

Grundlagen der Datenflußanalyse und Programmabhängigkeitsgraphen

Prof. Dr.-Ing. G. Snelting

Universität Karlsruhe
Lehrstuhl Programmierparadigmen

————— [Wozu Datenflußanalyse?] —————

Datenflußanalyse ist eine Familie von Verfahren zur

- Codeoptimierung:

```
x := 42;  
... // x wird hier  
... // nicht undefiniert    ⇒ x := 42;  
IF x<0 THEN  
  p(a,b,c);
```

- Parallelisierung:

```
FOR i:= 1 TO 100 DO  
  a[i] := a[i]+1;  
⇒  
  PARBEGIN  
    a[1] := a[1]+1;  
    ...  
    a[100] := a[100]+1;  
  PAREND
```

- Programmverstehen:

Welche Anweisungen beeinflussen den Wert von x in Zeile 4711 ?

- Sicherheitsprüfung:

... siehe Vortrag ...

Es gibt viele verschiedene Datenflußanalysen

Beispiele:

1. *Constant Propagation*: Wert von Variablen zur Übersetzungszeit bekannt \Rightarrow Kontrollkonstrukte statisch auswerten:

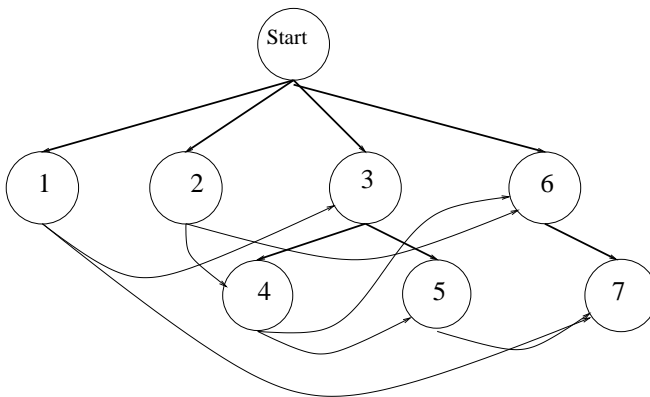
```
x := 42;  
y := 17;  
IF x<0 THEN  
  read(y);  
  x := y;  
END;  
IF y>0 THEN  
  p(x);
```

\Rightarrow

```
x := 42;  
y := 17;  
p(42);
```

2. *Programmabhängigkeitsgraph*: Kanten zwischen Definition / Verwendung einer Variablen bzw zwischen Kontrollprädikaten / kontrollierten Anweisungen

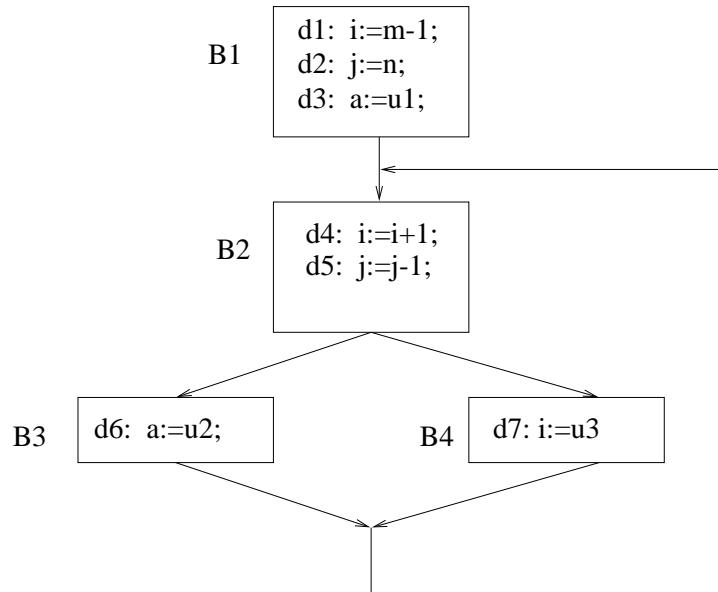
```
(1) x := 42;  
(2) y := 17;  
(3) IF x<0 THEN  
(4)   read(y);  
(5)   x := y;  
    END;  
(6) IF y>0 THEN  
(7)   p(x);
```



