Výřezy programů v procesu ladění a testování

RNDr. Tomáš Havlát, CSc
Katedra matematické informatiky PřF UJEP
Kotlářská 2, 611 37 Brno

1 Motivace

Ne vždy zajímá programátora pouze globální chování programu jako funkce transformující vstupní data na výstupní. Jednou z takových situací je například zjištění existence chyby při testovacím nebo rutinním běhu programu. V tom případě je nutné provést klasifikaci chyby (o jaký druh chyby jde) a lokalizovat místo v programu, kde se chyba projevila.

Klasifikaci chyby provádí buď programátor (program se zacyklil, výstupní data jsou nesprávná, ...), nebo více či méně věrohodně výpočetní systém příslušnou zprávou (dělení nulou, odmocnění záporné hodnoty, ...). Lokalizaci chyby v lepším případě provede rovněž výpočetní systém (odkazem do textu programu) nebo programátor sám (případně s pomocí trasování). Po klasifikaci a lokalizaci je nutno zjistit příčiny odhalené chyby. Z automatizovaných prostředků může být programátorovi nápomocen např. on-line ladicí systém (pokud je k dispozici), ale nejčastěji nastupuje klasické studium výpisu programu. Programátora v tomto případě nezajímá program jako celek, ale pouze ty příkazy, které mohly potenciálně ovlivnit vznik dané chyby.

Tím se dostáváme, zatím na intuitivní úrovni, k pojmu *výřezu programu*, který obsahuje z příkazů původního programu jen ty, které mají vliv (jsou potřebné) na jisté zvolené vlastnosti programu v době běhu. Při konstrukci výřezu S programu P budeme vždy vycházet 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. Nepotřebné (z daného hlediska) příkazy pak z programu vypustíme tak, aby vzniklý výřez byl programem, který má sledované vlastnosti (téměř) totožné s původním programem.

Než přistoupíme k formální definici výřezu, zavedeme potřebné pojmy.

2 Základní pojmy

Graf toku řízení (GTŘ) programu P je orientovaný graf (s právě jedním počátečním

a jedním koncovým uzlem), jehož uzly reprezentují (elementární) příkazy programu, hrany možnost předání řízení mezi jednotlivými příkazy.

V dalším často nebudeme rozlišovat mezi programem a jeho grafem toku řízení, resp. mezi příkazy programu a uzly grafu toku řízení.

Pro každý uzel u (= příkaz) grafu toku řízení definujme množinu jeho předchůdců PRED(u) resp. následníků SUCC(u), která obsahuje ty uzly, ze resp. do kterých vede z resp. do uzlu u hrana grafu toku řízení.

Označme jako DEF(p) resp. REF(p) množinu proměnných, kterým je přiřazena hodnota v příkazu p resp. referencovaných v příkazu p. (např. je-li p příkaz x:=x+y je DEF(p)={x}, REF(p)={x,y}).

Informační graf (IG) programu P s grafem toku řízení G je definován následovně:

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 definovaná, v uzlu v referencována a v GTŘ existuje cesta z uzlu u do uzlu v taková, že v žádném vnitřním uzlu této cesty není proměnná x definována.

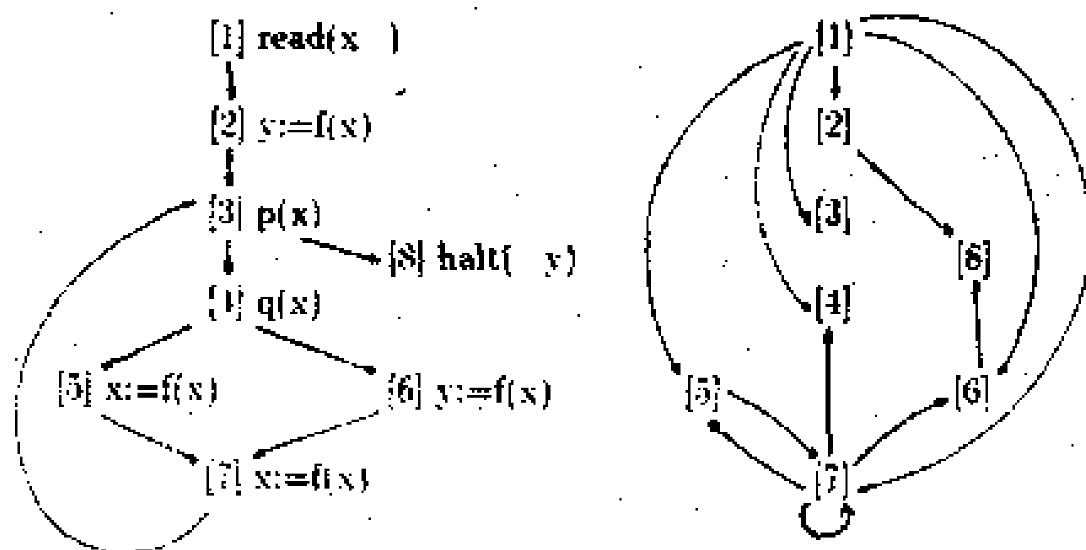

Figure 1: Graf toku řízení a informační graf

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:

