Vybrané postsyntaktické statické analýzy
TJD – Teorie programovacích jazyků

Zdeněk Letko
iletko@fit.vutbr.cz

28. května 2009

Obsah

1 Úvod

Tato práce se zabývá základy vybraných metod statické analýzy kódu. Pojem statická znamená, že zdrojový kód programu je analyzován bez jeho spuštění. Tyto analýzy mají široké uplatnění jak v překladačích, tak ve verifikaci programů. Analýza kódu vede k hlubšímu pochopení chování programu, který je reprezentován tímto kódem. V překladači lze takto získané informace využít zejména k optimalizaci výsledného programu a to jak z hlediska práce s pamětí, tak z hlediska pořadí vykonávání jednotlivých instrukcí. Ve verifikaci tyto informace slouží k rozhodnutí, zda program splňuje specifické vlastnosti, které mají vztah ke správné funkcionalitě programu.

Analýzy uváděné v této práci patří mezi tzv. *postsyntaktické* analýzy. To znamená, že nad zdrojovým kódem již byla provedena lexikální a syntaktická analýza[2, 1]. Vstupem *lexikální* analýzy je zdrojový kód programu, který lze charakterizovat jako sekvenci znaků. Lexikální analyzátor tuto sekvenci znaků převede na sekvenci *tokenů* (elementárních nositelů informace v daném jazyce) a některé irelevantní části kódu vyloučí z dalšího zpracování (například komentáře). Posloupnost tokenů je následně zpracována *syntaktickým* analyzátorem, který se snaží určit strukturu posloupnosti tokenů vzhledem ke gramatice daného programovacího jazyka. Výstupem syntaktické analýzy často bývá stromová struktura postihující jednotlivé konstrukce jazyka. V této reprezentaci lze již jednoduše určit jednotlivé příkazy jazyka, včetně jejich parametrů. U postsyntaktických analýz tak již můžeme mluvit o *příkazech* a jejich významu pro běh programu.

V následujících sekcích tohoto dokumentu je uveden stručný úvod do pěti různých postsyntaktických analýz. Nejprve je uvedena analýza toku řízení, jejímž výstupem je informace o pořadí zpracovávání jednotlivých příkazů včetně informací o větvení programu a případných skocích. Ve třetí sekci je představena analýza toku dat, která využije informace získané v analýze toku řízení ke zjištění způsobu, jakým daný program pracuje s daty. Výpočetní mechanismus, konkrétněji tokové rovnice, které k analýze využívá, lze úspěšně použít i k získávání dalších informací o programu. Analýza závislostí, které je věnována čtvrtá sekce, identifikuje logické (přesněji paměťové a časové) závislosti mezi jednotlivými příkazy a paměťovými místy, které program využívá. Pátá sekce se věnuje alias analýze, která se snaží určit, se kterým paměťovým místem (nebo množinou míst) může daný příkaz pracovat. Všechny předchozí analýzy jsou představeny v jednoduché podobě, kdy analyzují kód náležící pouze jedné proceduře (či neobsahující žádné procedury). V šesté sekci jsou představeny principy, kterými lze analýzy uvedené v předchozích sekcích rozšířit tak, aby podporovaly rozčlenění programu do více procedur

(či metod), jak je tomu u moderních programovacích jazyků.

2 Analýza toku řízení

Tato sekce představuje *analýzu toku řízení*[2, 1, 5, 4], jednu z nejzákladnějších postsyntaktických analýz, která slouží k extrakci pořadí vykonávání jednotlivých příkazů programu. Pro správné pochopení vykonávání programu je nutné identifikovat cesty toku řízení. Výsledek této analýzy, tok řízení, lze vizuálně reprezentovat například vývojovým diagramem, který je pro člověka dobře pochopitelný. V této sekci budou prezentovány dva algoritmy, které se často používají pro získání informací o toku řízení v programu. Prvním je algoritmus, který využívá relace dominance a iterativního výpočtu. Druhý algoritmus rekurzivně rozděluje proceduru na menší části podle struktury indukované použitými příkazy. Tento přístup se také někdy označuje termínem *strukturální analýza*.

První dva kroky obou algoritmů jsou shodné. Oba algoritmy nejdříve identifikují tzv. *základní bloky* programu a posléze jejich propojením získají grafovou reprezentaci toku řízení. Základním blokem programu rozumíme nejdelší sekvenci příkazů neobsahující větvení, která má právě jeden vstupní bod, jímž je první instrukce v sekvenci, a právě jeden výstupní bod, jímž je poslední instrukce v sekvenci. Jinými slovy se jedná o nejdelší možné části kódu, kde se nevyskytuje větvení. Algoritmus nalezení základních bloků pracuje ve dvou krocích:

(1) V prvním kroku algoritmus detekuje všechny uzly, které jsou vstupními body základních bloků, tj. (a) vstupní uzel procedury nebo rutiny, (b) cíl větvení nebo skoku, (c) instrukce přímo následující za instrukcí větvení nebo návratu (return).

(2) Ve druhém kroku algoritmus buduje základní bloky, které obsahují instrukce od vstupního bodu až po nejbližší následující vstupní bod (ten již patří do následujícího základního bloku).

Činnost uvedeného algoritmu ukážeme na příkladu. Mějme proceduru napsanou v jazyce C viz. obrázek 1a, která počítá fibonačiho čísla. Podobu této procedury v jednoduchém mezikódu lze vidět na obrázku 1b a příslušný vývojový diagram na obrázku 2. V prvním kroce algoritmus identifikuje vstupní uzly (zde 1, 12, 5, 6, 7, 8) a ve druhém kroce jsou vybudovány základní bloky, které lze nalézt na obrázku 3a (v tomto případě B1, B2, B3, B4, B5, B6).

Grafem toku řízení rozumíme orientovaný graf $G = (N, E)$, kde N představuje množina uzlů, respektive základních bloků, a E je množina hran toku řízení, přičemž platí, že $E \subseteq N \times N$. Vrchol, obsahující první instrukci pro-

```
unsigned int fib(m)
   unsigned int m;
{  unsigned int f0 = 0, f1 = 1, f2, i;
   if (m <= 1) {
      return m;
   }
   else {
      for (i = 2; i <= m; i++) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

(a) Zdrojový kód C.