Ausgangspunkt der Analyse ist stets der Kontrollflußgraph (CFG):

```
i := m-1;  
j := n;  
a := u1;  
WHILE e2 DO  
  i := i+1;  
  j := j-1;  
  IF e1 THEN  
    a := u2  
  ELSE  
    i := u3  
  END;  
END;  
END;
```

⇒



Knoten des CFG: *Basic Blocks* - elementare Anweisungssequenzen ohne Kontrollflußänderung

Kante von B_i nach B_j : B_j kann unmittelbar nach B_i ausgeführt werden

CFG kann i.a. beliebige Zyklen enthalten (GOTOs!)

Datenflußanalyse orientiert sich an der Struktur des CFG

Ein bekanntes Datenflussproblem: Welche Zuweisung hat potentiellen Effekt auf eine gegebene Anweisung?

- Eine Variable x wird durch eine Zuweisung, Read-Anweisung, Ausgabe-Parameter usw. *definiert*.
- Definitionen (Zuweisungsstellen) $D = \{d_1, d_2, \dots\}$;
 $D_x \subseteq D$: Menge aller Definitionen von x
- Eine Definition $d \in D_x$ *killt* alle anderen Definitionen von x
- Eine Definition d *erreicht* einen Basic Block B , wenn sie nicht vorher gekillt wird. Dh es gibt Pfad im CFG $d \rightarrow^* B$, auf dem keine weitere Definition von x liegt

Für jeden Basic Block B werden zunächst bestimmt:

1. $gen(B)$: Menge der in B neu erzeugten Definitionen
2. $kill(B)$: Menge der in B gekillten Definitionen

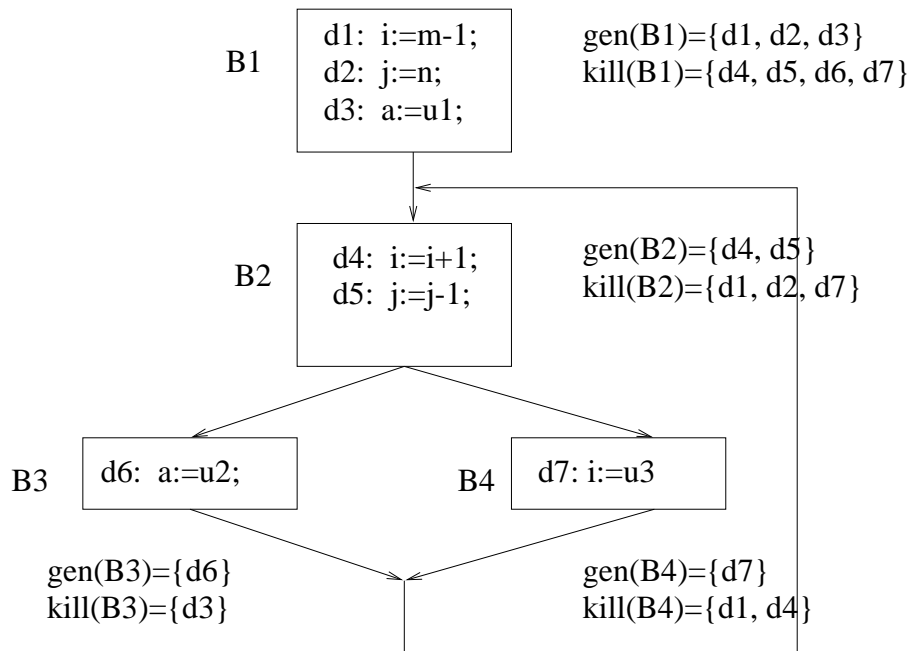
Bsp: $B \cong d : x := E$; (B enthält nur eine Zuweisung)

$$gen(B) = \{d\}, \quad kill(B) = D_x - \{d\}$$

im allgemeinen ist

$$kill(B) = \bigcup_{x \in vars(B)} D_x - (gen(B) \cap D_x)$$

im Beispielprogramm:



