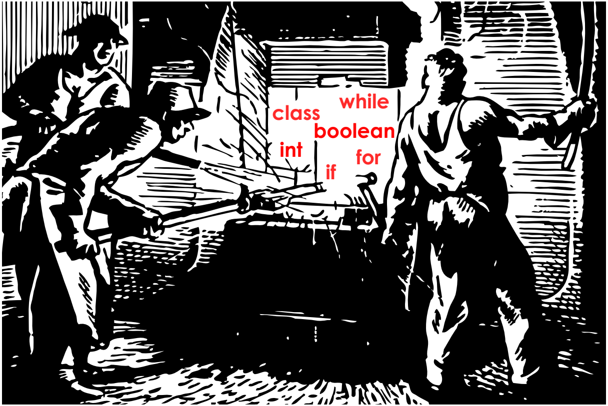
Compiler 2

2. Block: Datenflußanalyse



Organisatorisches

Organisatorisches

TECHNISCHE UNIVERSITÄT DARMSTADT

Ab jetzt auszugsweise Material aus

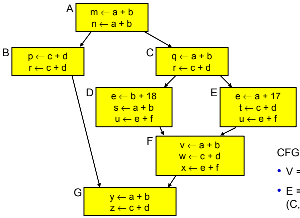
Advanced Compiler Design and Implementation

von Steven S. Muchnick, erschienen 1997 bei Morgan-Kaufman

Relationen im CFG

Relationen im CFG

TECHNISCHE UNIVERSITÄT DARMSTADT



CFG G = (V,E)

- V = {A,B,C,D,E,F,G}
- E = {(A,B),(A,C),(B,G),(C,D), (C,E),(D,F),(E,F),(F,E)}
- |V| = 7, |E| = 8

- Anhand des CFGs lassen sich nun Aussagen über Beziehungen zwischen Blöcken treffen
- Eine sehr wichtige:

"Welche Blöcke *x* werden vor einem bestimmten Block *y* in jedem Fall ausgeführt?"

- Fachbegriff: Welche Blöcke *x* dominieren den Block *y*?

Dominanz 1

Dominanz

x dominiert y genau dann, falls jeder Pfad vom Eingangsknoten des CFGs zum Knoten y den Knoten x enthält. Geschrieben als $x \gg y$.

Gilt immer: $x \gg x$

Strikte Dominanz

x dominiert y strikt, falls $x \gg y$ und $x \neq y$.
Geschrieben als $x \underline{\gg} y$.

Dominanz 2

Dominatoren

$\textsc{Dom}(y) = \{x \in \text{CFG} | x \gg y\}$ ist die Menge der Dominatoren von y.

Unmittelbarer Dominator (*immediate dominator*)

$x = \textsc{IDom}(y)$ ist der im CFG y am nächsten gelegene Dominator x aus $\textsc{Dom}(y)$ mit $x \neq y$.

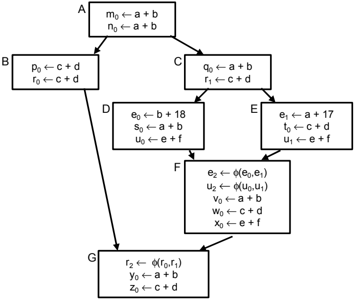
Dominanz 3

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sehr nützliche Relation
  - Finden von Schleifen
  - Zielauswahl für Code-Bewegung
  - Umwandlung in SSA-Form

Dominatormengen

Block	Dom	IDom
A	A	–
B	A,B	A
C	A,C	A
D	A,C, D	C
E	A,C, E	C
F	A,C,F	C
G	A,G	A



➥Berechnung der Relation: Kommt noch . . .

1. Beispiel: Copy Propagation

Copy Propagation (CP)

- Viele Algorithmen legen Zwischenvariablen an
  - `a := x + y; t1 := a;`
- Zwischenvariablen
  - benötigen viel Speicher, viele Register
  - verursachen viele Kopieranweisungen `y := x`
  - sind in vielen Fällen unnötig

➥Beseitigen durch

1. Copy Propagation ($\rightarrow$ Muchnick 12.5)
2. Dead Code Elimination

Idee Copy Propagation (CP)

- Versuche zur Compile-Zeit Aussagen über Laufzeitverhalten zu machen
  - “Simulation” des Programmes
- Falls möglich, benutze immer Originalvariable statt Kopie
  - Eingabe: `a:=x+y; b:=x+y;`
  - Nach SSA/Red.Elim./AST: `a:=x+y; t1:=a; b:=t1;`
  - Nach CP: `a:=x+y; t1:=a; b:=a;`
  - Nach Dead Code-Elimination: `a:=x+y; b:=a;`
- Vorgehen
  - Stelle fest, wenn Originalvariablen zwischen ihrer Berechnung …
  - …und ihrer Verwendung nicht überschrieben werden

Lokale und globale Phasen

Datenstrukturen für Lokale CP 1

Speichere: Zuordnung von Originalvariablen w an Kopien v für eine Zuweisung $v := w$

Tupel (v, w)

- Zielvariable v
- Originalvariable w

ACP (*available copies*)

Die Menge der verfügbaren Kopieranweisungen ACP sind all die (v, w), bei denen weder v noch w zwischen Definition und der betrachteten Stelle des Programmes überschrieben wurden.

Datenstrukturen für Lokale CP 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Realisierung von ACP bestimmt Gesamtlaufzeit des Verfahrens abhängig von Anzahl der Kopieranweisungen n.

- Lineare Suche: $O(n^2)$
- Baumstruktur: $O(n \log n)$
- Hash: $O(n)$

Algorithmus für Lokale CP 1

Hilfsfunktion: Liefere zu verwendenden Operand für `opnd`, ggf. ausgetauscht durch eine in ACP vorhandene Originalvariable

```
func Copy_Value(opnd, ACP) : Var
  Operand            opnd;
  Set<Pair<Var,Var>> ACP;  // Menge der (v,w)
begin
  Pair<Var,Var> acp;       // ein (v,w)
  foreach acp in ACP do
    if opnd.kind == VARIABLE && opnd.name == acp.first() then
      return acp.second(); // gefunden, verwende Originalvar.
    endif
  endfor
  return opnd.name; // Ziel nicht gefunden, alter Opnd. zurück
end
```

Algorithmus für Lokale CP 2

Hilfsfunktion: Entferne eine überschriebene Variable `v` aus ACP

```
proc Remove_ACP(ACP, varname)
  var Set<Pair<Var,Var>> ACP;
      Var varname;
begin
  Set<Pair<Var,Var>> temp = ACP.copy(); // Löschen bei Iterat.
  Pair<Var,Var> acp;                    // Paar (v,w)
  forach acp in temp do
    if acp.first() == varname || acp.second() == varname then
      ACP.remove(acp);
    endif
  endfor
end
```

