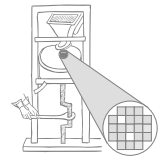


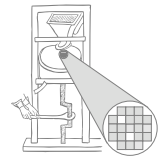
Optimierende Compiler

Datenflussanalyse

Andreas Koch

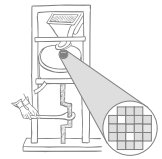
FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2008



Organisatorisches

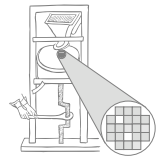
Vorlesung



Am Donnerstag, dem 15.05.2008 muss die Vorlesung leider entfallen.

Nutzen Sie die Zeit zum Bearbeiten des Programmierprojekts!

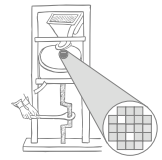
Organisatorisches



Ab jetzt auszugsweise Material aus

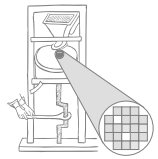
Advanced Compiler Design and Implementation

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



Copy Propagation

Copy Propagation (CP)

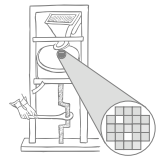


- Viele Algorithmen legen Zwischenvariablen an
  - `a := x + y; t1 = a;`
- Zwischenvariablen
  - benötigen viel Speicher, viele Register
  - verursachen viele Kopieranweisungen `y := x`
  - sind in vielen Fällen unnötig

➥Beseitigen durch

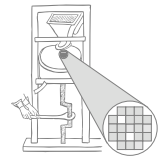
1. Copy Propagation (→ Muchnick 12.5)
2. Dead Code Elimination

Idee Copy Propagation



- 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/DVNT/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

Datenstrukturen für Lokale CP 1



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

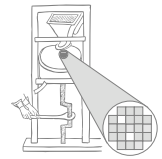
Tupel (v, w)

- 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.

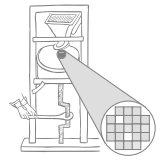
Datenstrukturen für Lokale CP 2



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

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

Algorithmus für Lokale CP 1

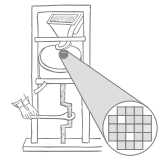


Hilfsfunktion: Liefere zu verwendenden Operand für **opnd**, ggf. ausgetauscht durch in ACP vorhandene Originalvariable

A. Koch

```
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

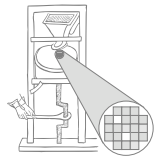


Hilfsfunktion: Entferne eine überschriebene Variable `v` aus ACP

A. Koch

```
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
```

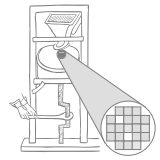
Algorithmus für Lokale CP 3