```
1       receive m (val)
2       f0 ← 0
3       f1 ← 1
4       if m <= 1 goto L3
5       i ← 2
6   L1: if i <= m goto L2
7       return f2
8   L2: f2 ← f0 + f1
9       f0 ← f1
10      f1 ← f2
11      i ← i + 1
12      goto L1
13  L3: return m
```

(b) Mezikód.

Obrázek 1: Analyzovaná procedura (převzato z [1]).

cedury, je označen jako počáteční. Hrana mezi dvěma uzly grafu $a, b \in N$ jdoucí z bloku a do b existuje v grafu tehdy, pokud platí alespoň jedna z následujících podmínek: (1) Poslední instrukcí bloku a je instrukce (ne)podmíněného skoku na první instrukci bloku b. (2) Poslední instrukcí bloku a je podmíněný příkaz skoku, za kterým bezprostředně následuje instrukce bloku b.

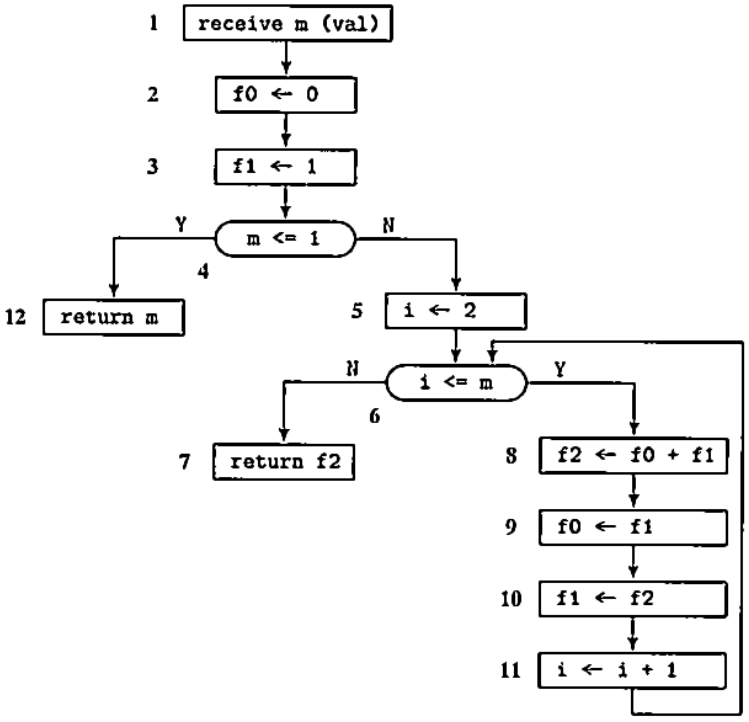
V grafu toku lze pro každý uzel b identifikovat množiny předchůdců $pred(b)$ a následníků $succ(b)$ dané následujícími vztahy:

$$pred(b) = \{n \in N | \exists (b, n) \in E\}$$

$$succ(b) = \{n \in N | \exists (n, b) \in E\}$$

Pro jednodušší analýzu se graf řízení rozšiřuje přidáním dvou význačných uzlů, které se označují *entry* a *exit*. Uzel *entry* nemá žádného předchůdce a hrany z něj vedou do vstupního uzlu procedury či rutiny reprezentované grafem toku. Analogicky se uzel *exit* stává jediným konečným uzlem grafu procedury a vedou do něj hrany ze všech uzlů b, jejich původní $succ(b) = \emptyset$. Toto rozšíření umožňuje jednodušší definici algoritmů pracujících s grafem toku, neboť lze snadno detekovat vstupní a výstupní bod analyzované procedury. Tato úprava také zjednoduší implementaci meziprocedurální analýzy, o které bude řeč v 6. sekci.

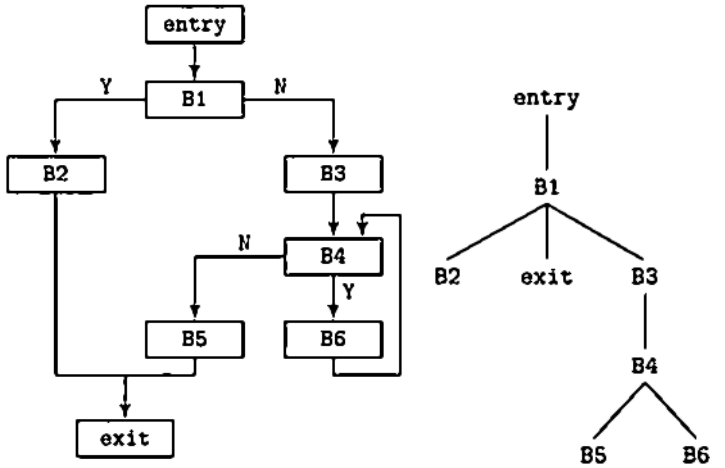
Pokud připustíme přítomnost příkazu volání procedury v našem programu, pak je výše zmíněná konstrukce grafu toku řízení optimistickou podobou, kdy není příkaz volání procedury považován za příkaz větvení. To se může změnit, pokud analyzovaný program podporuje například výjimky, které může volaná procedura generovat. Stejně tak existují jazyky (například Java), kde téměř každá instrukce (rozumějme příkaz) může generovat výjimku.



Obrázek 2: Graf toku řízení bez základních bloků (převzato z [1]).

Pesimistický algoritmus konstrukce grafu toku řízení by graf rozšířil ještě o hrany, které by vedly od místa vyvolání výjimky k místu jejich obsloužení, resp. k *exit* uzlu, pokud by kód neobsahoval obsluhu této výjimky. To by ale znamenalo, že by každý příkaz tvořil samostatný základní blok. Proto se častěji využívá optimistického přístupu.

Nyní, když jsme identifikovali základní bloky, následuje identifikace smyček, kterou první algoritmus řeší pomocí *relace dominance* a iterativního výpočtu. Základní blok a dominuje bloku b (a *dom* b), jestliže na každé cestě z uzlu *entry* do uzlu b se vyskytuje blok a. Relace *dom* je reflexivní, antisymetrická, tranzitivní a lze si ji představit jako stromovou strukturu. Řekneme, že blok a přímo dominuje bloku b, jestliže a *dom* b a $a \neq b$ a neexistuje žádné c, které by se vyskytovalo na cestě z a do b. Je zřejmé, že relaci dominance lze snadno spočítat iterativním způsobem z relace přímé dominance. Smyčku v programu pak lze identifikovat pomocí nalezení tzv. *zpětných hran* v grafu toku řízení. Zpětná hrana $(a, b) \in E$ je hrana, pro níž platí, že b *dom* a. Jinými slovy, to je hrana, jejíž konečný uzel dominuje počátečnímu uzlu. Blok b v takovém případě označujeme jako začátek smyčky. Smyčku pak tvoří všechny bloky, z nichž je po průchodu bloku b blok a dosažitelný. Je dobré si také uvědomit, že bloky smyčky vlastně tvoří silně souvislou komponentu grafu G. Graf toku řízení pro dříve uvedený příklad výpočtu fibonačiho čísel vidíme



(a) Graf toku řízení. (b) Strom dominance.

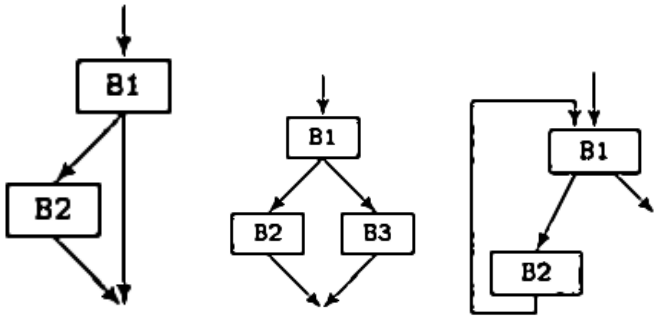
Obrázek 3: Graf toku řízení a relace dominance (převzato z [1]).

na obrázku 3a a příslušný strom indukovaný relací dominance je zobrazen na obrázku 3b.

Podobně jako relaci dominance, lze nad grafem toku řízení definovat i jiné relace (postdominance, striktní dominance, ...), které pak lze využít pro získání dalších informací o toku řízení v programu[1]. Relace postdominance bude potřeba pro analýzu závislostí řízení v sekci 4, a tak ji zde ještě definujme. Základní blok a je v relaci postdominance se základním blokem b, což zapisujeme *apdomb*, právě tehdy, když každá cesta z b do *exit* prochází uzlem a. Jinými slovy se jedná o dominanci, jen počítanou proti směru hran grafu G.

Druhý algoritmus, založený na identifikaci struktury procedury, relaci dominance nepočítá, místo ní rekurzivně dělí graf toku na tzv. *regiony*. Výsledný graf regionů tvoří strom, který se označuje termínem *řídicí strom*. Řídicí strom je graf, jehož kořenem je uzel reprezentující region obsahující celý graf toku analyzovaného kódu (procedury). Listy tohoto grafu reprezentují jednotlivé základní bloky a vnitřní uzly reprezentují regiony odpovídající řídicím strukturám, které se v programu vyskytují (`if-then-else`, `while-loop`, ...). Příklady struktur v jejich nejjednodušší podobě jsou uvedeny na obrázku 4. Hrany mezi uzly řídicího stromu vyjadřují vnoření regionu či základního bloku do nadřazeného regionu.

řídicí strom je konstruován pomocí jednoduchých transformací grafu toku systémem zdola nahoru. Děje se tak v následujících krocích: (1) Pomocí prohledávání do hloubky je proveden postorder průchod grafu toku. (2) Následně se každý uzel ztotožní se vstupním uzlem nějaké řídicí struktury v její nejjednodušší podobě (například pro strukturu `if-then` to je graf o třech uz-



(a) `if-then`. (b) `if-then-else`. (c) `while`.

Obrázek 4: Nejjednodušší podoby vybraných struktur (převzato z [1]).

lech, kde vstupní uzel obsahuje vyhodnocení podmínky a následující dva uzly reprezentují možné cíle větvení viz. obrázek 4a). (3) Ztotožněný uzel a odpovídající podgraf je nahrazen abstraktním uzlem reprezentujícím danou strukturu. (4) Body (1) až (3) se opakují, dokud není původní graf toku redukován do jednoho abstraktního uzlu, který je zároveň kořenem řídicího stromu. (5) Z abstraktních uzlů a základních bloků je vybudován řídicí strom.

