

Compiler II

Kontrollflussgraphen und Static Single Assignment-Form

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2012

- Unterbrechung des normalen Compile-Flusses
- Einführung einer neuen Zwischendarstellung
- Stoff der 2. Aufgabe

Ab jetzt Auszüge aus:

A. Koch

Single-Pass Generation of Static Single Assignment Form for Structured Languages

MARC M. BRANDIS and HANSPETER MÖSSENBÖCK

ACM Transactions on Programming Languages and
Systems 16(6): 1684-1698, Nov.1994

- Erzeugung von SSA-Form aus strukturierten Programmierprachen
- Sehr gut zu lesen

Practical Improvements to the Construction and Destruction of Static Single Assignment Form

BRIGGS, COOPER, HARVEY, SIMPSON

SOFTWARE: PRACTICE AND EXPERIENCE, VOL. 28(8),
128 (July 1998)

- Umwandeln aus der SSA-Form (→ nächste Woche)
- Recht gut zu lesen

Efficiently Computing Static Single Assignment Form and the Control Dependence Graph

A. Koch

CYTRON, FERRANTE, ROSEN, WEGMAN, ZADECK

ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 13 , Issue 4 (October 1991)

- *Das SSA-Paper schlechthin*
- Keine ganz einfache Lektüre mehr
- Aber viele Details nur hier behandelt
 - Z.B. Behandlung von Arrays etc.

Kontrollflussgraphen als IR

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;
```

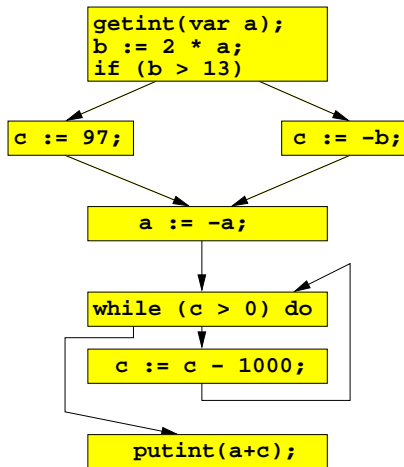
- Basisblöcke alleine **nicht** ausreichend als allgemeine Zwischendarstellung
 - Kontrollfluss fehlt völlig
- Erweiterung auf **Graph** von Basisblöcken
 - Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
 - Kanten symbolisieren **Kontrollfluß**
- Sehr gut für viele Optimierungen brauchbar
- Häufig verwendete Zwischendarstellung im Optimierer

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**
 - In Triangle: if/then/else, while/do
 - Strukturierte Programmierung
 - Allgemeiner Fall deutlich komplizierter
 - goto
 - setjmp()/longjmp()

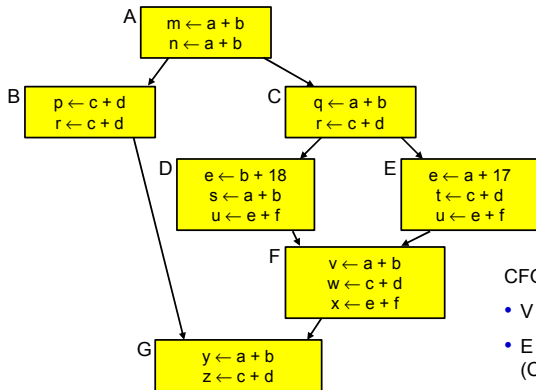
Beispiel Kontrollflußgraph 1

```
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);
```



A. Koch

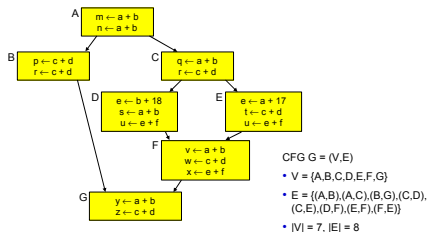
Beispiel Kontrollflußgraph 2



CFG $G = (V, E)$

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

Relationen im CFG



- 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

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*)

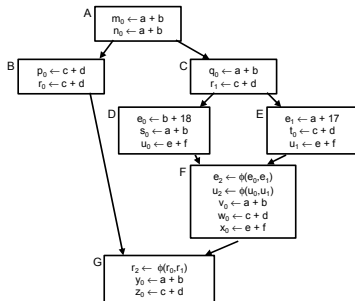
$\text{IDOM}(y)$ ist der im CFG y am **nächsten** gelegene Dominator aus $\text{DOM}(y)$.

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

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 ...

Vorschau Redundanzeliminierung

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;
```

