*:

$$LIVE^0(u) = \emptyset,$$

$$\forall i > 0: \quad LIVE^i(u) = \bigcup_{v \in SUCC(u)} LIVE^{i-1}(v) \cap NOTKILL(u) \cup GEN(u)$$

Zřejmě každé částečné řešení $LIVE^i$ bude obsaženo v konečném řešení *LIVE*, tj. pro $i \geq 0$ je $LIVE^i \subseteq LIVE$, a tedy i $\bigcup_{i \geq 0} LIVE^i \subseteq LIVE$. Na druhé straně (platí pro *LIVE*, nikoliv však obecně), je-li nějaká proměnná x v *LIVE*, pak musí existovat $i \geq 0$ takové, že x je v $LIVE^i$, tj. $LIVE \subseteq \bigcup_{i \geq 0} LIVE^i$ a tudíž $LIVE = \bigcup_{i \geq 0} LIVE^i$. Z toho plyne, že $LIVE^0(u) := GEN(u)$ je rovněž korektní inicializací.

Uvedená idea řešení je jednoduchá i v sekvenčním podání: po inicializaci $LVTOP = \emptyset$ (resp. $GEN(u)$) vyhodnocujeme rovnice pro jednotlivé uzly v libovolném pořadí a z hlediska korektnosti a ukončení se nabízí tyto možnosti:

1. Udržovat *seznam uzlů*, kterým byla zaslána nová (různá od předešlé) informace. Vyhodnocování končí, jakmile je seznam prázdný (už není co nového šířit - informace se stabilizovala).
2. Uzly libovolným způsobem uspořádat a vyhodnocovat jim odpovídající rovnice cyklicky v daném pořadí (systémem *round-robin*). Jestliže po průchodu přes všechny uzly nezaznamenáme ani u jednoho změnu nově vypočtené informace, pak se systém stabilizoval a další průchody by už nic nového nepřinesly. (Jelikož u *LIVE* problému se informace šíří zpětně od uzlu k jeho předchůdcům, je vhodným - z hlediska efektivity - uspořádáním uzlů *Postorder*).

Algoritmus - Iterativní LIVE

Vstup: pro všechny uzly u *GEN*(u), *NOTKILL*(u)
Výstup: pro všechny uzly u *LVTOP*(u)

```
begin   {uzly z N jsou očíslovány 1,2,...,n v pořadí Postorder}
    {inicializace:}
    for i:=1 to n do LVTOP(i) := ∅ od;
    {iterace:}
    change := true;
    while change do
        change := false;
        for i:=1 to n do
            NEWLVTOP := ⋃ {LVTOP(k); k∈SUCC(i)} ∩ NOTKILL(i) ∪ GEN(i);
            if NEWLVTOP ≠ LVTOP(i) then change:=true;
                                        LVTOP(i) := NEWLVTOP fi;
        od;
    od;
end;
```

Ukončení a korektnost algoritmu "Iterativní LIVE" je intuitivně zřejmá; formálně bude ukázána v čl.6.3 (algoritmus je instancí obecného modelu). Jeho složitost je $O(d(G). \mid E \mid)$ operací $\cup$ a $\cap$ (viz [Hec77]); **while** se provede nejvýše $(d+2)$krát, šíří-li se *GEN* po acyklické cestě s d zpětnými hranami; **for** představuje $\mid E \mid - \mid N \mid$ operací $\cup$ a $\cap$ pro *NOTKILL* a *GEN*).

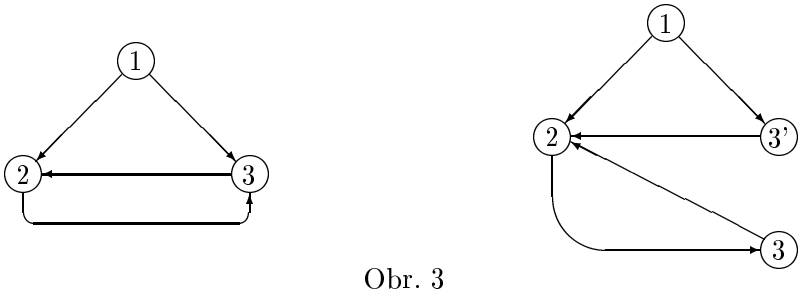
4 Eliminační metody

Iterativní algoritmy řešení problémů ATD jsou výhodné z hlediska porozumění principu, z hlediska snadné implementace a v neposlední řadě proto, že se dají aplikovat na libovolné GTŘ bez omezení jejich struktury. Avšak při šíření informace nevyužívají téměř žádná fakta o struktuře GTŘ. Na základě těsné analogie s řešením systému lineárních rovnic, který lze řešit jak iterativně, tak eliminací, vzniká otázka, zda podobný přístup nelze uplatnit i zde (viz [Tar81]). Navíc lze očekávat, že tyto (nerekurzivní) eliminační metody mohou poskytnout řešení i u těch složitějších problému ATD, kdy iterace sice konverguje, ale ne dostatečně rychle (přesné řešení nenalezneme v konečném čase). Výhodou eliminačních algoritmů je jejich obvykle lepší časová složitost (až lineární), někdy poskytují přesnější řešení. Jsou však podstatně komplikovanější (i k implementaci).

Společnou ideou eliminačních algoritmů je využít struktury grafu toku řízení tak, aby byl sumarizován efekt šíření informace pro nějakou (obecně nekonečnou) množinu cest mezi dvěma uzly, např. pro nějaký vhodný model (jednoduchého) cyklu. Pak *lokální informaci GEN, KILL*, která udává lokální vlastnost (např. živost) na vstupech jednotlivých bloků postupně *globalizujeme*: začneme u nejvnitřnějšího cyklu a skládáním efektů lokálních *GEN* a *KILL* najdeme výsledné *GEN* a *KILL* pro vstupní bod celého cyklu. Množinu uzlů tvořících tento cyklus pak nahradíme jediným uzlem (*GEN* a *KILL* pro jeho vstup jsme již našli) a celá situace se opakuje až dospějeme k triviálnímu grafu (tvořenému jediným uzlem) a jeho *GEN* a *KILL*. Pro triviální graf nyní platí $LIVE = GEN$. Po tomto průchodu 1 (přenos informace ve směru lokální $\rightarrow$ globální) provádíme průchod 2 (globální $\rightarrow$ lokální), při kterém na základě sumarizované informace v *GEN* a *KILL* počítáme postupně detailnější informaci *LIVE*: expandujeme triviální graf v obráceném pořadí až dospějeme k původnímu grafu toku řízení a informaci *LIVE* pro každý jeho uzel.

Nejjednodušším modelem cyklu je tzv. *silně souvislý (pod)graf*, kdy požadujeme, aby z každého uzlu vedla cesta do libovolného uzlu. Tento model je však dosti obecný a nestrukturovaný, což se projeví jak na komplikovanosti algoritmu, tak hlavně na jeho časové složitosti. Přesto takové algoritmy existují [Ear72].

Výhodnější se ukazuje omezit se na programy, kde každý cyklus má právě jeden vstup (duplikací kódu ve vnitřní reprezentaci programu lze splnění této podmínky dosáhnout i pro programy, které obsahují cykly s více vstupy - viz obr.3). Uveďme dvě metody založené na modelech cyklu s jedním vstupem. První z nich (intervalová analýza) je relativně nejnázornější ilustrací eliminace, druhá (algoritmus Grahamové



Obr. 3

a Wegmana) si zaslouží pozornost díky své časové složitosti.

4.1 Intervalová analýza

Modelem cyklu je tzv. *interval*, což je množina uzlů I definovaná takto:

1. existuje právě jeden uzel $h \in I$ (*hlava* intervalu I), který leží na každé cestě z uzlu nepatřícího do I do uzlu v I (jediný vstup do cyklu);
2. I je souvislý;
3. $I - \{h\}$ neobsahuje cykl (tj. všechny cykly v I musí obsahovat h).

Uveďme algoritmus, který pro daný graf toku řízení G a daný uzel h konstruuje maximální interval s hlavou h. Zde i dále pro množinu uzlů M označíme $SUCC(M) = \bigcup_{x \in M} SUCC(x)$.

Algoritmus MI - konstrukce maximálního intervalu

Vstup: graf toku řízení G; jeho uzel h
Výstup: MAXI(h) - maximální interval s hlavou h
begin I:={h};
while $\exists$ x $\in$ *SUCC*(I)-{I} takový, že *PRED*(x) $\subset$ I **do** I:=I $\cup$ {x} **od**;
MAXI(h):=I
end;

Stojí za povšimnutí pořadí, v jakém jsou uzly do intervalu I přidávány. Toto tzv. *intervalové pořadí* je linearizací částečného uspořádání daného orientací hran v intervalu (bez hran vedoucích do hlavy h). Pro průchod 1 analýzy toku dat bude důležité, že při zpracování uzlů v intervalovém pořadí analyzujeme daný uzel $x \neq h$ až když jsou analyzováni všichni jeho předchůdci. Analogicky v průchodu 2 při zpracování uzlů v obráceném intervalovém pořadí je uzel analyzován až po analýze všech svých následníků. Budeme se snažit o nalezení úplného uspořádání s uvedenými vlastnostmi pro všechny uzly GTŘ. Pomůže nám v tom následující algoritmus, který GTŘ rozdělí na vzájemně disjunktní intervaly (uvnitř každého je intervalové pořadí známo).

Algoritmus IP - rozdělení GTŘ na intervaly

Vstup: graf toku řízení G
Výstup: INTS(G) - množina disjunktních intervalů tvořících rozklad uzlů grafu G
Pomocné proměnné a procedury:
H množina potenciálních hlav intervalů
DONE množina hlav nalezených intervalů
MI(h) procedura, která nalezne maximální interval MAXI(h) s hlavou h, (viz algoritmus MI)

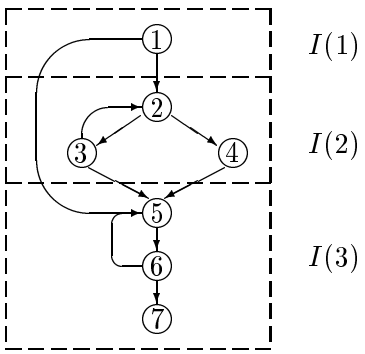
begin H := {počáteční uzel GTŘ G}; DONE := $\emptyset$; INTS(G) := $\emptyset$;
while H $\neq \emptyset$
do nechť x je libovolný uzel z H;
H := H - {x}; DONE := DONE $\cup\{x\}$; **call** MI(x);
INTS(G) := INTS(S) $\cup$ MAXI(I); {přidej nové potenciální hlavy:}
H := H $\cup$ (*SUCC*(MAXI(x)) - MAXI(x) -DONE)
od
end;

Příklad: Pro GTŘ na obr.4 je výstupem z algoritmu IP tento rozklad na intervaly: $I(1) = \{1\},\ I(2) = \{2,3,4\},\ I(5) = \{5,6,7\}$.

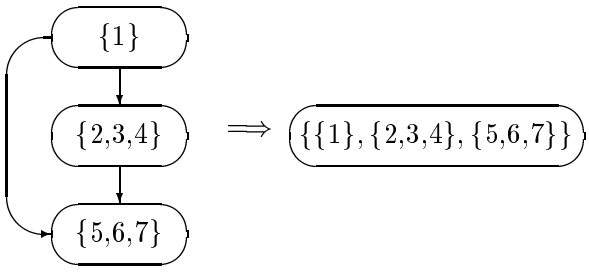
Uvnitř každého intervalu je dáno intervalové pořadí uzlů. K dosažení úplného uspořádání celé množiny uzlů GTŘ musíme ještě nalézt (intervalové) uspořádání mezi jednotlivými intervaly. Vhodnou metodou je uvažovat graf, jehož uzly jsou zkonstruované intervaly a celý postup opakovat.

Pro GTŘ G definujeme *odvozený* GTŘ $I(G)$ takto:

1. Uzly grafu $I(G)$ jsou intervaly grafu G, tj. INTS(G).
2. Počátečním uzlem grafu $I(G)$ je MAXI(s), kde s je počáteční uzel grafu G.
3. V grafu $I(G)$ vede hrana z uzlu J do uzlu K, právě když v G vede hrana z nějakého uzlu intervalu J do hlavy intervalu K.



Obr. 4



Obr. 5

Posloupnost grafů $G_0, G_1, \ldots, G_m$ nazveme *odvozenou posloupností* pro GTŘ G, jestliže $G = G_0$, $G_{i+1} = I(G)$ $(0 \leq i \leq m-1)$, $G_{m-1} \neq G_m$ a $I(G_m) = G_m$. Graf G_i nazveme odvozeným grafem řádu i, G_m limitním grafem a číslo $m+1$ intervalovým řádem grafu G. GTŘ G se nazývá *reducibilní*, právě když jeho limitní graf je triviální. V opačném případě hovoříme o nereducibilním GTŘ. Obr.5 navazuje na obr.4 a ilustruje zbytek odvozené posloupnosti.

Intervalové pořadí uzlů reducibilního GTŘ G získáme opakovaným odvozováním intervalového pořadí grafu G_i ze známého intervalového pořadí grafu G_{i+1} (intervalové pořadí triviálního grafu G_m je předem dáno): interval J v pořadí grafu G_i nahradíme jeho uzly v tom pořadí, jak byly do J přidávány.

Při ATD založené na intervalové analýze je nutno pracovat s lokální informací *KILL*, která je sdružena nikoli s uzly, ale s hranami (intervaly mají více než jeden výstup). Tedy $NOTKILL(x, y)$ je množina všech proměnných takových, pro něž existuje cesta ze vstupu uzlu x do vstupu uzlu y neobsahující redefinice těchto proměnných.

Rovnice pro *LIVE* mají pak tento tvar:

$$(LIVE) \quad LIVE(x) = \bigcup_{y \in SUCC(x)} (LIVE(y) \cap NOTKILL(x, y)) \cup GEN(x)$$

Intervalová analýza - průchod 1 (lokální → globální):

Jádrem tohoto průchodu je algoritmus I1, který určí pro interval I jeho $GEN(I)$ a $NOTKILL(I, J)$ na základě *GEN* a *NOTKILL* uzlů tohoto intervalu. Průchod 1 spočívá ve volání procedury I1 pro každý interval grafu G_0 (v jejich intervalovém pořadí), posléze se tento proces opsakuje pro G_1 atd. až je aplikován na G_{m-1}, který je tímto procesem redukován na limitní graf odvozené posloupnosti $G_m = (\{x^*\}, \emptyset, x^*)$ a určena jeho $GEN(x^*)$.

Intervalová analýza - průchod 2 (globální → lokální):

V tomto průchodu počítáme *LIVE* pro stále menší oblasti původního grafu. Postupujeme od G_m, pro který je evidentně korektní přiřazení $LIVE(x^*) := GEN(x^*)$, přes $G_{m-1}, ..., G_1$ až při G_0 dostáváme *LIVE* pro každý uzel GTŘ G_0. Celá druhá fáze spočívá v opakovaném volání procedury I2, která, na základě *LIVE* pro interval $I(h)$ a množin *LIVE* pro jeho následníky J, určí (s pomocí *GEN* a *NOTKILL*) množiny *LIVE* pro jednotlivé uzly tohoto intervalu. Poznamenejme, že *GEN* a *NOTKILL* byly určeny již v průchodu 1 a předpoklad o existenci (tj. již vypočtených) $LIVE(I)$ a $LIVE(J)$ je splněn díky pořadí, v němž je I2 postupně aktivována, tj. od G_m po G_0.