Variací na právě představenou strukturní analýzu je *intervalová analýza*, která funguje podobně jako strukturní analýza, avšak podporuje jen omezený počet struktur. Proto se v tomto případě pro regiony hledané v grafu toku používá termínu *interval*. Příkladem intervalové analýzy je analýza, která rozlišuje tři druhy intervalů: Interval odpovídající smyčce, interval odpovídající tzv. nevhodnému regionu (silně souvislá komponenta, která není smyčkou) a interval odpovídající necyklickému regionu.

První algoritmus, využívající relaci dominance, postačuje pro navazující analýzy (například pro analýzu toku dat, viz následující sekce), ovšem nehodí se pro optimalizační algoritmy, které při optimalizaci mění strukturu procedur. Algoritmus užívající iterativní výpočet se s některými změnami není schopen vyrovnat a je třeba relace přepočítat. Strukturální analýza tímto neduhem netrpí, protože je při změně struktury procedury nutno změnit pouze odpovídající podstrom. Strukturální analýza je také rychlejší v případech, kdy program používá jednoduché konstrukce. Její hlavní nevýhodou je náročnější implementace, a to nejen konstrukce grafu toku, ale i navazujících analýz.

3 Analýza toku dat

Analýza toku dat[5, 3, 2, 1] využívá graf toku řízení vytvořený analýzou z předchozí sekce k získávání informací o tom, jak daná procedura pracuje s daty. Problém, respektive způsob jeho řešení, se zakóduje do tzv. *tokových*

rovnic, které postihují vliv jednotlivých instrukcí a bodů větvení programu na řešení problému. Následně jsou tyto rovnice řešeny iterativním způsobem tak, že je respektována struktura programu daná grafem toku řízení.

Existuje velké množství analýz využívajících metodu toku dat, která je představena v následujících odstavcích. Příkladem takové analýzy může být *analýza živosti proměnných*[1, 5]. Proměnnou označíme jako živou v daném bodě výpočtu, pokud existuje možnost, že hodnotu této proměnné bude v budoucnu nějaký příkaz číst. Živou proměnnou tedy nelze z paměti odstranit před provedením tohoto příkazu. Naopak, pokud není proměnná v daném bodě výpočtu živá, lze její hodnotu zahodit a paměťové místo, které ji obsahuje, využít pro jiná data. *Bodem výpočtu* se zde rozumí stav programu mezi vykonáváním dvou po sobě následujících příkazů.

Hodnoty, se kterými analýza toku dat pracuje, tvoří svazovou strukturu L se dvěma operacemi spojení $\sqcap$ a průsek $\sqcup$, které musí splňovat podmínky dané uzávěrovými vlastnostmi, komutativitou a asociativitou. Připomeňme, že tyto operace indukují relaci částečného uspořádání $\sqsubseteq$ na množině prvků L. V tomto svazu také musí existovat unikátní největší $\top$ a nejmenší $\bot$ prvek.

Tokové rovnice implementují funkci $f : L \to L$ definovanou nad svazovou strukturou. Tato funkce vlastně modeluje efekt jednotlivých programových konstrukcí vzhledem ke svazu L. Pro funkci f se vyžaduje splnění podmínky monotónnosti. Pokud je svaz L konečný a funkce f monotónní, lze velmi snadno matematickou indukcí ukázat, že iterativní aplikace funkce f vede k nalezení řešení v konečném počtu kroků. Řešením je pak největší pevný bod v případě, kdy výpočet začíná od $\top$ a funkce f je nerostoucí, respektive nejmenší pevný bod v případě, kdy výpočet začíná od $\bot$ a funkce f je neklesající. Proto se také tato metoda výpočtu označuje jako *metoda pevných bodů*.

Analýzy využívající tokové funkce lze rozdělit do několika skupin podle směru výpočtu a podle charakteru zpracovávání větvení. Směr výpočtu tokových rovnic může být buď *dopředný*, kdy se začíná od *entry* uzlu grafu řízení a výpočet následuje směr hran grafu toku řízení, nebo *zpětný*, kdy se začíná od *exit* uzlu grafu řízení a výpočet postupuje proti směru hran. Větvení může být zpracováno tak, že informace je platná ve všech cestách (*musí* platit) nebo existuje cesta, kde je platná (*může* platit).

Algoritmus výpočtu toku dat pracuje následujícím způsobem. Vstupem je graf toku $G = (N, E)$ s $entry, exit \in N$ a svaz L. Pro každý základní blok programu reprezentovaný uzlem $b \in N$ algoritmus počítá informace toku dat vstupující do uzlu $IN(b)$ a informace toku dat vystupující z uzlu $OUT(b)$. Platí, že $IN(b), OUT(b) \in L$. Rovnice IN a OUT jsou tokové rovnice, které

mají při dopředné analýze tvar:

$$IN(b) = \begin{cases} Init & \text{pro } b = entry \\ \bigsqcup\limits_{p \in pred(b)} OUT(p) & \text{jinak} \end{cases}$$

$$OUT(b) = F_{type}(IN(b))$$

kde $Init$ je inicializační hodnota na vstupu do procedury, obvykle to bývá hodnota $\top$ nebo $\bot$. Funkce $pred(b)$ vrací množinu uzlů, ze kterých v příslušném grafu toku řízení vede hrana do uzlu b. *Operátor cest* $\bigsqcup$ modeluje efekt kombinace tokových informací na vstupu uzlu, kam vede více hran. Záleží na konkrétní analýze, jestli je na tomto místě použit operátor $\bigsqcup$, čili se bude hledat cesta, pro kterou nějaká vlastnost platí, nebo operátor $\bigsqcap$, kdy se bude hledat vlastnost platící pro všechny cesty. A konečně, funkce $F_{type}()$ je transformační funkcí, která realizuje vliv instrukce reprezentované uzlem b na tokovou informaci.

Algoritmus končí, jestliže se při posledním průchodu všemi uzly grafu hodnoty funkce $OUT(b)$ v žádném uzlu nezmění. Výstupem algoritmu jsou množiny hodnot z L platné pro jednotlivé uzly grafu G (výstupy příslušných funkcí $IN(b)$) a množiny hodnot z L platné pro jednotlivé hrany grafu G (výstupy funkcí $OUT(b)$, kde blok b je blok, ze kterého hrana vychází).

Tento algoritmus je velmi neefektivní, protože opakovaně prochází uzly, které již hodnoty tokových rovnic nemění. Velmi často se proto používá optimalizovaná varianta, které se říká *worklist algoritmus*. Worklist algoritmus udržuje seznam uzlů, které je třeba zpracovat. Na začátku výpočtu seznam obsahuje pouze uzel $entry$. V každém kroku výpočtu je první uzel b_1 ze seznamu vyjmut a zpracován. Po zpracování jsou v případě, kdy dojde ke změně hodnoty funkce $OUT(b_1)$, do seznamu přidány všechny uzly, které podle tokové relace následují uzel b_1. Výpočet končí, jestliže seznam neobsahuje žádný uzel určený ke zpracování.