↳ Redundante Berechnung von $a - d$ vermieden

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

Durchnumerieren der LHS-Variablen
(\rightarrow Variablenversionen)

A. Koch

Eingabe-Code

```
a0 ← x0 + y0
* b0 ← x0 + y0
a1 ← 17
* c0 ← x0 + y0
```

Value Numbering

```
a03 ← x01 + y02
* b03 ← x01 + y02
a14 ← 17
* c03 ← x01 + y02
```

Umgeschrieben

```
a03 ← x01 + y02
* b03 ← a03
a14 ← 17
* c03 ← a03
```

- Wert 3 verfügbar als a_0^3
- Hier nur etwas mehr Verwaltungsaufwand
- Aber echte Probleme kommen noch!
 - Beim Überschreiten von Basisblockgrenzen
 - Eine Lösung: **Static Single Assignment-Form** von CFGs

Static Single Assignment-Form

```
// Normal           // SSA-Form
v := 0;            v1 := 0;
x := v + 1;        x1 := v1 + 1;
v := 2;            v2 := 2;
y := v + 3         y1 := v2 + 3
```

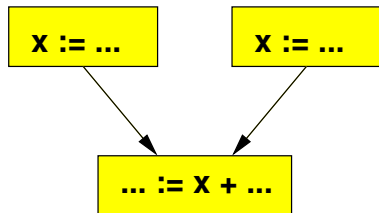
A. Koch

- Zur Compile-Zeit (also **statisch**)
- Jeder Wert wird an genau **eine eigene** Variable zugewiesen
 - Erzeuge eindeutige Namen für gleiche Zuweisungsziele
 - Numerierte Variablen sind **Wertinstanzen** der ursprünglichen Variablen
- Jeder Operand hat somit **genau eine** Definition in BB
- Letzte Definition ist die aktuelle

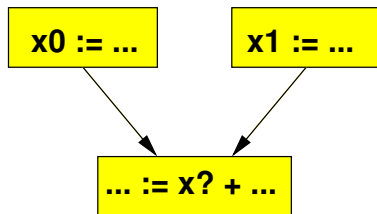
- Was, wenn mehrere “letzte” Definitionen? (z.B. then/else-Zweige: mehrere BBs)
- Sogenannte *merge points*
- Zusammenführen von mehreren “letzten” Definitionen

A. Koch

Ursprünglicher CFG



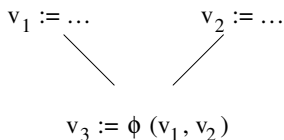
“Letzte” Definition von x ?



Was passiert, wenn zwei Werte der gleichen Variable aufeinanderstoßen?

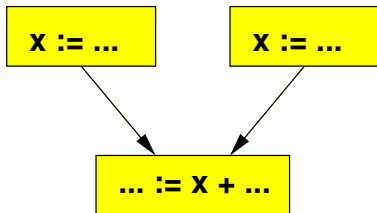
- An sogenanntem *merge* oder *join*-Punkten im Kontrollflußgraphen

➔ Auflösung über Phi-Funktion

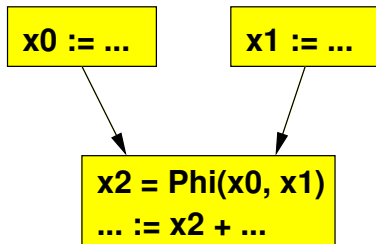


- Für jeden Kontrollzweig einen Parameter
 - Den jeweiligen Wert
- Liefert als Ergebnis den Wert entsprechend der genommenen Kante
 - Von welchem Zweig kamen wir?
 - Welcher Wert ist also der richtige?