Vlastní I2 je založena na tom, že uzly z $I(h) - \{h\}$ jsou zpracovávány v reverzi intervalového pořadí: při aktivaci pro $LIVE(x)$ máme již vypočteny $LIVE(y)$ pro $y \in SUCC(x)$ a můžeme tedy použít výše uvedenou rovnici $(LIVE)$; nastává totiž právě jeden z těchto případů:

1. $y \in I - \{h\}$, pak $LIVE(y)$ již bylo určeno díky zpracování uzlů v reverzi intervalového pořadí;

Algoritmus I1

Vstup: (1) Interval I s hlavou h
(2) $\forall x \in I : GEN(x),\ \forall x \in I . \forall y \in SUCC(x) : NOTKILL(x, y)$
Výstup: $GEN(I),\ \forall J \in SUCC(J) :\ NOTKILL(I, J)$
Pomocné proměnné: $\forall x \in I$: množina PATH(x) těch proměnných, pro které existuje cesta ze vstupu hlavy intervalu do vstupu uzlu x neobsahující redefinice těchto proměnných.

begin
{inicializace : }
GEN(I):=*GEN*(h); PATH(h):=Var;
{Var je množina všech proměnných v daném programu}
{globalizace *GEN*:}
for ∀ x∈ I-{h} v intervalovém pořadí **do**
PATH(x) := ⋃ {PATH(y) ∩ *NOTKILL*(y,x); y ∈ *PRED*(x) };
GEN(I) := *GEN*(I) ∪ (*GEN*(x) ∪ PATH(x))
od; {dále hJ značí hlavu intervalu J}
{globalizace *NOTKILL*:}
for ∀J taková, že hJ ∈ *SUCC*(I) **do**
NOTKILL(I,J) := ⋃ {PATH(y) ∩ *NOTKILL*(y,hJ); y∈*PRED*(hJ)∩I}
od
end;

2. y je hlavou I a v tom případě $LIVE(y) = LIVE(I)$, která byla určena již v grafu o jeden řád vyšším;
3. y je hlavou nějakého intervalu J, který je následníkem intervalu I a v tom případě použijeme, analogicky jako v bodu 2, množinu $LIVE(J)$.

Při úvahách o složitosti musíme nejprve uvážit složitost nalezení odvozené posloupnosti grafů (analýzy toku řízení). Algoritmus rozkladu GTŘ $G = (N, E, s)$ na intervaly pracuje zřejmě v čase úměrném počtu hran vstupního GTŘ, tj. $O(\mid E \mid)$ a je-li m intervalový řád G (předpokládáme navíc $\mid E \mid \gg m$), pak dostáváme celkem $O(\mid E \mid .m)$. Poznamenejme, že existují anomální grafy s $m = O(\mid E \mid^2)$, v praxi však zřídka bývá $m > 6$ a v průměru je 2,75 [Knu71].

Uvažujme složitost vlastní analýzy toku dat. Značí-li n počet uzlů a e počet hran v odvozené posloupnosti, pak tato analýza má složitost $O(e)$. Existují sice grafy s $e = O(\mid E \mid^2)$, avšak pro reálné programy je typické $O(e) = O(\mid E \mid)$.

Z teoretického hlediska je tedy nejhorší případ intervalové analýzy lepší než nejhorší případ iterativních metod, avšak na reálných programech je jejich složitost přibližně stejná, přičemž iterativní metody jsou snáze implementovatelné.

4.2 Komprese cest - algoritmus Grahamové a Wegmana

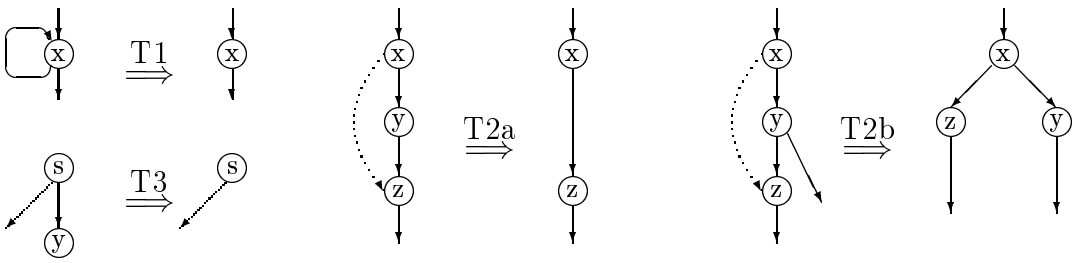
Algoritmus I2

Vstup: (1) Interval I s hlavou h
(2) $\forall x \in I : GEN(x),\ \forall x \in I . \forall y \in SUCC(x) : NOTKILL(x, y)$
(3) $LIVE(I),\ \forall J \in SUCC(I) : LIVE(J)$
Výstup: $\forall x \in I : LIVE(x)$
begin
$LIVE$(h) := $LIVE$(I); {viz bod 2.} {dále hJ značí hlavu intervalu J}
for $\forall J \in SUCC(I)$ **do** $LIVE(hJ) := LIVE(J)$ **od**; {viz bod 3.}
for $\forall x \in I - \{h\}$ v reverzi intervalového pořadí **do**
$LIVE(x) := \bigcup\{LIVE(y) \cap NOTKILL(x, y); y \in SUCC(x)\} \cup GEN(x)$
od; {viz bod 1.}
end;

Eliminační metody nejsou triviální, ale mohou být co do složitosti velmi efektivní (až témeř lineární), což ilustruje metoda Grahamové a Wegmana.

Algoritmy založené na kompresi cest pracují opět jen nad reducibilními grafy a vyznačují se tím, že definují vždy několik relativně jednoduchých transformací, jež zachovávají reducibilitu grafu a řešení pro původní graf je z transformovaného (odvozeného) grafu snadno získatelné. Postupnou aplikací základních transformací dostaneme z původního, reducibilního GTŘ, limitní triviální graf. Tyto algoritmy, tak jako všechny eliminační metody, pracují opět dvouprůchodově, přičemž nezbytnou analýzu toku řízení (tj. nalezení posloupnosti transformací redukující daný graf na limitní) lze provádět souběžně s prvním průchodem analýzy toku dat. Z metodologických důvodů však tyto dva úkoly od sebe oddělíme.

Typickým a rychlým reprezentantem těchto metod je algoritmus [Gra76], který používá tyto transformace T1 až T3 (viz též obr. 6):



Obr. 6

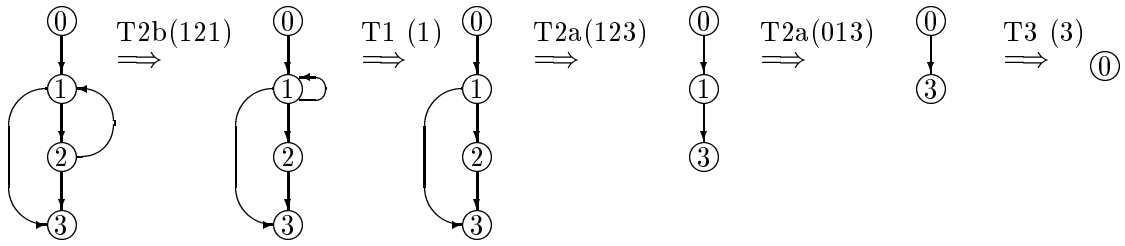
T1 odstraňuje hranu tvořící cyklus délky 1;
T2 zkracuje cestu délky 2 obsahující uzly x, y, z na cestu délky 1, přičemž nastávají tyto dvě možnosti:
(a) T2a eliminuje prostřední uzel y, jestliže jeho jediným následníkem je třetí uzel, to jest z;

(b) T2b zkracuje cesty bez eliminace uzlů v ostatních případech;
T3 eliminuje následníka y uzlu s, má-li y za následníka nejvýše sebe sama.

Nechť $G = (N, E, s)$ je GTŘ. Pak transformace T1 až T3 definujeme takto:
Transformace T1: jestliže pro nějaký $x \in N$ existuje hrana $e = (x, x) \in E$ a existuje právě jeden $u \in N - \{x\}$ takový, že $(u, x) \in E$, pak $\mathrm{T1}(G, e) \stackrel{\mathrm{def}}{=} (N, E - \{e\}, s)$;
Transformace T2: jestliže pro nějaký $y \in N - \{s\}$ existuje právě jeden $x \in N - \{y\}$ takový, že $(x, y) \in E$ a existuje-li libovolná hrana $e = (y, z) \in E$ $(y \neq z)$, pak $\mathrm{T2}(G, e) \stackrel{\mathrm{def}}{=} (N', E', s)$ kde
(a) nemá-li y jiného následníka než uzel z, pak $N' = N - \{y\}$ a $E' = E \cup \{(x, z)\} - \{(x, y), (y, z)\}$;
(b) v opačném případě $N' = N,\ E' = E \cup \{(x, z)\} - \{(y, z)\}$;
Transformace T3: nechť v G každá hrana, která není cyklem délky 1, vychází z uzlu s. Existuje-li $x \in N$ takový, že $e = (s, x)$ je jedinou hranou vstupující do x, pak $\mathrm{T3}(G, e) \stackrel{\mathrm{def}}{=} (N - \{x\}, E - \{e\}, s)$.

Lze ukázat, že graf toku řízení G je reducibilní v intervalovém smyslu, právě když je redukovatelný opakovanými aplikacemi transformací T1 až T3 na triviální graf.

Je-li dán GTŘ G, pak nejprve provádíme analýzu toku řízení: na G postupně aplikujeme uvedené transformace a současně konstruujeme tzv. *rozbor grafu G* - uchováváme pořadí použitých transformací ve tvaru posloupnosti prvků (t, S), kde $t \in \{\mathrm{T1}, \mathrm{T2a}, \mathrm{T2b}, \mathrm{T3}\}$ udává typ transformace a S specifikuje uzly, na něž je t aplikována (viz též příklad na obr. 7).



Obr. 7

Tyto transformace lze na graf toku řízení aplikovat ad hoc, avšak složitost algoritmu provádějícího analýzu toku dat je závislá na jejich pořadí. Autoři uvádí (viz [Gra76]) velice důmyslný algoritmus (využívající mimo jiné relace dominance - viz například [Tar74]) tak, aby byl minimalizován počet transformací T2, které jsou, díky výpočtu globálnějších *GEN* a *NOTKILL* i lokálnějších *LIVE*, z hlediska časového nejnáročnější.

Máme-li k dispozici rozbor GTŘ G, můžeme opět použít dvouprůchodového algoritmu: Fáze 1 počítá množiny *GEN* a *NOTKILL* podle níže uvedených pravidel, a to v pořadí, jak udává rozbor grafu G. Fáze 2 začíná opět přiřazením $LIVE(s) := GEN(s)$, kde s je jediný uzel limitního grafu. Následně, v revezi pořadí daného rozborem, počítáme postupně stále lokálnější množiny *LIVE* dle dále uvede-

ných pravidel. Po skončení této fáze máme určeny množiny *LIVE* pro všechny uzly původního, reducibilního grafu toku řízení G.

Pravidla pro výpočet *GEN* a *NOTKILL* při redukci grafu (fáze 1):

T1: žádný výpočet (**skip**)
T2a: $GEN(x) := GEN(x) \cup (NOTKILL(x,y) \cap GEN(y))$
$(NOTKILL(x,z) := NOTKILL(x,y) \cup (NOTKILL(x,y) \cap NOTKILL(y,z))$
T2b: $(NOTKILL(x,z) := NOTKILL(x,y) \cup (NOTKILL(x,y) \cap NOTKILL(y,z))$
T3: $GEN(s) := GEN(s) \cup (NOTKILL(s,x) \cap GEN(x))$
(při čtení pravidel doporučujeme konzultovat obr. 6)

Pravidla pro výpočet *LIVE* při zpětné expanzi grafu (fáze 2):

T1: žádný výpočet (**skip**)
T2a: $LIVE(y) := GEN(y) \cup (NOTKILL(y,z) \cap LIVE(z))$
T2b: $LIVE(y) := LIVE(y) \cup (NOTKILL(y,z) \cap LIVE(z))$
T3: $LIVE(x) := GEN(x)$
(při čtení pravidel pro *LIVE* je vhodné obrátit orientaci šipek $\Longrightarrow$ na obr.6)

Algoritmus GW - komprese cest

```
Vstup:   (1) Reduciblní GTŘ G = (N, E, s)
         (2) ∀x ∈ N : GEN(x); ∀x ∈ N : ∀y ∈ SUCC(x) : NOTKILL(x,y) .
         (3) seznam ROZBOR = < (t1,S1), ..., (tr,Sr) > redukující graf G,
             kde r =| ROZBOR | ;
Výstup:  ∀x ∈ N : LIVE(x);
begin    {fáze 1: lokální ⟶ globální}
for i := 1 to r do (t,S) := ROZBOR(i) od;
    case t of  T1:  skip;
               T2a: with S do GEN(x) := GEN(x) ∪ ( NOTKILL(x,y) ∩ GEN(y) );
                               NOTKILL(x,z) := NOTKILL(x,z) ∪
                                               (NOTKILL(x,y) ∩ NOTKILL(y,z))
                    od;
               T2b: with S do NOTKILL(x,z) := NOTKILL(x,z) ∪
                                               (NOTKILL(x,y) ∩ NOTKILL(y,z))
                    od;
               T3:  GEN(s) := GEN(s) ∪ (NOTKILL(s, S.x) ∩ GEN(S.x) )
    end  {case}
od; {for} {konec fáze 1; fáze 2: globální ⟶ lokální :}
LIVE(s) := GEN(s);
for i := r downto r do (t,S) := ROZBOR(i) od;
    case t of  T1:  skip;
               T2a: with S do LIVE(y) := GEN(y) ∪ (NOTKILL(y,z)∩LIVE(z)) od;
               T2b: with S do LIVE(y) := GEN(y) ∪ (NOTKILL(y,z)∩LIVE(z)) od;
               T3:  with S do LIVE(x) := GEN(x) od;
```

end {case}
od; {for}
end;

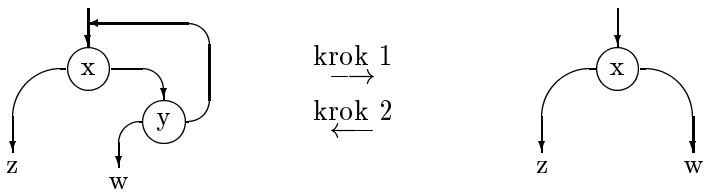
Algoritmus GW je časové složitosti $O(\mid E \mid . log \mid E \mid)$ a na strukturovaných programech je lineární, tj.$O(\mid E \mid)$, přičemž pro reálné (a tím spíše pro strukturované) programy je $\mid E \mid = O(\mid N \mid)$. Důkazy uvedených tvrzení viz [Gra76]. Existuje dokonce efektivnější implementace (založená na [Tar75] - viz [Tar81]), která je složitosti $O(\mid E \mid .\alpha(\mid E \mid, \mid N \mid))$, kde funkce α je vztažena k inverzi Ackermannovy funkce; je tudíž z praktického hlediska (asymptoticky) lineární, avšak implementačně velmi náročný. Navíc není dosud známo, zda je tato efektivnější verze aplikovatelná na problémy ATD typu "zpět".