```
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 // benutzende 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 if (i == "LHS := RHS") then // Zuweisung
      Remove_ACP(ACP, i.LHS.name); // entferne überschr. Var.
      if (RHS instanceof Var && LHS != RHS) then // Kopie?
        ACP.add(new Pair(LHS, RHS));
      endif
    endif
  endfor
end
```

A. Koch

Beispiel Lokale CP

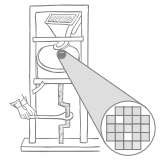


A. Koch

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⟩}	

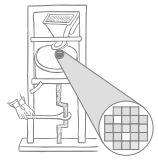
Muchnick, Fig 12.25

Ansatz Globale CP



- 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



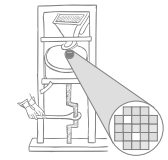
COPY(b)

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.

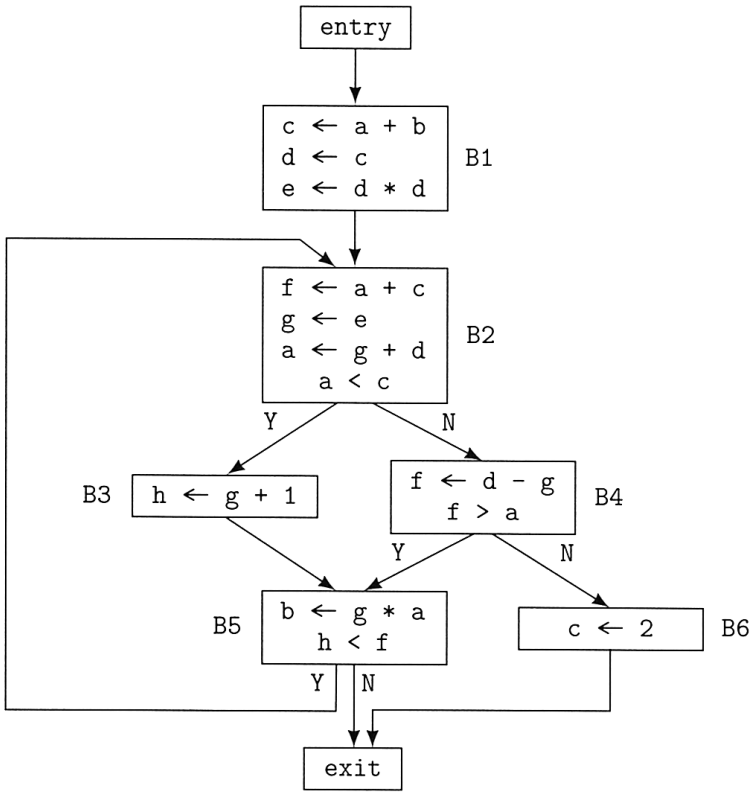
KILL(b)

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

Beispiel: COPY und KILL



Beispiel



Mengen

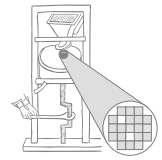
$$
\begin{aligned}
COPY(\texttt{entry}) &= \emptyset \\
COPY(\texttt{B1}) &= \{\langle \texttt{d}, \texttt{c}, \texttt{B1}, 2\rangle\} \\
COPY(\texttt{B2}) &= \{\langle \texttt{g}, \texttt{e}, \texttt{B2}, 2\rangle\} \\
COPY(\texttt{B3}) &= \emptyset \\
COPY(\texttt{B4}) &= \emptyset \\
COPY(\texttt{B5})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

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

A. Koch

Quelle: Muchnick, pp. 359-360

Datenflußgleichungen für globale CP 1



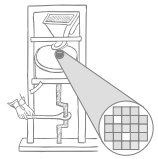
CPIN(b)

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.

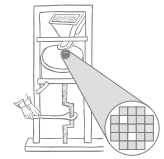
Datenflußgleichungen für globale CP 1



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
  - $\text{CP}_{\text{IN}}(\text{entry})$: Startblock hat keine Kopieranweisungen zur Verfügung
  - $\text{CP}_{\text{IN}}(b), b \neq \text{entry}$: Alle anderen Blöcke haben alle in der ganzen Prozedur auftretenden Kopieranweisungen zur Verfügung
    - Wird schrittweise eingeschränkt

Datenflußgleichungen für globale CP 2

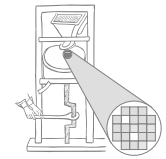


$$\begin{aligned} \text{CPIN}(b) &= \bigcap_{d \in \text{pred}(b)} \text{CPOUT}(d) \\ \text{CPOUT}(b) &= \text{COPY}(b) \cup (\text{CPIN}(b) - \text{KILL}(b)) \end{aligned}$$

mit Initialisierung

$$\begin{aligned} \text{CPIN}(\text{entry}) &= \emptyset \\ \text{CPIN}(b) &= \bigcup_{d \in \text{Blocks}} \text{COPY}(d) \text{ ,für } b \neq \text{entry} \end{aligned}$$

Beispiel: Initialisierung

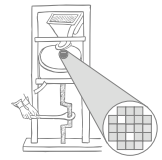


A. Koch

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

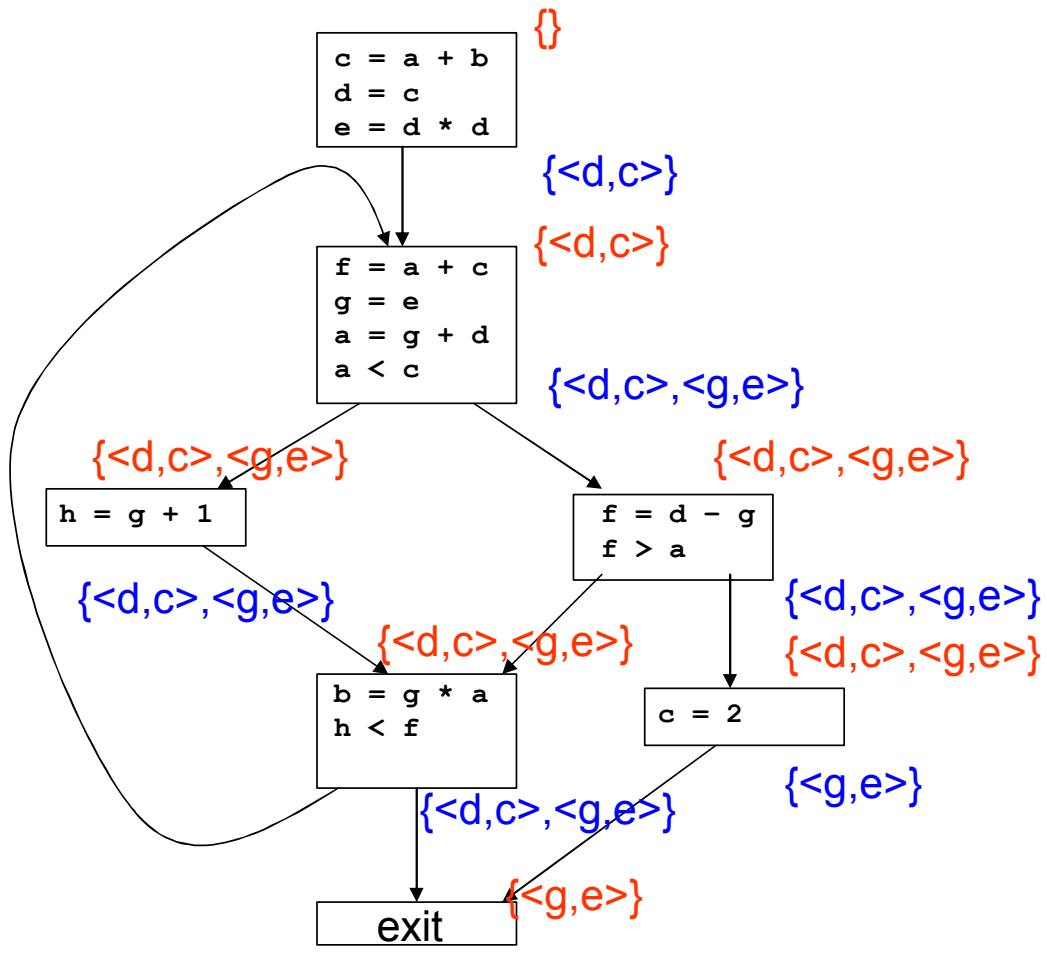
$$
\begin{aligned}
\textsc{CPin}(\text{entry}) &= \emptyset \\
\textsc{CPin}(\text{B1}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\textsc{CPin}(\text{B2}) &= \{\langle d, c, B1, 2\rangle, \langle g, e, B2, 2\rangle\} \\
\textsc{CPin}(\text{B3}) &= \{\langle d, c, B1, 2\rangle,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

Beispiel: Ergebnis der iterativen Berechnung



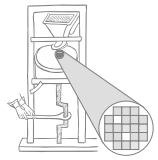
Rot: CPIN, Blau: CPOUT

A. Koch



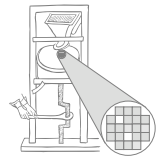
Quelle: White, GMU CS640, Scalar Optimization

Weitere Vorgehensweise bei der globalen CP



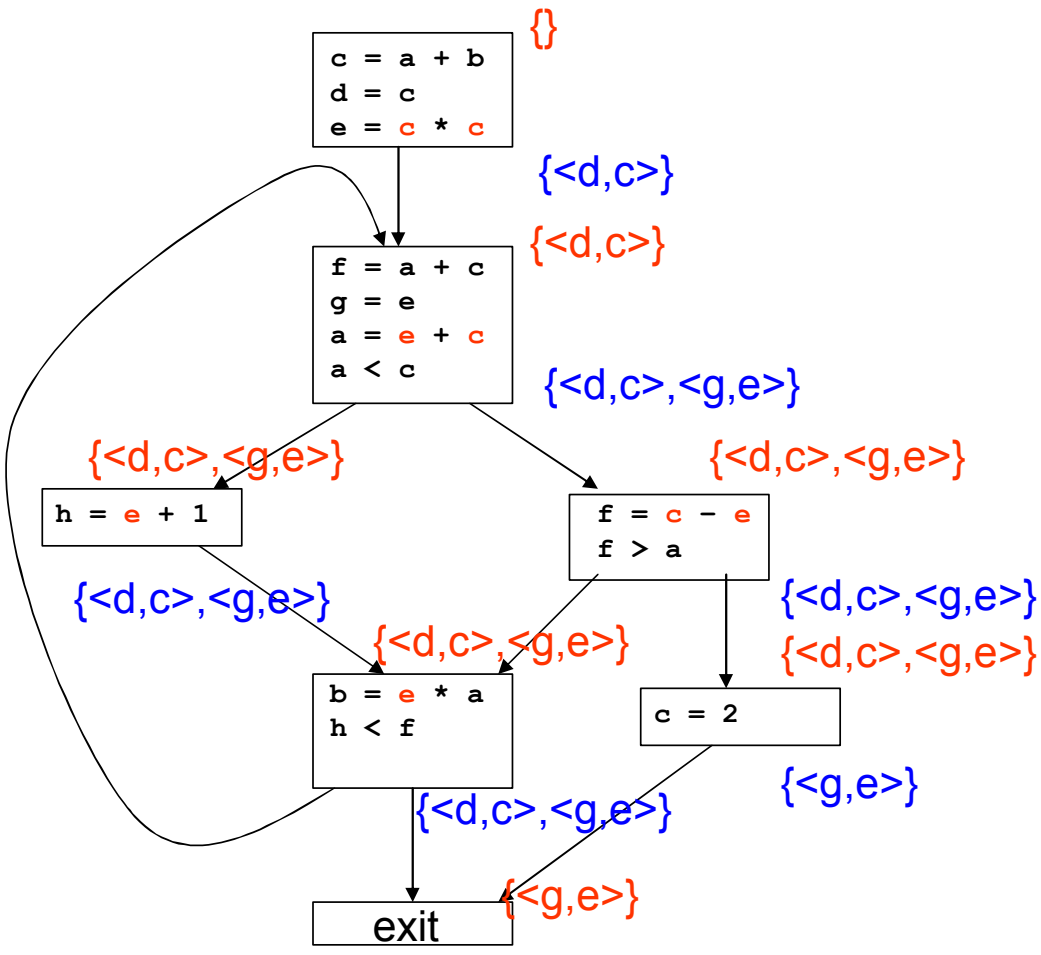
- Werte berechnete Daten nun pro Block aus
- Vorgehen: `Local_Copy_Prop` beginnt nun nicht mehr mit leerer ACP-Menge
- . . . sondern: Initialisiere ACP-Menge für Block b aus $\text{CP}_{\text{IN}}(b)$
- Analog zu: VN auf EBB und Region (DVNT)

Beispiel: Ergebnis der globalen CP

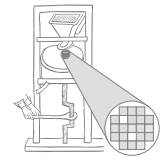


Rot: CPIN, Blau: CPOUT

A. Koch

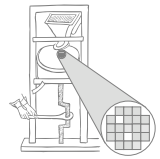


Quelle: White, GMU CS640, Scalar Optimization



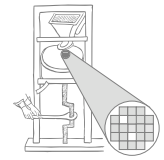
Iterative Datenflussanalyse

Iterative Datenflussanalyse



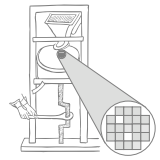
- Weiterführung der Techniken aus GCSE: AVAIL etc.
- Treffe Aussagen
  - . . . über Laufzeitverhalten von Programm
  - . . . zur Compile-Zeit
- Mittel der Wahl
  - Gleichungssysteme
  - Lösungsverfahren: Hier iterative, gibt aber auch andere
- Anwendung
  - Finde Anwendungsstellen von Optimierungen
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

Weiteres Beispiel: *Live Variables*