Der Effekt jedes Basic Blocks B wird durch *Transferfunktion* beschrieben: $f_B : 2^D \rightarrow 2^D$

$$f_B(X) = \text{gen}(B) \cup (X - \text{kill}(B))$$

„Eine Definitionsmenge X wird in B teilweise getilgt, andererseits kommen neue Definition hinzu“

Gesucht für jeden B : $\text{in}(B), \text{out}(B) \in 2^D$ mit

$$\text{out}(B) = f_B(\text{in}(B)), \quad \text{in}(B) = \bigcup_{C \in \text{pred}(B)} \text{out}(C)$$

Dies Lösung sagt über jeden BB, welche Definitionen ihn beeinflussen können

Die Transferfunktionen für sichtbare Definitionen sind nur ein Beispiel einer allgemeinen Methodik:

- Relevante Information wird in Form von *abstrakten Werten* berechnet
- abstrakte Werte werden für Ein- und Ausgang jedes Basic Blocks berechnet
- abstrakte Werte sind stets Elemente eines Verbandes $(L; \leq, \sqcap, \sqcup)$ mit endlicher Höhe

\rightsquigarrow *Extrafolien Grundlagen der Ordnungs- und Verbandstheorie*

Idee: wenn man im Verband von oben nach unten geht, wird die durch Verbandselement repräsentierte Information immer genauer. \top steht für Unwissen, \perp steht für Widerspruch (zuviel Information)

Bem. Leider ist die Literatur nicht einheitlich, und manche Verbände stehen in manchen Büchern auf dem Kopf.

Mathematisch ist das belanglos (dualer Verband)

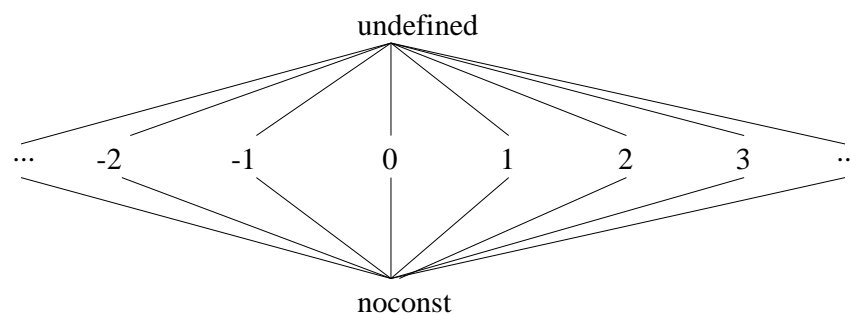
- Beispiel 1: Verband für sichtbare Definitionen:

$$L_S = (L; \leq, \sqcap, \sqcup) = (2^D; \supseteq, \cup, \cap)$$

ist endlicher Potenzmengenverband.

steht auf dem Kopf: $\emptyset = \perp$ repräsentiert Nichtwissen

- Beispiel 2: Verband für Konstantenpropagation L_P beschreibt folgende Fälle:
 1. man weiss nichts über den Wert einer Variable an einem Programmpunkt ($\top = \text{undefined}$)
 2. die Variable hat an einem bestimmten Programmpunkt genau einen bekannten Wert n
 3. die Variable hat an einem Punkt 2 oder mehr verschiedene Werte ($\perp = \text{noconst}$)



der *flache Verband*: unendlich breit, Höhe 3

formal: $L_P = \mathbb{IN} \cup \{\top, \perp\}$, $x \leq y \iff x = y \vee x = \perp$

Dieser Verband steht nicht auf dem Kopf :-)

Wieso Verbände?

- es gibt Supremum und Infimum (wichtig für zusammenlaufende CFG-Kanten)
- monotone Funktionen in Verbänden endlicher Höhe haben stets Fixpunkte
- Potenzmengenverbände können effizient durch Bitvektoren implementiert werden

Sei $(L; \leq, \sqcap, \sqcup)$ der Verband der abstrakten Werte

Zu jedem Basic Block B wird eine *Transferfunktion*

$$f_B : L \rightarrow L \in F = \{f_B \mid B \text{ basic Block} \}$$

angegeben, die die Wirkung von B beschreibt

ferner wird verlangt:

1. $id_L \in F$ (Identität \rightsquigarrow Nullanweisung)
2. $x \leq y \Rightarrow f_B(x) \leq f_B(y)$ (Monotonie \rightsquigarrow Konvergenz)
3. $f_B \circ f_{B'} \in F$ (Abgeschlossenheit)
4. $f_B \sqcap f_{B'} \in F$ (punktweises Infimum)
5. evtl. $f_B(x \sqcap y) = f_B(x) \sqcap f_B(y)$ (Distributivität)

Distributive Analysen sind genauer (s.u.)