4.3 Eliminační metody založené na grafových gramatikách

S cílem zjednodušit a zrychlit analýzu toku řízení (a tím i analýzu toku dat) se mnoho autorů snaží dále omezit třídu grafů, které by byly danou metodou redukovatelné. Zde budeme prezentovat nástin jedné z těchto metod, detailní popis lze najít v [Far76].

Pomocí grafové gramatiky (GG) se definuje třída tzv. téměř strukturovaných grafů (odpovídá třídě RE_1 - viz [HaK77] či, zhruba, řídícím strukturám jazyka Ada, Modula). Pravidla GG slouží k nalezení rozboru daného téměř strukturovaného GTŘ. Vlastní ATD probíhá (po nalezení rozboru grafu) ve známém dvouprůchodovém algoritmu, kdy s každým pravidlem GG je spojen výpočet globalizace *GEN* a *NOTKILL* (krok 1) a též lokalizace *LIVE* (krok 2). Algoritmus je na uvedené třídě lineární, v ostatních případech selhává.

Vzhledem k tomu, že pravidel GG je celkem devět a algoritmus hledání rozboru je poněkud komplikovaný (důkaz jeho korektnosti velmi komplikovaný), uvádíme na obr.8 pouze příklad pravidla pro cyklus s dvěma výstupy:



Obr. 8

Pravidla pro výpočet *GEN* a *NOTKILL* při redukci grafu:

krok 1: $GEN(x) := GEN(x) \cup (NOTKILL(x,y) \cap GEN(y))$
$NOTKILL(x,z) := NOTKILL(x,z) \cap NOTKILL(x,y)$
$NOTKILL(x,w) := NOTKILL(x,y) \cap NOTKILL(y,w)$

Pravidla pro výpočet *LIVE* při zpětné expanzi grafu:

krok 2: $LIVE(y) := GEN(y) \cup (NOTKILL(y,x) \cap LIVE(x)) \cup \cup (NOTKILL(y,w) \cap LIVE(w))$

5 Analýza toku dat vyšší úrovně

Všechny dosud zmíněné metody pracovaly s detailní (nízko-úrovňovou) reprezentací programu - grafem toku řízení. Vzniká otázka, zda lze uvedené analýzy provádět na vyšší úrovni reprezentace programu, například na syntaktickém stromu programu. Mnoho takových metod skutečně bylo navrženo ([Ros77], [Bab8], [Ros80], [Ros82], [Zad84]). Jejich filosofie staví na faktu, že většina současných programovacích jazyků má bohatý repertoár řídicích struktur, které explicitně (již v době definice jazyka) udávají tok řízení. U dříve uvedených metod se tato informace při budování grafu toku řízení ztrácí a pak ji více či méně pracně získáváme (v době kompilace) zpět konstrukcí rozboru GTŘ nebo ji (u iterativních metod) nebereme v úvahu a platíme za to ztrátou časové efektivity.

Metody analýzy toku dat vyšší úrovně pracují pro základní problémy ATD na strukturovaných programech v lineárním čase a oproti eliminačním metodám mají další výhodu ve své robustnosti: není-li graf toku řízení (téměř) struturovaný či reducibilní, pak neselhávají, ale stále poskytují řešení, i když - zhruba řečeno - s rostoucím počtem "vhodně volených" příkazů **goto** roste jejich časová složitost. Neposlední výhodou je, že lokální změna v programu (způsobená například kompilátorem provádějícím optimalizační transformace) má za následek opakování analýzy jen pro ten podstrom syntaktického stromu, v němž byla změna provedena; tzv. *analýza na žádost* (např. [Bab78a], [Zad84]). Nevýhodu lze spatřovat v tom, že jisté optimalizace vyžadují reprezentaci programu na velmi nízké úrovni - pak je nutno syntaktický strom uchovávat v podstatě po celou dobu překladu.

Základní rysy metod analýzy toku dat vyšší úrovně jsou založeny na vzájemně rekurzívních procedurách, analogicky jako například syntaktická analýza metodou rekurzívního sestupu. S každým symbolem gramatiky sdružíme množiny (atributy) *GEN*, *NOTKILL* a *LIVE*, *LVBOT*, kde *GEN* a *NOTKILL* pro terminály v syntaktickém stromu jsou dány - čtenář obeznámený s atributovými gramatikami (viz např. [Knu68]) na ně může nahlížet jako na syntetizované atributy. Globalizaci *GEN* a *NOTKILL* jistého strukturovaného příkazu odpovídá syntetizování těchto atributů z lokálních *GEN* a *NOTKILL* jeho složek. Analogickou úvahou pro *LIVE* a *LVBOT* šířícím se od příkazu k jeho složkám, dospějeme k chápání *LIVE* a *LVBOT* jako dědičných atributů.

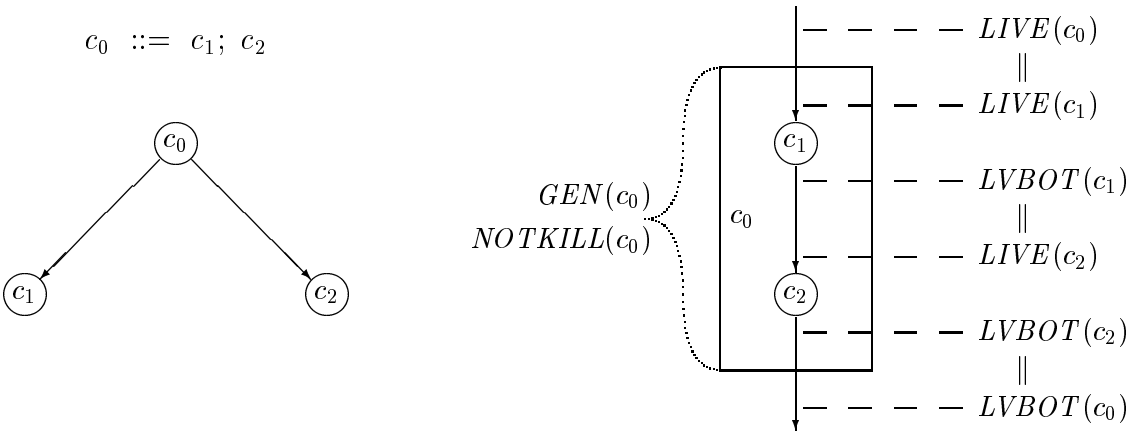
Ilustrujme základní rysy analýzy toku dat vyšší úrovně na příkladové gramatice s těmito pravidly:

(1) p ::= **begin** c **end**
(2) c_0 ::= c_1; c_2
(3) c_0 ::= **if** e **then** c_1 **else** c_2
(4) c_0 ::= **while** e **do** c_1
(5) c ::= e
(6) c_0 ::= $/label/$: c_1
(7) c_0 ::= **goto** $/label/$
(8) c ::= $/read - command/$
(9) c ::= $/write - command/$

Blíže nespecifikované neterminály (*e, /read - command/,...*) lze pro naše účely považovat za terminální symboly - jejich *GEN* a *NOTKILL* je snadné zjistit lokální analýzou. Tedy pro příkazy popsané pravidly (1), (5), (8) a (9) je situace jednoduchá: za předpokladu, že známe *LVBOT* těchto příkazů, platí tato rovnice:

$$(LIVE) \quad LIVE(c) := GEN(c) \cup (LVBOT(c) \cap NOTKILL(c))$$

U ostatních případů je situace složitější; ilustrujme ji na sekvenci - viz pravidlo (2) a obr. 9.



Obr. 9 a) syntaktický strom b) tok řízení a informace ATD

Z obr. 9 je zřejmé, že platí rovnosti:

$$(SEKVENCE) \quad \begin{aligned} LVBOT(c_2) &= LVBOT(c_0) \\ LVBOT(c_1) &= LIVE(c_2) \\ LIVE(c_0) &= LIVE(c_1) \end{aligned}$$

Na rozdíl od jednoduchých příkazů je naše situace komplikovanější tím, že neznáme *GEN* a *NOTKILL* příkazu c_0 a nemůžeme použít rovnici (*LIVE*) přímo. Podle způsobu řešení této situace lze metody analýzy toku dat vyšší úrovně rozdělit na tyto dva typy:

(1) Globalizovaná informace se hledá analogicky jako u eliminačních metod. V průchodu 1 globalizuje lokální informaci:

$$\begin{aligned} GEN(c_0) &:= GEN(c_1) \cup (NOTKILL(c_1) \cap GEN(c_2)) \\ NOTKILL(c_0) &:= NOTKILL(c_1) \cap NOTKILL(c_2) \ . \end{aligned}$$

V průchodu 2 (nyní shora-dolů) se globální $LIVE(c_0)$ a $LVBOT(c_0)$ lokalizují do c_1 a do c_2 :

$$\begin{array}{lcl} LVBOT(c_2) & := & LVBOT(c_0) \\ LIVE(c_2) & := & GEN(c_2) \cup (NOTKILL(c_2) \cap LVBOT(c_2)) \\ LVBOT(c_1) & := & LIVE(c_2) \\ LIVE(c_1) & := & GEN(c_1) \cup (NOTKILL(c_1) \cap LVBOT(c_1)) \end{array}$$

(První a třetí pravidlo plyne ze $(SEKVENCE)$, druhé a čtvrté jsou dány vztahem $(LIVE)$). Úplný výčet pravidel pro metody tohoto typu lze nalézt například v [Ken81] pro strukturované příkazy (bez **goto**, v [Ros80] je prezentována metoda umožňující analýzu i v případech, kdy jazyk obsahuje jak příkazy **goto**, tak i jeho omezené varianty (**exit, leave**, ...), ovšem za cenu vyšších nároků na čas.

(2) *GEN* a *NOTKILL* se neglobalizuje, a tyto metody vykazují jisté iterativní rysy. Je-li volána procedura, která má za úkol analyzovat sekvenci se skutečným parametrem c_0, musí se pro každou jeho složku volat procedura, jež ji analyzuje. Zřejmě je výhodnější volat nejprve proceduru pro analýzu c_2, a pak pro c_1: za předpokladu, že známe $LVBOT(c_0)$ a přijmeme konvenci, že po výpočtu $LIVE$ daného uzlu se tato hodnota posílá do $LVBOT$ jeho předchůdce, můžeme určit $LIVE(c_2)$; volání pro c_1 má již k dispozici $LVBOT(c_1)$ a dává $LIVE(c_1)$, což je hledaná $LIVE(c_0)$ (a $LVBOT$ jeho předchůdce). Pravidla pro sekvenci jsou tedy dána rovnicemi $(SEKVENCE)$, kde výskyt $LIVE(.)$ na pravé straně odpovídá rekurzívnímu volání procedury $LIVE$ pro daný typ příkazu.

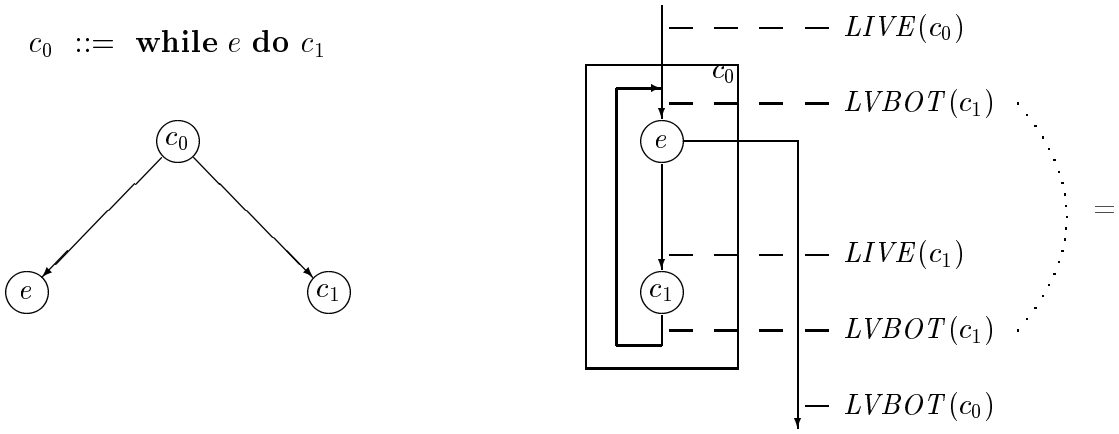
Pravidla pro větvení **if** e **then** c_1 **else** c_2 jsou dána tím, že e má dva následníky c_1 a c_2 a s nimi sdružené množiny $LIVE(c_1)$ a $LIVE(c_2)$, jejichž sjednocení je rovno $LVBOT_(e)$, a tím, že vyhodnocení výrazu e nezabíjí žádnou proměnnou:

$$(IF) \qquad \begin{array}{lcl} LVBOT(c_1) & := & LVBOT(c_0) \\ LVBOT(c_2) & := & LVBOT(c_0) \\ LIVE(c_0) & := & GEN(e) \cup LIVE(c_1) \cup LIVE(c_2) \end{array}$$

Pravidla pro c_0 ::= **while** e **do** c_1 nalezneme na základě analogických úvah (viz též obr. 10):

$$(WHILE) \qquad \begin{array}{lcl} LVBOT(c_1) & := & GEN(e) \cup LIVE(c_1) \cup LVBOT(c_0) \\ LIVE(c_0) & := & GEN(e) \cup LIVE(c_1) \cup LVBOT(c_0) \end{array}$$

V první rovnici se na pravé straně vyskytuje dosud neznámá hodnota $LIVE(c_1)$. Pro výpočet $LVBOT$ argumentu c_1 položíme při inicializaci "nejhorší předpoklad" $LIVE(c_1) = \emptyset$. Po výpočtu dle první rovnice a následném výpočtu $LIVE(c_1)$ se tento výsledek porovnává s předpokladem, s nímž jsme výpočet provedli, což je zároveň test na ukončení iterace (v případě rovnosti). Jinak provádíme další iteraci, tentokrát již s aktualizovaným předpokladem – poslední vypočtenou hodnotou $LIVE(c_1)$ (obr. 10).



Obr. 10 a) syntaktický strom b) tok řízení a informace ATD

Pro pravidlo c_0 ::= /*label*/ : c_1 je situace velice jednoduchá:

$$(LABEL) \quad \begin{array}{lcl} LVBOT(c_1) & := & LVBOT(c_0) \\ LIVE(c_0) & := & LIVE(c_1) \end{array}$$

Příkazu c_0 ::= **goto** /*label*/ odpovídá v syntaktickém stromu list, a tedy nepotřebujeme specifikovat rovnice pro $LVBOT($ složka $)$. Musíme však nalézt rovnice pro $LIVE(c_0)$, protože tuto hodnotu má příkaz posílat svému předchůdci. Je-li tímto příkazem řízení předáno na příkaz c, pak je zřejmě $LVBOT(c_0) = LIVE(c)$, a protože c_0 žádnou proměnnou ani nepoužívá ani nezabíjí, dostáváme $LIVE(c_0) = LVBOT(c_0)$. Rovnice pro toto pravidlo je tedy tvaru

$$(GOTO) \qquad LIVE(c_0) := LIVE(c),$$

kde c_0 je tvaru **goto** n a c je tvaru n: c .