Uvedený algoritmus a tokové rovnice lze snadno upravit pro výpočet zpětné analýzy toku dat. V takovém případě výpočet začíná od uzlu $exit$ a postupuje proti směru hran v grafu. Nepočítají se tedy následníci, ale předchůdci jednotlivých uzlů, a tokové rovnice mají následující tvar:

$$IN(b) = F_{type}(OUT(b))$$

$$OUT(b) = \begin{cases} Init & \text{pro } b = exit \\ \bigsqcup\limits_{p \in succ(b)} IN(p) & \text{jinak} \end{cases}$$

kde funkce $succ(b)$ vrací množinu uzlů do nichž vede hrana z uzlu b.

V mnoha analýzách toku dat se pracuje se svazem definovaným nad množinou všech podmnožin 2^A množiny A s definovanými množinovými operacemi sjednocení $\cup$ a průnik $\cap$. Nejmenší prvek $\bot$ vyjadřuje prázdnou množinou $\emptyset$ a největší prvek $\top$ reprezentuje celou množinou A. Transformační funkce $F_{type}(b)$ pro blok b lze pak pro dopřednou analýzu definovat podle následujícího vztahu:

$$F_{type}(b) = (IN(b) \bigcup GEN(b)) \setminus KILL(b)$$

kde funkce $GEN(b)$ vrací množinu prvků z A, které jsou platné po vykonání bloku b a funkce $KILL(b)$ vrací množinu prvků z A, které po provedení bloku b již neplatí. Tyto analýzy se také označují jako *bit-vektorové metody*, protože jejich výsledky lze reprezentovat vektorem, kdy každému prvku množiny A náleží jedna složka vektoru, a fakt, že aktuální informace obsahuje prvek z A lze reprezentovat jedničkou na příslušném místě vektoru.

Dosud uvedené iterativní přístupy, které se velmi často používají zejména pro svou implementační jednoduchost, pracovaly s grafem toku řízení. Existují ale také algoritmy, které pracují se stromem řízení získaným strukturální analýzou uvedenou v předchozí sekci. Z historických důvodů se tyto algoritmy označují jako *eliminační metody* podle Gaussovy eliminační metody, která se jim velmi podobá.

Vstupem algoritmu výpočtu dopředné analýzy je strom toku řízení $G = (N, E)$ a svaz L. Výpočet probíhá ve dvou průchodech všech uzlů grafu. V prvním průchodu, který je realizován od listových uzlů (základní bloky) ke kořeni (abstraktní uzel reprezentující celou proceduru), algoritmus konstruuje tokové funkce shrnující tok dat v podstromu začínajícím v daném uzlu. Pro základní bloky b jsou tyto funkce identické s $F_{type}(b)$ funkcemi z iterativního výpočtu. Funkce abstraktních uzlů jsou konstruovány pomocí kompozice tokových funkcí následníků počítaného uzlu. Například pro jednoduchou konstrukci `if-then-else` má příslušná toková funkce tvar:

$$F_{if-then-else} = (F_{then} \circ F_{if/Y}) \sqcap (F_{else} \circ F_{if/N})$$

kde operátor kompozice $\circ$ je definován vztahem $(f \circ g)(x) = f(g(x))$ a svazová operace $\sqcap$ je volena podle toho, jakou informaci počítáme, zda informaci, která musí platit na všech cestách, nebo informaci, která může platit na některé cestě (podobně jako tomu bylo u iterativního výpočtu). Funkce F_{then}, $F_{if/Y}$, F_{else} a $F_{if/N}$ odpovídají jednotlivým cestám v grafu toku řízení.

Druhý průchod všemi uzly grafu G je vykonáván od kořene k listům. Během tohoto průchodu jsou tokové funkce vytvořené v předcházejícím kroku využity pro šíření informace o toku dat. To se děje pomocí definování $IN(u)$

rovnic toku, které reprezentují vstupní hodnotu počítané informace z nadřazeného uzlu. Pro kořenový uzel platí $IN(root) = Init$, kde $Init$ je počáteční hodnota informace na vstupu do procedury (podobně jako u iterativního výpočtu to je nejčastěji hodnota $\top$ nebo $\bot$). Další rovnice jsou definovány typem abstraktního uzlu. Například pro náš případ `if-then-else` dostáváme rovnice:

$$IN(if) = IN(if - then - else)$$
$$IN(then) = F_{if/Y}(IN(if))$$
$$IN(else) = F_{if/N}(IN(if))$$

kde uzly if, $then$, $else$ jsou jednotlivé uzly grafu toku, které byly transformovány do abstraktního uzlu $if - then - else$.

Uvedený dopředný směr analýzy je v případě využití stromu toku řízení mnohem jednodušší než výpočet zpětného směru. To je dáno tím, že každý abstraktní uzel má právě jednoho rodiče (vstupní bod), ale může mít několik potomků (výstupní body) na základě řídicí struktury, kterou představuje. Pro jednoduché řídicí konstrukce, které mají jeden výstupní bod, lze jednoduše otočit tok řízení. Pro konstrukci `if-then-else` tak dostaneme po prvním průchodu:

$$F_{if-then-else} = (F_{if/Y} \circ F_{then}) \sqcap (F_{if/N} \circ F_{else})$$

A v druhém průchodu počítáme místo rovnice pro vstupní hodnoty IN, rovnice pro výstupní hodnoty OUT:

$$OUT(then) = OUT(if - then - else)$$
$$OUT(else) = OUT(if - then - else)$$
$$OUT(if) = F_{then}(OUT(then)) \sqcap F_{else}(OUT(else))$$

V případě struktur s více výstupními body je nutno v prvním průchodu identifikovat všechny možné cesty vedoucí od výstupních bodů ke vstupnímu bodu. Zejména v případě struktur se složitější vnitřní stavbou může být počet takových cest velmi vysoký. Pro cesty $p_1, \ldots, p_n$ pak první průchod algoritmu vytvoří funkci:

$$F_{struct} = F_{p_1} \sqcap \cdots \sqcap F_{p_n}$$

Při druhém průchodu se počítá rovnice:

$$OUT(struct) = F_{p_1}OUT(b_1) \sqcap \cdots \sqcap F_{p_n}OUT(b_n)$$

kde uzly $b_1, \ldots, b_n$ jsou uzly grafu toku před redukcí, které odpovídají výstupním uzlům struktury. Funkce $F_{p_n}OUT(b_n)$ je pak počítána pro cestu

složenou z bloků $bl_1, \ldots, bl_k$ grafu toku před redukcí, kde $b_n = bl_k$. Dostáváme tak kompozici funkcí jednotlivých bloků:

$$F_{p_n}OUT(b_n) = F_{bl_1}(F_{bl_1}(\ldots F_{bl_k}(out(bl_k))\ldots))$$

Některé úlohy vyžadují obousměrnou analýzu, čili šíření informace jak v dopředném, tak zpětném směru ve stejném okamžiku. Takové analýzy jsou pak většinou realizovány kombinací již představených dopředných a zpětných analýz.

V praxi se většinou využívají iterační podoby algoritmů hlavně pro jejich jednoduchost. Výhodou algoritmu pracujícího se stromem toku řízení je efektivita výpočtu pokud procedura neobsahuje složité struktury. V případě strukturální analýzy k tomu dochází málokdy, ale v případě intervalové analýzy se může jednat o častý jev. Stejně jako u analýzy toku řízení platí, že analýza využívající stromovou reprezentaci lépe reaguje na změny způsobené například optimalizačními algoritmy, protože přepočítává hodnoty jen v podstromu, kterého se změna týká.