$$REACHES(s) = \emptyset$$

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

kde

1. s je počáteční uzel GTŘ

2. NOTKILL(v) je množina příkazů, v nichž je definována hodnota těch proměnných, které nejsou redefinovány v uzlu v

3. GEN(v)={v} jestliže DEF(v)≠ ∅, jinak GEN(v)=∅

Na řešení takového systému rovnic existuje celá řada metod analýzy toku dat, čtenář je najde například v [Hecht77].

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∈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 následující 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

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

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

Analogicky tomu, jak jsme v informačním grafu zaznamenali potencionální vliv příkazů přiřazujících vypočtenou hodnotu na příkazy, které tuto hodnotu mohou referencovat, bychom v dalším chtěli zachytit vliv větvících příkazů na ty příkazy, jejichž provedení či neprovedení je jistým způsobem ovlivněno výběrem větve ve větvícím příkazu.

Označme jako **DK(u)** příkaz, který je *nejbližším dominátorem* vzhledem ke konci příkazu u (*dominátor* vzhledem ke konci daného příkazu u je příkaz p, který leží na každé cestě grafem toku řízení z příkazu u do koncového příkazu). Algoritmy pro nalezení dominátorů jsou uvedeny opět v [Hecht77].

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≠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.

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)≠y
3. GEN(v)={v} pro větvící příkazy v, jinak GEN(v)= ∅

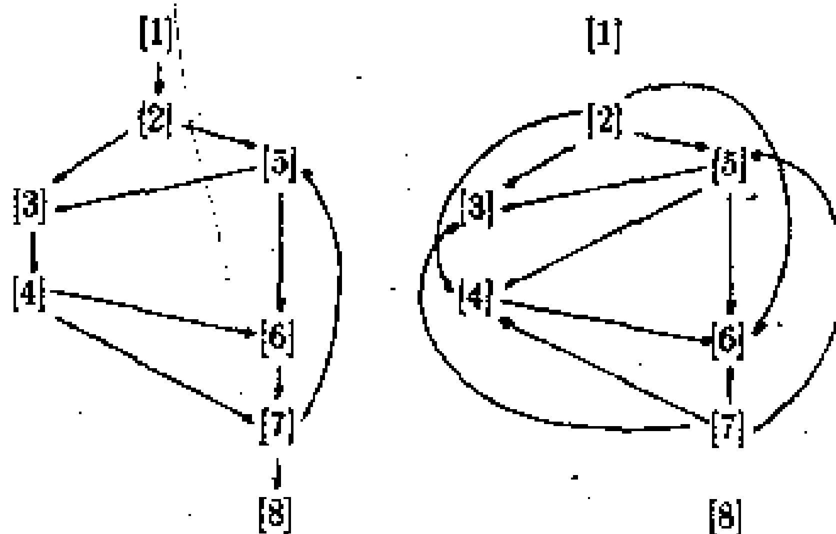

Figure 2: Graf toku řízení a graf vlivu větvení

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 uzlu v právě tehdy, když u∈BRANCH(v).

3 Výřezy programů

V motivační části jsme dospěli k tomu, že výřez programu má být programem, který má jisté, námi zvolené, vlastnosti (téměř) totožné s původním programem, a že sledovaná vlastnost indukuje přímo či nepřímo množinu příkazů, které jsou pro zachování této vlastnosti u výřezu apriori nezbytné.

Budeme tedy předpokládat, že výřez je zadán množinou **AUST** apriori potřebných příkazů. Další příkazy potřebné pro zachování sledované vlastnosti u výřezu získáme

na základě informací obsažených v informačním g afu a v grafu vlivu větvení programu algoritmem **SUST** uvedeným na obrázku 3, který určí množinu **UST** všech potřebných příkazů.