Ursprünglicher CFG



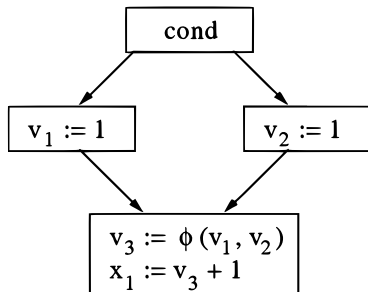
Auflösung durch ϕ -Funktion



Beispiel SSA-Form: IF-Statement

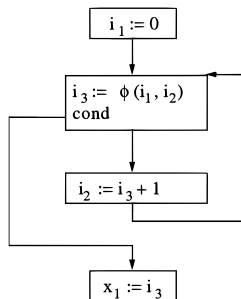
```
IF cond THEN  
    v := 1  
ELSE  
    v := 2  
END;  
x := v + 1
```

A. Koch



Beispiel SSA-Form: WHILE-Statement

```
i := 0;  
WHILE cond DO  
  i := i + 1  
END;  
x := i
```



Beachte: Entscheidung, ob Wert von **vor** oder **nach** dem Schleifenkörper genommen wird.

- Für jeden Wert genau eine Definition
- Jede Zuweisung legt neuen Wert an
- Kein Auslöschen (*kill*) von Werten möglich
- Wenn zwei Ausdrücke textuell gleich sind
- ... liefern sie das gleiche Ergebnis

Drei Teilprobleme

- 1 Eindeutige Namen für Werte
 - Einfach durchnummerieren
- 2 Einfügen von Phi-Funktionen
 - Holzhammermethode
 - An jedem join-Point für **alle** Variablen Phi-Funktionen einfügen
 - Erzeugt **sehr viele** Phi-Funktionen, die meisten unnötig
- 3 Umbenennen von benutzten Variablen in passende Werte
 - Wieder recht einfach
 - Referenziert letzte Definition

Allgemeine Lösung

- Cytron et. al. 1991
- Vorgehen: Berechnen von Dominatorgrenzen
- “Gerade nicht mehr” von Knoten X dominierte Knoten
- Hier nicht mehr klar, ob Definitionen aus X noch gelten
- Einfügen von Phi-Knoten nur für die Variablen, bei denen entschieden werden muß
 - Aufeinandertreffen von **verschiedenen** Definitionen an Dominatorgrenzen
- Algorithmus nicht trivial ...

- Keine GOTOs
- Nur strukturierte Anweisungen
 - IF
 - CASE
 - WHILE
 - REPEAT
 - FOR

➡ Viel einfacheres und schnelleres Vorgehen möglich

➡ Brandis/Mössenböck 1994

Unser Ansatz für Triangle!

Aus Zeitgründen in der Vorlesung keine **detaillierte** Behandlung von

- Arrays
- Records
- Prozeduraufrufen
- Verschachtelten Geltungsbereichen

Alles handhabbar

... aber aufwändig und lenkt von Kernideen ab.

Bei Interesse (oder Bedarf!): Cytron et al., Abschnitt 3.1

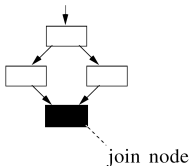
<i>Assignments (original form)</i>	<i>Assignments (SSA form)</i>	<i>Current values</i>	
		<i>v</i>	<i>x</i>
		v_0	x_0
$v := 0;$	$v_1 := 0;$	v_1	x_0
$x := v + 1;$	$x_1 := v_1 + 1;$	v_1	x_1
$v := 2$	$v_2 := 2$	v_2	x_1

- Jede Zuweisung an v erzeugt neuen Wert v_i
- Nach Zuweisung ist v_i **aktueller** Wert von v
- Ersetze alle **folgenden** Verwendungen von v durch v_i
- Verwaltung z.B. in extra Tabelle während Umformung

