



A. Koch

Optimierende Compiler

6. Einführung in Code-Optimierung

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2006

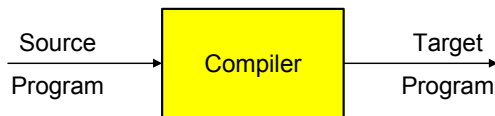
Organisatorisches



A. Koch

- Ab jetzt vorgehen nach Cooper & Torczon
 - Daraus ab jetzt auch die meisten Zeichnungen
- Geplant: Behandle Kapitel 8-10
- Unterstützt durch Papers
 - Werden als PDF auf Web-Seite gelegt

Auswirkung von Optimierung



A. Koch

Optimierer versucht:

- Overhead von hoher Abstraktionsebene der Eingabesprache zu reduzieren
- Eingabeprogramm effizient auf Zielmaschine abzubilden
 - Vertusche Schwächen der Hardware-Architektur
 - Stärken der Hardware-Architektur auszunutzen
- Sonderfälle auszunutzen

➔ Ziel: Effizienz eines erfahrenen Assembler-Programmiers

Ergebnis der Optimierung



A. Koch

Viele Möglichkeiten, entsprechend Vorgabe des Benutzers

- Schnellster Code
- Kleinster Code
- Geringste Anzahl an Speicherzugriffen
- Geringste Anzahl von ausgelagerten Speicherseiten
- ...

➔ Optimierung formt Code entsprechend um

Charakteristika von Optimierungen



A. Koch

Sicherheit Die Bedeutung des Programmes (hier: extern beobachtetes Verhalten) darf **nicht** verändert werden.

Profitabilität Die Optimierung muß in Bezug auf das angestrebte Ziel eine **ausreichende** Verbesserung bringen.

Risiko Welche **negativen** Effekte an die Optimierung nach sich ziehen?

Anwendbarkeit Gibt es **ausreichend** viele Stellen, an denen die Optimierung angebracht werden kann?

Redundante Ausdrücke



Ursprünglicher Code

```
m := 2 * y * z;  
n := 3 * y * z;  
o := 2 * y - z;
```

Umgeschriebener Code

```
t0 := 2 * y;  
m := t0 * z;  
n := 3 * y * z;  
o := t0 - z;
```

A. Koch

Redundanter Ausdruck

Ein Ausdruck $x \text{ op } y$ ist an einer Position L redundant, wenn er in jedem Fall vor L berechnet wurde, und zwischen der Berechnung und L die Operanden x und y nicht verändert wurden.

Beschränkung auf Basisblock



A. Koch

Basisblock (BB)

Längste Folge von Anweisungen ohne Kontrollfluß.

Beispiel:

```
a := b + 42;  
if (a > 23) then  
  c := a - 46;  
  d := b * 15;  
else  
  c := a + 46;  
  d := 0  
  q := false;  
endif
```

Basisblöcke:

```
a := b + 42;
```

```
c := a - 46;  
d := b * 15;
```

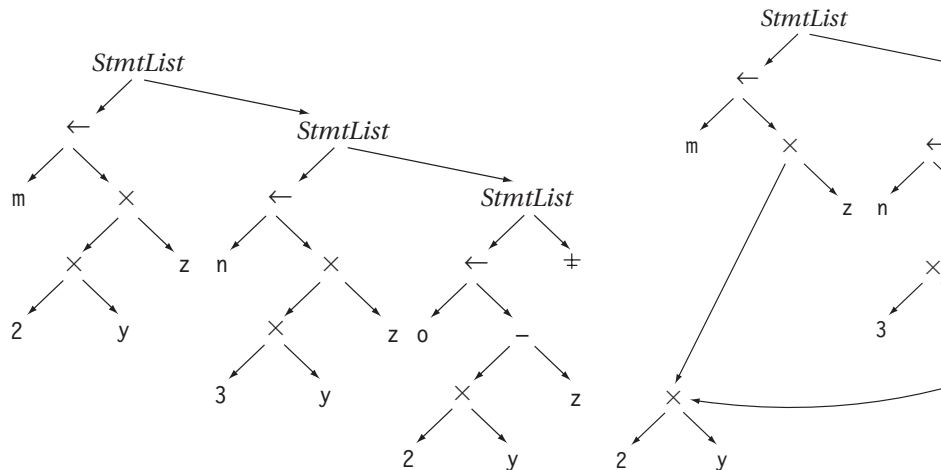
```
c := a + 46;  
d := 0  
q := false;
```

Erster Ansatz: AST-Ebene



Erkennen gleicher Teilbäume, dann Erweiterung auf DAG

A. Koch



Erster Ansatz: Realisierungsidee



A. Koch

- Hashing über ganze Unterbäume
- Gleicher Hash \rightarrow vergleiche Struktur genau
- Wenn gleich: Setze bei redundantem Auftreten Zeiger auf Definition um

Problem:

```
m := 2 * y * z;  
y := 3 * y * z;  
o := 2 * y - z;
```

Einfaches Hashing bei $o := \dots$

- findet Definition von $2 * y$ (bei $m := \dots$)
- sieht aber nicht Zuweisung bei $y := \dots$

Erster Ansatz: Reparatur



Braucht Mechanismus, um Zuweisungen zu beachten!

A. Koch

Idee

- Verwalte unterschiedliche Versionen von Variablen
- Erhöhe Versionsnummer bei jeder Zuweisung
- Hash nun über Variablennamen und Versionsnummer

```
m0 := 2 * y0 * z0;  
y1 := 3 * y0 * z0;  
o0 := 2 * y1 - z0;
```

- Trägt bei m_0 nun als bekannt den Ausdruck $2 * y_0$ ein
- Unterbindet Wiederverwendung als $2 * y_1$

Kleiner Exkurs



Durchnumerieren von Variablenversionen

- **Sehr** Nützlicher Trick!
 - Siehe Static Single Assignment-Form (SSA) später!

A. Koch

Wann wird potentiell neue Variablenversion erzeugt?

- Bei jeder Zuweisung (einfacher Fall!)
- Was bei Zeigeroperationen?
 - Woanders im Programm: `p = &y; ...; *p = 42;`
 - Jeder Zeiger **könnte** eine Variable verändern
- Einzige immer sichere Methode
 - Bei Schreiben via Zeiger Versionen **aller** Variablen erhöhen
- Effizienter: **Beweisen**, dass Zeiger nur auf bestimmte Variablen zeigen kann
 - Nicht trivial (*pointer disambiguation*).