4 Analýza závislostí

Zatímco analýza toku dat představená v předešlé sekci nachází široké uplatnění jak v překladačích, tak při formální verifikaci, následující analýza nachází využití zejména v překladačích, kde je hojně využívána při optimalizaci kódu.

Analýzu závislostí[1, 2] lze rozdělit na analýzu závislostí dat a analýzu závislostí řízení. Analýza závislostí řízení nám poskytuje informaci o podmínkách kladených na pořadí vykonávání instrukcí programu. Díky jejím výsledkům lze při optimalizaci pozměnit pořadí vykonávání instrukcí bez změny výsledku výpočtu. Podobně analýza závislostí dat poskytuje informace o podmínkách, které je nutno dodržet při vykonávání instrukcí přístupu do paměti. Závislostní analýza také hraje důležitou roli při paralelizaci a vektorizaci algoritmů.

Fakt, že instrukce I_1 předchází při vykonávání instrukci I_2, zapíšeme operátorem $\lhd$, tedy $I_1 \lhd I_2$. Závislost je vztah mezi dvěma instrukcemi, který vyžaduje dodržení pořadí jejich vykonání. *řídicí závislost* je omezení plynoucí z toku řízení. Například instrukce je prováděna pouze tehdy, pokud je splněn predikát podmínky větvení. *Datová závislost* je omezení plynoucí z toku dat mezi příkazy. Například zápis hodnoty musí předcházet jejímu čtení. Nejdříve se podívejme na možné závislosti dat a následně na řídicí závislost.

Rozlišujeme čtyři druhy závislosti dat mezi dvěma instrukcemi $I_1 \lhd I_2$ pracujícími se stejným paměťovým místem:

- *Závislost toku*, kde instrukce I_1 zapíše hodnotu, kterou I_2 čte.
- *Antizávislost*, při níž instrukce I_1 čte hodnotu, kterou I_2 posléze přenastaví.
- *Závislost výstupu*, kde obě instrukce I_1 i I_2 zapisují hodnotu proměnné.
- *Závislost vstupu*, při které obě instrukce I_1 i I_2 čtou hodnotu proměnné. Tato závislost se většinou neuvažuje, protože dostupná technika umožňuje souběžné čtení paměťového místa.

Zmíněné čtyři závislosti lze rozšířit ještě o dvě další:

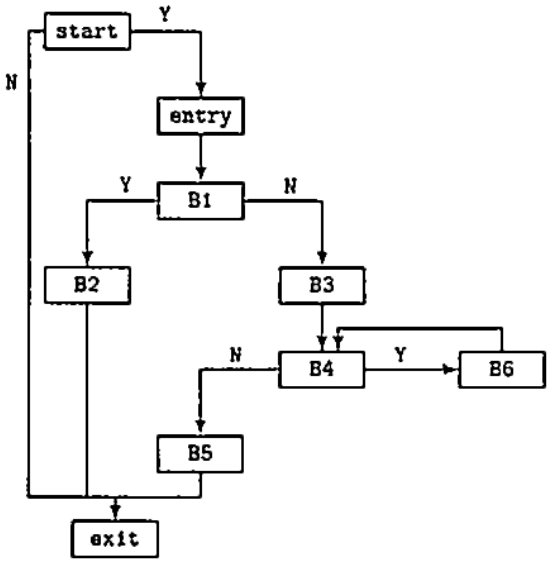
- Možnost, kdy nejsme schopni závislost rozhodnout. Například v případě, kdy nevíme, zda dvě proměnné s nimiž dané instrukce pracují neukazují na stejné paměťové místo (to lze zjistit pomocí alias analýzy uvedené v následující sekci).
- Tzv. *strukturální hazard*, což je situace, kdy dvě instrukce soutěží o stejné výpočetní zdroje (například aritmeticko logickou jednotku).

Závislosti lze snadno reprezentovat orientovaným grafem $G = (N, E)$, jehož uzly představují jednotlivé instrukce a hrany $(a, b) \in E$ reprezentují fakt, že existuje závislost mezi uzly $a, b \in N$, která vynucuje pořadí vykonání $a \triangleleft b$. Každá hrana nese označení typu závislosti, kterou představuje. Tento graf se nazývá *graf závislosti dat*. Algoritmus vytvářející tento graf jednoduše vyhodnocuje funkci $Conflict(I_1, I_2)$ pro všechny kombinace instrukcí. Funkce $Conflict$ má následující předpis:

$$Conflict : Instr \times Instr \rightarrow boolean$$

kde $Instr$ je množina instrukcí a *boolean* je boolovská hodnota. Funkce $Conflict(I_1, I_2)$ vrací *true*, pokud instrukce I_1 musí být vykonána před instrukcí I_2, kvůli správnému výstupu běhu programu.

Analýza závislostí v řízení nám poskytuje informaci o tom, na kterých rozhodnutích je závislé vykonání nějakého základního bloku při větvení programu. Tato analýza tedy pracuje se základními bloky a s predikáty, které reprezentují testy před větvením, jenž jsou vyhodnocovány na konci jednotlivých základních bloků. Výpočet závislostí v řízení tedy pracuje s grafem toku řízení a pro výpočet využívá relaci postdominance, představenou v sekci 2. Řekneme, že základní blok b je závislý na řízení bloku a právě tehdy, když v grafu řízení existuje cesta z a do b taková, že ke každému uzlu na této cestě kromě uzlu a je uzel b postdominantem. Uzel a je tedy prvním uzlem na této cestě, kterému uzel b není postdominantem.



Obrázek 5: Upravený graf toku řízení (převzato z [1]).

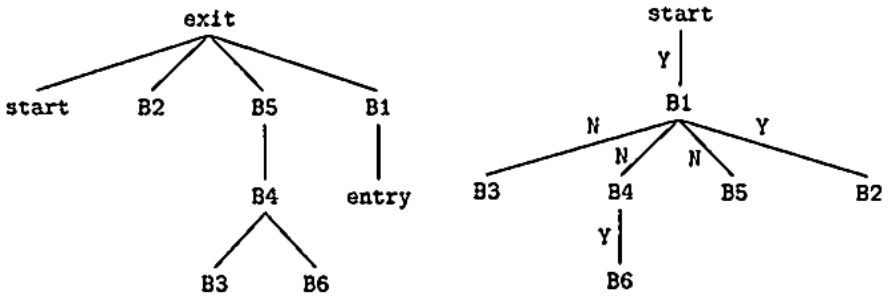
Výsledek analýzy závislostí v řízení opět můžeme reprezentovat grafem. Konstrukce *grafu závislosti v řízení* probíhá ve dvou krocích, kdy je nejdříve sestaven tzv. základní graf závislosti v řízení, který je posléze rozšířen o tzv. *regiony*. Konstrukce základního grafu, který se konstruuje z grafu toku řízení $G = (N, E)$, se skládá z následujících kroků:

(1) Graf toku řízení je rozšířen o nový počáteční uzel *start*, reprezentující základní blok s predikátem, jehož kladné vyhodnocení vede tok řízení do uzlu *entry* a záporné vyhodnocení do uzlu *exit*. Všechny hrany v rozšířeném grafu označíme písmenem Y, pokud hrana odpovídá toku řízení pří splnění predikátu na konci základního bloku odpovídajícímu výchozímu uzlu hrany. Obdobně jsou ostatní hrany označeny písmenem N, pokud predikát ve výstupním uzlu hrany není splněn. Vzniklý graf označme $G^+ = (N^+, E^+)$. Příklad rozšířeného grafu toku řízení je na obrázku 5.

(2) K takto rozšířenému grafu je vytvořen graf relace postdominance $G_{pdom} = (N_{pdom}, E_{pdom})$, který tvoří strom s kořenem v uzlu *exit*. Strom postdominance k rozšířenému grafu toku řízení z obrázku 5 je na obrázku 6a.

(3) V rozšířeném grafu G^+ je definována množina hran $S \subset E$ takových, že pro $(m, n) \in S$ uzel n není postdominátorem uzlu m. Čili platí, že $S = E^+ \backslash E_{pdom}$.

(4) Pro každou hranu $(m, n) \in S$ určíme nejmenšího společného předka l uzlů m a n v grafu G_{pdom}. Pokud je uzel m kořenem ($m = exit$), pak $l = m$. Uzel l je tedy buď uzel m nebo jeho rodič a všechny uzly v grafu G_{pdom} na cestě z l do n kromě uzlu l jsou závislé na uzlu m, označme je D_m. Výsledný graf závislosti řízení je tedy neorientovaný graf obsahující hrany (a, b), kde



(a) Strom postdominance. (b) Základní graf závislosti řízení.

Obrázek 6: Grafy postdominance a řídicích závislostí (převzato z [1]).

a jsou uzly m získané z množiny hran S a b jsou jednotlivé uzly z příslušné množiny D_m. Všechny hrany tohoto grafu jsou označeny písmeny Y nebo N podle boolovského vyhodnocení predikátu v uzlu a příslušné hrany (a, b). Graf znázorňující závislosti v našem příkladě je uveden na obrázku 6b.