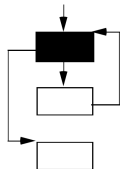
Bei strukturierten Programmiersprachen:

Alle Join-Knoten sind durch Konstrukte bereits vorgegeben

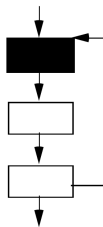
IF, CASE



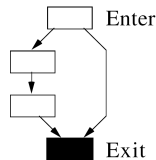
WHILE, FOR



REPEAT



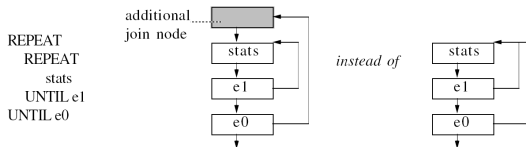
procedure



Platzhalter
(siehe später!)

- Strukturen können verschachtelt sein
 - Bearbeite von **innen** nach **aussen**
 - Innerster Join-Knoten ist **aktueller** Join-Knoten
- Erzeuge keine **speziellen** Knoten für Joins
- Verwende bisherige Blöcke weiter

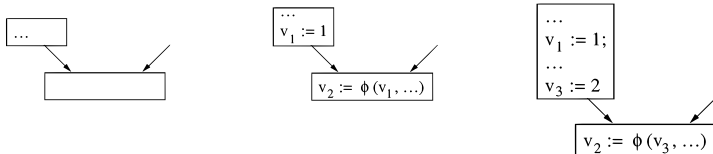
Ausnahme: Verschachtelte REPEAT-Anweisungen



- Für spätere Optimierung hilfreich
- Sonst kein Ziel für aus der inneren Schleife bewegte Berechnungen

- **Jede** Zuweisung gehört zu einem Zweig des Kontrollflußgraphen
- Jede Zuweisung erzeugt einen neuen Wert
 - Ggf. auch bei Prozeduraufruf (`var`, global, nicht-lokal)
- Irgendwann trifft der Wert auf einen **Join-Knoten**
- Dort **Unterscheidung** zwischen allen Werten für diese Variable

➔ Jede Zuweisung **erzeugt** oder **modifiziert** Phi-Funktion für Variable

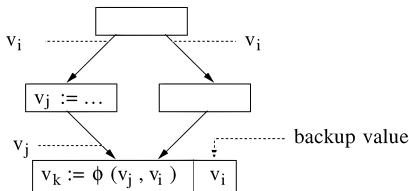


- Phi-Operand entsprechend dem **bearbeiteten** Zweig
- ... wird jeweils auf **letzten aktuellen** Wert gesetzt
- Phi-Funktionen treten **selber** in Zuweisungen auf
- Erzeugen also selber **neue** Werte
- Führen zu **weiteren** Phi-Funktionen in **nächstäußerem** Join-Knoten
- Ende bei Erreichen des Exit-Knotens

Vorgehen: Erzeugen eines CFGs in SSA-Form **je Prozedur** durch Traversieren des **ASTs**

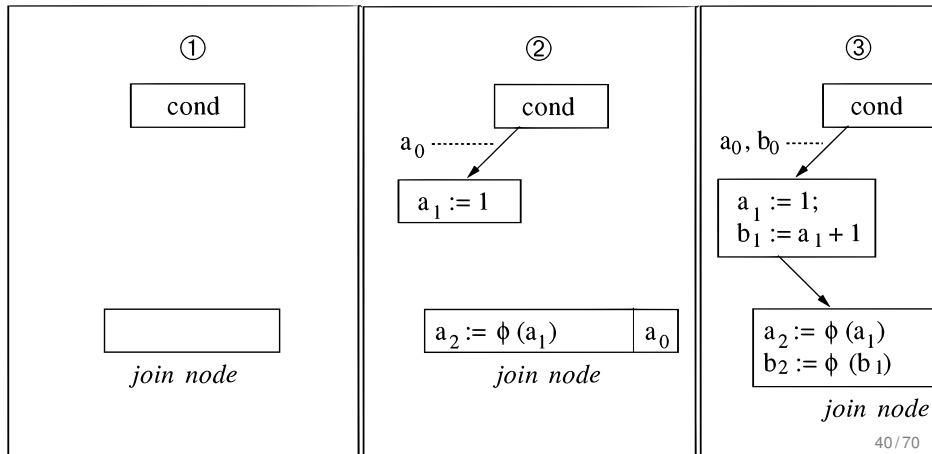
Könnte aber auch direkt beim Parsen geschehen.