Anderer Ansatz



A. Koch

- Verwaltet nun Folgen von Anweisungen
 - Häufige IR, z.B. in SUIF
- Leichter umzustellen

Idee

- Verwalte nun nicht Variablen**namen** in Ausdrücken
- ... sondern direkt ihre **Werte**
 - Jeder zur Laufzeit berechnete Wert wird durchnumeriert
 - Der Wert zweier Ausdrücke e_1 und e_2 bekommt dieselbe Nummer
 - ... wenn e_1 und e_2 **beweisbar immer** denselben Wert liefern

Value Numbering (VN)



- e_1 und e_2 **redundant**, wenn
 - sie ihre Operanden mit gleichen Operatoren verknüpfen
 - **und** die Operanden die gleichen Wertnummern haben

A. Koch

Implementierung mit Hash-Table

- Hashe Variablen und Konstanten mit Textdarstellung (`.spelling`)
 - Alternativ in Triangle: Adresse der Deklaration im AST
- Hashe ganze Ausdrücke bestehend aus Operatoren und Wertnummern
- Neue Werte (neue Hash-Keys) bekommen neue Wertnummer zugewiesen

Wichtig: Ausdrücke gleichen Hash-Wertes **müssen** zur Laufzeit gleichen Wert liefern.

Value Numbering: Hash-Kollisionen



Wie Kollisionen handhaben?

A. Koch

- Bei einfachen Ausdrücken $x \text{ op } y$ leicht vermeidbar
 - 4b für Operator
 - je 14b für Wertnummern der Teilausdrücke
 - insgesamt ein 32b Wort
 - Eindeutig bis max. 16384 Teilausdrücken
- Bei komplizierteren Ausdrücken
 $a \text{ op1 } b \text{ op2 } c \text{ op3 } \dots$
 - Kollisionen nicht mehr einfach vermeidbar
 - Zweistufiges Vorgehen: Hash+Vergleich
 - Alternative: Aufteilen in einfache Ausdrücke
 - Auch in Triangle (Baum aus Teilausdrücken)
- Hier Beschränkung auf einfache Ausdrücke

Beispiel Value Numbering



A. Koch

```
x := a + d;  
y := a;  
z := y + d;
```

| Ausdruck | ... auf Wertnummern | Wertnummer Ergebnis |
|----------|---------------------|---------------------|
| a | - | 1 |
| d | - | 2 |
| a + d | 1+2 | 3 |
| x | 3 | 3 |
| a | 1 | 1 |
| y | 1 | 1 |
| y + d | 1+2 | 3 |
| z | 3 | 3 |

Value Numbering Algorithmus



A. Koch

Für jeden Ausdruck e der Form $r_e \leftarrow x_e \text{ op}_e y_e$ im Block

- 1 Bestimme Wertnummern von x_e, y_e
- 2 Bestimme Hash-Wert für x_e, y_e, op_e
- 3 Hash-Wert bekannt?
 - Ja: ersetze e durch Kopie von bekanntem Ausdruck, trage dessen Wertnummer für r_e ein.
 - Nein: Trage Hash-Wert neu in Tabelle ein, vergebe neue Wertnummer und trage diese für r_e ein.

Diskussion Value Numbering (VN) 1



A. Koch

- Kommutative Operatoren
 - Schlage mit beiden Operandenreihenfolgen nach
- Auch Constant Folding während VN möglich
 - Flag in Tabelle für “konstanter Wert”
- Berücksichtige algebraische Eigenschaften beim Hashen
 - Viele Sonderfälle, baue Entscheidungsbaum für Operator
 - Auf Wertnummern, *nicht* auf Variablen

Algebraische Eigenschaften

$x \leftarrow y$, $x+0$, $x-0$, $x*1$, $x\div 1$, $x-x$, $x*0$,
 $x\div x$, $x\vee 0$, $x \wedge 0xFF\dots FF$,
 $\max(x, \text{MAXINT})$, $\min(x, \text{MININT})$,
 $\max(x, x)$, $\min(y, y)$, and so on ...

Diskussion Value Numbering (VN) 2



A. Koch

- Schreiben über Zeiger löscht *alle* Wertnummern
- Schreiben auf Arrayelement mit variablem Index
 - Löscht Wertnummern aller Elemente dieses Arrays
- Schreiben auf ganze Record-Variable
 - Löscht Wertnummern aller Komponenten dieser Record-Variable
- Prozeduraufruf
 - Löscht Wertnummern von `var`-Parametern
 - Löscht Wertnummern von globalen und nicht-lokalen Variablen

Beispiel VN mit Umschreiben



A. Koch

Bisher: Erkennen von Redundanzen, jetzt auch
Umschreiben des Codes

Eingabe-Code

```
a := b + c;  
b := a - d;  
c := b + c;  
d := a - d;
```

Value Numbering

```
a3 := b1 + c2;  
b5 := a3 - d4;  
c6 := b5 + c2;  
d5 := a3 - d4;
```

Umschreiben

```
a := b + c;  
b := a - d;  
c := b + c;  
d := b;
```

Probleme



Bisher: Zugriff auf Werte über Namen (von Variablen)

A. Koch

Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

- Zugriff auf Wert 3 über Variablennamen a
- Nicht mehr möglich!
- Möglichkeiten
 - Führe Buch über den Wert haltende Variablen (hier b)
 - Mache Sicherheitskopien von Variablen (a³ nach t³)
 - **Vergebe eindeutige Namen für Zuweisungen**
 - Kein Überschreiben mehr möglich

Eindeutig benannte Zuweisungsziele



Durchnumerieren der LHS-Variablen
(→ Variablenversionen)

A. Koch

Eingabe-Code

$a_0 \leftarrow x_0 + y_0$
* $b_0 \leftarrow x_0 + y_0$
 $a_1 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$

Value Numbering

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow x_0^1 + y_0^2$
 $a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow x_0^1 + y_0^2$

Umgeschrieben

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow a_0^3$
 $a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow a_0^3$

- Wert 3 verfügbar als a_0^3
- Hier etwas mehr Verwaltungsaufwand
- ... echte Probleme kommen noch!
 - Überschreiten von Basisblockgrenzen (→ SSA-Form)

Ist VN sicher?