Algorithmus für Lokale CP 3

Gelesenes Auftreten von Variablen

```
proc Local_Copy_Prop(b)
  Block b;
begin
  Set<Pair<Var,Var>> ACP = Set.empty();
  Instruction i;
  foreach i in b.instructions() do
    if (i instanceof Expression) then // verwendendes Auftreten
      if (i == "a + b") then // Bin.Exp.
        i.opnds.a.name := Copy_Value(i.opnds.a.name, ACP);
        i.opnds.b.name := Copy_Value(i.opnds.b.name, ACP);
      else if (i == "-a") then // Un.Exp.
        i.opnds.a.name := Copy_Value(i.opnds.a.name, ACP);
      else if (i == "f(a)") then // List.Exp.
        i.opnds.a.name := Copy_Value(i.opnds.a.name, ACP);
      else if ... // andere lesende Instruktionsarten
      endif
    else
       ... // schreibendes Auftreten
    endif
  endfor
end
```

Algorithmus für Lokale CP 4

Geschriebenes Auftreten von Variablen

```
proc Local_Copy_Prop(b)
  Block b;
begin
  Set<Pair<Var,Var>> ACP = Set.empty();
  Instruction i;
  foreach i in b.instructions() do
    if (i instanceof Expression) then // verwendendes Auftreten
      ...
    else if (i == "LHS := RHS") then // Zuweisung
      // Zuweisungen zerstören bestehende Kopien, Ausnahme:
      // triviale Kopien a:=a zerstören keine (*,a) (a,*)
      if (LHS != RHS)
        Remove_ACP(ACP, i.LHS.name); // entferne überschr. Var.
      endif
      if (RHS instanceof Var && LHS != RHS) then // Kopie?
        ACP.add(new Pair(LHS, RHS));
      endif
    endif
  endfor
end
```

Beispiel Lokale CP

Position	Code Before	ACP	Code After
		∅	
1	b ← a		b ← a
		{⟨b,a⟩}	
2	c ← b + 1		c ← a + 1
		{⟨b,a⟩}	
3	d ← b		d ← a
		{⟨b,a⟩,⟨d,a⟩}	
4	b ← d + c		b ← a + c
		{⟨d,a⟩}	
5	b ← d		b ← a
		{⟨d,a⟩,⟨b,a⟩}	

Muchnick, Fig 12.25

Ansatz Globale CP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Basiert auf Datenflussanalyse
  - Welche Kopieranweisungen erreichen Verwendungen ihrer LHS intakt?
  - Intakt: Weder LHS noch RHS überschrieben!
- Erweiterte Darstellung (v, w, b, p)
  - b ist Block der Zuweisung $v := w$
  - p ist Position der Zuweisung $v := w$ innerhalb des Blockes b (z.B. Nummer der Anweisung)

Hilfsmengen für globale CP

TECHNISCHE UNIVERSITÄT DARMSTADT

COPY(b)

Menge der (v, w, b, p), bei denen bei einer Kopieranweisung $v := w$ im Block b weder v noch w vor Ende des Blockes Ziel einer Zuweisung sind.

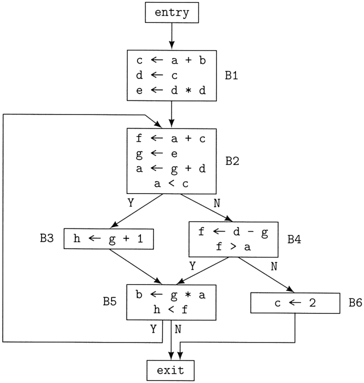
KILL(b)

Menge der (t, u, d, q) mit $d \neq b$, bei denen t und/oder u in Block d Ziel einer Zuweisung sind.

Beispiel: COPY und KILL

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiel



Mengen

$$
\begin{aligned}
COPY(\texttt{entry}) &= \emptyset \\
COPY(\texttt{B1}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle\} \\
COPY(\texttt{B2}) &= \{\langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\} \\
COPY(\texttt{B3}) &= \emptyset \\
COPY(\texttt{B4}) &= \emptyset \\
COPY(\texttt{B5}) &= \emptyset \\
COPY(\texttt{B6}) &= \emptyset \\
COPY(\texttt{exit}) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
KILL(\texttt{entry}) &= \emptyset \\
KILL(\texttt{B1}) &= \{\langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\} \\
KILL(\texttt{B2}) &= \emptyset \\
KILL(\texttt{B3}) &= \emptyset \\
KILL(\texttt{B4}) &= \emptyset \\
KILL(\texttt{B5}) &= \emptyset \\
KILL(\texttt{B6}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle\} \\
KILL(\texttt{exit}) &= \emptyset
\end{aligned}
$$

Quelle: Muchnick, pp. 359-360

Datenflußgleichungen für globale CP 1

TECHNISCHE UNIVERSITÄT DARMSTADT

CPIN(b)

Menge von Kopieranweisungen (t, u, d, q), die zu Beginn des Blocks b intakt sind.

CPOUT(b)

Menge von Kopieranweisungen (t, u, d, q), die am Ende eines Blocks b intakt sind.

Datenflußgleichungen für globale CP 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorgehensweise bei Aufstellen der Gleichungen

- Nur solche Kopieranweisungen sind am Anfang eines Blockes verfügbar ...
- ... die an allen Enden von Vorgängern verfügbar waren
- Startwerte für iterative Lösung
  - CPIN(entry): Startblock hat keine Kopieranweisungen zur Verfügung
  - CPIN(b), $b \neq$ entry: Alle anderen Blöcke haben alle in der ganzen Prozedur auftretenden Kopieranweisungen zur Verfügung
    - Wird schrittweise eingeschränkt

Datenflußgleichungen für globale CP 2

$$
\begin{aligned}
\mathrm{CP}\textsc{in}(b) &= \bigcap_{d \in \mathrm{pred}(b)} \mathrm{CP}\textsc{out}(d) \\
\mathrm{CP}\textsc{out}(b) &= \textsc{Copy}(b) \cup (\mathrm{CP}\textsc{in}(b) - \textsc{Kill}(b))
\end{aligned}
$$

mit Initialisierung

$$
\begin{aligned}
\mathrm{CP}\textsc{in}(\mathrm{entry}) &= \emptyset \\
\mathrm{CP}\textsc{in}(b) &= \bigcup_{d \in \mathrm{Blocks}} \textsc{Copy}(d) \text{ ,für } b \neq \mathrm{entry}
\end{aligned}
$$