Live Variables

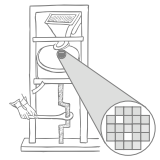
Live Variables - Definition



Live Variable

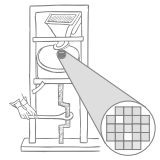
Eine Variable v ist *lebendig* (*live*) an einer Stelle p im Programm genau dann, wenn es im CFG einen Pfad von p zu einer Verwendung von v gibt, auf dem v *nicht* definiert wird.

Live Variables - Anwendung



- 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



LiveOut(b)

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

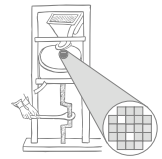
Damit Berechnung durch Gleichungssystem.

1. Teil
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

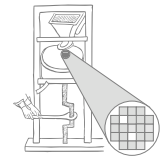


2. Teil: Rekursive Definition für innere Knoten

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

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

Live Variables - Interpretation

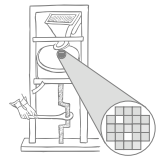


A. Koch

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

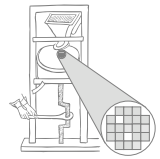
- $\textsc{LiveOut}(m)$ sind alle Variablen, die live am Anfang von Nachfolgerblöcken m sind
- Variable muss nur auf einem Pfad live sein ($\rightarrow \bigcup$)
  - Unterschied zu AVAIL!
- Jeder Nachfolgerknoten m trägt Variablen bei
  - In m benutzte Variablen, die vorher nicht redefiniert werden ($\textsc{UEVar}(m)$)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

Live Variables - Vorgehen



1. CFG aufbauen
   - Kennen wir bereits, für strukturierte Sprachen einfach
   - 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



A. Koch

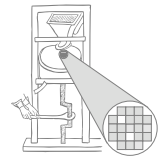
UEVAR(b) := $\emptyset$
VARKILL(b) := $\emptyset$

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 $v \notin$ VARKILL(b) **then**
      UEVAR(b) := UEVAR$(b) \cup \{v\}$
  VARKILL(b) := VARKILL$(b) \cup \{$variable$(LHS)\}$

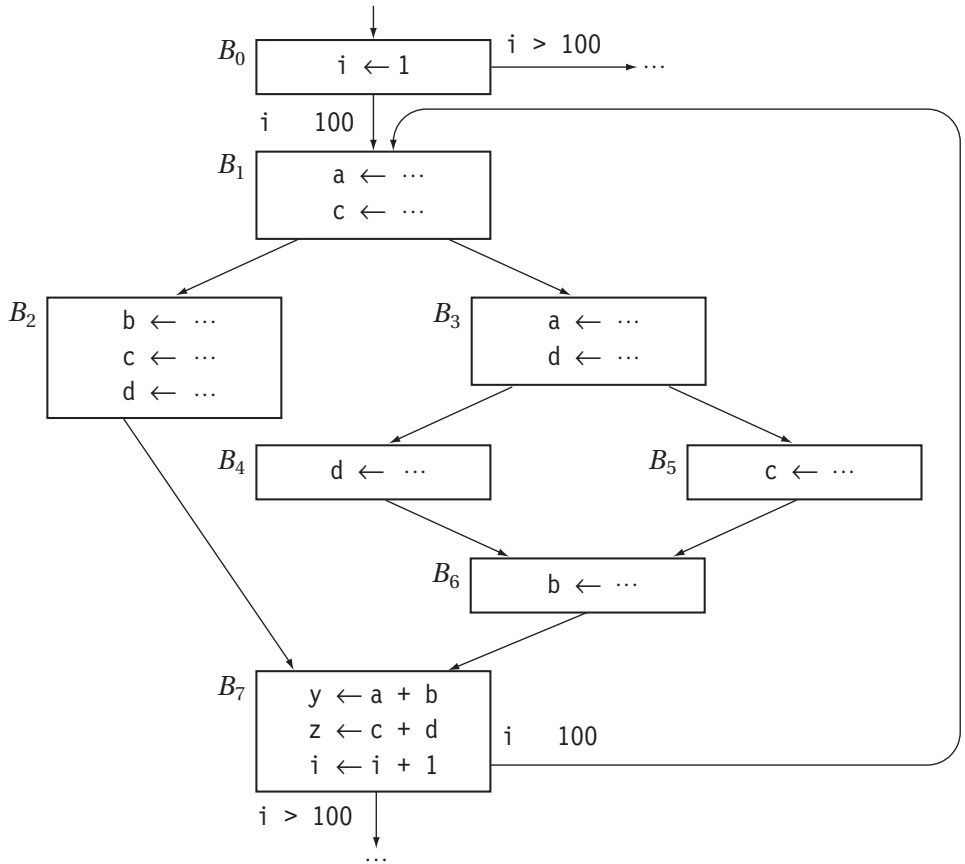
Hier vereinfacht: Nur Zuweisungen in Block
Analoges Vorgehen für andere Operationen, unterscheide

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

Live Variables - Beispiel 1



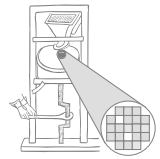
A. Koch



	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVAR	∅	∅	∅	∅	∅	∅	∅	{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

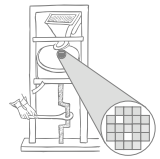


Analog zu AVAIL, ersetzt durch Berechnung von LIVEOUT

A. Koch

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**

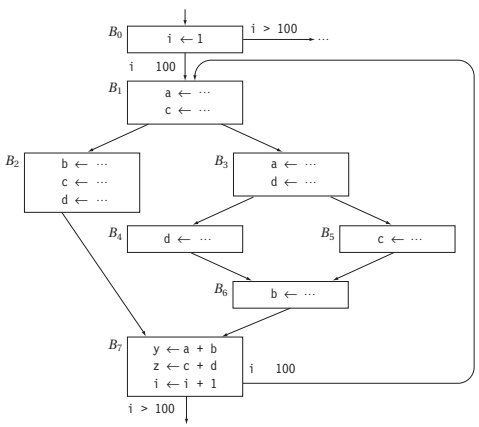
Live Variables - Beispiel 2



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

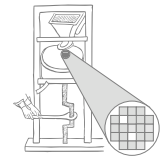
A. Koch

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVar	∅	∅	∅	∅	∅	∅	∅	{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}



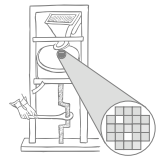
Quelle: C&T, pp.442-443

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

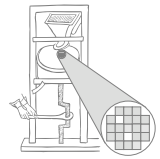


Vor Benutzung berücksichtigen:

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

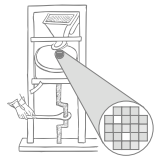
Im folgenden Diskussion am Beispiel LIVEOUT.

Terminierung der Datenflußanalyse



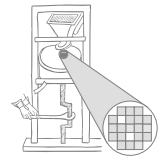
- LIVEOUT Mengen wachsen monoton, beginnend bei $\emptyset$
- Sie können nie schrumpfen
- Bei maximaler Größe umfasst eine LIVEOUT-Menge alle Variablen
- Da es nur endlich viele Variablen gibt, sind die LIVEOUT-Mengen beschränkt
- Die Iteration bricht also nach endlicher Zeit immer ab
  - Irgendwann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

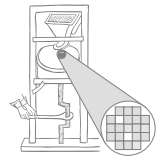
Effizienz der Datenflußanalyse 1



- 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 . . .
  - . . . vorwärts (AVAIL): Besuche soviele Vorgänger eines Knotens wie möglich, bevor der Knoten selbst besucht wird
  - . . . rückwärts (LIVEOUT): Besuche soviele Nachfolger eines Knotens wie möglich, bevor der Knoten selbst besucht wird

Effizienz der Datenflußanalyse 2

Vorwärts



Verschiedene Möglichkeiten für Abarbeitungsreihenfolgen

A. Koch

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

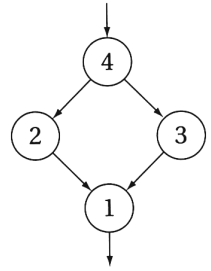
Beispiel: Reverse Post-Order

Step1: PostOrder

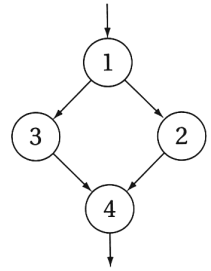
```
proc main() ≡
    count ← 1
    Visit(Entry)
end