Z tvaru výpočetních pravidel pro (a) příkaz while a pro (b) příkaz goto je vidět, že nemůžeme mluvit o atributové gramatice. V případě (a) by totiž byla atributová gramatika definována kruhem:

$$\begin{array}{lcll} LVBOT(c_1) & := & GEN(e) \cup LIVE(c_1) \cup LVBOT(c_0) & \text{viz}(WHILE) \\ LIVE(c_1) & := & GEN(c_1) \cup (NOTKILL(c_1) \cap LVBOT(c_1)) & \text{viz}(LIVE) \end{array}$$

V případě (b) není splněna podmínka závislosti atributů, kdy k výpočtu atributu smíme použít jen atributů předchůdce, sousedů či následníků, což však pro rovnici *(GOTO)* neplatí. Navíc i zde by případ zpětného skoku znamenal definici kruhem. Tedy i pro zpětné skoky musíme pracovat s předpoklady - stejným způsobem jako u příkazu while. Algoritmus tudíž iteruje, dokud se všechny předpoklady nestabilizují, tj. ve dvou po sobě jdoucích průchodech si nejsou rovny.

Pro analýzu složitosti uvedeného algoritmu je vhodné si uvědomit, že pro libovolný uzel u syntaktického stromu programu platí, že je-li proměnná $v \in LIVE(u)$ po $j-$té iteraci, pak $v \in LIVE(u)$ po každé další iteraci. (Dokáže se indukcí,

uvážíme-li, že *GEN* a *NOTKILL* jsou konstantní a iteraci začínáme s předpokladem $LIVE(u) = \emptyset$.) Důsledek je tento:

Pro každý program obsahující W příkazů cyklu a L příkazů s návěštím, na něž předává řízení zpětný skok, skončí algoritmus nejpozději po W+L+2 iteracích.
Nástin důkazu: Množiny předpokladů mají $W + L$ prvků. Pro libovolný průchod $j (j > 1)$ platí, že předpoklady pro $(j-1)-$ní a $j-$tý průchod jsou si rovny (a v tom případě algoritmus končí) nebo aspoň jeden předpoklad, který, řekněme proměnnou v, neobsahoval, ji nyní obsahuje. To značí, že po $k-$tém průchodu alespoň $k-1$ předpokladů proměnnou v obsahuje, a tedy po provedení $W+L+1$ průchodů musí všechny předpoklady proměnnou v obsahovat. Jinými slovy, množina předpokladů pro iteraci $W+L+1$ musí být rovna výsledkům této iterace a algoritmus končí.

Poznamenejme, že uvedený nejhorší případ nastává, musí-li se "dobrá zpráva" *GEN* o živosti nějaké proměnné šířit po acyklické cestě obsahující $W+L$ zpětných hran reprezentujících návraty v cyklech while a zpětné příkazy goto. Jelikož pomocí pouze cyklů while nelze (při vhodném modelování) vytvořit acyklickou cestu složenou ze zpětných hran délky větší než 1, lze ukázat, že neobsahuje-li program příkazy skoku, pak algoritmus končí po dvou průchodech - [Bab78].

6 Obecný model analýzy toku dat

Dosud jsme se zabývali jediným konkrétním problémem analýzy toku dat. Jelikož problémy ATD, jak ukážeme dále, jsou velmi podobné, je vhodné vybudovat jejich obecnější model a studovat jeho vlastnosti s tím, že algoritmy a jejich vlastnosti (ukončení, korektnost, složitost) pro konkrétní problémy dostaneme jako instance obecného modelu, u nichž již zmíněné vlastnosti nemusíme (problém od problému) dokazovat.

6.1 Řešení dalších základních problémů ATD

Dosažitelnost definic

Definice d proměnné x (např. *x:= ..., read(x)*) dosahuje vstupu (resp. výstupu) uzlu v právě když d je v uzlu u takovém, že existuje cesta z u do v podél níž není x redefinována.

Je vidět, že dosažitelnost definice d (proměnné x) je generována v uzlu u, jestliže d je v uzlu u a není v tomto uzlu následována svou redefinicí. Množinu takových definic v uzlu u označíme *GEN(u)*.

Dosažitelnost definice d proměnné x je zabita uzlem u, jestliže x je v uzlu u redefinována; množinu všech proměnných (uvažovaného programu), které nejsou v uzlu u redefinovány označíme *NOTKILL(u)*.

Konečně *RDTOP(u)*, resp. *RDBOT(u)* značí hledanou informaci, tj. množinu všech definic dosažitelných na vstupu, resp. výstupu uzlu u. Pak problém dosažitel-

nosti definic pro GTŘ *G=(N,E,s)* lze specifikovat systémem (vzájemně rekurzívních) rovnic *(RD1)* až *(RD3)*, resp. systémem *(REACH)* a *(RD3)*.

$$(RD1) \qquad RDBOT(u) = (RDTOP(u) \cap NOTKILL(u)) \cup GEN(u) \ \forall u \in N$$

$$(RD2) \qquad RDTOP(u) = \bigcup_{v \in PRED(u)} RDBOT(v) \quad \forall u \in N$$

je-li PRED(s) $= \emptyset$ *, pak RDTOP(u)* $= \emptyset$

$(RD3)$ *řešením REACHES problému je nejmenší řešení (vzhledem k inklusi) splňující (RD1) a (RD2).*

$$(REACH) \quad RDTOP(u) = (\bigcup_{v \in PRED(u)} RDTOP(v) \cap NOTKILL(v)) \cup GEN(v)$$

Problém dostupných výrazů

Výraz e je dostupný na vstupu, resp. výstupu uzlu v právě když na každé cestě z počátečního uzlu s do uzlu u byl již e vyhodnocen a dále již nebyla hodnota žádného jeho operandu měněna.

Dostupnost výrazu je generována jeho vyhodnocením v uzlu, například jeho výskytem na pravé straně přiřazovacího příkazu. Množinu všech výrazů vyhodnocovaných v uzlu u označíme *GEN(u)*.

Naopak dostupnost výrazu je zabita v uzlu, jestliže některý z jeho operandů je v tomto uzlu redefinován. Množinu výrazů, které nejsou zabity v uzlu u označíme *NOTKILL(u)*.

Problém dostupných výrazů specifikuje systém (vzájemně rekurzívních) rovnic *(AV1)* až *(AV3)*, resp. *(AVAIL)* a *(AV3)*, kde *AVTOP(u)*, resp. *AVBOT(u)* označuje hledanou informaci - množinu výrazů dostupných na vstupu, resp. výstupu uzlu u.

$$(AV1) \qquad AVBOT(u) = (AVTOP(u) \cap NOTKILL(u)) \cup GEN(u) \ \forall u \in N$$

$$(AV2) \qquad AVTOP(u) = \bigcap_{v \in PRED(u)} AVBOT(v) \quad \forall u \in N$$

$AVTOP(u) = \emptyset$

$(AV3)$ *řešením AVAIL problému je největší řešení (vzhledem k inklusi) splňující (AV1) a (AV2).*

$$(AVAIL) \qquad AVTOP(u) = (\bigcap_{v \in PRED(u)} AVTOP(v) \cap NOTKILL(v)) \cup GEN(v)$$

Problém využitelných výrazů

Výraz e je využitelný v bodě p (tj.na vstupu, resp. výstupu uzlu), právě když na všech cestách z p do koncového uzlu je e referencován a na žádné z těchto cest není redefinována hodnota některého jeho operandu).

Množiny *NOTKILL(u)* jsou definovány jako u problému dostupných výrazů a GEN(u) je množina všech výrazů, které jsou v uzlu u vyhodnoceny, avšak mezi vstupem do u a jejich vyhodnocením v u není měněna hodnota žádného jejich operandu.

Problém využitelných výrazů specifikuje systém *(VB1)* až *(VB3)*, resp. *(BUSY)* a *(VB3)*, kde *VBTOP(u)* resp. *VBBOT(u)* značí množinu výrazů využitelných na vstupu, resp. výstupu uzlu u.

$$(VB1) \qquad VBTOP(u) = (VBBOT(u) \cap NOTKILL(u)) \cup GEN(u) \; \forall u \in N$$

$$(VB2) \qquad VBBOT(u) = \bigcap_{v \in SUCC(u)} VBTOP(v) \quad \forall u \in N$$

je-li $SUCC(s) = \emptyset$ *, pak* $VBBOT(u) = \emptyset$

$(VB3)$ *řešením BUSY problému je největší řešení (vzhledem k inklusi) splňující (VB1) a (VB2).*

$$(BUSY) \qquad VBBOT(u) = (\bigcup_{v \in PRED(u)} VBBOT(v) \cap NOTKILL(v)) \cup GEN(v)$$

Systémy *(REACH), (AVAIL)* a *(BUSY)* lze řešit iteratívní metodou, jak byla uvedena ve 3.1, kterou je však nutno problém od problému modifikovat tak, jak udávají jednotlivé systémy. V 6.3 uvedeme obecný algoritmus, jehož konkrétními instancemi dostaneme algoritmy řešící nejen dosud uvedené, ale i další problémy analýzy toku dat.

Problém adaptace eliminačních metod může být komplikovanější (v závislosti na metodě a zejména pro problémy ATD typu "zpět"); obecný model uvedeme v 6.4.

6.2 Informace v analýze toku dat

(1) *Informaci* (např. množiny *LIVE*) *lze modelovat pomocí prvků polosvazu.*

Často potřebujeme popsat prvek Z, který by nesl informaci obsaženou buď v nějakém prvku X nebo v nějakém prvku Y, či duálně, informaci, která je společná prvku X i Y. Takový prvek budeme nazývat *spojením* (či duálně *průsekem*) X a Y. U LIVE-problému chceme, aby živost na výstupu byla spojením Z informací od následníků $X, \ldots, Y$.

Dvojici $(S, \sqsubseteq)$ nazveme *částečně uspořádanou množinou (posetem)*, je-li na S definována relace $\sqsubseteq$ částečného uspořádání ("je menší nebo rovno než" či "má menší nebo stejný informační obsah než"); $x \sqsubset y$ značí $x \sqsubseteq y$ a $x \neq y$. Spojením prvků x a y nazveme, pokud existuje, prvek $z \in S$ takový, že $x \sqsubseteq z$, $y \sqsubseteq z$, přičemž neexistuje prvek $z' \in S$, takový, že $x \sqsubseteq z' \sqsubset z$ a $y \sqsubseteq z' \sqsubset z$. Duálně - záměnou $\sqsubseteq$

za $\sqsupseteq$ definujeme průsek. Mají-li $x, y \in S$ jediné spojení (resp. průsek), značíme jej $\bigvee\{x, y\}$ (resp. $\bigwedge\{x, y\}$), či infixově $x \vee y$ (resp. $x \wedge y$).

Svazem $(S, \bigvee, \bigwedge)$ nazveme poset $(S, \sqsubseteq)$, kde každé dva prvky mají jediné spojení a jediný průsek. Jestliže navíc spojení $\bigvee H$ prvků množiny H a průsek $\bigwedge H$ existují pro všechna H, $H \subseteq S$, nazýváme S *úplným svazem*. Tento má nejmenší prvek (*infimum*, nulu) $\bot = \bigwedge S$ a největší prvek (*supremum*, jedničku) $\top = \bigvee S$ a značíme jej $S(\sqsubseteq, \bigvee, \bigwedge, \bot, \top)$.

Příklad 1a : Nechť $\mathcal{P}(\mathcal{S})$ je množina všech podmnožin množiny S. Je-li S konečná, pak $\mathcal{P}(\mathcal{S})(\subseteq, \bigcup, \bigcap, \emptyset, \mathcal{S})$ je úplný svaz. Tedy i množiny *LIVE*, resp. *LVBOT, LVTOP* (tj. systém podmnožin konečné množiny proměnných libovolného daného programu) s operacemi $\bigcup$ a $\bigcap$, tvoří úplný svaz.

Je-li dán svaz $(S, \bigvee, \bigwedge)$, pak k výchozímu posetu $(S, \sqsubseteq)$ lze dospět tak, že položíme $x \sqsubseteq y$ právě když $x \vee y = y$, resp. $x \wedge y = x$. Často vystačíme s jednodušší strukturou - *polosvazem (S,*)*, kde S je neprázdná množina a $*$ binární operace, která je idempotentní ($x * x = x$), komutativní a asociativní na S. Je-li $(S, \bigvee, \bigwedge)$ svaz, pak $(S, \bigvee)$ a $(S, \bigwedge)$ jsou *spojový* a *průsekový* polosvaz.

Nejvýše spočetnou podmnožinu $C \subseteq S$, $C = \{x_0, x_1, \ldots, x_n, \ldots\}$ nazveme *rostoucím řetězcem*, jestliže $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$. $(S, \sqsubseteq)$ nazveme *dobře založená* množina (spňující podmínku rostoucích řetězců), jestliže pro $x_i \in S$, $i = 0, 1, \ldots$ a $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ existuje m takové, že $x_m = x_{m+1} = \ldots$. Duálně lze definovat klesající řetězec.

Příklad 1b : *Problém šíření konstant* (CP-*problém*).
Tento problém spočívá v tom, že máme pro každý uzel GTŘ určit ty proměnné, jimž je na vstupu do daného uzlu (při libovolném výpočtu podél libovolné cesty od počátečního uzlu do uzlu daného) vždy přiřazena stejná konstantní hodnota. Nechť *Var* je konečná množina proměnných (daného programu) a C je množina možných hodnot konstant. Pro CP-problém zvolíme za informaci sdruženou se vstupem do uzlu podmnožinu množiny $Var \times C$; například množina $\{(A, 1), (B, 2)\}$ sdružená s nějakým uzlem značí, že podél libovolné cesty z počátečního uzlu do daného uzlu nabývá proměnná A vždy hodnotu 1 a B hodnotu 2. Zřejmě pro $L = \mathcal{P}(Var \times \mathcal{C})$ je $(L, \bigwedge)$ polosvaz: L je množina funkcí z konečné množiny *Var* do C a $\bigwedge$ je množinový průnik nad uspořádanými dvojicemi tvaru $(V, r) \in L$; L má infimum $\emptyset$ a nemá supremum.

Tedy s každým bodem (vstupem, resp. výstupem uzlu) grafu toku řízení sdružujeme jistý prvek polosvazu; operace spojení reprezentuje efekt spojení informace pocházející od následníků/předchůdců (analogicky pro průsek).

Zbývá modelovat efekt výpočtu v uzlu (efekt lokální informace). Jde vlastně o funkci, která transformuje informaci (prvek polosvazu) na vstupu/výstupu uzlu na informaci na výstupu/vstupu uzlu.