Beispiel: Initialisierung

$$
\begin{aligned}
\textsc{Copy}(\text{entry}) &= \emptyset \\
\textsc{Copy}(\text{B1}) &= \{\langle d, c, B1, 2\rangle\} \\
\textsc{Copy}(\text{B2}) &= \{\langle g, e, B2, 2\rangle\} \\
\textsc{Copy}(\text{B3}) &= \emptyset \\
\textsc{Copy}(\text{B4}) &= \emptyset \\
\textsc{Copy}(\text{B5}) &= \emptyset \\
\textsc{Copy}(\text{B6}) &= \emptyset \\
\textsc{Copy}(\text{exit}) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\text{CP}\textsc{in}(\text{entry}) &= \emptyset \\
\text{CP}\textsc{in}(\text{B1}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\text{CP}\textsc{in}(\text{B2}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\text{CP}\textsc{in}(\text{B3}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\text{CP}\textsc{in}(\text{B4}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\text{CP}\textsc{in}(\text{B5}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\text{CP}\textsc{in}(\text{B6}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\text{CP}\textsc{in}(\text{exit}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\}
\end{aligned}
$$

Einfaches iteratives Lösungsverfahren

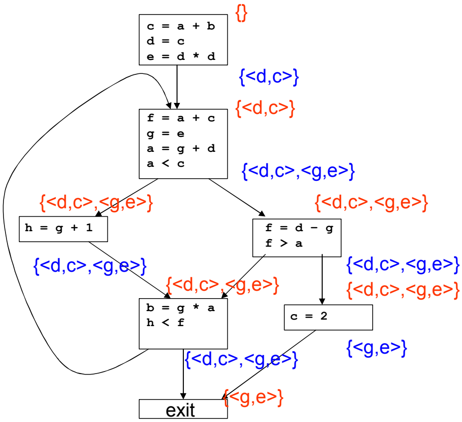
TECHNISCHE UNIVERSITÄT DARMSTADT

```
worklist := cfg.getBlocks();

while ( worklist.notEmpty() ) {
  b := worklist.removeFirst();
  recompute CPin(b);
  recompute CPout(b);
  if (CPout(b) changed)
    worklist.add(b.getSuccessors());
}
```

Beispiel: Ergebnis der iterativen Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

Rot: CPIN, Blau: CPOUT



Quelle: White, GMU CS640, Scalar Optimization

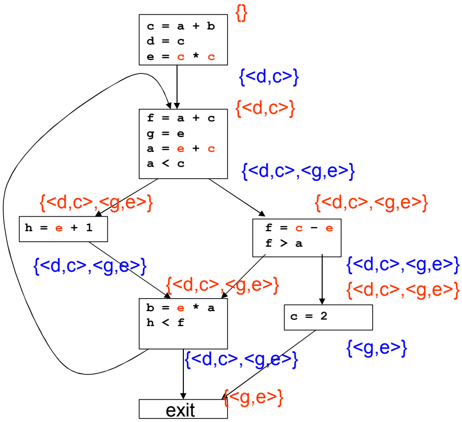
Weitere Vorgehensweise bei der globalen CP

- Werte berechnete Daten nun pro Block aus
- Vorgehen: `Local_Copy_Prop` beginnt nun nicht mehr mit leerer ACP-Menge
- ... sondern: Initialisiere ACP-Menge für Block b aus $\text{CP}_{\text{IN}}(b)$
- Analog zu: VN auf EBB und Region (DVNT), $\rightarrow$ 7. Block

Beispiel: Ergebnis der globalen CP

TECHNISCHE UNIVERSITÄT DARMSTADT

Rot: CPIN, Blau: CPOUT



Quelle: White, GMU CS640, Scalar Optimization

2. Beispiel: Konstanten propagieren

(Zu) Einfaches Verfahren

Konstanten propagieren 1

- *Constant Propagation*
- Weiterführung von Constant Folding
- Nun hinweg über Anweisungsgrenzen und Merge Points

- Beispiel für andere Formulierung von DF-Problemen
  - Außerhalb des üblichen GEN/KILL Schemas

So nicht benutzen!

- Gibt mächtigere Formulierungen
- Die dann auch wieder in Standardschema passen

Konstanten propagieren 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Darstellung durch Paare (x, v)
  - x ist Variable
  - v ist ein Wert, entweder
    - …ein einzelner bekannter Wert c (Konstante passend zum Datentyp von x)
    - …oder $\perp$ (steht hier für das Auftreten mehrerer verschiedener Werte)
- $\textsc{Const}(b)$ sind alle bisher gesammelten Aussagen zu Beginn des Blocks b
- Damit darstellbar:
  - x ist unbekannt: $(x, v) \notin \textsc{Const}(b)$
  - x ist konstant mit Wert c: $(x, c) \in \textsc{Const}(b)$
  - x ist variabel: $(x, \perp) \in \textsc{Const}(b)$

Konstanten propagieren 3

TECHNISCHE UNIVERSITÄT DARMSTADT

- Initialisieren
  - $\text{CONST}(entry) = \{(p_1, \perp), (p_2, \perp), \ldots\}$ für alle Funktions/Prozedur-Parameter p_n
  - $\text{CONST}(b) = \emptyset$ für $b \neq entry$
- Dann in Reihenfolge Anweisungen in jedem Block b untersuchen

Für `x := y`

$$
\begin{array}{l}
\text{ADD} := \emptyset \\
\text{DEL} := \emptyset \\
\textbf{if } (\mathrm{x}, v_1) \in \text{CONST}(b) \textbf{ then } \text{DEL} := \{(x, v_1)\} \\
\textbf{if } (\mathrm{y}, v_2) \in \text{CONST}(b) \textbf{ then } \text{ADD} := \{(x, v_2)\} \\
\text{CONST}(b) := (\text{CONST}(b) \setminus \text{DEL}) \cup \text{ADD}
\end{array}
$$

Konstanten propagieren 4

TECHNISCHE UNIVERSITÄT DARMSTADT

Für `x := y op z`, analog auch `x := y op` *Const.*

$\textsc{Add} := \emptyset$
$\textsc{Del} := \emptyset$
if $(x,v_1) \in \textsc{Const}(b)$ **then** $\textsc{Del} := \{(x, v_1)\}$
if $(y,v_2) \in \textsc{Const}(b) \wedge (z,v_3) \in \textsc{Const}(b)$ **then**
$\quad\textsc{Add} := \{(x, v_2 + v_3)\}$
else // *1. Pessimismus*
$\quad\textsc{Add} := \{(x, \perp)\}$
$\textsc{Const}(b) := (\textsc{Const}(b) \setminus \textsc{Del}) \cup \textsc{Add}$

- Mit $\perp$ op x = x op $\perp$ = $\perp$ (2. Pessimismus)
- Gegen 2. Pessimismus: Hier auch Sonderregeln möglich
  - $c \cdot 0 = 0,\ c - c = 0,\ c \cdot 1 = c, \ldots$
- Transformation von $\textsc{Const}(b)$ in Block b: $\textsc{Const}_{out}(b) = F_b(\textsc{Const}_{in}(b))$

Konstanten propagieren 5

Bei Überschreiten von Blockgrenzen:
Mehrere Aussagen zu *x* aus verschiedenen Vorgängern *d* treffen zusammen

Konfluenzoperator: $\wedge$ (*meets*, Infimum, Durchschnitt) über alle Vorgängermengen