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
```



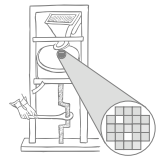
Postorder



Reverse Postorder

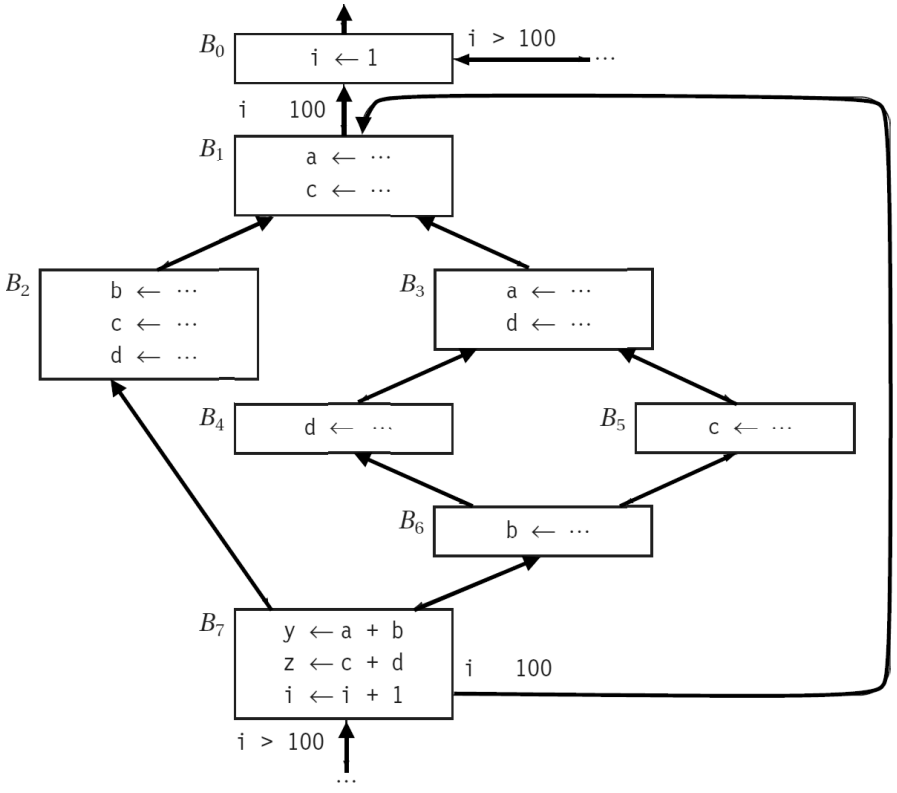
Quelle: Purdue CS502 Dataflow, C&T p.446

Effizienz der Datenflußanalyse 3
Rückwärts



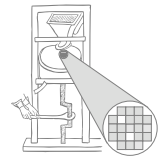
- z.B. Depth-First Search
- besser RPO auf reversem CFG (Kanten umgekehrt)

A. Koch



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

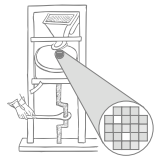
Effizienz der Datenflußanalyse 4



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

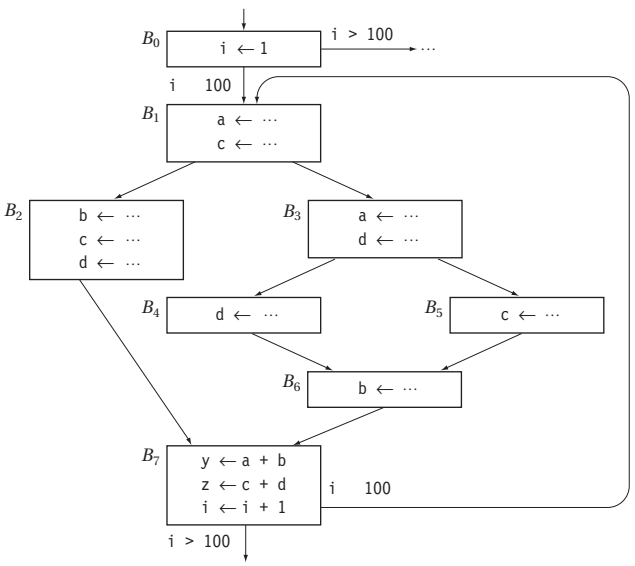


$$\textsc{LiveOut}(b) = \bigcup_{m \in \text{succ}(b)} \textsc{UEVar}(m) \cup (\textsc{LiveOut}(m) \cap \overline{\textsc{VarKill}(m)})$$

A. Koch

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVAR	∅	∅	∅	∅	∅	∅	∅	{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

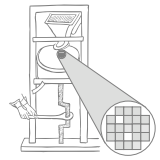


Iteration	LIVEOUT(n) B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	∅
2	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}

Quelle: C&T pp.442-447

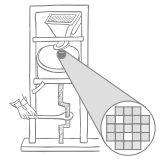
Konvergiert jetzt in 3 Iterationen (statt 5)!

Schwächen der Datenflußanalyse 1



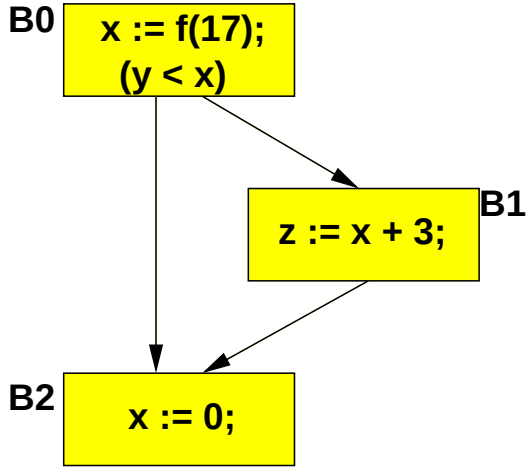
- Fundamentale Annahme bei Datenflußberechnung:
- Alle Blöcke können ausgeführt werden

Gegenbeispiel zur Annahme 1



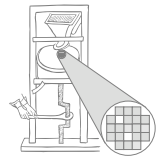
A. Koch