(2) *S každým uzlem (hranou) sdružíme funkci* $f : S \to S$, *kde* S *je polosvaz.*

Příklad 2a : (pokračování 1a) Pro určování živých proměnných platí rovnice *(LIVE)* transformující *LVBOT* - spojení *LIVE* následníků na *LIVE* uzlu, kde pro libovolný, ale pevně daný uzel jsou jeho *GEN* a *NOTKILL* konstantní. Označme je g a k. Pak uvedená transformace je vlastně funkcí $f_{<g,k>}$ takovou, že

$$f_{<g,k>}: \ S \to S, \quad \text{kde } f_{<g,k>}(x) = (x \cap k) \cup g$$

Je-li každému uzlu u přiřazena odpovídající lokální funkce f_u, pak *LIVE*-problém lze modelovat spojovým polosvazem $(S, \bigvee)$, kde $S = \mathcal{P}(Var)$ a rovnice *(LIVE)* je nyní tohoto tvaru:

$$\textit{(LIVE)} \qquad LIVE(x) \ = \ \bigvee\{f_y(LIVE(y));\ y \in SUCC(x)\}$$

Příklad 2b : (pokračování 1b) Přiřaďme každému uzlu funkci definovanou následovně. Pro jednoduchost předpokládejme, že uzly obsahují příkazy tvaru $A{:=}B\ op\ C$ nebo $A{:=}r$, kde $r \in C$, $op \in \{+, -, *, /\}$ a $A, B, C \in Var$. Funkce reprezentující uzel je kompozicí funkcí reprezentujících jednotlivé příkazy uvnitř uzlu. Pro $x \in L$ položme:

1. $f_{<A:=r>}(x) = y$, kde $\begin{array}{l} y(V) = x(V),\ \forall V \in Var - \{A\} \quad a \\ y(A) = r \end{array}$
   tj. y dostaneme z x tak, že vypustíme dvojici s první složkou A a přidáme dvojici (A, r).
2. $f_{<A:=BopC>}(x) = y$, kde $\begin{array}{l} y(V) = x(V),\ \forall V \in Var - \{A\} \quad a \\ y(A) = b\ op\ c\ ,\ je - li\ x(B) = b\ a\ x(C) = c\ , \\ y(A) \text{ není definováno - v ostatních případech .} \end{array}$

Je-li s každým uzlem u sdružena funkce f_u uvedeného tvaru, pak CP problém lze modelovat průsekovým polosvazem $(L, \bigwedge)$ - viz příklad 1b - a rovnice určující množinu *CP* proměnných konstantních na vstupu do uzlu x lze psát v tomto tvaru:

$$(CP) \qquad CP(x) = \bigwedge\{f_y(CP(y));\ y \in PRED(x)\}$$

V uvedených příkladech (i u ostatních problémů ATD) by funkce f_u reprezentující efekt výpočtu v uzlu měly být konzistentní se sémantikou daného jazyka. U základních problémů ATD je tato konzistence ověřitelná na syntaktické úrovni, proto se tyto problémy někdy nazývají syntakticky orientované; ostatní problémy ATD se nazývají sémanticky orientované (např. CP problém). Sémantika analýzy toku dat je speciální (a jednoduchým) příkladem tzv. předvýpočtové sémantiky; podmínky konzistence těchto sémantik se standardní sémantikou jazyka lze nalézt v [Kře85b]. V konkrétním a jednoduchém případě analýzy toku dat však lze, alespoň na intuitívní úrovni požadovat, aby funkce f_u byly z tohoto hlediska "rozumně navržené".

Nechť $(S, \sqsubseteq)$ je poset. Funkci $f: \ S \to S$ nazveme *monotónní*, jestliže $\forall x, y \in S$: $(a \sqsubseteq b) \Rightarrow (f(x) \sqsubseteq f(y))$. Snadno se nahlédne, že f je monotónní na polosvazu $(S, \sqsubseteq)$ právě když $\forall x, y \in S: \ f(x \wedge y) \sqsubseteq f(x) \wedge f(y)$. Je-li $\forall x, y \in S: \ f(x \wedge y) = f(x) \wedge f(y)$, nazveme fukci f *distributivní*. f je $(\wedge)$ *spojitá* funkce, jestliže pro každý klesající

řetězec $C \subseteq S$ platí $f(\bigwedge C) = \bigwedge f(C)$. Platí-li uvedená rovnost pro libovolnou množinu $C \subseteq S$, pak f je *aditivní* (nekonečně distributivní) funkce. Zřejmě aditivita imlikuje spojitost, která implikuje distributivitu, která imlikuje monotónii.

Je-li $(L, \sqsubseteq)$ poset, funkce $f : L \to L$ monotónní, pak $x \in L$ nazveme *pevným bodem funkce f*, právě když $x = f(x)$. *Nejmenším pevným bodem* funkce f nazveme (pokud existuje) prvek $\bigwedge\{x \in L;\ f(x) \sqsubseteq x\}$, *největším pevným bodem* f nazveme prvek $\bigvee\{x \in L;\ f(x) \sqsupseteq x\}$.

6.3 Model pro iterativní metody

Je-li dán dobře založený polosvaz $(L, \sqsubseteq)$ s nejmenším prvkem $\bot$ a množina funkcí $F = L \to L$, pak F nazveme *prostorem monotónních funkcí sdruženým s* L, právě když jsou splněny tyto podmínky:

(M1) $\forall x, y \in L.\ \forall f \in F :\ f(x \wedge y) \sqsubseteq f(x) \wedge f(y)$ (tj. každá funkce z F je monotónní);
(M2) $\exists id \in F.\ \forall x \in L :\ id(x) = x$ (v F existuje identická funkce id);
(M3) $\forall f, g \in F \Rightarrow f.g \in F$ (F je uzavřena vzhledem ke kompozici);
(M4) $\forall x \in L.\ \exists f \in F :\ x = f(\bot)$ (L je roven uzávěru $\{\bot\}$ vzhledem ke kompozici a průseku).

Dvojici (L, F) nazveme *(algebraický) monotónní kontext analýzy toku dat.* (Duálně definujeme prostor F pro spojový polosvaz.)

Podmínka *(M1)* odpovídá intuitivnímu požadavku, aby většímu informačnímu obsahu argumentu x odpovídal i větší informační obsah funkční hodnoty $f(x)$; *(M2)* odpovídá situaci, kdy uzel neobsahuje žádný příkaz, který by ovlivňoval informaci při průchodu tímto uzlem. *(M3)* uvažuje zpracování informace při průchodu sekvencí dvou po sobě následujících uzlů a *(M4)* zabraňuje tomu, aby L obsahoval irelevantní informaci (nadbytečné prvky) - infimum $\bot$ reprezentuje prázdnou informaci na vstupu vstupního uzlu před započetím analýzy.

Instancí (monotónního) kontextu nazveme čtveřici $I = (L, F, G, M)$, kde (L, F) je (monotónní) kontext, $G = (N, E, s)$ je GTŘ a $M : N \to F$ je funkce přiřazující každému uzlu z N operaci z F. (Pro eliminační případ obvykle $M : E \to F$.)

Důležitou podtřídu monotónních kontextů tvoří *distributivní kontexty*, tzn., že všechny funkce z F jsou distributivní:

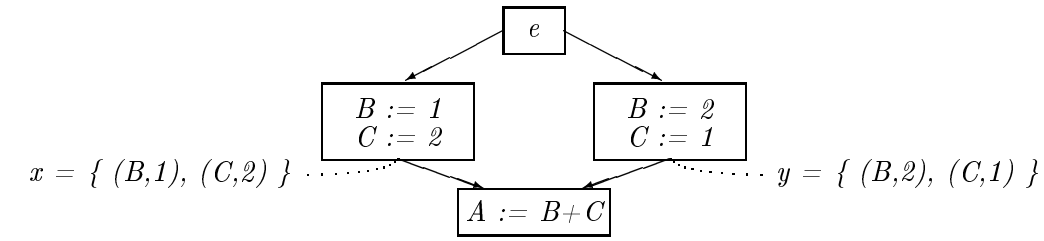
(D1) $\forall x, y \in L.\ \forall f \in F :\ f(x \wedge y) = f(x) \wedge f(y).$

Intuitivně lze uvedenou rovnost interpretovat tak, že při operaci průseku nedochází ke ztrátě informace. V podmínce *(M1)*, která je implikována podmínkou *(D1)*, nerovnost $\sqsubseteq$ reprezentuje pouze konzervativní, tj. bezpečnou aproximaci "přesného" řešení.

Příklad 3a : Model pro *LIVE* problém, prezentovaný v příkladu 2a, tvoří distributivní kontext.

Příklad 3b : Model pro CP problém z příkladu 2b je monotónní kontext, který není distributivní. Ilustrujme to následujícím protipříkladu instance monotónního kontextu z obr. 11).

Kontexty tohoto typu jsou monotónní a distributivity lze dosáhnout volnou interpretací operátorů, tj. jestliže výsledek aplikace n-árního operátoru na dvě různé n-tice není nikdy stejný - [Kam77]. Odlišný přístup je uveden v [Kil73], kdy za cenu časové složitosti je uvažována i komutativita operací.



kde $< A := B + C > (x \wedge y) = \emptyset$,
ale $< A := B + C > (x) \wedge < A := B + C > (y) = \{(A, 3)\}$.

Obr. 11

Konečně poznamenejme, že podmínka dobré založenosti L v definici monotónního kontextu umožňuje nalézt průsek řetězce $C = \{c_1, c_2, \ldots\}$ v konečném čase; pak totiž existuje přirozené m takové, že $\bigwedge\{c_i; c \in C\} = \bigwedge\{c_i; 1 \leq i \leq m\}$.

Máme-li formulován problém ve tvaru systému rovnic (srovnej příklady 2a a 2b)

$$\begin{array}{lcl} x_1 & = & f_1(x_1, \ldots, x_n) \\ & \ldots & \\ x_n & = & f_n(x_1, \ldots, x_n) \end{array} \qquad resp. \quad X = f(X) \qquad (6.1)$$

můžeme použít k nalezení extremálního (pro spojení minimálního - viz *LIVE* problém, požadavek *(L3)*) pevného bodu funkce f základní iterační algoritmus MFP:

1. inicializuj X ;
2. **while** $f(X) \neq X$ **do** $X := f(X)$ **od** ;
3. **return** X $\{= f(X)\}$.

Je-li totiž funkce f monotónní, tj. všechny f_i z (4.1) jsou monotónní, pak množina $C = \{X^i; i \geq 0\}$, kde (v případě spojového polosvazu):

$$X^0 = (\bot, \ldots, \bot) \qquad a \quad X^{i+1} = f(X^i), \quad i \geq 0 \qquad (6.2)$$

tvoří řetězec, a tedy algoritmus MFP končí.

Ukažme ještě, že supremum řetězce C je vhodné považovat za řešení systému *(4.1)* (srovnej s definicí *LIVE*): Označíme-li X_{iter} řešení systému *(4.1)*, pak $\forall i. i \geq 0 : \; X^i \sqsubseteq X_{\text{iter}}$, tj. každé X^i nese částečnou informaci konzistentní s konečným řešením X_{iter} zadaným specifikací *(4.1)*. Obráceně, obsahuje-li X_{iter} jistý fakt, pak musí existovat nějaké (a zde konečné) $i \geq 0$ takové, že tento fakt je v X^i obsažen, což dává $X_{\text{iter}} \sqsubseteq \bigvee\{X^i; i \geq 0\}$, a tedy $X_{\text{iter}} = \bigvee\{X^i; i \geq 0\}$. Systém *(4.1)*, jak víme z *LIVE* problému, může mít více řešení (pevných bodů), ale iterace *(4.2)* najde nejmenší z nich: je-li totiž X_{lib} nějakým řešením *(4.1)*, pak musí být pevným bodem *(4.1)*. Indukcí ověříme, že $X^i \sqsubseteq X_{\text{lib}}, i = 0, 1, \ldots$ ($\bot = X^0 \sqsubseteq X_{\text{lib}}$ z definice $\bot$; pro $i > 0$ předpoklad $X^{i-1} \sqsubseteq X_{\text{lib}}$ a monotónie f dává žádané $X^i = f(X^{i-1}) \sqsubseteq f(X_{\text{lib}} = X_{\text{lib}})$. Je tedy X_{lib} horní závorou C - viz *(4.2)*, ovšem X_{iter} je supremum C, což značí $X_{\text{iter}} \sqsubseteq X_{\text{lib}}$. Na základě těchto úvah je tedy korektní algoritmus $\bigvee$MFP.

Algoritmus - $\bigvee$MFP

Vstup: Instance $I = (L, F, G, M)$ monotónního kontextu.
Výstup: Řešení pro instanci I

```
begin    {uzly grafu G očíslovány 1,2,...,n }
{inicializace:}  for i:=1 to n do X(i) := ⊥ od;
{iterace:}       change := true;
                 while change do
                     change := false;
                     for i:=1 to n do
                         NEW := ⋁ {f_k(X(k)); k ∈ PS(i)}
                         if NEW ≠ X(i) then change:=true; X(i) := NEW fi;
                     od;
                 od;
                 return ( X(1), ... , X(n) )
end.
```

V algoritmu $\bigvee$MFP zkratka PS značí a) pro problémy typu vpřed předchůdce uzlu, tj. PS::=*PRED*, b) pro problémy typu zpět následníky, tj. PS::=*SUCC*. V případě a) je vhodné volit pořadí uzlů *rPostorder*, v b) pořadí *Postorder*. (Je tedy vidět, že algoritmus Iterativní LIVE z 3.1 je instancí obecného algoritmu $\bigvee$MFP.)

Algoritmus $\bigvee$MFP lze snadno modifikovat na algoritmus $\bigwedge$MFP; za zmínku stojí fáze inicializace: Nechť v $G = (N, E, s)$ má uzel s pořadové číslo 1. Pak inicializace má tento tvar:

```
X(1) := ⊥;  {při startu žádná informace na vstupu s }
 for i:=2 to n do X(i):=⊤ od   {⊤ je duální k ⊥ }
```

Nemá-li L supremum $\top$, lze takový prvek přidat uměle - viz např. [Kam76], [Hec77].

Všimněme si složitosti uvedených algoritmů. Případ $\bigwedge$MFP je zřejmě díky možné neexistenci suprema (viz inicializace) obecnější (transformace výsledků na duální případ je triviální). Musíme se omezit jen na konstatování faktů, důkazy lze nalézt v [Kam77] - jsou obdobné jako naše úvahy o složitosti v čl. 3.1.

Je-li dána libovolná instance $I = (L, F, G, M)$ distributivního kontextu pro problém ATD typu vpřed a uspořádání rPostorder pro $G = (N, E, s)$, pak algoritmus $\bigwedge$MFP končí nejvýše po $d(G) + 3$ iteracích právě když pro (L, F) platí:

$$\forall x \in L.\ \forall f, g \in F:\ fg(\bot) \ \sqsupseteq\ g(\bot) \wedge f(x) \wedge x\ . \qquad (6.3)$$

Tvrzení lze zesílit v případě, že L má jedničku: za stejných předpokladů algoritmus $\bigwedge$MFP konverguje nejvýše po $d(G) + 2$ iteracích právě když pro (L, F) platí:

$$\forall x \in L.\ \forall f, g \in F:\ fg(\bot) \ \sqsupseteq\ g(\bot) \wedge f(\bot)\ . \qquad (6.4)$$

V případě existence suprema jsou *(6.3)* a *(6.4)* ekvivalentní. Poznamenejme, že podmínka *(6.3)* je implikována podmínkou

$$\forall x \in L.\, \forall f \in F:\ f(x) \sqsupseteq x\ . \qquad (6.5)$$

a implikuje podmínku

$$\forall x \in L.\, \forall f \in F:\ ff(\bot) \sqsupseteq f(x) \wedge x\ . \qquad (6.6)$$

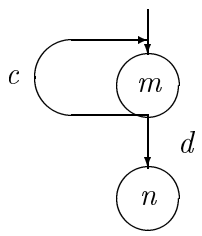
avšak není ekvivalentní ani s jednou z nich (srovnejte *(6.5)* a *(6.6)* s podmínkami kladenými na kontexty eliminačních metod v čl. 6.4).