A. Koch

- Hash-Tabelle beginnt leer
 - Ausdrücke werden bei Durchgehen des Blocks eingetragen
 - Falls $(op, VN(x), VN(y))$ in Tabelle vorkommt
 - Ist Ausdruck mindestens einmal bereits in Block vorgekommen
 - x und y sind nicht neubelegt worden
 - Algorithmus verwendet **Wertnummern** statt Variablen!
- ➔ Falls $(op, VN(x), VN(y))$ eine Wertnummer hat, kann er gefahrlos benutzt werden

Algorithmus

- **beweist** inkrementell, dass $(op, VN(x), VN(y))$ redundant
- modifiziert Code, aber invalidiert nicht Tabelle

Ist VN profitabel?



A. Koch

- Wenn Wiederbenutzung billiger ist als Neuberechnung
 - Übliche Annahme
 - Bei Registermaschinen potentiell problematisch (*register spill*)
- Zusätzliches Constant Folding ist *immer* profitabel
 - Neuberechnung braucht immer 1+ zusätzliche Register
 - Load Immediate braucht genau 1 zusätzliches Register
 - Immediate Instruktion braucht 0 zusätzliche Register

```
ADD R0, #8
```
- Algebraische Eigenschaften
 - Entfernte Operationen sind immer nützlich ($x + 0$)
 - Vereinfachung hängt von Zielmaschine ab ($2*x, x+x$)
 - Kann aber leicht berücksichtigt werden

Wo und wie ist VN anwendbar?



A. Koch

- Potentiell anwendbar auf alle Ausdrücke eines Blocks
- Wie passende Stellen finden?
- Linearer Durchlauf über Block
- Konstruiert Modell des dynamischen Programmzustands
- Bei jeder Operation verschiedene Möglichkeiten prüfen

Zusammenfassung

- VN führt erschöpfende Suche durch
- Folge: Nur begrenzter Rechenaufwand je Operation akzeptabel

Kontrollflußgraph 1



A. Koch

Engl. *control flow graph* (CFG)

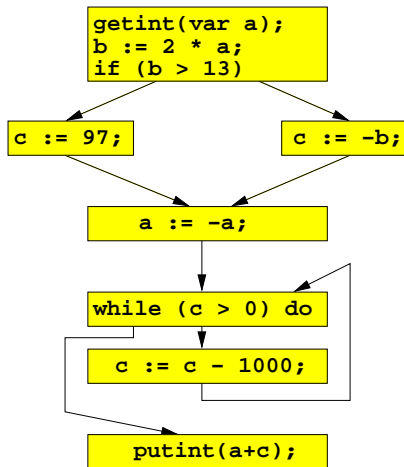
- Knoten sind Basisblöcke
- Kanten sind Sprünge zwischen den Blöcken
- Sprünge treten also nur am Ende eines Blocks auf!
- Sprungziel ist immer ein Blockanfang
 - Triangle: if/then/else, while/do
 - Deutlich komplizierter
 - goto
 - setjmp()/longjmp()

Beispiel Kontrollflußgraph 1



A. Koch

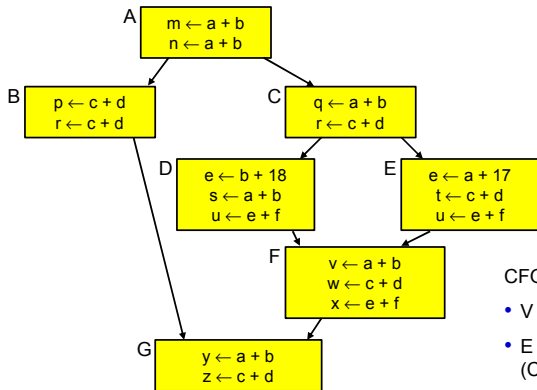
```
getint(var a);  
b := 2 * a;  
if (b > 13) then  
  c := 97;  
else  
  c := -b;  
a := -a;  
while (c > 0) do  
  c := c - 1000;  
putint(a+c);
```



Beispiel Kontrollflußgraph 2



A. Koch



CFG $G = (V, E)$

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



- Arbeiten auf Basisblöcken (BB)
- Für jeden Basisblock gilt
 - Alle Anweisungen werden sequentiell abgearbeitet
 - Falls eine Anweisung ausgeführt wird, werden alle Anweisungen ausgeführt.
- Können sehr genaue Analysen durchführen
- Beweisen dabei i.d.R. stärkere Aussagen als auf größeren Bereichen möglich

Bisheriges VN Vorgehen



Local Value Numbering (LVN)

- Jeweils ein Basisblock betrachtet
- Gute lokale Resultate
- Aber keine Arbeit über Blockgrenzen

A. Koch

Erweitern des Redundanzbegriffes

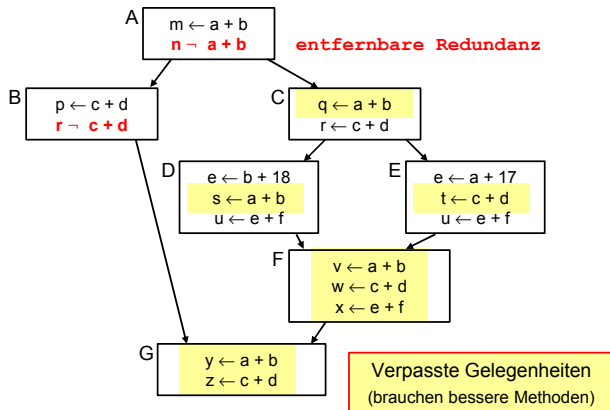
Redundanz über Blockgrenzen

Ein Ausdruck $x \text{ op } y$ ist an einer Stelle L genau dann redundant, wenn er auf **jedem** Pfad vom Startknoten des CFGs zur Stelle L evaluiert worden ist und die Werte seiner Teilausdrücke x und y nicht verändert wurden.