$\textsc{Const}_{in}(b) := \emptyset$
for each $x \in \bigcup_{d \in \text{pred}(b)} \textsc{Const}_{out}(d)$ **do**

  if $\exists_c \forall_{d \in \text{pred}(b)} (x, c) \in \textsc{Const}_{out}(d)$ **then**
    $\textsc{Const}_{in}(b) := \textsc{Const}_{in}(b) \cup \{(x, c)\}$
  else // *Pessimismus*
    $\textsc{Const}_{in}(b) := \textsc{Const}_{in}(b) \cup \{(x, \perp)\}$

Konstanten propagieren 6

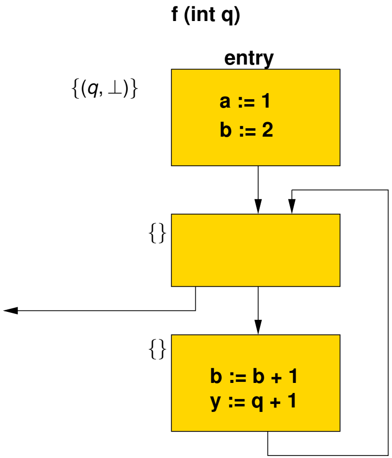
TECHNISCHE UNIVERSITÄT DARMSTADT

Damit vorwärtsgerichtetes Datenflußproblem formulierbar:

$$\textsc{Const}(b) = \bigwedge_{d \in \text{preds}(b)} F_d(\textsc{Const}(d))$$

Konstanten propagieren 7

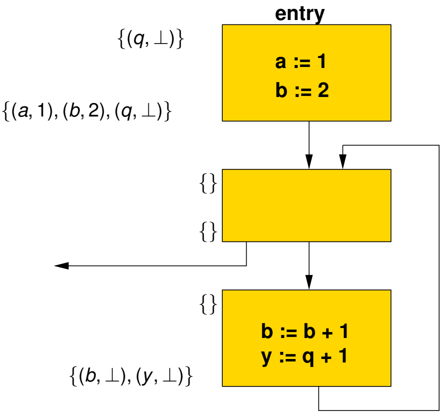
Beispiel: Initialisierung



Konstanten propagieren 8

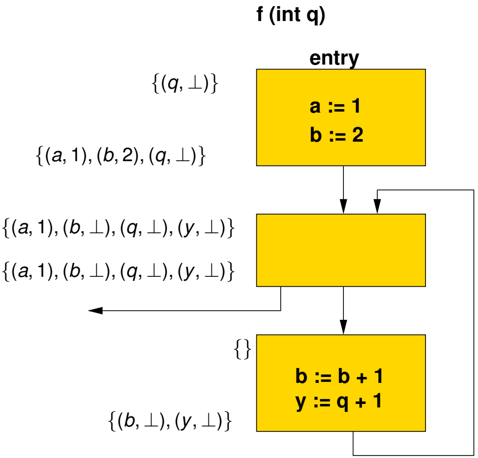
Beispiel: Schritt 1

f (int q)



Konstanten propagieren 9

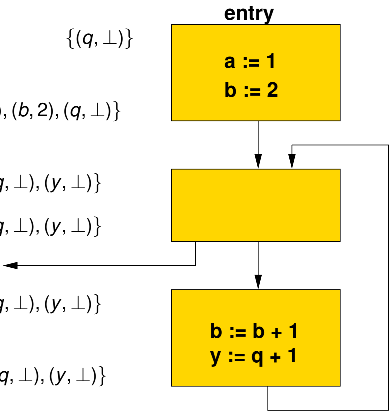
Beispiel: Schritt 2



Konstanten propagieren 10

Beispiel: Fixpunkt

f (int q)



$\{(q, \perp)\}$

$\{(a, 1), (b, 2), (q, \perp)\}$

$\{(a, 1), (b, \perp), (q, \perp), (y, \perp)\}$

$\{(a, 1), (b, \perp), (q, \perp), (y, \perp)\}$

$\{(a, 1), (b, \perp), (q, \perp), (y, \perp)\}$

$\{(a, 1), (b, \perp), (q, \perp), (y, \perp)\}$

Konstanten propagieren 11

- CONST(b) kann groß werden, ist aber endlich
- Relevanz
  - Hier nur Beispiel für ungewöhnlicheres DF-Problem
- Besser: Sparse Conditional Constant Propagation
  - Ignoriert Einfluß nicht-ausführbarer Blöcke

Iterative Datenflussanalyse

Iterative Datenflussanalyse

TECHNISCHE UNIVERSITÄT DARMSTADT

- Treffe Aussagen
  - ... über Laufzeitverhalten von Programm
  - ... zur Compile-Zeit
- Mittel der Wahl
  - Gleichungssysteme
  - Lösungsverfahren: Hier iterative, gibt aber auch andere
- Anwendung
  - Finde Anwendungsstellen von Optimierungen
  - Beweise, das Anwendung sicher ist

Weiteres Beispiel: *Live Variables*

Live Variables

Live Variables - Definition

TECHNISCHE UNIVERSITÄT DARMSTADT

Live Variable

Eine Variable v ist *lebendig* (*live*) an einer Stelle p im Programm genau dann, wenn es im CFG mindestens einen Pfad von p zu einer Verwendung von v gibt, auf dem v *nicht* definiert wird.

Live Variables - Anwendung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nur live Variables müssen in Prozessorregistern gehalten werden
- Können bei der SSA-Konstruktion zur Eliminierung von Phi-Funktionen dienen
- Können zur Erkennung von uninitialisierten Variablen dienen
  - Lokale Variable ist live bei Prozedureintritt
- Können Basis direkter Optimierungen sein
  - Store-Anweisungen nur für live Variables, überflüssig für andere

Live Variables - Berechnung 1

$\text{LiveOut}(b)$

Menge aller Variablen, die bei Austritt aus Block *b* live sind.

Damit Berechnung durch Gleichungssystem.

1. Teil
$\text{LiveOut}(b_n) = \emptyset$, mit b_n Endknoten des CFG

Bei Prozedurende sind alle (lokalen) Variablen nicht mehr live.