Významem regionu je spojení uzlů, které jsou závislé na stejném vyhodnocení stejného predikátu, do jednoho abstraktního uzlu grafu. Díky tomu má každý predikát nejvýše dva následníky.

Analýza závislostí se v mnohém komplikuje, pokud se v analyzovaném programu vyskytnou zanořené smyčky, rekurze či složitější datové struktury. V současné době existuje velké množství specializovaných algoritmů, které jsou schopny za cenu náročnějšího výpočtu rozhodnout, zda existuje mezi dvěma příkazy (základními bloky) závislost nebo nikoli. Přesněji řečeno algoritmy se snaží dokázat, že mezi příkazy závislost není, aby mohlo být využito progresivnějších optimalizačních metod.

Blízkým a v posledních letech široce studovaným tématem je analýza s názvem *program slicing*[3], která se snaží o nalezení podmnožiny celého programu, která má vliv na zvolenou proměnnou nebo větev programu.

5 Alias analýza

Alias analýza[1, 2, 3], někdy také označovaná termínem *analýza ukazatelů*, slouží k určení takových paměťových míst, která mohou být přístupná z několika proměnných v programu. Jinak řečeno se tato analýza snaží identifikovat množinu příkazů programu, jenž pracují se stejným paměťovým místem. Tato informace je velmi důležitá pro mnoho dalších analýz a optimalizací, které při svých výpočtech pracují s paměťovými místy či obecně s daty a ukazateli. Náročnost detailní alias analýzy je ve většině moderních programovacích jazyků, které podporují široké spektrum operací s ukazateli, velmi vysoká. Proto se často počítá pouze některá z aproximací ideálního

řešení.

U aproximací se rozlišuje, zda dvě proměnné *mohou* ukazovat na stejné paměťové místo, tedy, že existuje taková větev v grafu toku, kde k této situaci dochází, nebo *musí* ukazovat na stejné paměťové místo, tedy, že ve všech větvích grafu toku k této situaci dojde. Dále se rozlišuje, zda byl při výpočtu zohledněn graf toku řízení procedury, pak mluvíme o výpočtu *zohledňujícím* tok řízení, nebo byl ignorován, pak mluvíme o výpočtu *nezohledňujícím* tok řízení. Kombinací těchto přístupů dostáváme následující čtyři úrovně přesnosti informace alias analýzy:

(1) Graf řízení *nezohledňující* analýza proměnných, které *mohou* ukazovat na stejné místo. V tomto případě lze výsledek analýzy chápat jako binární relaci nad množinou proměnných $alias \in Var \times Var$ takovou, že $x\ alias\ y$ platí tehdy, když proměnné x a y mohou ukazovat na stejné paměťové místo, a to kdykoli během provádění procedury. Tato relace je symetrická a není tranzitivní. O tranzitivitě nelze mluvit, protože dvojice proměnných mohou ukazovat na stejné místo v různých částech kódu procedury a tudíž v různých časových okamžicích provádění procedury.

(2) Graf řízení *nezohledňující* analýza proměnných, které *musí* ukazovat na stejné místo. I zde je výsledkem analýzy binární relace $alias \in Var \times Var$, ovšem taková, že x a y musí během vykonávání procedury ukazovat na stejné paměťové místo. Tato relace je symetrická a tranzitivní.

(3) Graf řízení *zohledňující* analýza proměnných, které *mohou* ukazovat na stejné místo. V tomto případě dostáváme množinu binárních relací, kterou je však lépe reprezentovat funkcí $alias(p, v) = SL$, kde p je stav programu (mezi dvěma po sobě následujícími příkazy), v je proměnná a SL je množina paměťových míst, kam může proměnná v ukazovat. Alias informaci platnou pro dané místo v programu lze pak získat pomocí množinové operace $\cap$.

(4) Graf řízení *zohledňující* analýza proměnných, které *musí* ukazovat na stejné místo. I v tomto případě lze reprezentovat výsledek analýzy pomocí funkce tvaru $alias(p, v) = l$, kde dvojice p a v odpovídá předešlému případu a l reprezentuje jedno unikátní místo v paměti, kam právě proměnná v ukazuje. Tato analýza poskytuje nejpřesnější výsledky a její výpočet je tudíž nejnáročnější.

Výpočet alias analýzy silně závisí na programovacím jazyce, který je analyzován. Různé jazyky dovolují různé množství operací s ukazateli. Proto lze alias analýzu rozdělit na dvě části: jazykově specifickou a jazykově nezávislou.

Úkolem jazykově specifické části je detekovat operace, kdy může dojít k situaci, že dvě proměnné ukazují na stejné paměťové místo. K tomu může dojít například v následujících situacích:

- Pokud může dojít k překrytí paměti alokované pro dva různé objekty.

- Když v programu existují odkazy na pole, části polí nebo jednotlivé elementy polí.
- Jestliže program obsahuje ukazatele.
- Pokud jsou hodnoty proměnných předávány do procedur odkazem.

Na náročnost analýzy mají vliv i další specifika jazyka, jakými je například přítomnost dynamicky alokované paměti. V tomto případě je nutné se rozhodnout, zda tuto vlastnost při výpočtu alias informace zohlednit, například pojmenováním každého alokovaného místa a jeho sledováním, nebo tuto vlastnost jazyka ignorovat a o všech dynamicky alokovaných místech prohlásit, že mohou být přístupná pomocí příslušné množiny proměnných.

Jazykově nezávislá část pak šíří informaci o efektu detekovaných operací do zbylého programu. Často se k tomuto šíření využívá mechanismus analýzy toku dat uvedený v kapitole 3.

Princip výpočtu alias analýzy pomocí mechanismu analýzy toku dat demonstrujme na příkladě analýzy vybraných konstrukcí jazyka C. Nechť naše alias analýza *zohledňuje* graf řízení a počítá množiny míst, kam *mohou* ukazovat jednotlivé proměnné. Vstupem algoritmu je graf toku, který má jeden vstupní uzel *entry*, jeden výstupní uzel *exit* a ostatní uzly reprezentují jednotlivé příkazy. Stav systému, který chápeme jako stav mezi dvěma po sobě následujícími příkazy, lze ztotožnit s pojmenováním hrany grafu toku.

Jazykově závislá část výpočtu tak pracuje s množinou stavů systému P, s množinou objektů X, které mohou obsahovat hodnoty (proměnné, pole, ...) a s množinou všech abstraktních paměťových míst M. Algoritmus počítá dvě funkce, které mapují stavy a objekty na abstraktní paměťová místa, konkrétněji jsou tvaru $P \times X \rightarrow 2^M$. Funkce $ovr_p(x)$ udává množinu abstraktních paměťových míst, která mohou být využita objektem $x \in X$ ve stavu $p \in P$. Funkce $ptr_p(x)$ vrací paměťová místa, na která může x ukazovat ve stavu p. Při výpočtu je použita i funkce $ref_p(x)$, která udává množinu míst, která mohou být dostupná z x v bodě p po libovolném počtu dereferencí.