Beispiel LVN



A. Koch



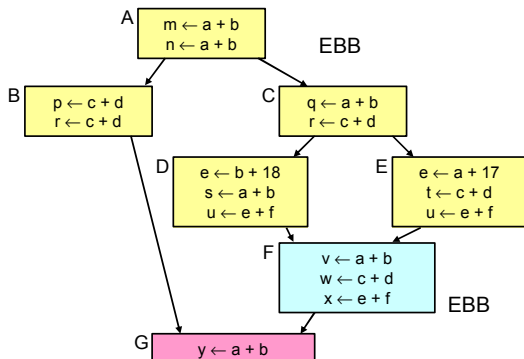
Superlokale Methoden 1



Arbeiten auf Extended Basic Blocks (EBBs)

- EBB $B = \{b_1, b_2, \dots, b_n\}$, mit BBs b_i
- Nur b_1 darf im CFG mehrere oder keine Vorgänger haben
- Alle anderen b_i haben genau einen Vorgänger
- EBB ist Baum aus BBs mit b_1 als Wurzel

A. Koch

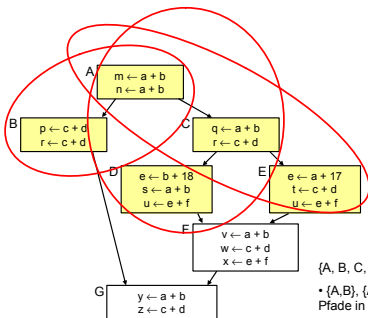


Superlokale Methoden 2



- Idee: Benutze **Pfade** beginnend bei b_1 durch EBB wie einen BB
- Auf jedem Pfad: Genau ein Vorgänger, baue auf dessen Analysen auf

A. Koch



{A, B, C, D, E} ist ein EBB

• {A,B}, {A,C,D}, und {A,C,E} sind Pfade in {A, B, C, D, E}

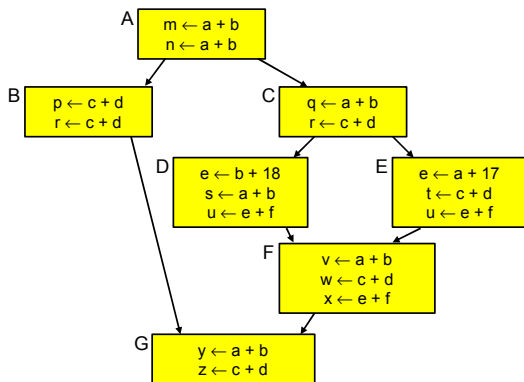
{F} und {G} sind auch EBBs

• ... haben aber nur triviale Pfade

Superlokales VN 1



A. Koch



- Idee: Wende **lokale** Methode auf jeden **Pfad** an
- Hier: Bearbeite (A, B) , (A, C, D) , (A, C, E)
- Baue auf Ergebnisse des Vorgängers auf
- **Aber**: Hilft noch nicht für F und G (woher kommt man?)



- Vermeide mehrfache Neuberechnung gleicher Daten
 - Hier A und C
- Braucht Fähigkeit, Einträge ungültig zu machen
 - Beispiel: Von (A,B) nach (A,C) müssen Daten von B entfernt werden
- Eine Realisierung:
Symbol-Tabelle mit Geltungsbereichen
 - Siehe Kontextanalyse
 - Öffne Geltungsbereich bei Anhängen von Block an Pfad
 - Schließe Geltungsbereich bei Entfernen von Block aus Pfad

Idee SSA



Eingabe-Code

```
a ← x + y
* b ← x + y
a ← 17
* c ← x + y
```

Value Numbering

```
a3 ← x1 + y2
* b3 ← x1 + y2
a4 ← 17
* c3 ← x1 + y2
```

Umgeschrieben

```
a3 ← x1 + y2
* b3 ← a3
a4 ← 17
* c3 ← a3 (oops!)
```

A. Koch

Problem: Zuordnung von Wertnummern zu Namen

- Um überschriebene Namen zu umgehen (siehe Beispiel!)
- Muß auch je Geltungsbereich aktualisiert werden

Besser

- Eindeutige Namen für jede Definition
- Nun eindeutiger Name je Wertnummer



- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle
- Problem: Was, wenn mehrere “letzte” Definitionen?
(z.B. then/else-Zweige: mehrere BBs)



Zusammenführen von mehreren “letzten” Definitionen

- Füge ϕ -Funktionen an Zusammenführungspunkten im CFG ein
- Sogenannte *merge points*
- ϕ -Funktion liefert den Wert mit dem Index der Nummer der letzten genommenen Kante

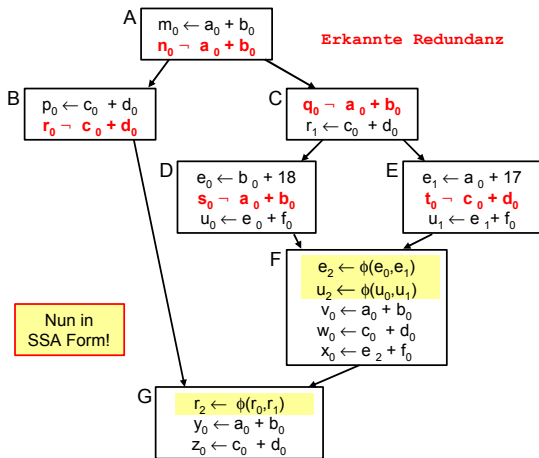


Wie CFG in SSA-Form umwandeln? **Kommt noch ...**

Beispiel Superlokale VN 1



A. Koch



Beispiel Superlokale VN 2



A. Koch

Mehr Redundanzen erkannt, aber nicht alle

- F und G eigene EBBs, hier $a + b$ unbekannt
- $e + f$ berechnet in allen Vorgängern von F, aber mit unterschiedlichen Werten

Erweiterung der Anwendungsbereiche

- VN: Leicht von BB auf EBB erweiterbar
- Klappt aber nicht immer. z.B.
 - Verändern bereits bearbeiteter Blöcke

Noch größere Anwendungsbereiche 1



A. Koch

- Superlokale VN verwirft gesamte Tabelle bei Merge Point
 - Nächste Verfeinerung: Über Merge Points hinweg arbeiten
- ➔ **Regionaler** Anwendungsbereich
- Größer als superlokal
 - Kleiner als gesamte Prozedur