- Beschränkung auf Prozedurebene
- Bei uns vereinfacht: Parameter nicht betrachtet

Live Variables - Berechnung 2

2. Teil: Rekursive Definition für innere Knoten

$$\textsc{LiveOut}(b) = \bigcup_{m \in \mathrm{succ}(b)} \textsc{UEVar}(m) \cup (\textsc{LiveOut}(m) \cap \overline{\textsc{VarKill}(m)})$$

- Rechnet rückwärts von Nachfolger zu Vorgängerknoten
- $\textsc{UEVar}(m)$ (*upwards exposed*): Vor ihrer Definition in Block m benutzte Variablen
- $\textsc{VarKill}(m)$ sind alle im Block m definierten Variablen

Live Variables - Interpretation

$\textsc{LiveOut}(b) =$

$$\bigcup_{m \in \mathrm{succ}(b)} \textsc{UEVar}(m) \cup (\textsc{LiveOut}(m) \cap \overline{\textsc{VarKill}(m)})$$

- $\textsc{LiveOut}(m)$ sind alle Variablen, die live am Anfang von Nachfolgerblöcken von *m* sind
- Variable muss nur auf einem Pfad live sein ($\rightarrow \bigcup$)
- Jeder Nachfolgerknoten *m* trägt Variablen bei
  - In *m* benutzte Variablen, die vorher nicht redefiniert werden ($\textsc{UEVar}(m)$)
  - Variablen die
    - *m* selbst live verlassen ($\textsc{LiveOut}(m)$)
    - ...und in *m* nicht redefiniert werden ($\textsc{VarKill}(m)$)

Live Variables - Vorgehen

1. CFG aufbauen
   - Falls nötig um einen eindeutigen Endknoten anreichern
2. Per-Block Daten vorberechnen (UEVAR und VARKILL)
3. Iterativen Fixpunkt-Algorithmus für LIVEOUT anwenden

Live Variables - Vorberechnung für Block *b*

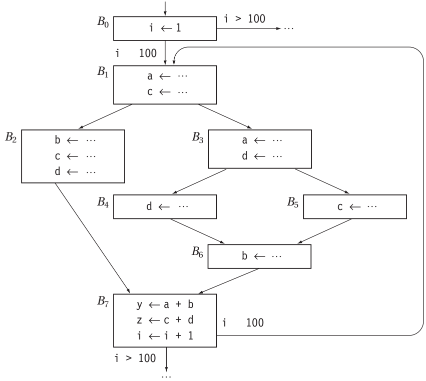
$\text{UEV}\textsc{ar}(b) := \emptyset$
$\text{V}\textsc{ar}\text{K}\textsc{ill}(b) := \emptyset$

for i := 1 **to** number of operations in block b **do**
  parse operation i into "*LHS* := *RHS*"
  for $v \in$ variables referenced in *RHS* **do**
    if $v \notin \text{V}\textsc{ar}\text{K}\textsc{ill}(b)$ **then**
      $\text{UEV}\textsc{ar}(b) := \text{UEV}\textsc{ar}(b) \cup \{v\}$
  $\text{V}\textsc{ar}\text{K}\textsc{ill}(b) := \text{V}\textsc{ar}\text{K}\textsc{ill}(b) \cup \{\text{variable}(LHS)\}$

Hier vereinfacht: Nur Zuweisungen in Block
Analoges Vorgehen für andere Operationen, unterscheide

- Lesen (RHS) von Variablen
- Schreiben (LHS) von Variablen

Live Variables - Beispiel 1



	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVAR	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{a,b,c,d,i}
$\overline{\text{VARKILL}}$	{a,b,c,d,y,z}	{b,d,i,y,z}	{a,i,y,z}	{b,c,i,y,z}	{a,b,c,i,y,z}	{a,b,d,i,y,z}	{a,c,d,i,y,z}	{a,b,c,d}

Quelle: C&T, pp. 442-443

Live Variables - Iterative Lösung

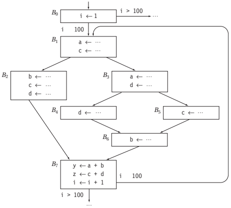
N := number of blocks - 1
for i := 0 **to** N **do**
    LiveOut(i) := $\emptyset$
changed := **true**
while changed **do**
    changed := **false**
    for i := 0 **to** N **do**
        recompute LiveOut(i)
        if LiveOut(i) changed **then**
            changed := **true**

Live Variables - Beispiel 2

$\textsc{LiveOut}(b) =$

$$\bigcup_{m \in \mathrm{succ}(b)} \textsc{UEVar}(m) \cup (\textsc{LiveOut}(m) \cap \overline{\textsc{VarKill}(m)})$$

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVAR	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{a,b,c,d,i}
$\overline{\textbf{VARKILL}}$	{a,b,c,d,y,z}	{b,d,i,y,z}	{a,i,y,z}	{b,c,i,y,z}	{a,b,c,i,y,z}	{a,b,d,i,y,z}	{a,c,d,i,y,z}	{a,b,c,d}



Quelle: C&T, pp.442-443

	LIVEOUT(*n*)							
Iteration	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$	{a,b,c,d,i}	$\emptyset$	$\emptyset$	$\emptyset$	{a,b,c,d,i}	$\emptyset$
2	$\emptyset$	{a,i}	{a,b,c,d,i}	$\emptyset$	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
3	{i}	{a,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
4	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
5	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}

Diskussion

Eigenschaften des iterativen Datenflußlösers

Vor Benutzung berücksichtigen:

- Terminiert die Analyse?
- Beantwortet das berechnete Ergebnis die gestellte Frage?
- Wie schnell läuft die Analyse?

Im folgenden Diskussion am Beispiel LIVEOUT.

Terminierung der Datenflußanalyse

TECHNISCHE UNIVERSITÄT DARMSTADT

- LIVEOUT Mengen wachsen monoton, beginnend bei $\emptyset$
- Sie können nie schrumpfen
- Bei maximaler Größe umfasst eine LIVEOUT-Menge alle Variablen
- Da es nur endlich viele Variablen gibt, sind die LIVEOUT-Mengen beschränkt
- Die Iteration bricht also nach endlicher Zeit immer ab
  - Irgendwann ändert sich nichts mehr
  - Worst-case: Alle LIVEOUT-Mengen umfassen alle Variablen

Korrektheit der Datenflußanalyse

- LIVEOUT berechnet lokale Eigenschaft
  - Zwischen Block und seinen Nachfolgern
- Vereinigt Ergebnisse der Nachfolger
  - Wenn *v* live auf irgendeiner Nachfolgekante ist, dann *v* in LIVEOUT
- Kann Zusammenhang zwischen lokalen Eigenschaften und der Definition von Live Variables hergestellt werden?
  - Diese ist ja über alle Pfade definiert!
- Beweis über Verbandalgebra (*lattice algebra*)
  - Hier nicht behandelt ($\rightarrow$ Kam/Ullman JACM 1976)

Effizienz der Datenflußanalyse 1

TECHNISCHE UNIVERSITÄT DARMSTADT

- Überlegung: Das Ergebnis der iterativen Lösung des Datenflußproblems ist unabhängig von der Bearbeitungsreihenfolge der Blöcke
- Die Reihenfolge beeinflußt aber die nötige Anzahl von Iterationen
- Also: Suche nach schnellerer Abarbeitungsreihenfolge
- Idee: Bei Vorgehen ...
  - ... rückwärts (LIVEOUT): Besuche soviele Nachfolger eines Knotens wie möglich, bevor der Knoten selbst besucht wird