Beispiel: sichtbare Definitionen. $L_S = (L; \leq, \sqcap, \sqcup) = (2^D; \supseteq, \cup, \cap)$, $f_B : 2^D \rightarrow 2^D$, $f_B(X) = gen(B) \cup (X - kill(B))$
 die f_B sind monoton und distributiv. Beweis:

1. $X \leq Y \Rightarrow Y \subseteq X \Rightarrow f_B(Y) = gen(B) \cup (Y - kill(B)) \subseteq gen(B) \cup (X - kill(B)) = f_B(X) \Rightarrow f_B(X) \leq f_B(Y)$
2. $f_B(X \sqcap Y) = f_B(X \cup Y) = gen(B) \cup ((X \cup Y) - kill(B)) = (gen(B) \cup (X - kill(B))) \cup (gen(B) \cup (Y - kill(B))) = f_B(X) \sqcap f_B(Y)$

— [Aufstellen und Lösen der Datenflußgleichungen] —

- gesucht: stabile Werte $in(B), out(B) \in L$ am Anfang und Ende jedes Basic Blocks B mit

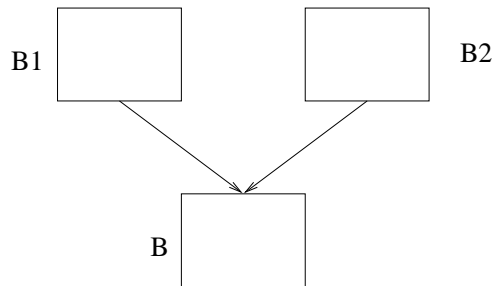
$$out(B) = f_B(in(B))$$

Dies definiert ein Gleichungssystem!

- Hintereinanderausführung von Basic Blocks entspricht Komposition der Transferfunktionen:

$$f_{B;B'}(D) = f_{B'}(f_B(D)) \quad in(B') = out(B)$$

- falls zwei Pfade im CFG zusammenlaufen, muss das Infimum ihrer Transferfunktionen berechnet werden:



$$f_{(B1||B2)}(X) = (f_{B1} \sqcap f_{B2})(X) = f_{B1}(X) \sqcap f_{B2}(X)$$

Im speziellen Fall „sichtbare Definitionen“ müssen die entsprechenden Definitionsmengen $\in 2^D$ vereinigt werden:

$$in(B) = out(B1) \cup out(B2) = f_{B1}(in(B1)) \cup f_{B2}(in(B2))$$

Beispiel: Sichtbare Definitionen für das Programm S. 4.
 Transferfunktion von S. 6.

⇒ Gleichungssystem für alle Werte $in(B_i), out(B_i)$

$$in(B1) = \emptyset$$

$$\begin{aligned} in(B2) &= out(B1) \cup out(B3) \cup out(B4) \\ &= f_{B1}(in(B1)) \cup f_{B3}(in(B3)) \cup f_{B4}(in(B4)) \end{aligned}$$

$$in(B3) = in(B4) = out(B2) = f_{B2}(in(B2))$$

Lösungsverfahren: *Fixpunktiteration*. Init: $in(B_i) = \emptyset$

Mengen werden als Bitvektoren dargestellt

Beispieleinträge:

$$\begin{aligned} in_1(B2) &= out_0(B1) \cup out_0(B3) \cup out_0(B4) = \\ &= 1110000 \vee 0000010 \vee 0000001 = 1110011; \end{aligned}$$

$$\begin{aligned} out_1(B2) &= gen(B2) \cup (in_1(B2) - kill(B2)) = \\ &= 0001100 \vee (1110011 \wedge \overline{1100001}) = 0011110 \end{aligned}$$