Z praktického hlediska je vhodné si (znovu) uvědomit, že cyklická souvislost $d(G)$ reducibilního GTŘ je menší, nejvýše rovna intervalovému řádu G, který je pro GTŘ reálných programů na základě empirických výsledků ([Knu71]) v průměru 2,75 a lze tedy soudit, že 5 až 6 iterací je horní mez pro praktické problémy ATD splňující *(6.4)*, resp. *(6.3)*.

Všimněme si stručně případu, kdy kontext ATD není distributívní. Definice problémů ADT požaduje tzv. *MOP* (meet-over-all-paths) *řešení* tohoto tvaru (srovnejte s rovnicí (*) v čl. 3.1): $\bigvee\{f_p(\bot); p \in PATH(u)\}$ pro všechny uzly u (duálně $\bigwedge\{f_p(\bot); p \in PATH(u)\}$), kde pro cestu p obsahující po řadě uzly $u_j, \ldots, u_k$ značí (pro $f_i = M(u_i)$) $f_p = f_k, \ldots, f_j$. Uvedený $\bigwedge$MFP algoritmus dává pro distributivní kontext MOP řešení, avšak z příkladu 3b nedistributivního kontextu pro CP problém je vidět, že v obecném (monotónním) případě MOP řešení není extremálním pevným bodem, který nalezne $\bigwedge$MFP algoritmus. V každém případě však MFP řešení X_{iter} je menší nebo rovno MOP řešení (viz [Kam77]) a je tedy jeho bezpečnou aproximací.

6.4 Model pro eliminační metody

Eliminační metody se snaží *sumarizovat efekt šíření informace* podél jistých cest mezi dvojicemi uzlů. Uvažované množiny cest se však u jednotlivých metod různí. To, co mají společné, je nutnost sumarizovat efekt (tj. $\bigvee$ nebo $\bigwedge$) nekonečně mnoha cest pomocí konečného počtu operací na F (včetně funkce *eval:* $F \times L \to L$, $eval(f, x) = f(x)$, kterou používají i iterační metody).



Obr. 12

Uvažujme jednoduchý příklad z obr. 12 a mějme instanci kontextu $I = (L, F, G, M)$. Zobrazení M rozšířené z hran na cesty značme opět M. Je vidět, že z uzlu m do uzlu n existuje nekonečná množina cest: $P = \{d, (c, d), (c, c, d), \ldots\} = \{c^*d\}$. Je-li instancí I hraně c přiřazena funkce f, hraně d funkce g, pak hledaná množina

$$M(P) = \{M(p); p \in P\} = \{g, gf, gff, \ldots, gf^i, \ldots\}$$

je obecně nekonečná. V ideálním případě bychom rádi pracovali s tzv. *spojitými kontexty*, kdy pro prostor funkcí F platí tyto podmínky:

(S1) F obsahuje identickou funkci id;
(S2) F je uzavřena na kompozici, průsek a na operaci *, kde $f^*(x) = \bigwedge\{f^i(x); i \geq 0\}$;
(S3) každá funkce $f \in F$ je spojitá.

Problém eliminační metod spočívá v nalezení funkce f^* sumarizující efekt cyklu v konečně mnoha krocích. Zřejmě nejslabší rozumný a reálný model je *(eliminační) monotónní kontext*, kdy pro prostor funkcí F platí tyto podmínky:

(EM1) F obsahuje identickou funkci id;
(EM2) F je uzavřena na kompozici a průsek;
(EM3) každá funkce $f \in F$ je monotónní;
(EM4) *aproximace* f^*: $\forall f \in F.\ \exists f^a \in F:$ (i) $\forall x \in L.\ \forall i \geq 0:\ f^a(x) \sqsubseteq f^i(x)$
(*ii*) $f(x) \sqsupseteq x \ \Rightarrow\ f^a(x) \sqsupseteq x$.

Prezentujme další (postupně slabší) podmínky kladené na monotónní prostor F.

Nejjednodušším řešením úkolu aproximace F^* (sumarizace $M(P)$) je předpokládat *idempotentní kontext (L,F)*, kdy platí podmínky *(EM)* a navíc platí podmínka

(I1) $\forall f \in F:\ f.f = f$,

tedy $f^* = f$ (srovnejte s *(6.5)*). Intuitivně odpovídá idempotence tomu, že cykl stačí projít jedenkrát. V příkladě z obr. 12 je pak $M(P) = \{g, gf\}$ a lze ji sumarizovat jako $g \wedge (g.f)$. Klasickým případem, který využívá idempotence, je intervalová analýza uvedená v čl. 4.1.

Pro některé problémy ATD idempotentní kontext nenalezneme a hledáme tudíž slabší podmínku pro aproximaci f^*, a to i za cenu jisté (avšak bezpečné) ztráty informace. Monotónní kontext *(L,F)* nazveme *rychlý* (fast), jestliže platí:

(F1) $\forall f \in F:\ f.f \sqsupseteq\ f \wedge id$.

Pak aproximujeme $g.(f \wedge id\ \sqsubseteq\ \bigwedge M(P)$, neboť $(f \wedge id) \sqsubseteq f^i,\ i \geq 0$ (srovnejte s *(6.6)*).

CP problém (jak lze ověřit) však není ani rychlý. Pro práci s takovými kontexty lze využít triku z [Gra77]: *rychlým uzávěrem* f^+ libovolné funkce $f \in F$ nazveme $f^+ = \bigwedge\{(f \wedge id)^i; i \geq 0\}$. Předpoklad, že platí $f \in F\ \Rightarrow\ f^+ \in F$ a že f^+ lze najít v konečně mnoha krocích dovoluje pracovat s kontexty, které nejsou ani rychlé. V našem příkladě dostaneme $g.f^+\ \sqsubseteq\ \bigwedge M(P)$ jako bezpečnou aproximaci řešení.

Pojmy idempotentního (kdy $=$ nahradíme $\sqsupseteq$ - viz též *(6.5)*) a rychlého kontextu lze zobecnit na

(I2) *k-ohraničený kontext*: $\forall f \in F:\ f^k\ \sqsupseteq\ \bigwedge\{f^i;\ 0 \leq i \leq k-1\}$ a na
(F2) *k-semi-ohraničený kontext*:
$\forall x, y \in L.\ \forall f \in F:\ f^k(x)\ \sqsupseteq\ (\bigwedge\{f^i(x);\ 0 \leq i \leq k-1\} \wedge f^k(y))$.

Je vidět, že k-ohraničenost implikuje k-semi-ohraničenost, která implikuje *(k+1)*-ohraničenost. Lze ukázat (viz [Tar81]), že pro nalezení řešení k-(semi-)ohraničené instance problému ATD stačí uvažovat jen cesty, které jsou k-(semi-)jednoduché.

Například *LIVE* problém je 1-semi-ohraničený; problémy tzv. binární relační

ATD (detekce společných podvýrazů, metoda pro CP problém poskytující *MOP* řešení - viz [Kil73]) jsou 2-ohraničené, ne však 1-semi-ohraničené. Složitost n-ární relační ATD je studována v [Jon81]. Typová analýza nemusí být k-ohraničená pro žádné k.

K zajímavému kontextu lze dospět zobecněním pojmu aproximace z rychlých kontextů. (L, F) nazveme *spěšný* (rapid) *kontext*, právě když existuje binární operace @ na F a přirozené $t_@$ takové, že platí:

(R1) $\forall f, g \in F : \; g.f^+ \sqsubseteq g@f \sqsubseteq \bigwedge\{g.f^i; i \geq 0\}$ a
$g@f$ lze získat z f a g pomocí nejvýše $t_@$ aplikací operací . a $\wedge$.

V [Ros80] je uvedena metoda ADT vyšší úrovně využívající spěšných kontextů, která v některých případech, kdy $g.f^+ \sqsubset g@f$, získává přesnější řešení než rychlé kontexty. Například *LIVE* problém je 1-semi-ohraničený; problémy tzv. binární relační ATD (detekce společných podvýrazů, metoda pro CP problém poskytující *MOP* řešení - viz [Kil73]) jsou 2-ohraničené, ne však 1-semi-ohraničené. Složitost n-ární relační ATD je studována v [Jon81]. Typová analýza nemusí být k-ohraničená pro žádné k.

K zajímavému kontextu lze dospět zobecněním pojmu aproximace z rychlých kontextů. (L, F) nazveme *spěšný* (rapid) *kontext*, právě když existuje binární operace @ na F a přirozené $t_@$ takové, že platí:

(R1) $\forall f, g \in F : \; g.f^+ \sqsubseteq g@f \sqsubseteq \bigwedge\{g.f^i; i \geq 0\}$ a
$g@f$ lze získat z f a g pomocí nejvýše $t_@$ aplikací operací . a $\wedge$.

V [Ros80] je uvedena metoda ADT vyšší úrovně využívající spěšných kontextů, která v některých případech, kdy $g.f^+ \sqsubset g@f$, získává přesnější řešení než rychlé kontexty.

7 Aplikace výsledků analýzy toku dat

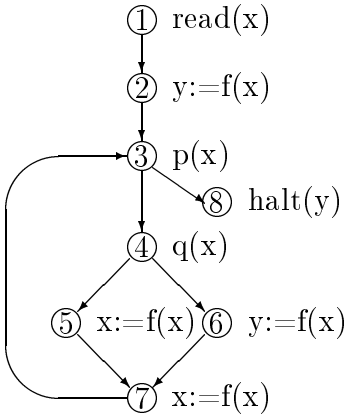
Dosud jsme se převážně věnovali metodám řešení problémů ATD. Slíbenou motivací pro vynaložené úsilí byla praktická aplikace výsledků získaných jejich řešením. Skutečně, již formulace některých problémů použití v praxi přímo nabízela, nebo praktické využití řešených problémů bylo v textu stručně uváděno v zájmu udržení pozornosti pragmaticky zaměřeného čtenáře.

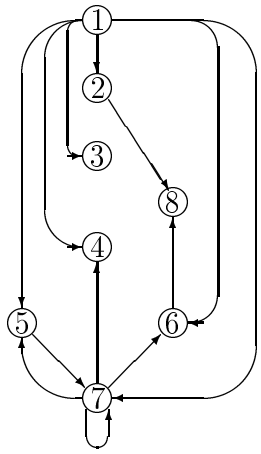
Zopakujme hlavní oblasti aplikace ATD. Již klasickou aplikací, která byla zároveň motivací pro teoretický výzkum ATD a dlouhodobou doménou její aplikace je optimalizace kódu generovaného při překladu. Dalšími oblastmi, do kterých aplikace ATD později pronikly, je proces vývoje programu, zejména pak fáze testování a ladění. Expanze aplikací ATD do procesu vývoje programu je jen logickým důsledkem faktu, že každý programátor je sám o sobě analyzátorem toku dat. Připomeňme proces lokalizace a zjištění příčiny chyby v programu nebo provádění modifikací v programu. Programátor analýzu toku dat provádí na základě intuice a zkušeností, modelem programu je mu obvykle balík výpisu programu ve zdrojovém textu. Provedení metod ATD prezentovaných v této práci lze svěřit počítači a připravit tak rychle a přesně podklady pro vlastní tvůrčí úsilí programátora.

7.1 Informační graf programu

Ke grafu toku řízení G, jehož uzly reprezentují příkazy programu, je *informační graf* (IG) definován takto:

1. množina uzlů IG je totožná s množinou uzlů GTŘ;
2. z uzlu u vede do uzlu v hrana označená proměnnou x právě tehdy, když proměnná x je v uzlu u definována, v uzlu v referencována a v grafu G existuje cesta z uzlu u do uzlu v taková, že v žádném vnitřním uzlu této cesty není proměnná x definována.





Obr. 13 Graf toku řízení Informační graf (IG)

Globální informace o toku dat potřebná k sestrojení informačního grafu je informace o dosažitelnosti jednotlivých definic proměnných na vstupech uzlů grafu toku řízení. Pro každý uzel u nás zajímá množina $REACHES(u)$ příkazů, v nichž je vypočtena hodnota, která je stále ještě dostupná i na vstupu do uzlu u. Požadované množiny obdržíme řešením systému rovnic *(RD1)* až *(RD3)*, resp. *(REACH)* a *(RD3)* (viz čl. 6.1).

Na základě množin $REACHES(u)$ snadno sestrojíme pro každý příkaz p a každou proměnnou x v něm referencovanou množinu $DEFS(x,p)$ příkazů, které mohou počítat hodnotu proměnné x použitou v příkazu p. V informačním grafu tedy vede hrana označená proměnnou x z příkazu u do příkazu v právě tehdy, když $u \in DEFS(x,v)$.

Informačním grafem je pak pro každý příkaz p a proměnnou x, jíž je příkazem p přiřazena hodnota, dána množina $USES(x,p)$ příkazů, které mohou referencovat tuto hodnotu proměnné x. Mezi množinami $DEFS$ a $USES$ je tento vztah:

$$p \in DEFS(x,r) \Leftrightarrow r \in USES(x,p).$$

Poznamenejme, že informaci obsaženou v množinách $REACHES$ můžeme vztáhnout místo k uzlům grafu toku řízení k jeho hranám. Pro libovolnou hranu e s počátečním uzlem u platí

$$REACHES(e) = REACHES(u) \cap NOTKILL(u) \cup GEN(u).$$

Takto pojatá informace může být důležitá například při modifikaci programu vsouváním nového příkazu. Je ihned patrno, které toky dat tím případně přerušujeme.

7.2 Šíření konstant

Ukažme si použití globálních informací o toku dat obsažených v informačním grafu pro řešení *problému šíření konstant*, který spočívá v nalezení příkazů produkujících konstantní hodnotu (všechny referencované proměnné mají konstantní - a předem známou - hodnotu). Problém šíření konstant byl již použit jako příklad v předchozích kapitolách. Zde ukážeme jeho řešení na základě informací obsažených v množinách *DEFS* a *USES*.

Algoritmus CP - šíření konstant

Vstup: $PRIK$ - množina příkazů programu, množiny *DEFS* a *USES* všech příkazů z $PRIK$, boolevské hodnoty $CONST(x,p)$ pro všechny příkazy p a proměnné $x \in Rf(p) \cup Df(p)$: $CONST(x,p) = true$ právě tehdy, když $x \in Df(p)$ a $Rf(p) = \emptyset$ (výraz na pravé straně příkazu p je složen výhradně z literálů), hodnoty výrazů z pravých stran příkazů p uložené va $VAL(x,p)$ vždy, když $CONST(x,p) = true$.

Výstup: modifikované hodnoty $CONST(x,p) : CONST(x,p) = true$ právě tehdy, když proměnná x při provádění příkazu p nabývá pouze známé konstatní hodnoty (ta je pak uložena ve $VAL(x,p)$).