- 1 Bei Erreichen von IF: Erzeuge neuen Join-Knoten
 - Wird Phi-Funktionen aus THEN/ELSE enthalten
 - Wird später in den CFG eingehängt
- 2 Bearbeite THEN-Zweig, für eine Zuweisung an v
 - 1. Mal: Lege **leere** Phi-Funktion (Identität) für v an, sichere Wert v_i **vor** IF zusammen mit Phi-Funktion
 - Sonst: Setze Phi-Operand auf jeweils **aktuellen** Wert v_j
- 3 Bearbeite ELSE-Zweig
 - Setze **aktuelle** auf **gesicherte** Werte (pre-IF) zurück
 - Dann gleiches Vorgehen wie im THEN-Zweig



Phi-Knoten für IF-Anweisungen 2

①
IF cond THEN a := 1; b := a + 1
②
ELSE a := a + 1; c := 2
③
END



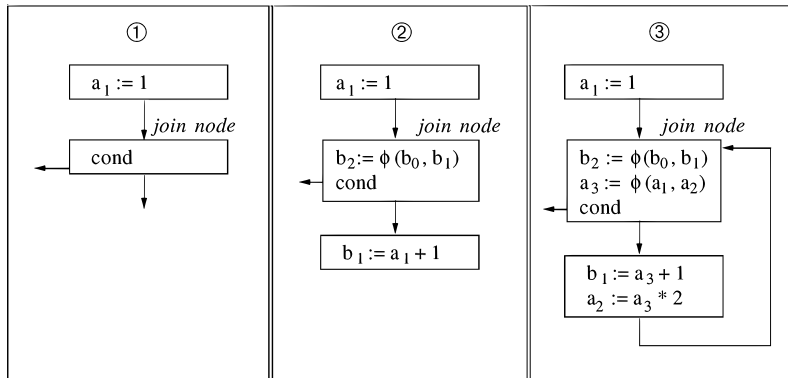
Nach Abarbeiten von THEN und ELSE-Zweigen:
Festlegen des Join-Blocks (*commit*)

- Join-Block **selber** bearbeiten
- Werte **Zuweisungen** von Phi-Funktionen aus
- Trage neue Phi-Funktionen in **nächstäußeren** Join-Block ein
 - Join-Block der umschließenden Kontrollstruktur
- Trage dort LHS der Phi-Zuweisungen als aktuelle Werte der Variablen ein
- Hänge aktuellen Join-Block in CFG ein

- Join-Knoten von WHILE-Anweisung ist **Kopfknoten**
 - Zusammentreffen von Schleifeneintritt und Rückwärtskante im CFG
- Bearbeitung des Schleifenkörpers analog zur IF-Anweisung, **aber**
- Bei Eintragen einer neuen Phi-Funktion in Kopfknoten
- ... entsteht **neuer** aktueller Wert
- Alle lesenden Benutzungen der Variable **im Schleifenkörper** durch **aktuellen** Wert ersetzen
 - Verwalte Liste aller im Schleifenkörper benutzten Werte
 - Sogenannte *use chain*
 - Kann für schnelle Korrektur (Ändern der Versionsnummer) benutzt werden

Phi-Knoten für WHILE-Anweisungen 2

①
②
③
 $a := 1; \text{WHILE cond DO } b := a + 1; a := a * 2 \text{ END}$



Beachte: Ersetzung von a_1 durch a_3 !