Block B	$in_0(B)$	$out_0(B)$	$in_1(B)$	$out_1(B)$	$in_2(B)$	$out_2(B)$
$B1$	0000000	1110000	0000000	1110000	0000000	1110000
$B2$	0000000	0001100	1110011	0011110	1111111	0011110
$B3$	0000000	0000010	0001100	0001110	0011110	0001110
$B4$	0000000	0000001	0001100	0010111	0011110	0010111

⇒ am Anfang von $B4$ sind $d3, d4, d5, d6$ sichtbar,
 insbesondere die beiden Zuweisungen an a !

Jeder B_i ist aus elementaren Zuweisungen zusammengesetzt. Es ist

$$L = \{(x, c) \mid x \in Vars(P), c \in L_C\}$$

Transferfunktion für Zuweisung $x := E$: $f_B : 2^L \rightarrow 2^L$

$$Z = \{(v_1, c_1), (v_2, c_2), \dots\}$$

$$x := E$$

$$f_B(Z) = Z \setminus \{(x, c_x)\} \cup \{(x, eval(E))\}$$

mit

$$eval(E) = \begin{cases} value(E) & \forall v \in Vars(E) : (v, c_v) \in Z \\ & \wedge c_v \notin \{\top, \perp\} \\ \top & \exists v \in Vars(E) : (v, \top) \in Z \\ \perp & \exists v \in Vars(E) : (v, \perp) \in Z \end{cases}$$

wobei $\top = \textit{undefined}$, $\perp = \textit{noconst}$.

\perp bedeutet: die Variable ist garantiert keine Konstante

Bem. $(x, eval(E))$ entspricht $gen(B)$, (x, c_x) entspricht $kill(B)$

f_B ist monoton, aber nicht distributiv! (Übung)

Betrachtet man im Beispiel S. 4 $m, n, u1, u2, u3$ als Konstanten, i, j, a als Variablen, so ergibt sich das Gleichungssystem

$$in(B1) = \{(i, \top), (j, \top), (a, \top)\}$$

$$in(B2) = out(B1) \sqcap out(B2) \sqcap out(B3)$$

$$= f_{B1}(in(B1)) \sqcap f_{B3}(in(B3)) \sqcap f_{B4}(in(B4))$$

$$in(B3) = in(B4) = out(B2) = f_{B2}(in(B2))$$

mit

$$X \sqcap Y = \{(v, r) \mid (v, a) \in X, (v, b) \in Y, r = a \sqcap b \in L_C\}$$

$$\cup \{(v, a) \in X \mid (v, b) \notin Y\} \cup \{(v, b) \in Y \mid (v, a) \notin X\}$$

$$\text{Bsp: } \{(i, 42), (j, \top)\} \sqcap \{(j, 17), (i, 13), (a, 1)\} = \\ \{(i, \perp), (j, 17), (a, 1)\}$$

Nun führen wir die Fixpunktiteration durch (Übung!)

Ergebnis u.a.:

$$in(B2) = \{(i, (m - 1) \sqcap (m - 1 + 1) \sqcap u3), (j, n \sqcap (n + 1)), (a, u1 \sqcap u2)\} = \{(i, \perp), (j, \perp), (a, \perp)\}$$

\implies Am Anfang von $B2$ sind die Variablen i, j, a garantiert keine Konstanten, sondern haben zur Laufzeit mindestens zwei verschiedene Werte

Falls IF/WHILE-Ausdrücke konstant werden, so verschwinden Pfade, dh *das Gleichungssystem ändert sich!* Man muss dann die Iteration mit dem neuen System wiederholen

———— [Korrektheit der Fixpunktiteration] ————

Satz. (Kam/Ullman 77) Seien P_1, P_2, \dots alle Pfade von *Start* zu Programmpunkt s ; seien die Transferfunktionen $f_B \in F$ distributiv; sei $fix(s) \in L$ der durch Fixpunktiteration für $out(s)$ berechnete Wert. Dann ist

$$fix(s) = \bigsqcap_i f_{P_i}(\top)$$

Dies ist deswegen bemerkenswert, weil es i.a. unendlich viele Pfade gibt.