Noch größere Anwendungsbereiche 2



A. Koch

Überlegungen am Beispiel von F: Zusammenführung von D und E

- Keine Beschränkung auf einzelne der Tabellen D oder E möglich
 - Kontrollfluß könnte anderen Pfad nehmen, dann Ausdruck nicht redundant
- Zusammenfügen der Tabellen von D und E
 - Welche Einträge streichen? (z.B. $b+18$ und $a+17$)
 - Liegen auf unterschiedlichen Pfaden
 - Gleiche Ausdrücke in beiden Pfaden
 - Aber mit unterschiedlichen Wertnummern ($e+f$)
 - **Kompliziert** und rechenintensiv

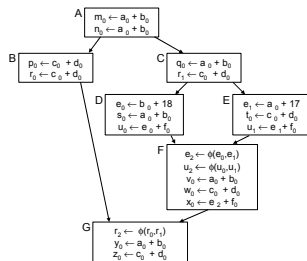
Noch größere Anwendungsbereiche 3



A. Koch

Anderer Ansatz: Beschränke Tabelle auf Fakten, die **unabhängig** vom konkreten Pfad gelten

- Beide Pfade zu F haben gleichen **Präfix** (A,C)
- Alle Operationen in A und C sind **immer** vor Erreichen von F ausgeführt worden
- Letzter "sicherer" Stand ist also Tabelle nach C
- Als Ausgangspunkt für F benutzen



Noch größere Anwendungsbereiche 4



A. Koch

Wie mit Zuweisungen in D und E umgehen?

- SSA-Form umgeht Verlorengehen von Werten
 - Eigene Variable für jeden Wert
 - D und E können **zusätzliche** Werte erzeugen
 - Aber können **keine** alten Werte streichen
 - Konflikte werden über ϕ -Funktionen aufgelöst

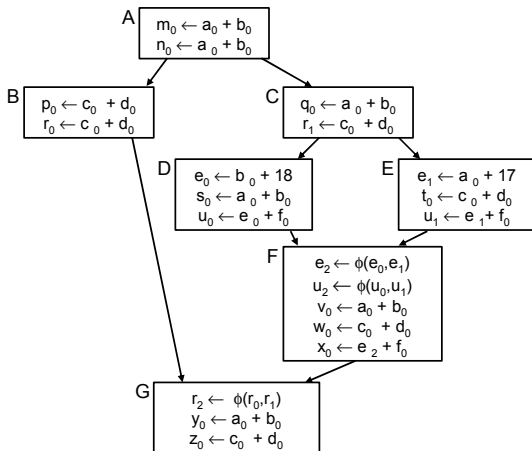
Noch größere Anwendungsbereiche 5



Effekt der Vorgehensweise: Verwende C als Ausgangspunkt für F

A. Koch

- Erkennt jetzt Redundanz von $a_0 + b_0$ und $c_0 + d_0$ in F
- Verpasst aber $e_2 + f_0$, da **zwischen** C und F berechnet



Noch größere Anwendungsbereiche 6

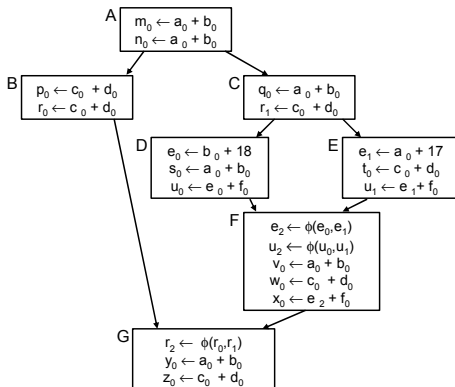


Verfahren benötigt: Letzten gemeinsamen Vorfahren über alle Pfade zu einem Block

Benutze Tabelle von X bei Eintritt in Y: $X \leftarrow Y$

A. Koch

- $- \leftarrow A$
- $A \leftarrow B$
- $A \leftarrow C$
- $C \leftarrow D$
- $C \leftarrow E$
- $C \leftarrow F$
- $A \leftarrow G$



➔ X muß Y dominieren



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 $x \ggg y$.



Dominatoren

$\text{DOM}(y) = \{x \in \text{CFG} \mid x \gg y\}$ ist die Menge der **Dominatoren** von y .

Unmittelbarer Dominator (*immediate dominator*)

$\text{IDOM}(y)$ ist der im CFG y am **nächsten** gelegene Dominator aus $\text{DOM}(y)$.

Dominanz 3

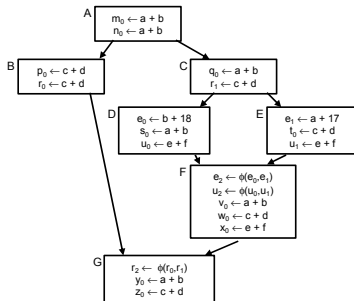


- Sehr nützliche Relation
 - Finden von Schleifen
 - Umwandlung in SSA-Form
 - Zielauswahl für Code-Bewegung

A. Koch

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 ...

Anwendung auf VN



A. Koch

- Geben sei ein Block b
- Jeder Block d in $\text{DOM}(b) - \{b\}$ wurde **vor** b ausgeführt
- Die VN-Tabelle jedes d **könnte** zur Vorbelegung von b verwendet werden
- Welche wäre die beste Wahl?
- Die von Block $e = \text{IDOM}(b)$!
 - e wird von allen anderen Blöcken aus $\text{DOM}(b) - \{b\}$ dominiert
 - Kann all deren Informationen verwenden
 - Hat damit die meisten Informationen

➔ Dominator VN Technique (DVNT oder kurz DVN)



Regionaler Algorithmus

A. Koch

- Rechne superlokalen Algorithmus auf EBBs, verwendet dabei
 - Verschachtelte Hash-Tabellen für Geltungsbereiche
 - SSA-Form
- **Neu:** Initialisiere Tabelle für Knoten x
 - Mit Tabelle von $IDOM(x)$
 - Trage so Wissen über Merge Points hinweg
- Wie vorher möglich:
 - Constant folding
 - Ausnutzung algebraischer Eigenschaften

➡ Größerer Anwendungsbereich sollte zu besseren Ergebnissen führen

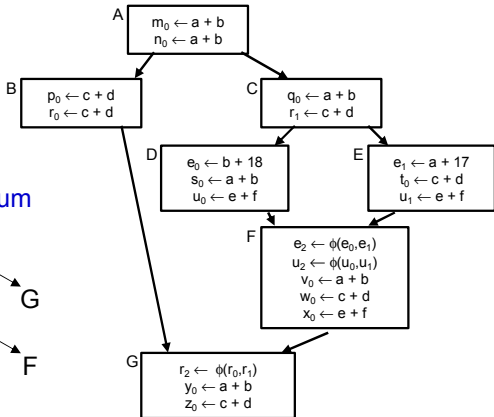
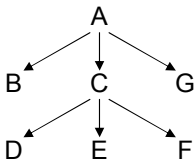