Vyjasněme nyní otázku, jak vypouštet z GTŘ G programu příkazy, které nepatří do množiny potřebných příkazů UST. Nepotřebné příkazy se dají rozdělit do navzájem disjunktních skupin tak, že
a) neexistuje hrana, která v G vede z uzlu patřícího do jedné skupiny do uzlu patřícího do jiné skupiny
b) pro každou skupinu existuje právě jeden uzel z množiny UST takový, že do něj vedou hrany z uzlů skupiny.
Vypuštění nepotřebných příkazů z GTŘ G můžeme tedy realizovat tak, že pro každou skupinu vypustíme podgraf indukovaný uzly skupiny, hrany, které vedou z uzlů skupiny do nějakého (právě jednoho) uzlu u z UST, a hrany, které vedou z uzlů množiny UST do uzlů skupiny nahradíme hranami vedoucími do uzlu u.

Tím je *výřez programu* zkonstruován. Konstrukce je korekt..í, výřez je programem.

Ilustrujme nyní zavedené pojmy a postupy. Uvažme program P na obr. 4 a množinu apriori potřebných příkazů AUST={1,7,8,11..13}. Algoritmem SUST nalezená množina všech potřebných příkazů UST={1,3,4,7,8,11,12,13,14}. Zmíněné skupiny nepotřebných příkazů jsou {2}, {5,6}, {9,10}. Odpovídající výřez je rovněž na obr. 4.

4 Vlastnosti a použití výřezů

První důležitou vlastností libovolného výřezu S programu P (odvoditelnou z konstrukce výřezu) je, že **vždy, když program P zastaví** pro libovolná vstupní data, pak **výřez S** pro tatáž vstupní data **rovněž zastaví**. Obrácená implikace bohužel neplatí: zacyklí-li se např. program P na obr. 3 v cyklu tvořeném příkazy 5 a 6, jeho výřez S může zastavit. Tuto vlastnost výřezů musíme mít vždy napaměti při aplikacích výřezů.

Při formulaci dalších vlastností výřezů budeme pracovat s pojmem *výpočet... posloupnosti* programu P pro daná vstupní data. Výpočetní posloupnost je 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 tatáž 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

```
Vstup:  1) AUST-mnozina apriori potrebnych prikazu
        2) mnoziny DEFS(x,p), BRANCH(p)
Vystup: mnozina UST vsech potrebnych prikazu

begin
   UST := ∅; POM := AUST;
   while POM ≠ ∅ do
      necht p je libovolny prikaz 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 koncovy prikaz (zastaveni)}
end.
```