Falls f nicht distributiv, gilt nur \geq statt $=$, d.h. der Fixpunkt stellt eine konservative Approximation dar.

Für manche nicht-distributive Probleme ist die exakte Lösung nicht berechenbar zB Konstantenpropagation

Es gibt Verallgemeinerungen des Satzes für interprozedurale Analyse (Knoop et al 95)

———— [Programmabhängigkeitsgraph (PDG)] ————

x, y seien Anweisungen bzw Basic Blocks; $gen(x), in(x)$ wie oben, $var(d)$ sei die in einer Definition definierte Variable; $uses(x)$ seien die in x benutzten Variablen.

Def. (Datenabhängigkeit)

$$x \rightarrow y \iff \exists d \in gen(x) : d \in in(y) \wedge var(d) \in uses(y)$$

Def. (Dominanz) x dominiert y , wenn jeder CFG-Pfad von *Start* zu y über x führt.

Def. (Postdominanz) y postdominiert x , wenn jeder CFG-Pfad von x zu *Stop* über y führt.

Def. (Kontrollabhängigkeit)

$$x \rightarrow y \iff \exists \text{ Pfad } P \text{ von } x \text{ nach } y \text{ im CFG } \forall z \in P, z \neq x, z \neq y : y \text{ postdominiert } z \wedge y \text{ postdominiert nicht } x$$

Bem. In strukturierten Programmen sind kontrollabhängige Anweisungen Y gerade jene im Rumpf eines `if`, `while` usw; sie hängen von der regierenden Bedingung x ab

Def. Der PDG (S, \rightarrow) hat alle Anweisungen als Knoten und Daten-/Kontrollabhängigkeiten als Kanten.

Beispiel: siehe S. 3

Bem. Falls es Prozeduren, Adressen (Aliasing) oder komplexe Datenstrukturen gibt, wird der PDG wesentlich komplizierter

Beispiel: Programm für elektronische Waage (↔ PTB)