```
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

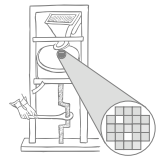


Falls der Compiler beweisen kann, dass immer $y \geq x$ ...

- 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

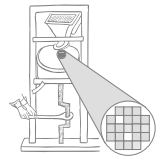
Kann aber nicht allgemein gelöst werden ($\rightarrow$ Halteproblem)!

Schwächen der Datenflußanalyse 2



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

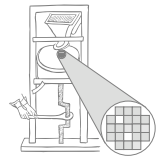
Schwächen der Datenflußanalyse 3



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

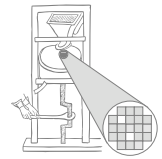
Schwächen der Datenflußanalyse 4



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!

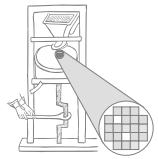
Schwächen der Datenflußanalyse 5



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

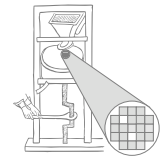
Schwächen der Datenflußanalyse 6



Prozeduren

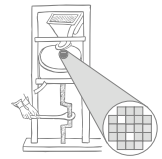
- Auch bei Beschränkung der Analyse auf eine Prozedur
- Jeder Prozeduraufruf kann verändern (abhängig von Sprache):
  - Nur Var-Parameter
  - Nicht-Lokale Variablen
  - Globale Variablen
  - Bei Unterstützung von Zeigern: Gesamten Speicherinhalt
- Unterprozeduren verkomplizieren die Situation noch

➥Analyse muss “worst case” Annahmen machen



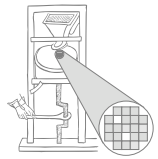
Sammlung von Datenflußproblemen

Verfügbare Ausdrücke



- Available Expressions
- AVAIL(b): Menge der Ausdrücke, die Block b erreichen
- Vorgestellt im 6. Block (Einführung in Code-Optimierung)
- Vorwärtsgerichteter Fluß über berechnete Ausdrücke
- Konkrete Anwendung: Global Common Subexpression Elimination

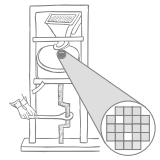
Erreichende Definitionen 1



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.

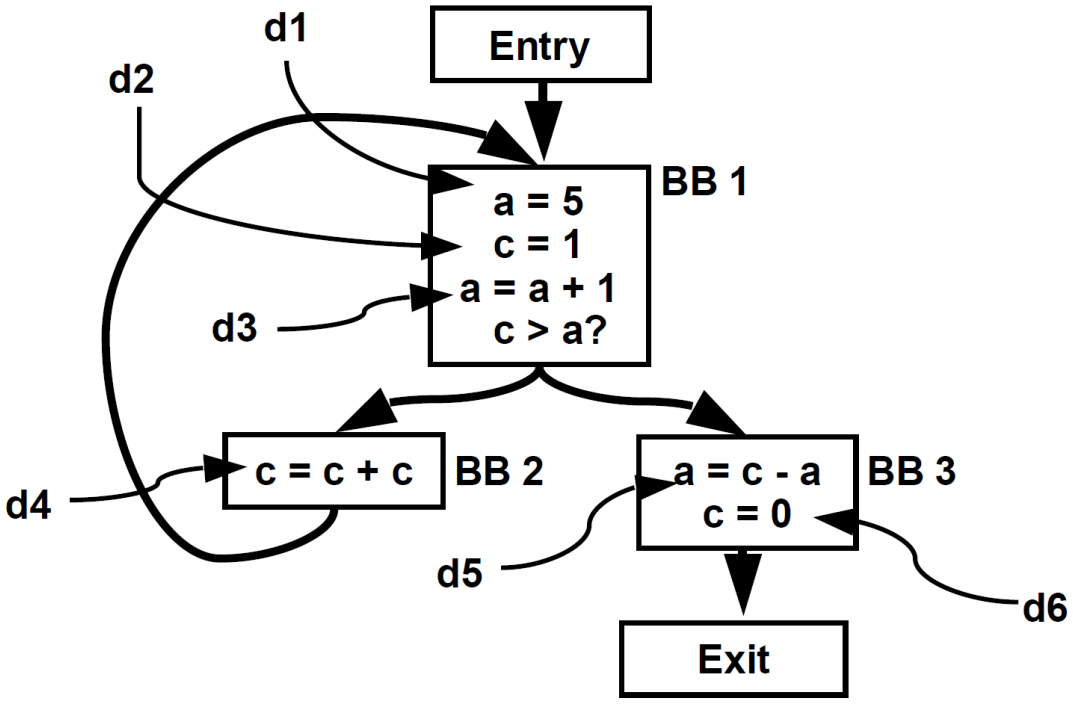
- REACHES(b): Menge der Definitionen, die Block b erreichen.
- Vorwärtsgerichteter Fluß über Zuweisungen an Variablen
- *Reaching Definitions*