Effizienz der Datenflußanalyse 2

Vorwärts

Verschiedene Möglichkeiten für Abarbeitungsreihenfolgen

- Vorwärts: z.B. Breadth-First-Search,
  aber besser Reverse Post-Order (RPO)

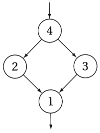
Beispiel: Reverse Post-Order

Step1: PostOrder

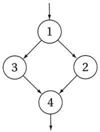
```
proc main() ≡
   count ← 1
   Visit(Entry)
end

proc Visit(v) ≡
   mark v as visited
   foreach successor s of v not yet visited
      Visit(s)
   end
   PostOrder(v) ← count++
end
```

Step 2: rPostOrder

```
foreach v ∈ V do
   rPostOrder(v) ← |V| − PostOrder(v)
end
```



Postorder



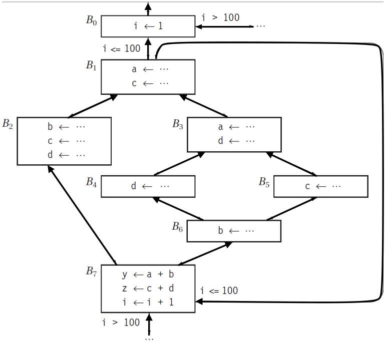
Reverse Postorder

Quelle: Purdue CS502 Dataflow, C&T p.446

Effizienz der Datenflußanalyse 3

Rückwärts

- z.B. Depth-First Search
- besser RPO auf reversem CFG (Kanten umgekehrt)



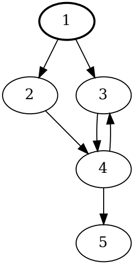
Post-Order auf rev. CFG: B0, B1, B2, B3, B5, B4, B6, B7

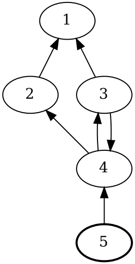
RPO auf rev. CFG: B7, B6, B4, B5, B3, B2, B1, B0

Effizienz der Datenflußanalyse 3

Rückwärts

- RPO auf reversem CFG (Kanten umgekehrt)
  $\neq$ PO auf normalem CFG





PO: 3, 5, 4, 2, 1

RPO auf rev.: 5, 4, 2, 3, 1

Effizienz der Datenflußanalyse 4

TECHNISCHE UNIVERSITÄT DARMSTADT

Abspeichern als Permutation in Array $P = [7, 6, 5, 4, 2, 3, 1, 0]$

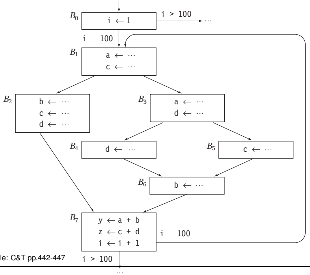
N := number of blocks - 1
for i := 0 **to** N **do**
  LIVEOUT(i) := $\emptyset$
changed := **true**
while changed **do**
  changed := **false**
  for i := 0 **to** N **do**
    recompute LIVEOUT($P[i]$)
    if LIVEOUT($P[i]$) changed **then**
      changed := **true**

Effizienz der Datenflußanalyse 5

$$\text{LIVEOUT}(b) = \bigcup_{m \in \text{succ}(b)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVAR	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{a,b,c d,i}
$\overline{\text{VARKILL}}$	{a,b,c, d,y,z}	{b,d,i, y,z}	{a,i, y,z}	{b,c,i, y,z}	{a,b,c, i,y,z}	{a,b,d, i,y,z}	{a,c,d, i,y,z}	{a,b, c,d}

Reihenfolge: B7, B6, B5, B4, B2, B3, B1, B0



Quelle: C&T pp.442-447

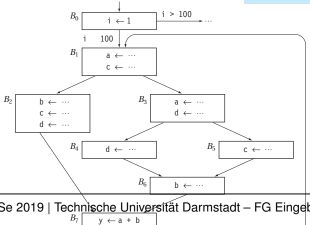
Iteration	LIVEOUT(n) B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	{i}	{a,c,i}	{a,b,c, d,i}	{a,c, d,i}	{a,c, d,i}	{a,c, d,i}	{a,b,c, d,i}	$\emptyset$
2	{i}	{a,c,i}	{a,b,c, d,i}	{a,c, d,i}	{a,c, d,i}	{a,c, d,i}	{a,b,c, d,i}	{i}
3	{i}	{a,c,i}	{a,b,c, d,i}	{a,c, d,i}	{a,c, d,i}	{a,c, d,i}	{a,b,c, d,i}	{i}

Konvergiert jetzt in 3 Iterationen (statt 5)!

Effizienz der Datenflußanalyse 6

- Weiterhin unnötige Berechnungen. . .
  - Finale Iteration: Komplett unnötig
  - Vorhergehende Iterationen: Blöcke deren Nachfolger sich nicht geändert haben
- Verbesserung: Work list basierter Algorithmus
  Prioritätswarteschlange mit Prioritäten in RPO

```
pqueue := cfg.getBlocksInRPO();

while ( pqueue.notEmpty() ) {
  b := pqueue.removeFirst();
  recompute Sets(b);
  if (Sets(b) changed)
    pqueue.add(b.getNeighbours());
}
```



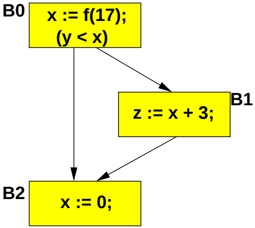
Iteration	LIVEOUT(*n*) B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	∅
2	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}

Schwächen der Datenflußanalyse 1

- Fundamentale Annahme bei Datenflußberechnung:
- Alle Blöcke können ausgeführt werden

Gegenbeispiel zur Annahme 1

```
x := f(17);
if (y < x) then
  z := x + 3;
endif
x := 0;
```



- x ist Live in B0, da es in B1 gelesen werden kann
- x wird aber in B2 Killed
- Falls B1 nie ausgeführt wird, ist x nicht Live außerhalb von B0

Gegenbeispiel zur Annahme 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Falls der Compiler beweisen kann, dass immer $y \geq x$ ...

- würde die Anweisung `z := x+3` nie ausgeführt werden
- Falls dann auch noch der Aufruf `f(17)` keine Seiteneffekte hat
- ... können Blocks B0 und B1 komplett entfernt werden

Kann aber nicht allgemein gelöst werden ($\rightarrow$ Halteproblem)!

Schwächen der Datenflußanalyse 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- LIVEOUT: Wird immer über alle Nachfolger berechnet
- Berechnet wird so nur eine Zusammenfassung der tatsächlich möglichen Abläufe

Schwächen der Datenflußanalyse 3

TECHNISCHE UNIVERSITÄT DARMSTADT

Probleme bei Arrays

- Zugriff `A[i,j,k]` auf ein einzelnes Element
- Datenflussanalyse kennt aber keine konkreten Werte für *i*, *j*, *k*
- Abstraktion: Betrachte gesamtes Array als eine Variable
  - `... := A[i,j,k]` zählt als Verwendung des gesamten Arrays
  - `A[i,j,k] := ...` zählt als Definition des gesamten Arrays

Schwächen der Datenflußanalyse 4

Benutzung dieser ungenauen Ergebnisse muß konservativ erfolgen!