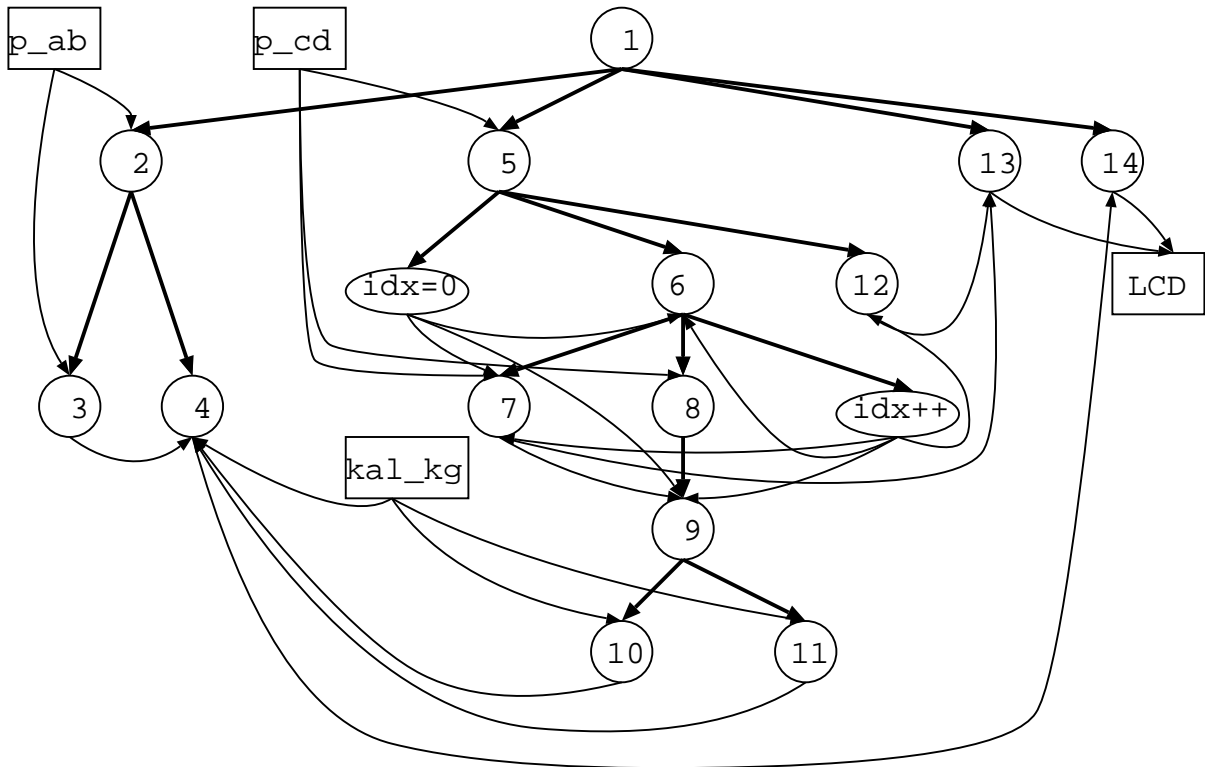
```
void main() {
    int p_ab[2] = {0, 1};
    int p_cd[1] = {0};
    char e_puf[8];
    int u; int idx;
    float u_kg; float kal_kg = 1.0;

(1) while(TRUE) {
(2)     if ((p_ab[CTRL2] & 0x10)==0) {
(3)         u = ((p_ab[PB] & 0x0f) << 8) + (unsigned int)p_ab[PA];
(4)         u_kg = u * kal_kg;
            }
(5)     if ((p_cd[CTRL2] & 0x01) != 0) {
(6)         for (idx=0;idx<7;idx++) {
(7)             e_puf[idx] = p_cd[PA];
(8)             if ((p_cd[CTRL2] & 0x10) != 0) {
(9)                 switch(e_puf[idx]) {
(10)                    case '+': kal_kg *= 1.01; break; /* unerlaubter */
(11)                    case '-': kal_kg *= 0.99; break; /* Datenfluss */
                        }
                }
            }
(12)     e_puf[idx] = '\0';
            }
(13)     printf("Artikel: %07.7s\n",e_puf);
(14)     printf("    %6.2f kg    ",u_kg);
            }
}
```

p_cd: Keyboard-Eingaberegister; Kontrollbits

p_ab: Messwert-Eingaberegister

zugehöriger PDG:



fett: Kontrollabhängigkeiten

dünn: Datenabhängigkeiten

Welche Anweisungen y können Programmpunkt x beeinflussen, und welche tun dies garantiert nicht?

Def. Der Rückwärtsslice $BS(x)$ enthält alle Anweisungen, die x beeinflussen können.

im PDG ist

$$BS(x) = \{y \mid y \rightarrow^* x\}$$

Bem. $BS(x)$ darf zu groß sein, aber niemals zu klein (Prinzip der konservativen Approximation). In der Praxis will man natürlich möglichst kleine $BS(x)$

Beispiel: für obiges Programm ist $(5) \in BS((14))$.

Deshalb potentielle Beeinflussung des angezeigten Messwertes durch das Keyboard.

Def. Vorwärtsslice $FS(x) = \{y \mid x \rightarrow^* y\}$

Chop $CH(x, y) = \{z \mid x \rightarrow^* z \rightarrow^* y\} = BS(y) \cap FS(x)$

Bem. Diese einfachen Formeln gelten nicht mehr im interprozeduralen Fall sowie für komplexe Datenstrukturen

Datenflußanalyse ist ein weites Feld, in das enorme Forschungsanstrengungen geflossen sind und fließen

Erwähnt seien

- interprozedurale Analyse
- Analyse von Arrays
- Analyse von Pointern
- effiziente Implementierungstechniken
- inkrementelle / bedarfsgetriebene Datenflußanalyse
- automatische Parallelisierung
- Program Slicing
- Anwendungen in Wartung, Reengineering, Sicherheitsanalyse, ...

Datenflußanalyse = abstrakte Interpretation + Model Checking

(B. Steffen)

Literatur: Nielson/Nielson/Hankin: Principles of Program Analysis. Springer 1999