Erreichende Definitionen 2



A. Koch

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



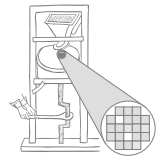
$$\text{DEDEF}(BB1) = \{d2, d3\}$$

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

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

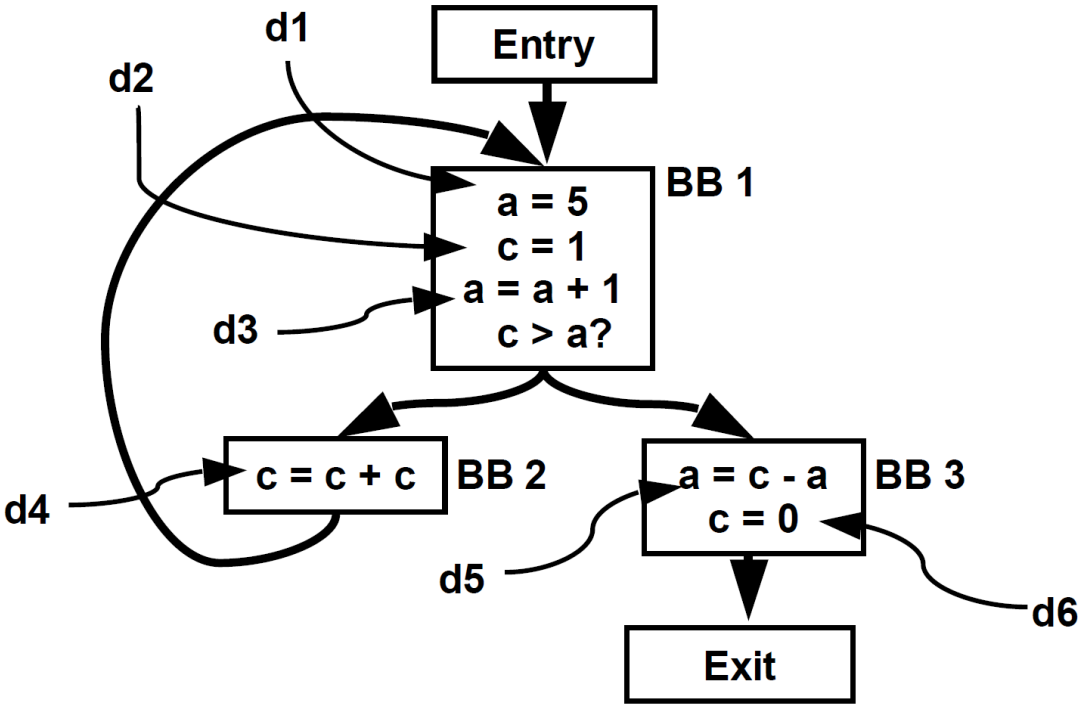
Quelle: UToronto ECE540S Data Flow Analysis

Erreichende Definitionen 3



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

A. Koch



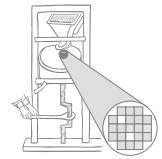
$$\text{DEFKILL}(BB1) = \{d5, d4, d6\}$$

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

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

Quelle: UToronto ECE540S Data Flow Analysis

Erreichende Definitionen 4



A. Koch

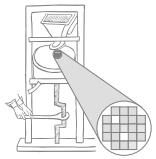
Datenflußgleichungen

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

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

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

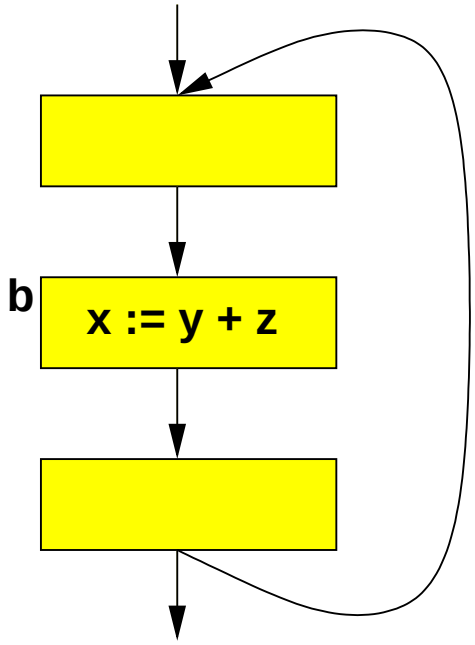
Erreichende Definitionen 5



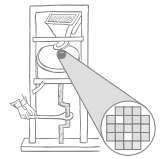
A. Koch

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

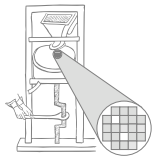


Sehr Rege (*very busy*)

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.

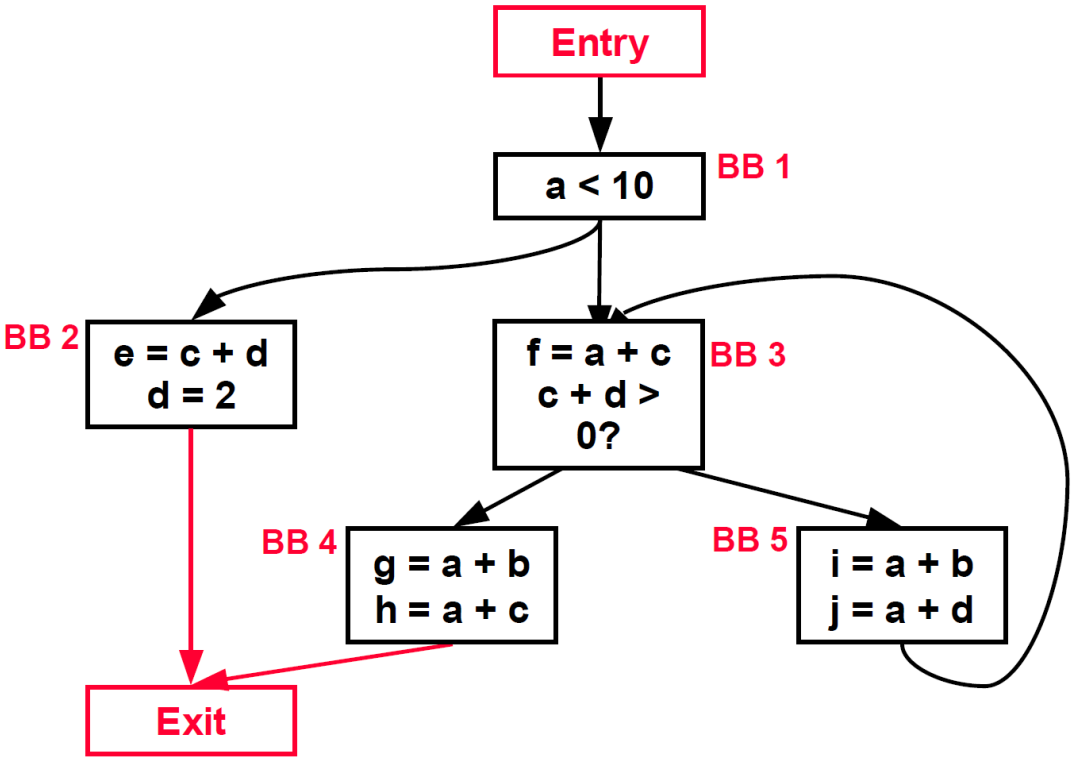
- VERYBUSY(b): Menge der Ausdrücke, die am Ende von b sehr rege sind
- Rückwärtsgerichteter Fluß über Ausdrücke
- *Very Busy Expressions*