Pomocné procedury: COMPUTE pro výpočet hodnoty výrazu u něhož známe hodnoty referencovaných proměnných.

```
begin POM:={u∈PRIK| CONST(Df(u),u)};
        {příkazy produkující zřejmě konstantní hodnotu}
   while POM ≠ ∅ do nechť u je libovolný příkaz z POM; POM:=POM - {u};
        for each r ∈ USES(Df(u),u) do konst:=true;
           for each q ∈ DEFS(Df(u),r) - {u} while konst do
              if not(CONST(Df(u),q) and VAL(Df(u),q) = VAL(Df(u),u) )
                then konst:=false fi
           od;
           if konst then CONST(Df(u),r):=true; VAL(Df(u),r) := VAL(Df(u),u);
                    if (∀ y ∈ Rf(r) : CONST(y,r) ) then
                      CONST(Df(r),r) := true; VAL(Df(r),r) := COMPUTE(r);
                      POM := POM ∪ {r};
                    fi
           fi
        od
   od
end.
```

7.3 Anomálie toku dat

Známou aplikací ATD je problém detekcí anomálií toku dat ([Kře79], [BeŠ84]). Rozeznáváme tři druhy anomálií toku dat. Nejčastější je *anomálie ur* - referencování proměnné, která nemá definovanou hodnotu. Další anomálie souvisejí s nevyužitím hodnoty přiřazené proměnné - aniž by proměnná byla referencována, je jí buď přiřazena nová hodnota - *anomálie dd*, anebo se její hodnota stává nedefinovanou - *anomálie du*. Anomálie toku dat je příznakem možné chyby v programu. Z informací o toku dat, které jsou obsaženy v informačním grafu, můžeme odvodit následující výskyty anomálií.

Na všech cestách do uzlu u, v němž je referencována proměnná x, se vyskytuje anomálie ur právě tehdy, když $x \in Rf(u) \wedge DEFS(x,u) = \emptyset$.

Na všech cestách z uzlu u, v němž je definována proměnná x, se vyskytuje anomálie du nebo dd právě tehdy, když $x \in Df(u) \wedge USES(x,u) = \emptyset$.

Další informace o výskytu anomálií můžeme získat jednoduchou modifikací problému dosažitelnosti ($REACHES$). Řešením systému rovnic pro $REACHES$ (viz čl.6.1.) se změněnými lokálními informacemi GEN a $NOTKILL$ obdržíme pro každý uzel u množiny $REACHES1(u)$ příkazů, v nichž se hodnota nějaké proměnné stává nedefinovanou a existuje z nich cesta do uzlu u, podél níž hodnota této proměnné není definována, resp. $REACHES2(u)$ příkazů, v nichž je definována hodnota nějaké proměnné a existuje z nich cesta do uzlu u, na níž nad touto proměnnou není provedena žádná akce (referencování, definování, oddefinování).

V případě $REACHES1$ ponecháme množiny $NOTKILL(v)$ beze změny, množiny $GEN(v)$ nahradíme množinami $GEN1(v) = \{v\}$, jestliže $Undf(v) \neq \emptyset$; jinak $GEN1(v) = \emptyset$.

V případě $REACHES2$ naopak ponecháme beze změny množiny $GEN(v)$ a množiny $NOTKILL(v)$ nahradíme množinami $NOTKILL2(u)$ příkazů, v nichž je definována hodnota těch proměnných, které nejsou redefinovány ani referencovány v uzlu v.

Na alespoň jedné cestě do uzlu u, v němž je definována resp. oddefinována proměnná x, se vyskytuje anomálie dd, resp. du právě tehdy, když

$$x \in Df(u) \wedge \{p \mid p \in REACHES2(u) \wedge x \in Df(p)\} \neq \emptyset \text{ , resp.}$$

$$x \in Undf(u) \wedge \{p \mid p \in REACHES2(u) \wedge x \in Df(p)\} \neq \emptyset.$$

Na alespoň jedné cestě do uzlu u, v němž je referencována proměnná x, se vyskytuje anomálie ur právě tehdy, když

$$x \in Rf(u) \wedge \{p \mid p \in REACHES1(u) \wedge x \in Undf(u)\} \neq \emptyset.$$

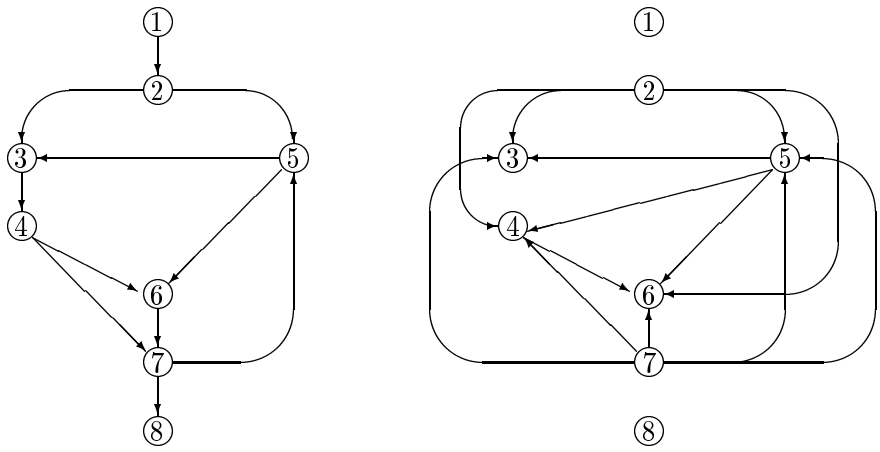
7.4 Graf vlivu větvení

Analogicky tomu, jak jsme v informačním grafu zaznamenali potencionální vliv příkazů přiřazujících vypočtenou hodnotu na příkazy, které tuto hodnotu mohou referencovat, bychom chtěli zachytit vliv větvících příkazů na ty příkazy, jejichž provedení či neprovedení je jistým způsobem ovlivněno výběrem větve ve větvícím příkazu. Nejde tedy o problém toku dat, ale o problém toku řízení. Oba problémy však spolu úzce souvisí, informace o toku dat v kontextu informací o toku řízení umožní další aplikace.

Označme *DK(u)* příkaz, který je *nejbližším dominátorem* vzhledem ke konci příkazu u.

Graf vlivu větvení (GVV) programu s grafem toku řízení G je definován takto:

1. množina uzlů GVV je totožná s množinou uzlů GTŘ
2. z uzlu u vede do uzlu v ($v \neq u$) hrana právě tehdy, když v grafu toku řízení existuje cesta z uzlu u do uzlu v taková, že $DK(u)$ neleží na této cestě.



Obr.14 Graf toku řízení Graf vlivu větvení

Informaci potřebnou pro sestrojení grafu vlivu větvení získáme analogickým způsobem jako pro informační graf. Pro každý příkaz u nás zajímá množina *INFL(u)* příkazů p z nichž vede cesta do příkazu u taková, že neobsahuje příkaz $DK(p)$. Řešíme tedy systém

$$INFL(s) = \emptyset$$

$$INFL(u) = \bigcup_{v \in PRED(u)} \{(INFL(v) \cap NOTKILL(v)) \cup GEN(v)\}$$

kde

1. s je počáteční uzel GTŘ
2. $NOTKILL(v)$ je množina příkazů p, pro něž $DK(p) \neq v$
3. $GEN(v) = \{v\}$ pro větvící příkazy v, jinak $GEN(v) = \emptyset$

Z množin $INFL(u)$ pak snadno sestrojíme pro každý příkaz p množinu

$$BRANCH(p) = (INFL(p) \cap NOTKILL(p)) - \{p\}$$

V grafu vlivu větvení vede hrana z uzlu u do v právě tehdy, když $u \in BRANCH(v)$.

7.5 Výřezy programů

Výřez programu je (na intuitivní úrovni) program, který bude obsahovat z příkazů původního programu jen ty, které mají vliv na jisté zvolené vlastnosti programu v době běhu. Při konstrukci výřezu S programu P vycházíme z nějaké množiny příkazů původního programu, jejichž potřebnost bude apriori explicitně či implicitně indukována sledovanými hledisky. Úspěšné provedení těchto příkazů může záviset na provedení jiných příkazů, které počítají hodnoty proměnných v nich referencovaných či vybírají větev, po které se dospěje, resp. nedospěje k jejich provedení. Potřebnost příkazů vzhledem ke sledovaným aspektům se tedy bude šířit grafem toku řízení opět prostřednictvím informací obsažených v informačním grafu a v grafu vlivu větvení. Nepotřebné (z daného hlediska) příkazy pak z programu P vypustíme tak, aby vzniklý výřez S byl programem, který má zvolené vlastnosti (téměř) totožné s původním programem.

Uveďme algoritmus, který k množině *AUST apriori potřebných příkazů* nalezne množinu *UST všech potřebných příkazů*. Všimněme si, že za vždy potřebné budeme považovat všechny příkazy zastavení.

Algoritmus SUST–hledání potřebných příkazů
Vstup: 1) *AUST*–množina apriori potřebných příkazů
2) množiny *DEFS*(x,p), *BRANCH*(p)
Výstup: množina *UST* všech potřebných příkazů

begin
UST := ∅; POM := *AUST*;
while POM ≠ ∅ **do**
nechť p je libovolný příkaz z POM; POM := POM – {p};
UST := *UST* ∪ {p};
for each q ∈ *DEFS*(x,p) ∪ *BRANCH*(p) **do**
if q ∉ *UST* **then** POM := POM ∪ {q} **fi**;
od;
od;
UST := *UST* ∪ {p : p je koncový příkaz (zastavení)}
end.

Vyjasněme otázku, jak vypouštet z grafu toku řízení $G = (N, E, s)$ programu P příkazy, které nepatří do množiny potřebných příkazů *UST*. Je dokázáno, že (je-

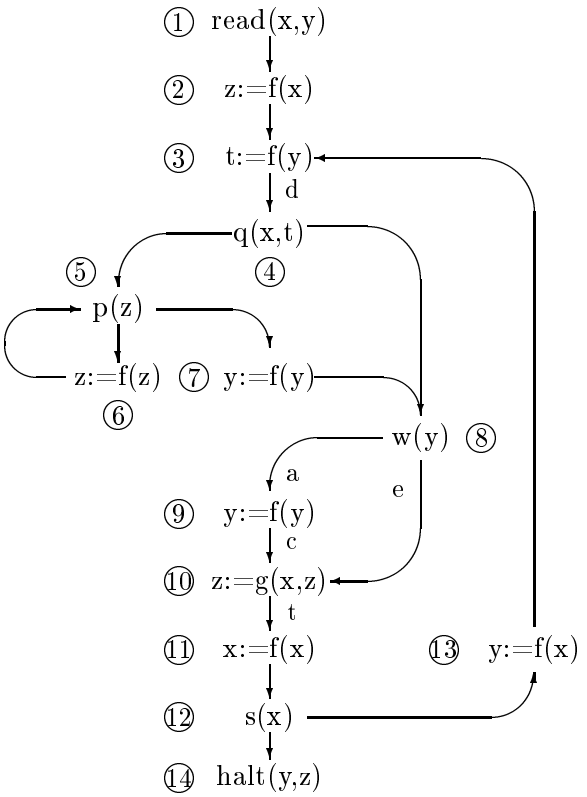
li $N - UST \neq \emptyset$) existuje a dá se nalézt rozklad množiny $N - UST$ s těmito vlastnostmi:

1. neexistuje hrana e, která v G vede z uzlu patřícího do jedné třídy rozkladu do uzlu patřícího do jiné třídy;
2. pro každou třídu T rozkladu existuje právě jeden uzel $u \in UST$ takový, že do něj vedou hrany z uzlů množiny T.

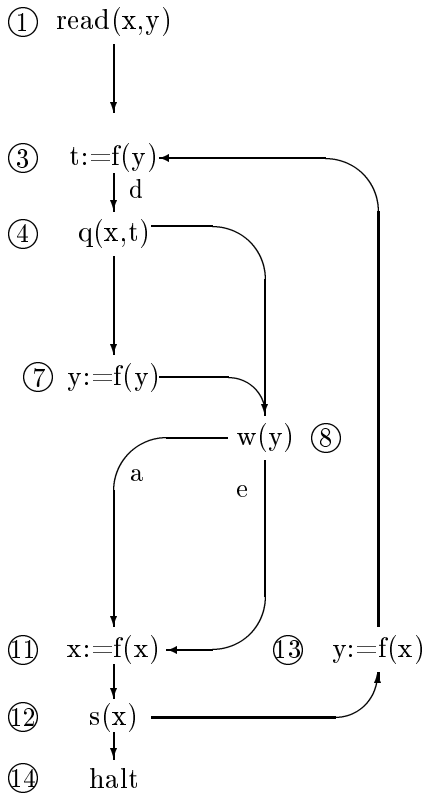
Vypuštění nepotřebných příkazů z grafu toku řízení G můžeme tedy realizovat tak, že pro každou třídu T vypustíme podgraf indukovaný množinou uzlů T, hrany, které vedou z uzlů množiny T do nějakého (právě jednoho) uzlu u z UST, a hrany, které vedou z uzlů množiny UST do uzlů množiny T nahradíme hranami vedoucími do uzlu u.

Tím je *výřez programu* zkonstruován. Konstrukce je korektní, výřez je programem.

Ilustrujme nyní zavedené pojmy a postupy. Uvažujme program P na obr. 15 a množinu apriori potřebných příkazů $AUST = \{1, 7, 8, 11, 13\}$. Algoritmem SUST nalezená množina všech potřebných příkazů je $UST = \{1, 3, 4, 7, 8, 11, 12, 13, 14\}$. Hledaný rozklad množiny $N - UST$ je {2}, {5,6}, {9,10}. Odpovídající výřez je rovněž na obr. 16.



Obr.15 Program P



Obr.16 Výřez S programu P

První důležitou vlastností libovolného výřezu S programu P (odvoditelnou z konstrukce výřezu) je, že pro libovolná vstupní data *vždy, když program P zastaví*, pak *výřez S rovněž zastaví* (pro tatáž vstupní data). Obrácená implikace neplatí: zacyklí-li se např. program P na obr. 15 v cyklu tvořeném příkazy 5 a 6, jeho výřez S může zastavit. Tuto vlastnost výřezů musíme mít vždy na paměti při aplikacích výřezů.

Při formulaci dalších vlastností výřezů budeme pracovat s pojmem *výpočetní posloupnosti* programu P pro daná vstupní data. Pro tento účel budeme *výpočetní posloupností* rozumět posloupnost stavů, kde stav je dán hranou (místem v programu) GTŘ (vedoucí z příkazu, který byl právě proveden, do příkazu, který má být právě proveden), uzlem do kterého tato hrana vede a posloupností momentálních hodnot proměnných. *Stav výpočtu* je tedy dán čítačem instrukcí (zdvojeným) a stavem paměti.

Vraťme se k vlastnostem výřezů. Uvažujme výpočetní posloupnost L programu P pro nějaká vstupní data a výpočetní posloupnost T jeho výřezu S pro táž data. Vypusťme z posloupnosti L programu P všechny stavy, které v čítači instrukcí obsahují uzly (nepotřebné příkazy), které nejsou ve výřezu S. Pokud se program P nezacyklil v nepotřebných příkazech, pak takto upravená posloupnost —označme ji LL— koresponduje s posloupností T výřezu S následujícím způsobem. V i-tém prvku posloupnosti LL je v čítači instrukcí *tentýž uzel* jako v i-tém prvku posloupnosti T, tudíž je v obou případech prováděn týž příkaz a navíc *hodnoty proměnných referencovaných v tomto příkazu jsou* opět v obou stavech *shodné*.

Tato vlastnost, spolu s prvně uvedenou, již umožňují některé aplikace výřezů. Klasická optimalizační metoda, eliminace mrtvého kódu, spočívá v konstrukci výřezu programu, přičemž za množinu apriori potřebných příkazů zvolíme množinu všech výstupních příkazů programu. Výřez, oproštěný od neproduktivního kódu je pak v důsledku vlastností výřezu funkčně ekvivalentní výchozímu programu, až na anomální případy, kdy se program zacyklí právě v neproduktivním kódu.