Jazykově závislý výpočet pak probíhá na daném grafu toku, kdy jsou analyzovány uzly (respektive jim odpovídající příkazy), ze kterých vychází hrana označená p. Pro každou instrukci, která má vliv na alias informaci, je vypočtena hodnota funkce $ptr_p(x)$. Protože takových příkazů je v případě jazyka C velké množství, vezměme jen některé příklady:

- Přiřazuje-li příkaz ukazateli x hodnotu *null*, pak $ptr_p(x) = \emptyset$.
- Je-li příkaz tvaru `x1 = x2`, pak $ptr_p(x1) = ptr_p(x2) = ptr_{p'}(x2)$, kde p' označuje předcházející stav.

- Je-li příkaz tvaru $x = \&a[expr]$, kde x je ukazatel a $a[expr]$ je buňka pole, pak $ptr_p(x) = ovr_p(a)$.
- Pokud příkaz představuje volání funkce $f()$, pak $ptr_p(x) = ref'_p(x)$ pro všechny ukazatele x, které jsou použity jako argumenty funkce x.

Jazykově nezávislá část analýzy pak využívá nástroje pro analýzu toku dat k šíření získaných ptr_p a ovr_p množin pro příslušné proměnné x. Globální flow funkce označme Ptr a Ovr. Bez dalšího popisu uveďme jejich tvary. Pro stavy s jedním předchozím stavem p':

$$Ovr(p,x) = \begin{cases} ovr_p(x) & \text{jestliže je } x \text{ ovlivněn předchozím příkazem} \\ Ovr(P',x) & \text{jinak} \end{cases}$$

$$Ptr(p,x) = \begin{cases} ptr_p(x) & \text{jestliže je } x \text{ ovlivněn předchozím příkazem} \\ Ptr(P',x) & \text{jinak} \end{cases}$$

U stavů s více předchozími stavy jsou hodnoty Ptr (respektive Ovr) získány sjednocením množin Ptr (Ovr) všech předcházejících stavů $p'_1, p'_2, \ldots$.

Uvedený příklad lze snadno upravit pro výpočet ostatních typů alias analýz a to buď úpravou *ptr* funkcí, aby vracely místo množiny paměťových míst právě jedno místo, nebo nahrazením grafu toku obyčejnou posloupností příkazů.

Problematika analýzy ukazatelů a paměti obecně je v současné době hluboce studovanou oblastí analýzy programů. Různé přístupy jsou schopny za cenu vysoké náročnosti výpočtu poskytovat přesné informace o tom, jak program s pamětí pracuje. Bohužel, náročnost výpočtu je v těchto případech natolik vysoká, že se nedají pro komplexní programy na současném hardware efektivně využít.

6 Meziprocedurální analýza

U všech předchozích analýz jsme uvažovali o analýze části kódu odpovídající jedné proceduře, respektive programů, které procedury neobsahují. Moderní programovací jazyky však umožňují dělit bloky kódu do procedur či metod, čímž se kód stává přehlednějším, lépe udržovatelným a znovupoužitelným. V této sekci budou představeny techniky, kterými lze předchozí analýzy rozšířit tak, aby podporovaly programy obsahující více procedur.

Nejjednodušším a v mnoha případech používaným způsobem řešení *meziprocedurální analýzy*[2, 1, 5] je tzv. *automatický rozvoj procedur* (anglicky inlining). Tato metoda nahradí volání procedury kopií těla procedury. Vzniká tak program sestávající z jedné procedury, který již dříve zmíněné přístupy

umí analyzovat. Nevýhodou tohoto řešení je, že tak vznikají velké programy, které některé druhy analýz nejsou schopny s dostupnými prostředky pojmout. Tuto metodu také nelze použít v případě přítomnosti rekurzivního volání. Vznikl by tak nekonečný cyklus rozvíjející tuto rekurzy. Do jisté míry a s újmou na přesnosti analýz lze však tento problém řešit omezením počtu rekurzivních vnoření.

Mnohem robustnějším řešením je využití *grafu volání*. Graf volání pro program, který obsahuje množinu procedur $P = \{p_1, p_2, \ldots, p_n\}$, lze popsat čtveřicí $G = (N, S, E, r)$. Uzly grafu N reprezentují jednotlivé procedury, množina uzlů tedy obsahuje $N = \{p_1, p_2, \ldots, p_n\}$. S je množina identifikátorů tzv. *míst volání*, které budou popsány v následujících odstavcích. Množina $E = N \times N \times S$ je množinou ohodnocených hran, kde se hrana $e = (p_i, s_k, p_j)$ interpretuje jako volání procedury p_j z místa s_k v proceduře p_i. Posledním členem čtveřice je uzel $r \in N$ označující vstupní uzel grafu, kterým je nejčastěji procedura `main`.

Zaměřme se nyní na místa volání. Toto místo identifikuje *kontext*, ze kterého je procedura volána. Nejjednodušší meziprocedurální analýzy kontext úplně ignorují. V grafu volání tedy existuje pouze jeden uzel pro každou proceduru. V takovém případě může množina S obsahovat například čísla řádků obsahujících volání nějaké procedury. Příklad tohoto přístupu je ukázán na obrázku 7. Obrázek 7a ukazuje kostru programu a na obrázku 7b je znázorněn příslušný graf volání.

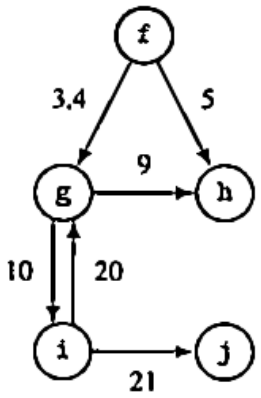
Mnohé analýzy potřebují pro zajištění dostatečné přesnosti výsledků znát kontext volání o něco přesněji. V takových případech mluvíme o *řetězcích volání*, které jednoznačně určují všechny volané procedury. Řetězec volání obsahuje přepis zásobníku volání v době, kdy je procedura volána. Řetězce mohou být v některých případech příliš dlouhé (v případě rekurze i nekonečné), a proto se často přistupuje k jejich omezování. Řetězce pak obsahují pouze k posledních záznamů ze zásobníku (označujeme je k-omezené), což má samozřejmě vliv na přesnost navazujících analýz.

Pro jednoduché programy je vytvoření grafu volání intuitivní, a proto zde nebude uveden algoritmus jeho vytvoření. Místo toho věnujme prostor nástinu řešení problémů, které mohou nastat při jeho vytváření. Prvním problémem může být fakt, že analyzátor nemá k dispozici celý kód programu. V takovém případě do grafu vložíme uzly reprezentující ty procedury, jejichž obsah neznáme, ovšem při pozdějších analýzách počítáme s nejhorší možnou variantou chování takových procedur.

Dalším problémem je přítomnost proměnných ovlivňujících tvorbu grafu. Pokud například neznáme hodnotu ukazatele, nad nímž je volána procedura, neznáme cíl volání. V tomto případě je nutné budovat graf volání inkrementálně. V první iteraci je vybudována největší možná část grafu na základě