Sehr rege Ausdrücke 2



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

A. Koch



$$\text{UEExpr}(BB1) = \emptyset$$

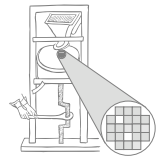
$$\text{UEExpr}(BB2) = \{c + d\}$$

$$\text{UEExpr}(BB3) = \{a + c, c + d\}$$

$$\text{UEExpr}(BB4) = \{a + b, a + c\}$$

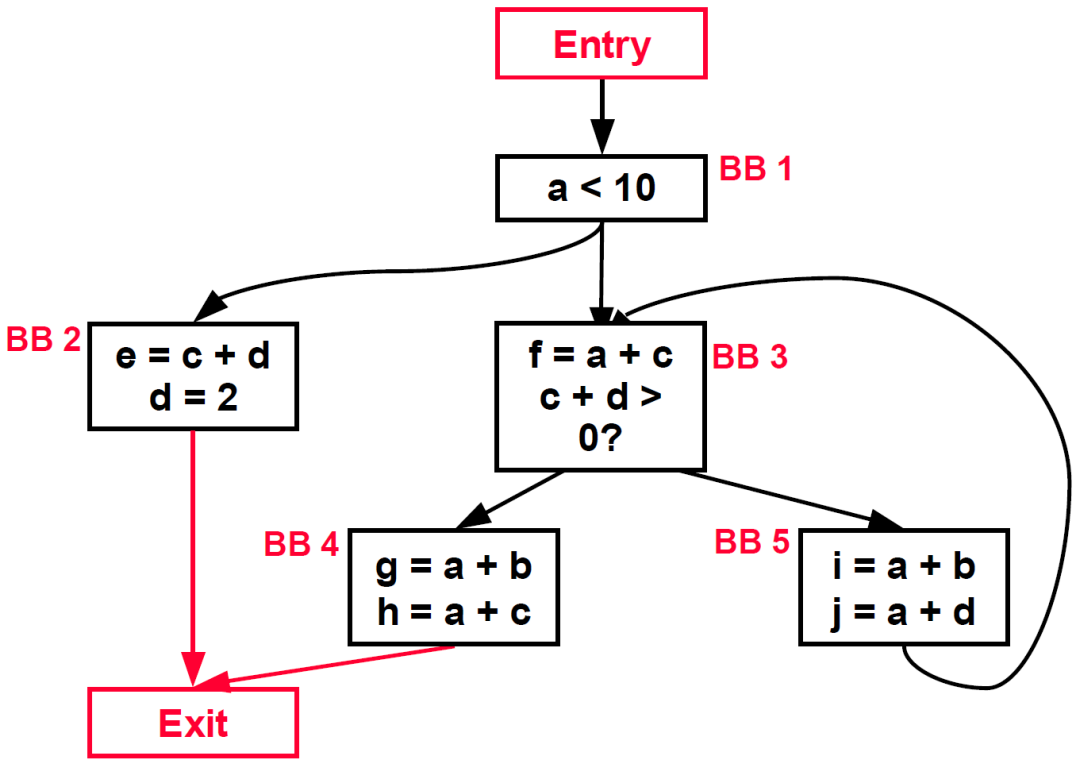
$$\text{UEExpr}(BB5) = \{a + b, a + d\}$$

Sehr rege Ausdrücke 3



EXPRKILL(b): Im Block b durch überschreiben der Operanden unbrauchbar gemachte Ausdrücke
($\rightarrow$ Berechnung von AVAIL)

A. Koch



EXPRKILL$(BB1) = \emptyset$

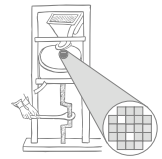
EXPRKILL$(BB2) = \{a+d, c+d\}$

EXPRKILL$(BB3) = \emptyset$

EXPRKILL$(BB4) = \emptyset$

EXPRKILL$(BB5) = \emptyset$

Sehr rege Ausdrücke 4



A. Koch

Datenflußgleichungen

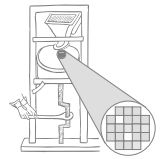
$\textsc{VeryBusy}(b_n) = \emptyset$

$\textsc{VeryBusy}(b) =$

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

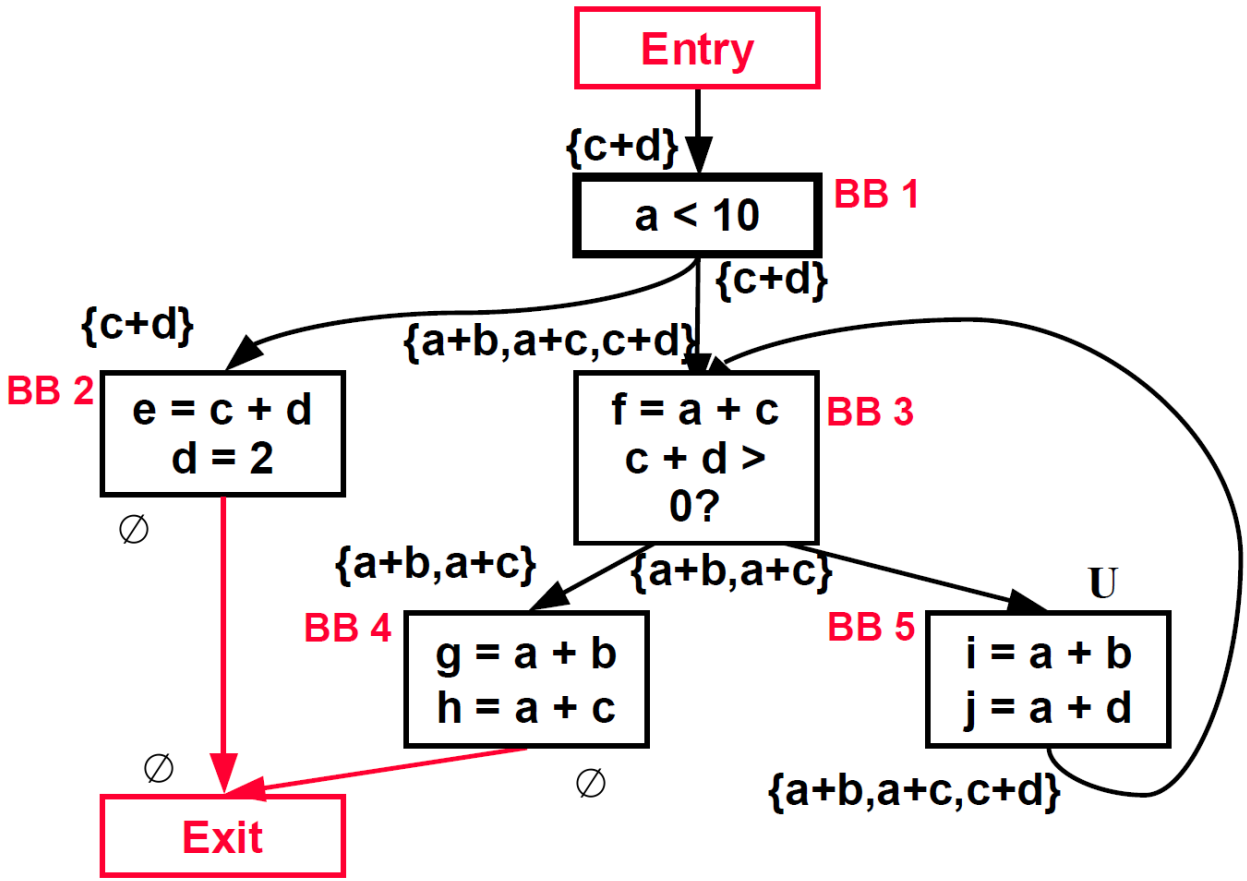
- Lösung mit iterativem Fixpunktverfahren
- Startwert für $b \neq b_n$: $\textsc{VeryBusy}(b) = U$, mit U Menge aller Ausdrücke in Prozedur