Arbeitet in Prefix-Reihenfolge auf **Dominatorbaum**

- Stellt sicher, das $IDOM(x)$ vor x bearbeitet ist

A. Koch

Dominatorbaum

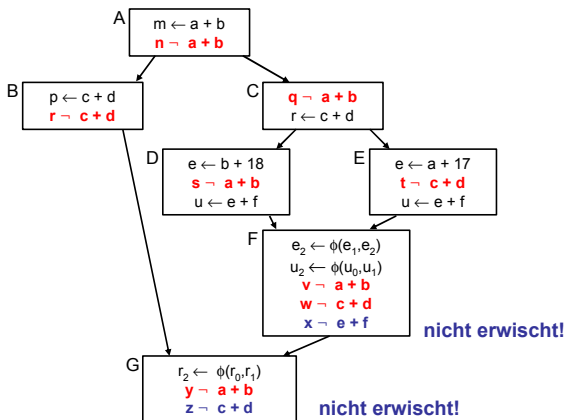


Mögliche Reihenfolge: A, B, G, C, D, E, F

Beispiel DVNT 3



A. Koch



- Erkennt nochmehr Redundanz
- Aber immer noch nicht alle Möglichkeiten
- Scheitert z.B. bei Schleifen
 - Rückwärtskanten im CFG



- Über CFG der **kompletten** Prozedur
- Allgemeine Vorgehensweise: Schritte trennen
- **Analyse** sammelt Informationen, auch über Zyklus hinweg
- **Transformation** erst, wenn alles fertig analysiert



Am Beispiel **Global Common Subexpression Elimination**

- Nun wieder **lexikalischer** Vergleich von **Namen**
- **Keine** Wertnummern mehr
- Keine SSA-Form mehr
- Demonstriert das wichtige Mittel der **Datenflussanalyse**



- **Global Value Numbering**
- Graphenbasierte Vorgehensweise
- Idee: Kongruente Wertgraphen in allen Zweigen des CFG finden
- Hier aber nicht weiter verfolgt
 - ... aus Zeitmangel
 - Siehe Muchnick Abschnitt 13.1.2

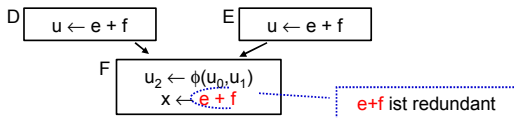
Grundidee



- Eine Auswertung eines Ausdrucks e an der Stelle p ist redundant
- ... genau dann, wenn
 - Jeder Pfad vom Prozedurstart zu p enthält eine Auswertung von e
 - **und** die Werte der Teilausdrücke von e haben sich von dort zu p nicht geändert

A. Koch

➔ Auswertung e an p liefert **gleichen** Wert wie frühere Auswertung(en)



Wie diese redundanten (Teil)ausdrücke finden?



Definition

Ein Ausdruck e ist **definiert** an einer Stelle p im CFG falls sein Wert an der Stelle p berechnet wird. p ist damit eine **Definitionsstelle** von e .



Auslöschung

Ein Ausdruck e wird **ausgelöscht** an einer Stelle p im CFG falls ein oder mehrere seiner Operanden an der Stelle p definiert werden. p ist damit eine **Auslöschungsstelle** von e .



Verfügbarkeit

Ein Ausdruck e ist **verfügbar** an einer Stelle p wenn jeder zu p führende Pfad im CFG, beginnend beim Prozeduranfang, eine vorhergehende Definition von e enthält und e nicht zwischen dieser Definitionsstelle und p ausgelöscht wird.

Identifikation von Ausdrücken



Hier lexikalisch, **nicht** mehr über Wertnummern!

A. Koch

```
x := e + f; // Definition von e+f
e := 5;     // Auslöschung e+f
y := e + f; // nicht redundant!
```

- Identifikation von Variablen über **Namen**
- Identifikation von Ausdrücken über
 - Berechne eindeutigen Hash-Wert über gesamten Ausdruck
 - Operanden**namen** und Operatoren
 - Numeriere Hash-Werte dann aufsteigend durch
- Anzahlen
 - Minimal: Anzahl Variablen plus Anzahl Konstanten
 - Maximal: Anzahl von (Teil)Ausdrücken im CFG



Ziel

Wenn ein Ausdruck e in einem Block **verfügbar** ist, braucht er dort nicht neu berechnet zu werden.

... nun über Basis-Blockgrenzen und Verzweigungen hinweg!



1. Schritt: Analyse

- Formuliere **Gleichungssystem** über den CFG der Prozedur
- Löse Gleichungssystem, um Menge **verfügbarer** Ausdrücke zu bestimmen

$AVAIL(b)$

$AVAIL(b)$ sei Menge der am Anfang von Block b verfügbaren Ausdrücke.



Sicherheit

A. Koch

- $x + y \in \text{AVAIL}(b)$ **beweist**, dass eine **vorherige** Auswertung von $x + y$ existiert
- Spätere Transformation muss über einen **Namen** auf diesen Wert zugreifen können
 - ... auf verschiedene Weisen realisierbar

Profitabilität

- Verursacht keine **zusätzlichen** Evaluationen
- Fügt aber **Kopieroperationen** ein
 - Im Prinzip billig
 - Viele können auch entfernt werden
 - Haben aber Einfluß auf Lebenszeiten (später ...)

Berechnung von $AVAIL(b)$ 1



Über zwei Hilfsfunktionen

$EXPRKILL(b)$

Die Menge der bei Ende des Blocks b ausgelöschten Ausdrücke.

$DEEXPR(b)$ (*downward exposed*)

Menge der im Block b definierten Ausdrücke, die bis zum Ende des Blocks nicht ausgelöscht werden.

Weiterleitung von Ausdrücken:

- Wenn Ausdruck e bei Eintritt in Block b verfügbar ist, und **nicht** in $EXPRKILL(b)$ ist, dann ist e auch nach Ende von b verfügbar.