- Fehlabschätzungen dürfen Korrektheit der Analyse in Bezug auf die gesuchte Aussage nicht beeinflussen
- Beispiele
  - Kann der Wert von `A[i,j,k]` nach Schreibzugriff auf `A[l,m,n]` verworfen werden?
  - ...Nein, denn der Schreibzugriff KILLed nicht notwendigerweise `A[i,j,k]`!
  - Könnte der Wert von `A[i,j,k]` nach Schreibzugriff auf `A[l,m,n]` beschädigt werden?
  - ...Ja, denn der Schreibzugriff könnte jedes Element von A verändern!

Schwächen der Datenflußanalyse 5

TECHNISCHE UNIVERSITÄT DARMSTADT

Analoge Problematik bei Zeigern

- Zuweisung via Zeiger kann potentiell jede Variable beeinflussen
- Kann weite Teile der Datenflussanalyse unbrauchbar machen
- Wird schlimmer bei Adressarithmetik (wie in C)
  - Nun nicht nur auf einzelne Variablen, sondern beliebig im Speicher
- Wird etwas besser bei fester Typisierung (keine Wandlung möglich)
  - Nun nur noch Variablen vom Typ des Zeigers betroffen

Schwächen der Datenflußanalyse 6

TECHNISCHE UNIVERSITÄT DARMSTADT

Prozeduren

- Auch bei Beschränkung der Analyse auf eine Prozedur
- Jeder Prozeduraufruf kann verändern (abhängig von Sprache):
  - Nur Var-Parameter
  - Nicht-Lokale Variablen
  - Globale Variablen
  - Bei Unterstützung von Zeigern: Gesamten Speicherinhalt
- Unterprozeduren verkomplizieren die Situation noch

➥Analyse muss “worst case” Annahmen machen

Sammlung von Datenflußproblemen

Verfügbare Ausdrücke

- Available Expressions
- AVAIL(*b*): Menge der Ausdrücke, die Block *b* erreichen
- Genauer im VL-Block: Redundanzeliminierung
- Vorwärtsgerichteter Fluß über berechnete Ausdrücke
- Konkrete Anwendung:
  Global Common Subexpression Elimination

Erreichende Definitionen 1

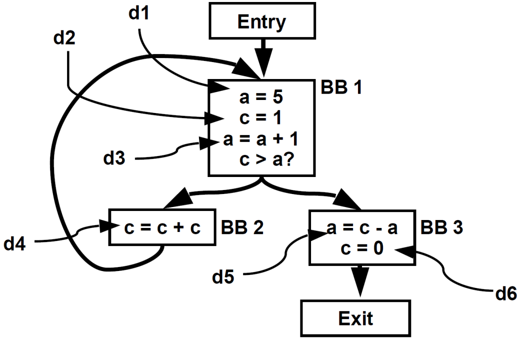
TECHNISCHE UNIVERSITÄT DARMSTADT

Eine Definition d einer Variablen v erreicht eine Operation i genau dann, wenn v in i gelesen wird und v auf einem Pfad von d zu i nicht redefiniert wird.

- REACHES(b): Menge der Definitionen, die Block b erreichen.
- Vorwärtsgerichteter Fluß über Zuweisungen an Variablen
- *Reaching Definitions*

Erreichende Definitionen 2

DEDEF(b) (*downward exposed definitions*): Definitionen in b, die nicht vor Blockende überschrieben werden



DEDEF($BB1$) = $\{d2, d3\}$

DEDEF($BB2$) = $\{d4\}$

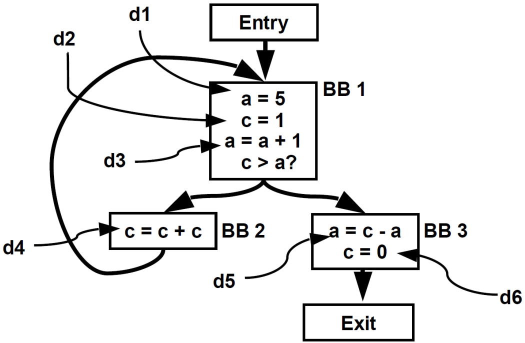
DEDEF($BB3$) = $\{d5, d6\}$

Quelle: UToronto ECE540S Data Flow Analysis

Erreichende Definitionen 3

TECHNISCHE UNIVERSITÄT DARMSTADT

DEFKILL(b): Im Block b überschriebene Definitionen anderer Blöcke aus der Menge aller Definitionen in der Prozedur



DEFKILL($BB1$) = $\{d5, d4, d6\}$

DEFKILL($BB2$) = $\{d2, d6\}$

DEFKILL($BB3$) = $\{d1, d3, d2, d4\}$

Quelle: UToronto ECE540S Data Flow Analysis

Erreichende Definitionen 4

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflußgleichungen

$\textsc{Reaches}(b_0) = \emptyset$

$\textsc{Reaches}(b) =$
$\quad \bigcup_{d \in \text{preds}(b)} (\text{DEDef}(d) \cup (\textsc{Reaches}(d) \cap \overline{\text{DefKill}(d)}))$

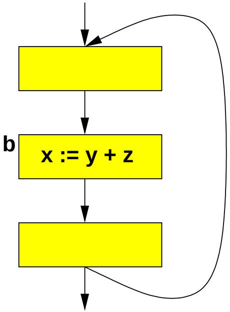
- Lösung mit iterativem Fixpunktverfahren
- Startwerte: $\textsc{Reaches}(b) = \emptyset$ für alle b

Erreichende Definitionen 5

TECHNISCHE UNIVERSITÄT DARMSTADT

Anwendungsbeispiel: Anweisung `x := y + z` in Schleifen-Body *b*

- Falls alle REACHES(*b*) für `y` und `z` außerhalb der Schleife
- ... kann gesamte Berechnung von `x` vor die Schleife gezogen werden
- Loop-Invariant Code Motion



Sehr rege Ausdrücke 1

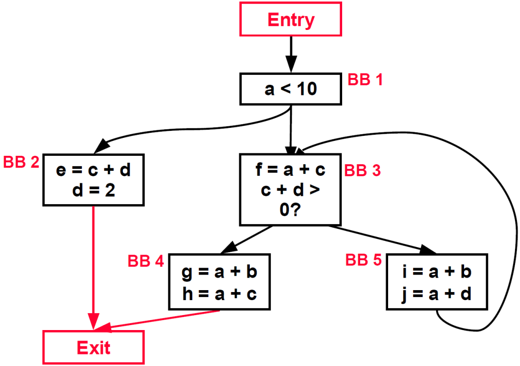
Sehr Rege (*very busy*)

Ein Ausdruck *e* ist sehr rege am Ende eines Blocks *b*, wenn er in allen Nachfolgern von *b* evaluiert und benutzt wird, und das einmalige Evaluieren von *e* am Ende von *b* das gleiche Ergebnis hätte wie die erstmalige Evaluation von *e* in den Nachfolgern von *b*.