Figure 3: Algoritmus SUST–hledání potřebných příkazů

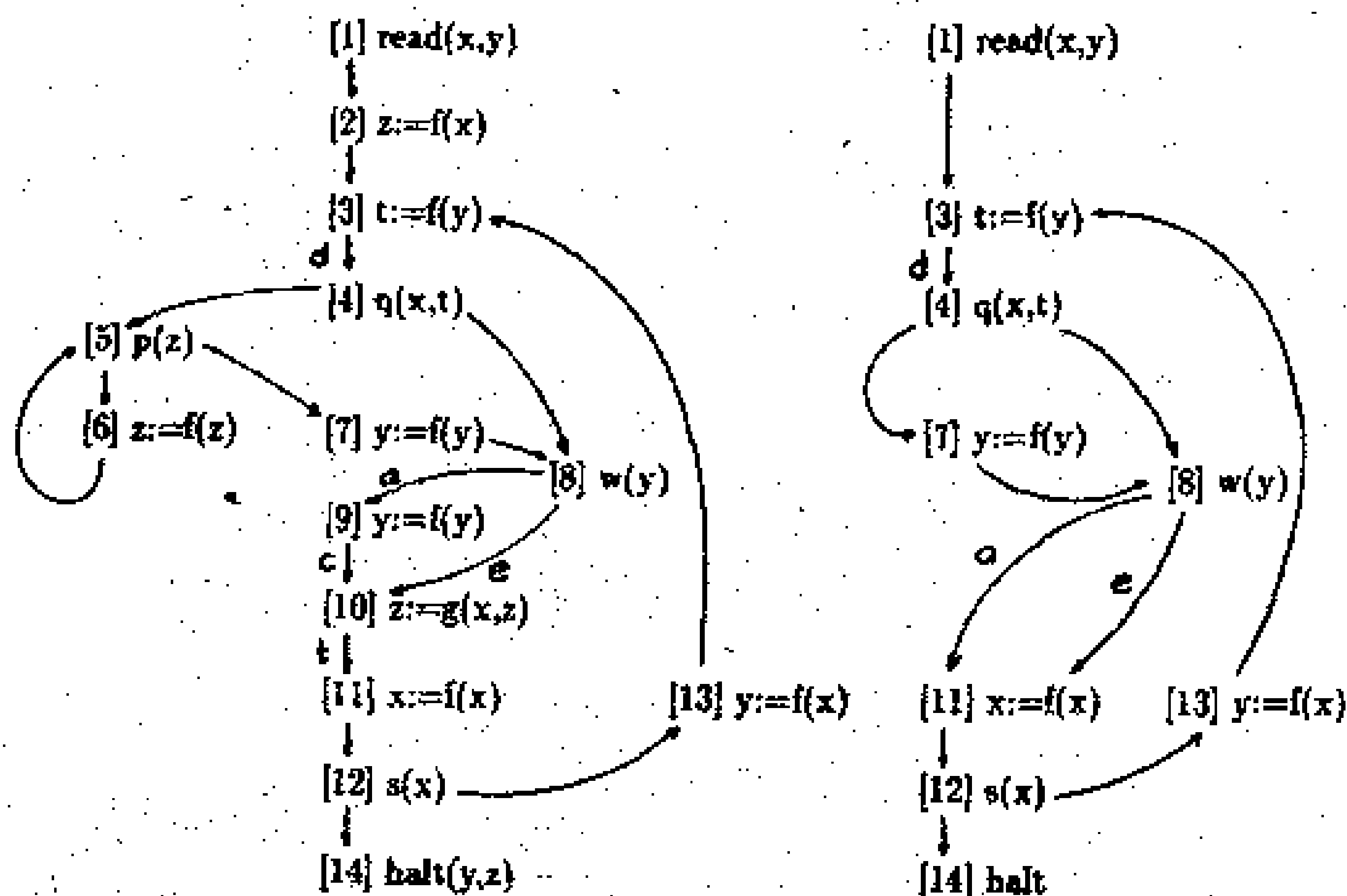

Figure 4: Program P a výřez S programu P

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, stejné služby jako program, ale v mnoha případech laciněji nám poskytne 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. **Stop-výřez** má vlastnost, že pro daná vstupní data **zastaví právě tehdy, když zastaví** původní **program** pro tatáž data. Stop-výřez programu z obr. 4 je [1] read(x); **repeat** [11] x:=f(x) **until** [12] s(x); [14] halt.

Zajímá-li nás v daném okamžiku vše (které příkazy a jakým způsobem), co souvisí s tokem řízení při výpočtu, pak je k dispozici tzv. *řídící výřez programu*, což je výřez, který obdržíme, když za množinu apriori potřebných příkazů vezmeme množinu všech testovacích příkazů. Řídící výřez, vedle toho, že jako stop-výřez zastaví pro daná data právě tehdy, když pro tato data zastaví program, je vhodný pro testování i studium řídící části programu. Může být nápomocen i např. 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 řadě dalších situací, s nimiž se střetáváme při testování a ladění, nás v prvé 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 informací obsažených v IG a GVV.

Protože elementární kriterium c=(e,V) vyjadřuje zájem o hodnoty proměnných množiny V v místě programu e, budou nutně součástí množiny potřebných příkazů p, které přiřazují hodnotu nějaké proměnné z V a jsou dosažitelné na hraně 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. Formálněji zapsáno, definujme pro elementární kriterium c množinu

AUST(c)={p: p∈REACHES(e)∧DEF(p)∩V≠∅} ∪BRANCH(w) ∪
∪ if w je testovací příkaz then {w} else ∅.

Množina apriori potřebných příkazů AUST pro zadané kriterium C je AUST(C)=∪ {AUST(c):c∈C}. Pro ilustraci se obraťme k programu na obr. 4 a zvolme kriterium C={(e,{x,y}),(c,{x})}. Pak AUST((e,{x,y}))={1,7,13}∪{12}∪{8}, AUST((c,{x}))={1,11}∪{8,12}∪∅, AUST(C)={1,7,8,11,12,13}. Příslušný výřez S je na obr. 4.

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 a 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 následujícím způsobem. 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. 4. 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 výpočetní 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 kriteria C elementární kriteria 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. 4). Označme jako 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.

5 Závěr

Všechny dosud uvedené vlastnosti výřezů otevírají široké aplikační možnosti vždy, kdy nás z nejrůznějších důvodů zajímají ty aspekty chování programu, které lze charakterizovat prostřednictvím množiny apriori potřebných příkazů resp. nějakým výřezovým kriteriem. 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.

References

[Hecht77] Hecht,M.S.: Flow Analysis of Computer Program, North Holland 1977

[Hav84] Havlát,T.: *Výřezy* programů, sborník SOFSEM 84

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

[BeŠ84] Benešovský,M.–Šmídek,M.: Testování programů. sborník SOFSEM 84