Berechnung von $AVAIL(b)$ 2



Damit nun definierbar

A. Koch

$$AVAIL(b) = \bigcap_{a \in \text{pred}(b)} \text{DEEXPR}(a) \cup (AVAIL(a) \cap \overline{\text{EXPRKILL}(a)})$$

$$AVAIL(b_0) = \emptyset$$

mit:

$\text{pred}(b)$: Vorgängerblöcke von b im CFG

b_0 Startblock des CFG

➔ Datenflußproblem, lösen mit Standardverfahren



Vor den Details erstmal einen Schritt zurück: Wie soll es weitergehen?

- 1 Berechne $AVAIL(b)$ für alle Blöcke b
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in $AVAIL(b)$
- 3 Innerhalb der Blöcke b dann **lokales Value Numbering**
 - Initialisiert block-lokale Tabelle mit Ausdrücken aus $AVAIL(b)$

Berechnung von AVAIL(b) 3



A. Koch

Baut auf Berechnung von DEEXPR und EXPRKILL auf

assume a block b with operations o_1, o_2, \dots, o_k

VAR KILL $\leftarrow \emptyset$

DEEXPR(b) $\leftarrow \emptyset$

for $i = k$ to 1

 assume o_i is " $x \rightarrow y + z$ "

 add x to VAR KILL

 if ($y \notin$ VAR KILL) and ($z \notin$ VAR KILL) then
 add " $y + z$ " to DEEXPR(b)

EXPR KILL(b) $\leftarrow \emptyset$

For each expression e

 for each variable $v \in e$

 if $v \in$ VAR KILL(b) then

 EXPR KILL(b) \leftarrow EXPR KILL(b) $\cup \{e\}$

assume a block b with operations o_1, o_2

VAR KILL $\leftarrow \emptyset$

DEEXPR(b) $\leftarrow \emptyset$

for $i = k$ to 1

 assume o_i is " $x \rightarrow y + z$ "

 add x to VAR KILL

 if ($y \notin$ VAR KILL) and ($z \notin$ VAR KILL) then
 add " $y + z$ " to DEEXPR(b)

EXPR KILL(b) $\leftarrow \emptyset$

For each expression e

 for each variable $v \in e$

 if $v \in$ VAR KILL(b) then

 EXPR KILL(b) \leftarrow EXPR KILL(b) $\cup \{e\}$

Rückwärts durch

Berechnung von $AVAIL(b)$ 4



A. Koch

- “**Foreach** expression e ”
über alle Ausdrücke der **Prozedur**
- Potentiell sehr langsam!
- Abhilfe
 - Hash-Map M von $v \rightarrow E$ bildet Variable ab auf benutzende Ausdrücke
 - Dann mit v über $VARKILL(b)$ iterieren
 - Für jede Variable v via $M(v)$ Ausdrücke E bestimmen
 - ... und E in $EXPRKILL(b)$ aufnehmen

Berechnung von $AVAIL(b)$ 5



A. Koch

Nun Anwendung eines iterativen Algorithmus zum Finden eines Fixpunktes

$Worklist \leftarrow \{ \text{all blocks in CFG} \}$

while ($Worklist \neq \emptyset$)

 remove a block b from $Worklist$

 recompute $AVAIL(b)$ as

$$AVAIL(b_i) = \bigcap_{x \in \text{pred}(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \overline{EXPRKILL(x)}))$$

if $AVAIL(b)$ changed **then**

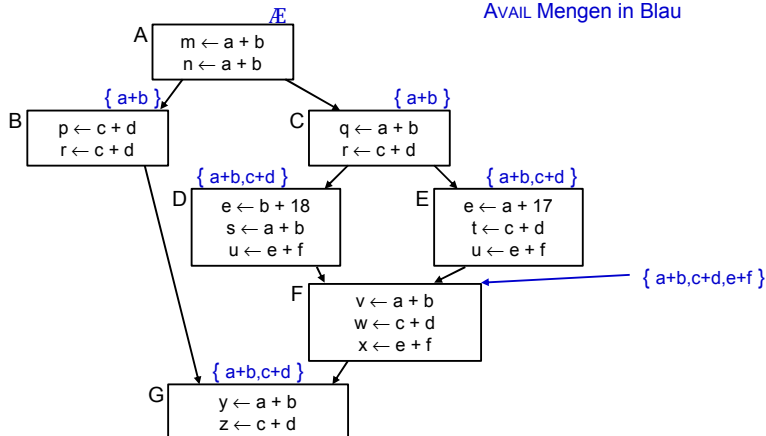
$Worklist \leftarrow Worklist \cup \text{successors}(b)$

Beweis der Terminierung: Später ...

Beispiel mit $AVAIL(b)$ -Mengen



A. Koch



Exkurs: Gewünschter Gesamtablauf



A. Koch

- 1 Berechne $AVAIL(b)$ für alle Blöcke b
 - **Erledigt!**
- 2 Vergebe dann eindeutige, CFG-globale Bezeichner für Ausdrücke in $AVAIL(b)$
- 3 Innerhalb der Blöcke b dann **lokales Value Numbering**
 - Initialisiert block-lokale Tabelle mit Ausdrücken aus $AVAIL(b)$

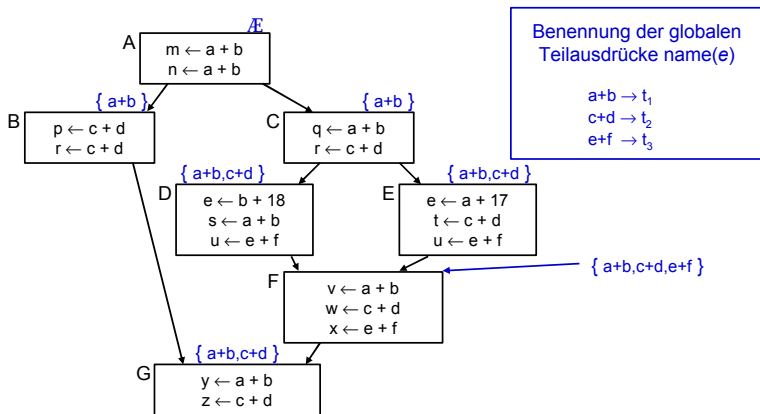
Nun eindeutige Namen t_i für global bekannte (Teil-)ausdrücke e_i vergeben

➔ Hashing über Ausdrücke und Durchnummerieren mit i

Beispiel mit benannten globalen CSEs



A. Koch



Damit jetzt Schritt 3 (Transformation) vornehmen!