- VERYBUSY(*b*): Menge der Ausdrücke, die am Ende von *b* sehr rege sind
- Rückwärtsgerichteter Fluß über Ausdrücke
- *Very Busy Expressions*

Sehr rege Ausdrücke 2

UEEXPR(b) (*upwards exposed expressions*): In b vor Überschreiben ihrer Operanden benutzte Ausdrücke.



Quelle: UToronto ECE540S Data Flow Analysis

$\text{UEE}\textsc{xpr}(BB1) = \emptyset$

$\text{UEE}\textsc{xpr}(BB2) = \{c + d\}$

$\text{UEE}\textsc{xpr}(BB3) = \{a + c, c + d\}$

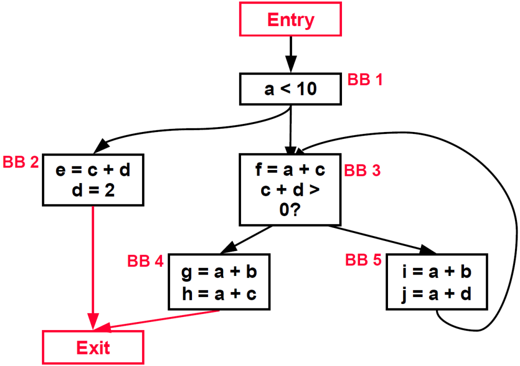
$\text{UEE}\textsc{xpr}(BB4) = \{a + b, a + c\}$

$\text{UEE}\textsc{xpr}(BB5) = \{a + b, a + d\}$

Sehr rege Ausdrücke 3

TECHNISCHE UNIVERSITÄT DARMSTADT

$\textsc{ExprKill}(b)$: Im Block b durch Überschreiben der Operanden unbrauchbar gemachte Ausdrücke



$\textsc{ExprKill}(BB1) = \emptyset$

$\textsc{ExprKill}(BB2) = \{a + d, c + d\}$

$\textsc{ExprKill}(BB3) = \emptyset$
$\textsc{ExprKill}(BB4) = \emptyset$

$\textsc{ExprKill}(BB5) = \emptyset$

Quelle: UToronto ECE540S Data Flow Analysis

Sehr rege Ausdrücke 4

TECHNISCHE UNIVERSITÄT DARMSTADT

Datenflußgleichungen

$\textsc{VeryBusy}(b_n) = \emptyset$

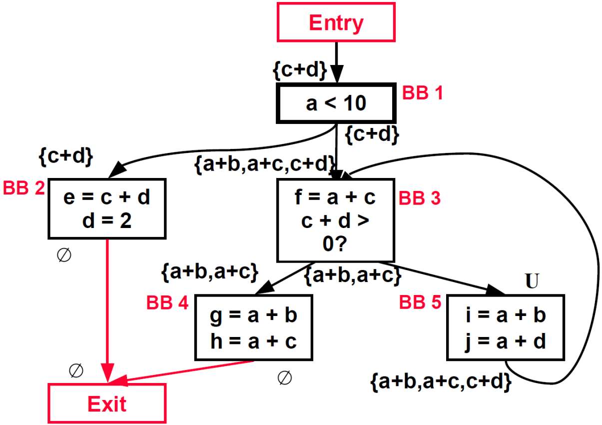
$\textsc{VeryBusy}(b) =$
$\quad \bigcap_{d \in \mathrm{succ}(b)} (\textsc{UEExpr}(d) \cup (\textsc{VeryBusy}(d) \cap \overline{\textsc{ExprKill}(d)}))$

- Lösung mit iterativem Fixpunktverfahren
- Startwert für $b \neq b_n$: $\textsc{VeryBusy}(b) = U$, mit U Menge aller Ausdrücke in Prozedur

Sehr rege Ausdrücke 5

TECHNISCHE UNIVERSITÄT DARMSTADT

Konkretes Beispiel



Sehr rege Ausdrücke 6

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anwendung zur Optimierung: Code Hoisting
- Ersetze Evaluationen der sehr regen Ausdrücke in Nachfolgern
- ...durch eine Evaluation in Vorgänger
- Macht Code nicht (direkt) schneller, aber kleiner

Verallgemeinerung

Gemeinsamkeiten 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Pfad-Vorwärts: Reaching definitions

$$\textsc{Reaches}(b) = \bigcup_{d \in preds(b)} (\textsc{DEDef}(d) \cup (\textsc{Reaches}(d) \cap \overline{\textsc{DefKill}(d)}))$$

Ein Pfad-Rückwärts: Live variables

$$\textsc{LiveOut}(b) = \bigcup_{m \in \mathrm{succ}(b)} (\textsc{UEVar}(m) \cup (\textsc{LiveOut}(m) \cap \overline{\textsc{VarKill}(m)}))$$

Alle Pfade-Vorwärts: Available expressions ($\rightarrow$ Redundanzelim.)

$$\textsc{Avail}(b) = \bigcap_{d \in preds(b)} (\textsc{DEExpr}(d) \cup (\textsc{Avail}(d) \cap \overline{\textsc{ExprKill}(d)}))$$

Alle Pfade-Rückwärts Very busy expressions

$$\textsc{VeryBusy}(b) = \bigcap_{d \in succ(b)} (\textsc{UEExpr}(d) \cup (\textsc{VeryBusy}(d) \cap \overline{\textsc{ExprKill}(d)}))$$

Gemeinsamkeiten 2

- Sehr ähnliche Struktur der Gleichungen
  - $f(x) = c_1 \ \mathrm{op}_1 \ (x \ \mathrm{op}_2 \ c_2)$
- Wie ausnutzen?
- Lösung aller solcher Datenflußprobleme
- Data Flow Framework
- Akzeptiert $c_1, c_2, \mathrm{op}_1, \mathrm{op}_2$, Konfluenzoperator als Parameter
- Lösen dann für Fixpunkt
- Vorteil: Nur ein Algorithmus muß mit viel Sorgfalt implementiert werden
- Kann dann alle vergleichbaren Probleme lösen

Es gibt aber auch Datenflußprobleme mit anderer Struktur!

Zusammenfassung

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Aufräumen nach Optimierung: Copy Propagation
- Iterative Datenflußanalyse
  - Live Variables
  - Erreichende Definitionen
  - Sehr rege Ausdrücke
  - Konstanten propagieren
- Diskussion
  - Reihenfolge
  - Schwächen
  - Gemeinsamkeiten