Sehr rege Ausdrücke 5



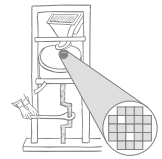
Konkretes Beispiel

A. Koch

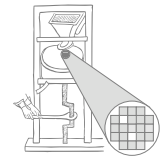


Quelle: UToronto ECE540S Data Flow Analysis

Sehr rege Ausdrücke 6

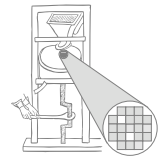


- Anwendung zur Optimierung: Code Hoisting
- Ersetze Evaluationen der sehr regen Ausdrücke in Nachfolgern
- . . . durch eine Evaluation in Vorgänger
- Macht Code nicht (direkt) schneller, aber kleiner



Verallgemeinerung

Gemeinsamkeiten 1



Ein Pfad-Vorwärts: Reaching definitions

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

Ein Pfad-Rückwärts: Live variables

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

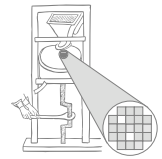
Alle Pfade-Vorwärts: Available expressions

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

Alle Pfade-Rückwärts Very busy expressions

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

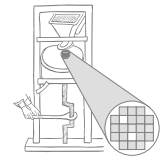
Gemeinsamkeiten 2



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

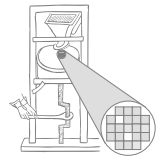
Es gibt aber auch Datenflußprobleme mit anderer Struktur!

Konstanten propagieren 1



- *Constant Propagation*
- Weiterführung von Constant Folding
- Nun hinweg über Anweisungsgrenzen und Merge Points
- Darstellung durch Paare (v, c)
  - v ist Variable
  - c ist entweder Konstante, oder $\perp$ (unbekannter Wert)
- CONSTANTS(b) sind alle bisher gesammelten Aussagen zu Beginn des Blocks b
- Damit darstellbar:
  - Keine Aussage über v machbar: $(v, c) \notin$ CONSTANTS(b)
  - v ist konstant mit Wert c: $(v, c) \in$ CONSTANTS(b)
  - v hat unbekannten (potentiell variablen) Wert: $(v, \perp) \in$ CONSTANTS(b)

Konstanten propagieren 2

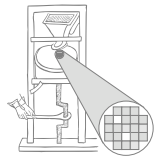


- Anfangs ist $\textsc{Constants}(b) = \emptyset$
- Dann in Reihenfolge Anweisungen in jedem Block b untersuchen

Für `x := y`

if (x,c) $\in$ $\textsc{Constants}(b)$ **do**
$\quad$ $\textsc{Constants}(b)$:= $\textsc{Constants}(b)$ - $\{(\mathrm{x}, c)\}$
if (y,c) $\in$ $\textsc{Constants}(b)$ **do**
$\quad$ $\textsc{Constants}(b)$:= $\textsc{Constants}(b) \cup \{(\mathrm{x}, c)\}$

Konstanten propagieren 3



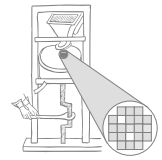
A. Koch

Für **`x := y op z`**

if (x,c) $\in$ CONSTANTS(b) **do**
CONSTANTS(b) := CONSTANTS(b) - $\{$(x, c)$\}$
if (y,c_1) $\in$ CONSTANTS(b) $\wedge$ (z,c_2) $\in$ CONSTANTS(b) **do**
CONSTANTS(b) := CONSTANTS(b) $\cup$ $\{$(x, c_1 op c_2)$\}$

- Mit $\bot$ op $x = x$ op $\bot = \bot$
- Analog **`x := y op`** *Const.*
- Hier auch Sonderregeln möglich
  - $c \cdot 0 = 0$, $c - c = 0$, $c \cdot 1 = c$, ...
- Transformation von CONSTANTS(b) in Block b:
  $\text{CONSTANTS}_{\text{out}}(b) = F_b(\text{CONSTANTS}_{\text{in}}(b))$

Konstanten propagieren 4



Bei Überschreiten von Blockgrenzen:
Mehrere Aussagen $C_{\text{out},b}$ treffen zusammen

Konfluenzoperator ist $\wedge$ (*meets*, Infimum, Durchschnitt)

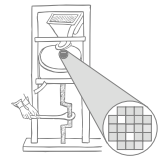
Definition Meets-Operator

$$(v, c_1) \wedge (v, c_2) = \begin{cases} (v, c_1) & : \text{ wenn } c_1 = c_2 \\ (v, \bot) & : \text{ sonst} \end{cases}$$

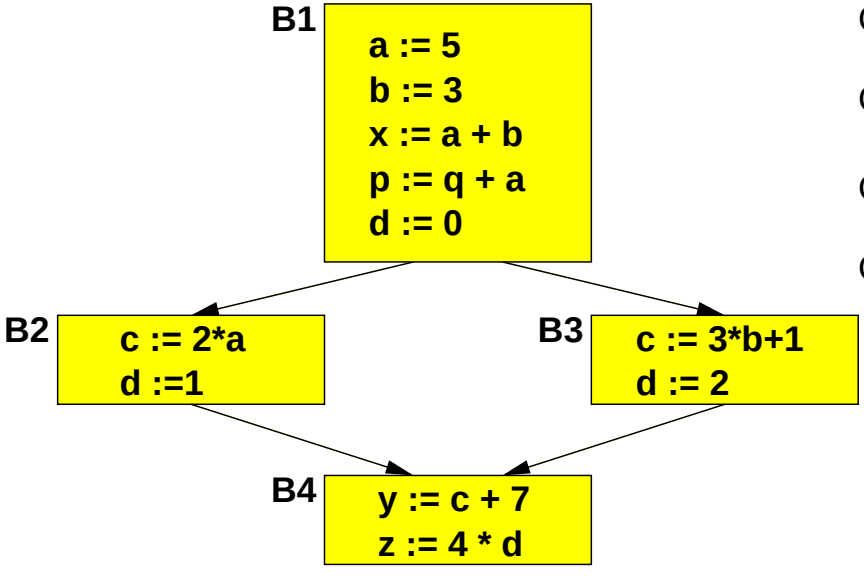
Damit vorwärtsgerichtetes Datenflußproblem formulierbar:

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

Konstanten propagieren 5



A. Koch



Cout(B1)={(a,5),(b,3),(d,0),(x,8)}

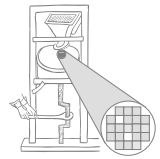
Cout(B2)={(a,5),(b,3),(c,10),(d,1),(x,8)}

Cout(B3)={(a,5),(b,3),(c,10),(d,2),(x,8)}

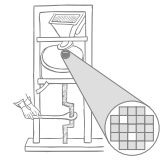
Cout(B4)={(a,5),(b,3),(c,10),(d,⊥),(y,17),(x,8),(z, ⊥)}

- Beachte: Keine Aussage über `p` möglich
- Grund: Keine Aussage über `q` möglich

Konstanten propagieren 6

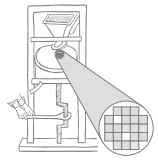


- CONSTANTS(b) kann groß werden, ist aber endlich
- Nur drei aufeinanderfolgende Zustände einer Variable
  1. Keine Aussage
  2. Konstant
  3. Variabel
- Relevanz
  - Hier nur Beispiel für ungewöhnlicheres DF-Problem
- Besser: Sparse Conditional Constant Propagation
  - Ignoriert Einfluß nicht-ausführbarer Blöcke



Zusammenfassung

Zusammenfassung



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