Transformation der Ausdrücke 1



A. Koch

Block-lokales Value Numbering in Block b

1. Phase: Finde Wiederbenutzungen eines Ausdrucks

- 1 Initialisiere Hash-Tabelle mit $AVAIL(b)$
 - Falls VN Versionsnummern benutzt:
Ausdrücke aus $AVAIL(b)$ umformen
z.B. $e + f \rightarrow e_0 + f_0$
- 2 Wenn Wiederverwendung $x := e_i$ erkannt
 - Ersetze e durch Verweis auf bekannten Namen t_i :
 $x := t_i$
 - Merke Wiederverwendung durch $USED[e] := true$

Transformation der Ausdrücke 2



A. Koch

2. Phase: Kopien der benutzten Werte unter bekanntem Namen erzeugen

- Für alle Ausdrücke e im Basisblock b
 - Wenn $e \in \text{DEEXPR}(b)$ und $\text{USED}[e]$
 - Füge nach letzter Definition von e in b ein: $t_i := e$

Gesamtergebnis



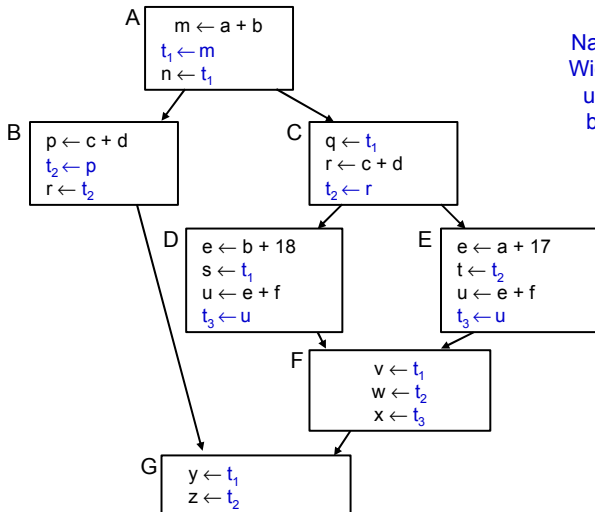
A. Koch

- **Lokale** Redundanzen durch lokales VN beseitigt
- **Globale** Redundanzen durch AVAIL-Mengen beseitigt
- Nicht ganz identischer Effekt, findet
 - Lokale Redundanzen durch Wertgleichheit
 - Globale Redundanzen durch gleiche Schreibweise

Effekt von globalem CSE im Beispiel



A. Koch

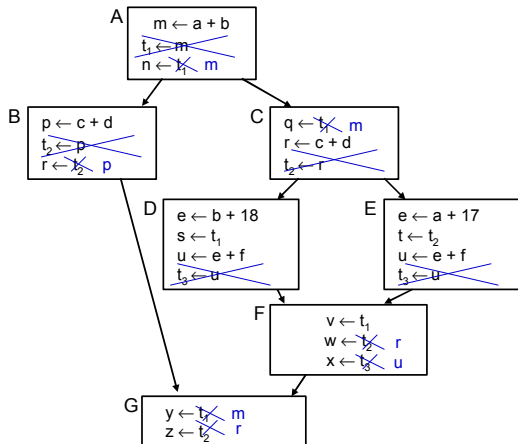


Nach Auffinden von
Wiederbenutzungen
und Kopieren der
benutzten Werte

Unnötige Kopien



A. Koch



Viele der Kopien unnötig,
hier sogar **alle**.

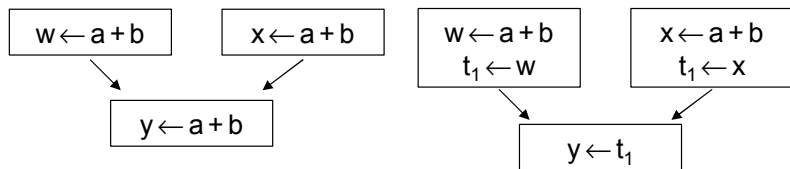
Können später aber entfernt
werden (copy propagation,
copy coalescing)

Gegenbeispiel



Hier wird Kopie gebraucht:

A. Koch



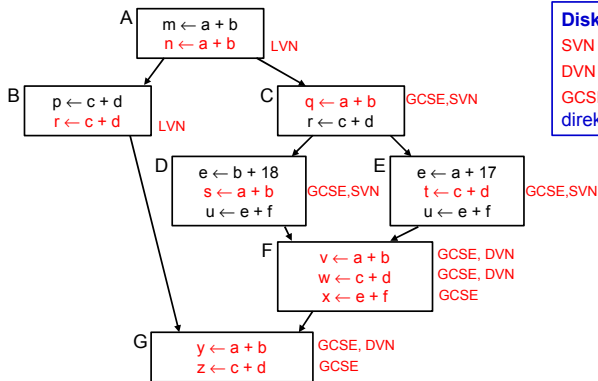
hier kann nicht "w oder x"
geschrieben werden

Unterschied zu letztem Beispiel:
In beiden Zweigen nur **eine** Variable (u).

Vergleich der Methoden 1



A. Koch



Diskussion:

SVN umfasst LVN

DVN umfasst SVN

GCSE & xVN sind nicht
direkt vergleichbar

Vergleich der Methoden 2



A. Koch

- GCSE ist **nicht** zwangsläufig die mächtigste Methode
- Hätte im Beispiel zwar alles gefunden
- Hat aber auch Schwächen gegenüber xVN
- Arbeit mit lexikalischem Vergleich
- Kann z.B. nicht erkennen:
 $(a + b) = (c + d)$, wenn $a = d$ und $b = c$
- GCSE versucht auf **lokaler** Ebene zu kompensieren
 - Verwendung von LVN innerhalb von Basisblöcken



- Charakteristika von Optimierungen
- Redundante Ausdrücke
- Versionen von Variablen (\rightarrow SSA)
- Value Numbering
- Lokal, super-lokal, regional
- Dominatoren
- Global Common Subexpression Elimination
- Datenflußanalyse
- Vergleich der Techniken