```
1     procedure f( )
2     begin
3        call g( )
4        call g( )
5        call h( )
6     end   || f
7     procedure g( )
8     begin
9        call h( )
10       call i( )
11    end   || g
12    procedure h( )
13    begin
14    end   || h
15    procedure i( )
16       procedure j( )
17       begin
18       end   || j
19    begin
20       call g( )
21       call j( )
22    end   || i
```

(a) Kód programu.



(b) Příslušný graf volání.

Obrázek 7: Jednoduchý graf volání (převzato z [1]).

explicitních volání. Pak je vygenerována množina proměnných, které je třeba vyhodnotit, aby mohl být graf rozšířen. Pro každou proměnnou je určena iniciální hodnota, která je posléze šířena do všech míst použití. Podle přesnosti použité analýzy a složitosti programu může proměnná nabývat jedné nebo i více hodnot. V případě množiny hodnot je v následné iteraci graf rozšířen o množinu hran vedoucích z daného místa volání. Algoritmus končí, když je množina proměnných, které jsou určeny k vyhodnocení po provedení iterace budování grafu, prázdná.

Nyní, když máme k dispozici graf volání, můžeme přistoupit k jeho využití. V závislosti na typu analýzy lze uzly (tedy procedury) grafu procházet v různém pořadí. Nejčastěji se volí průchod podle pořadí invokace procedur (respektive proti pořadí invokace), kdy je nejdříve zpracován vstupní uzel grafu (respektive listové uzly), a pak metodou BFS procházen celý graf.

U analýz, které berou v úvahu kontext volání, se často používají následující optimalizace uchovávání mezivýsledků pro jednotlivé procedury. První z nich je založen na *klonování* procedur. Pro každý řetězec volání je vytvořen klon volané procedury, do kterého vede hrana z místa volání. V případě, že procedura s takovýmto řetězcem volání už existuje, procedura není klonována a

do grafu je přidána hrana vedoucí z místa volání do této existující procedury.

Druhý přístup využívá *sumarizaci* procedur. Pro každou proceduru a příslušný řetězec volání je procedura analyzována a výsledek analýzy uložen v podobě shrnutí jejího obsahu vzhledem k prováděné analýze. Existuje-li již výsledek pro danou dvojici procedury a řetězce volání z dřívějšího výpočtu, procedura se znova nevyhodnocuje a použije se předpočítaná hodnota. Analýza grafu toku v tomto případě probíhá ve dvou fázích. (1) Průchodem zespodu navrch jsou vypočítány sumarizace efektů jednotlivých procedur. (2) Průchodem z vrchu dolů, jsou pak šířeny informace z míst volání k volaným procedurám. Rekurzi lze v tomto případě řešit výpočtem pevného bodu řešení.

Pro zkoumání manipulací s daty (například při analýze toku dat) je důležité vědět, jaké proměnné mohou být dotčeny jako *vedlejší efekt* provedení procedury a jaké parametry volané procedury nebudou změněny, tj. zůstanou konstantní. Vedlejší efekt procedury lze charakterizovat pomocí čtyř funkcí tvaru:

$$DEF, MOD, REF, USE : Procedure \times integer \rightarrow Var$$

Funkce $DEF(p, i)$ vrací množinu proměnných, které musí být definovány (tj. musí obsahovat hodnotu) před vykonáním i-té instrukce procedury p. Funkce $MOD(p, i)$ vrací množinu proměnných, které mohou být změněny vykonáním i-té instrukce procedury p. Funkce $REF(p, i)$ vrací množinu proměnných, jejichž hodnoty mohou být použity při vykonání i-té instrukce procedury p. A funkce $USE(p, i)$ vrací množinu proměnných, které mohou být použity při vykonání i-té instrukce procedury p.

Výpočet těchto funkcí je velmi jednoduchý, pokud máme k dispozici výsledky alias analýzy a spokojíme se s výpočtem, který nezohledňuje graf řízení dané procedury. Informace o vedlejších efektech vykonání procedury lze také použít při optimalizaci výpočtu meziprocedurální analýzy.

Představu o meziprocedurální podobě analýz uvedených v předchozích sekcích si lze celkem snadno udělat opětovným připomenutím jejich výpočtu. Všechny analýzy byly koncipovány tak, aby analyzovaná procedura měla právě jeden vstupní a jeden výstupní bod. V případě grafů byly tyto body reprezentovány uzly *entry* a *exit*. V prvním kroku analýzy se ve vstupním (respektive výstupním) uzlu definovaly iniciální hodnoty počítaných množin či hodnot. Při meziprocedurální analýze jsou tyto inicialní hodnoty vypočítány z informací dostupných v místě volání procedury. Výsledek analýzy procedury je pak šířen zpět do míst volání procedur a to jak v podobě hodnot z *exit* (respektive *entry*) uzlu, tak v podobě vedlejších efektů vykonání procedury.

7 Závěr

V předchozích kapitolách byly představeny některé základní principy zvolených postsyntaktických statických analýz kódu. O uvedené problematice existuje mnoho vědeckých článků a publikací. Pro každou z uvedených analýz bylo publikováno několik algoritmů, které se liší v přesnosti výsledků a náročnosti výpočtu. Tento text se u každé z uvedených analýz snažil uvést základní myšlenku a nastínit některé závažné problémy, které je nutno při dané analýze řešit. Většinou byl pro ilustraci uveden i vybraný algoritmus, který ilustroval jedno z možných řešení problému.

Reference

[1] Muchnick, S.: Advanced Compiler Design & Implementation, Academic Press, 1997.

[2] Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and tools, 2nd edition, Addison-Wesley, 2007.

[3] Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis, Springer, 2005.

[4] Češka, M., Hruška, T., Beneš, M.: Překladače, 3. vydání, SNTL, 1993.

[5] Schvartzbach, M.: Lecture Notes on Static Analysis. Dokument je dostupný na URL http://www.brics.dk/ mis/static.pdf [27. 5. 2009].