Aplikace výřezů (odvozené z dosud uvedených vlastností) v procesu testování a ladění programů jsou vhodné zejména v situacích, kdy programátora nezajímají ani tak výstupní hodnoty produkované programem, ale spíše tok řízení a příkazy, které tok řízení přímo či zprostředkovaně ovlivňují. Testujeme-li pouhé zastavení programu pro různá vstupní data, poskytne nám stejné služby jako program, ale v mnoha případech laciněji tzv. stop-výřez programu. *Stop-výřez programu* je výřez, který obdržíme, když za množinu apriori potřebných příkazů zvolíme množinu všech východů z cyklů. *Východem z cyklu* nazveme uzel v takový, že v GTŘ existuje cesta z uzlu v do uzlu v taková, že neobsahuje uzel $DK(v)$ – nejbližší dominátor vzhledem ke konci. Pro daná vstupní data *stop-výřez zastaví právě tehdy, když zastaví* původní *program* pro tatáž data. Stop-výřez programu z obr. 15 je
[1] read(x); **repeat** [11] x:=f(x) **until** [12] s(x); [14] halt.

Zajímá-li nás v daném okamžiku, které příkazy a jakým způsobem souvisí s tokem řízení při výpočtu, pak je k dispozici tzv. *řídící výřez programu*, což je výřez, který obdržíme, vezmeme-li za množinu apriori potřebných příkazů množinu všech

testovacích příkazů. Řídící výřez stejně jako stop-výřez zastaví pro daná data právě tehdy, když pro tato data zastaví program a je vhodný pro testování i studium řídící části programu. Může být např. nápomocen i při hledání vhodných množin vstupních dat. Podaří-li se nám totiž nalézt takovou množinu vstupů pro řídící výřez, že každá hrana jeho GTŘ se vyskytuje ve výpočetní posloupnosti pro nějaká data z této množiny, pak máme zaručeno, že i každá hrana (a tedy i příkaz) původního programu se vyskytuje ve výpočetní posloupnosti pro nějaká data z nalezené množiny.

V mnoha situacích, s nimiž se setkáváme při testování a ladění, nás v první řadě zajímají vedle toku řízení i proměnné, speciálně pak čím jsou ovlivňovány hodnoty, kterých mohou nabývat, když výpočet dospěje do určitých míst programu.

Předpokládejme, že náš zájem tohoto druhu je specifikován *kritériem* C, což je množina tzv. *elementárních kritérií* $c=(e,V)$, kde e je hrana GTŘ (místo v programu), V nějaká množina proměnných. I nyní můžeme použít výřezů programu, ale na rozdíl od předešlých aplikací v tomto případě z formulace požadavku nevyplývá tak přímočaře charakteristika množiny apriori potřebných příkazů. Pro její nalezení použijeme informace poskytnuté ATD. Protože elementární kriterium $c = (e, V)$ vyjadřuje zájem o hodnoty proměnných množiny V v místě programu e, budou nutně součástí množiny potřebných příkazů $AUST$ všechny příkazy p, které přiřazují hodnotu nějaké proměnné z V a jsou dosažitelné na hraně e ($p \in REACHES(e)$). Ze zřejmých důvodů bude třeba zařadit do $AUST$ ty testovací příkazy p, z nichž vede v GVV hrana do počátečního uzlu hrany e (označme ho w) a příkaz w v případě, že je testovacím příkazem. Formálně zapsáno, definujme pro elementární kriterium c množinu

$$AUST(c) = \{p \mid p \in REACHES(e) \wedge Df(p) \cap V \neq \emptyset\} \cup BRANCH(w) \cup$$
$$\cup \textbf{ if } w \text{ je testovací příkaz } \textbf{then } \{w\} \textbf{ else } \emptyset.$$

Množina apriori potřebných příkazů $AUST$ pro zadané kriterium C je $AUST(C) = \cup\{AUST(c) \mid c \in C\}$. Pro ilustraci se vraťme k programu na obr. 15 a zvolme kriterium $C = \{(e, \{x, y\}), (c, \{x\})\}$. Pak

$AUST((e, \{x, y\})) = \{1, 7, 13\} \cup \{12\} \cup \{8\}$,

$AUST((c, \{x\})) = \{1, 11\} \cup \{8, 12\} \cup \emptyset$,

$AUST(C) = \{1, 7, 8, 11, 12, 13\}$. Příslušný výřez S je na obr. 16.

Po nalezení množiny $AUST$ ke kriteriu C, což umožňuje zkonstruovat příslušný výřez, zbývá řešit ještě další problém. V uvedeném příkladě si všimněme, že ve výřezu není např. hrana c, která figuruje v kriteriu C. Vyvstává otázka, která místa (hrany) ve výřezu S odpovídají hraně c programu P.

Každé hraně e programu P proto přiřadíme podmnožinu $CORE(e)$ množiny hran výřezu S takto: Hrana g výřezu S je prvkem $CORE(e)$ právě tehdy, když v GTŘ programu P existuje cesta z počátečního uzlu hrany g do koncového uzlu hrany g taková, že hrana e leží na této cestě a žádné vnitřní uzly této cesty nepatří do výřezu S. Z definice vyplývá, že patří-li hrana e rovněž do výřezu S, pak $CORE(e) = \{e\}$. Jako příklad vezměme opět program P a jeho výřez S na obr. 15 a 16. Zde platí $CORE(d) = \{d\}$, $CORE(c) = \{a\}$, $CORE(e) = \{e\}$, $CORE(t) = \{a, e\}$, atd.

Uvažujme výpočetní posloupnost L programu P pro nějaká vstupní data a vý-

početní posloupnost T (pro tatáž data) jeho výřezu S, zkonstruovaného na základě kriteria C. Pro libovolné elementární kriterium $c = (e, V)$ z C označme jako $L(c)$ posloupnost, kterou obdržíme z posloupnosti L vypuštěním všech stavů, které v čítači instrukcí obsahují hranu různou od hrany e, a jako $T(c)$ posloupnost, kterou obdržíme z T, vypuštěním všech stavů, které v čítači instrukcí obsahují hranu nepatřící do $CORE(e)$. Pak pro i-té prvky posloupností $L(c)$ a $T(c)$ platí, že v nich *hodnoty proměnných množiny V jsou totožné.*

Zajímají-li nás hodnoty proměnných množiny V vždy před provedením příkazu p programu P nezávisle na tom, po které hraně výpočet k příkazu p dospěl, zařadíme do kritéria C elementární kritéria pro všechny vstupní hrany uzlu p, vždy ve dvojici s množinou V. Vlastnost příslušného výřezu musíme formulovat poněkud komplikovaněji s ohledem na to, že příkaz p nemusí být obsažen ve výřezu (viz např. uzel 10 na obr. 15). Označme $L(p)$ posloupnost obdrženou z L vypuštěním stavů, které neobsahují v čítači instrukcí uzel p, a jako $T(p)$ posloupnost obdrženou z T vypuštěním stavů, které v čítači instrukcí obsahují hranu nepatřící do $CORE(e)$ pro žádnou vstupní hranu e příkazu p. Pro i-té prvky posloupností $L(p)$ a $T(p)$ platí, že *hodnoty proměnných množiny V jsou totožné.*

Neformálně lze vlastnosti výřezu S programu P formulovat tak, že prohlížíme-li výpočetní posloupnosti L programu P a T výřezu S komparativním mikroskopem, který provádí příslušné selekce výpočetních posloupností a navíc umožňuje vidět pouze hodnoty určených proměnných, nerozlišíme je navzájem.

Všechny uvedené vlastnosti výřezů otevírají široké aplikační možnosti vždy, kdy nás z nejrůznějších důvodů zajínají ty aspekty chování programu, které lze charakterizovat prostřednictvím množiny apriori potřebných příkazů resp. nějakým výřezovým kritériem. Za těchto okolností je výhodnější z hlediska vynakládaného intelektuálního úsilí a strojového času pracovat s příslušným výřezem, který je co do sledovaných aspektů chování ekvivalentem původního programu a může být co do počtu příkazů a proměnných podstatně jednodušší než program sám.

Nezbytnou podmínkou aplikace výřezů v praxi je ovšem existence automatického generátoru výřezů. V tomto směru se jeví jako perspektivní využití (existujících) systémů pro manipulaci s programy ve formě grafů, jejichž představitelem je např. systém TPT [BeŠ84]. Nadstavbové komponenty tohoto systému realizují převod programů zapsaných v jazycích Pascal nebo Fortran do sítě, která obsahuje odpovídající graf volání, grafy toku řízení a graf syntaktický a lexikální se všemi atributy potřebnými k provádění analýzy toku dat. Systém rovněž umožňuje po manipulaci s těmito grafy zpětnou tranformaci do zdrojových jazyků, což je v případě výřezů dosti podstatné: výřezy je nutné produkovat ve formě vhodné jak pro uživatele, tak pro další zpracování na počítači, a tou je právě zdrojový text.

Literatura

[Aho76] *Aho, A. V. - Ullman, J. D*: Node Listing for Reducible Flow Graphs. JCSS,13, 1976.

[All72] *Allen,F.E.-Cocke,J.*: A Catalogue of Optimizing Transformation. In:Randall,R.(ed.): Design and Optimization of Compilers, Prentice-Hall, 1972.

[All76] *Allen,F.E.-Cocke,J.*: A Program Data Flow Procedure, CACM 19, 1976.

[Bab78] *Babich,W.A.-Jazayeri,M.*: The Method for Data Flow Analysis. Acta Informatica,10, 1978, s.265-272.

[Bar78] *Barth,J.M.*: A Practical Interprocedural Data Flow Analysis Algorithm. CACM,21, 1978.

[BeŠ84] *Benešovský,M.-Šmídek,M.*: Testování programů. In: sborník SOFSEM'84, VUSEI-AR Bratislava, 1984.

[Cho82] *Chow,A.-Rudmik,A.*: The Design of Data Flow Analyzer. SIGPLAN Not.,17, 1982, č.6, s.106-113.

[Cla81] *Clarke,L.A.-Richardson,.J.*: Symbolic Evaluation Methods for Program Analysis. In: [Muc81].

[Cou77] *Cousot,P.-Cousot,R.*: bstract Interpretation: A Unified Lattice Model. 4th ACM Symp.on Principles of Programming Languages, 1977, s.38-252.

[Cou79] *Cousot,P.-Cousot,R.*: Systematic Design of Program Analysis Frameworks. 6th ACM Symp.on Principles of Programming Languages, 1979, s.269-283.

[Cou81] *Cousot,P.*: Semantic Foundations of Program Analysis. In: [Muc81].

[Far76] *Farrow,R.-Kennedy,K.-Zucconi,L.*: Graph Grammars and Global Program Flow Analysis. Proc. 17th IEEE Symp.on Foundation of Comp.Sci., 1976.

[Flo67] *Floyd,R.W.*: Assigning Meanings to Programs. Proc.Symp. in Applied Math. of the AMS, Am.Math.Society, 967, s.19-32.

[Fos76] *Fosdick,L.D.-Osterweil,L.J.*: Data Flow Analysis in Software Reliability. Computing Surveys,8, 1976, č.3, s.305-330.

[Gra76] *Graham,S.L.-Wegman,M.*: A Fast and Usually Linear Algorithm for Global Flow Analysis. JACM,23, 1976, č.1, s.172-202.

[HaK82] *Havlát,T.-Křetínský,M.*: Statická analýza v testování. In: sborník LADĚNÍ 82, DT ČSVTS, Č.Budějovice, 1982.

[Hav84] *Havlát,T.*: Výřezy programů. In: sborník SOFSEM'84, VUSEI-AR Bratislava, 1984.

[Hav85] *Havlát,T.*: Výřezy programů jako prostředek pro ladění. In: sborník LADĚNÍ'85, DT ČSVTS, Č.Budějovice 1985.

[Hav86] *Havlát,T.*: Výřezy programových schémat. Kandidátská práce UJEP, Brno, 1986.

[Hec74] *Hecht,M.S.-Ullman,J.D.*: Characterization of Reducible Flow Graphs. JACM,21, 1974, č.3, s.367-375.

[Hec77] *Hecht,M.S.*: Flow Analysis of Computer Programs. North Holland, New York, 1977.

[Jon81] *Jones,N.D.-Muchnick,S.S.*: Complexity of Flow Analysis. In: [Muc81].

[Kam76] *Kam,J.B.-Ullman,J.D.*: Global Data Flow Analysis and Iterative Algorithms. SIAM J.on Comp.,5, 1976, s.158-180.

[Kam77] *Kam,J.B.-Ullman,J.D.*: Monotone Data Flow Analysis Frameworks. Acta Informatica,7, 1977, č.3, s.305-317.

[Ken81] *Kennedy,K.*: A Survey of Data Flow Analysis Techniques. In: [Muc81].

[Kil73] *Kildall,G.A.*: A Unified Approach to Global Program Optimization. ACM Symp. on Principles of Programming Languages, 1973, s.194-206.

[Knu68] *Knuth,D.E.*: Semantics of Context-free Languages. Mathematical Sys.Theory,2, 1968, č.2, s.127-145.

[Knu71] *nuth,D.E.*: An Empirical Study of FORTRAN Progarms. Soft.Pract. and Exp.,1, 1971, č.2, s.105-133.

[Kře79] *Křetínský,M.*: Metoda detekce anomálií toku dat. In: sborník SOFSEM'79, VUSEI-AR, Bratislava, 1979.

[Kře86] *Křetínský,M.*: Předvýpočtová analýza programů. Kandidátská práce UJEP, Brno, 1986.

[Muc81] *Muchnick,S.S.-Jones,N.D. (ed.)*: Program Flow Analysis: Theory and Applications, Prentice-Hall, 1981.

[Ros77] *Rosen,B.K.*: High-level Data Flow Analysis. CACM,20, 1977, č.10, s.712-724.

[Ros80] *Rosen,B.K.*: Monoids for Rapid Data Flow Analysis. SIAM J.on. Comp.,9, 1980, č.1, s.159-196.

[Ros81] *Rosen,B.K.*: Degrees of Availability as an Introduction to the General Theory of Data Flow Analysis. In: [Muc81].

[Ros82] *Rosen,B.K.*: A Lubricant for Data Flow Analysis. SIAM J.on Comp.,17, 1982, č.3, s.493-511.

[Tar74] *Tarjan,R.E.*: Finding Dominators in Directed Graphs. SIAM J.on Comp.,3, 1974.

[Tar75] *Tarjan,R.E.*: Efficiency of a Good but not Linear Set Union Algorithm. JACM,22, 1975, č.2, s.62-89.

[Tar81] *Tarjan,R.E.*: A Unified Approach to Path Problems. JACM,28, 1981, č.3, s.577-593.

[Tar81] *Tarjan,R.E.*: Fast Algorithm for Solving Path Problems. JACM,28, 1981, č.3, s.594-614.

[Ull73] *Ullman,J.D.*: Fast Algorithm for the Elimination of Common Subexpressions. Acta Informatica,2, 1973, č.3, s.191-213.

[Wei81] *Weiser,M.*: Program Slicing. Proc. of the 5th International Conference on Soft.Eng., 1981.

[Wei82] *Weiser,M.*: Programmers Use Slices When Debugging. CACM,25, 1982, č.7.

[Zad84] *Zadeck,F.K.*: Incremental Data Flow Analysis in Structured Editor. SIGPLAN Not.,19, 1984, č.6, s.132-143.