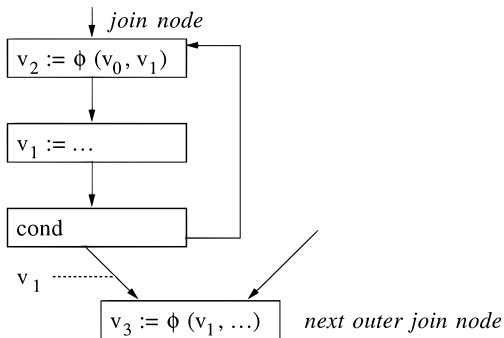
- Nach der Bearbeitung des Schleifenkörpers
- ... **Festlegen** der Phi-Zuweisungen im Join-Knoten
- Erzeugt neue Phi-Funktionen in nächstäußerem Join-Knoten
- Legt neue aktuelle Werte für nachfolgende Anweisungen fest
 - Im Beispiel: a_3 und b_2

CASE und FOR würden analog zu IF und WHILE bearbeitet

- Sonderfall!
- Konstrukt wird **nicht** über Join-Knoten verlassen
- Analog zu WHILE: Join-Knoten ist Schleifenkopf
 - Hier auch Phi-Zuweisungen untergebracht
- Aber Unterschied beim **Festlegen** des Join-Blocks!
- Aktueller Wert ist **nicht** Ziel der Phi-Zuweisung im Kopf (wie bei WHILE)
- ... sondern Wert zugeordnet der **Rückwärtskante**
 - Sonst wären Änderungen nach genau einem Schleifendurchlauf nicht sichtbar
- Gleichen Wert auch für Operanden **nächstäußerer** Phi-Funktion verwenden

Phi-Knoten für REPEAT-Anweisungen 2

REPEAT
 $v := \dots$
UNTIL cond



Beachte: Weiterverwendung von v_1 , **nicht** von v_2

- INSERTPHI

- Erzeugt neue oder modifiziert bestehende Phi-Zuweisung in Join-Knoten b
- Aufruf: $\text{INSERTPHI}(b, i, v_i, v_{old})$
 - Zur Bearbeitung von Zuweisung $v_i := \dots$
 - \dots die im i -ten, zum Block b führenden Zweig steht
 - v_{old} ist aktueller Wert **vor** dieser Zuweisung
 - Wird als Sicherheitskopie abgespeichert

- COMMITPHI

- Legt die Phi-Zuweisungen in einem Join-Knoten b fest
- Bestimmt **aktuelle** Werte
- Propagiert neue Phi-Zuweisungen in nächstäußeren Join-Knoten B , über die Kante I kommend

```
PROCEDURE InsertPhi (b: Node; i: INTEGER; vi, vold: Value);  
BEGIN  
  IF b contains no  $\phi$ -assignment for v THEN  
    Insert " $v_j := \phi(v_{old}, \dots, v_{old}) / v_{old}$ " in b;  
    IF b is a join node of a loop THEN  
      Rename all mentions of vold in the loop to vj  
    END  
  END;  
  Replace i-th operand of v's  $\phi$ -assignment by vi  
END InsertPhi;
```



```
PROCEDURE CommitPhi (b: Node);  
BEGIN  
  FOR all  $\phi$ -instructions " $v_i := \phi (v_0, \dots, v_n) / v_{old}$ " in b DO  
    IF b is a join node of a repeat THEN val :=  $v_n$  ELSE val :=  $v_i$  END;  
    Make val the current value of v;  
    InsertPhi(B, l, val,  $v_{old}$ )  
  END  
END CommitPhi;
```

Hier Annahme: Letzter Zweig n ist Rückwärtskante der REPEAT-Schleife

- Hier nicht gezeigt: Rücksetzen auf v_{old} bei Bearbeitung des nächsten Zweiges
- Variablen in Triangle durch Verweise auf Definitionen kennzeichnen
- Keine String-Vergleiche mehr nötig!
- Werte sind dann Tupel (Definition, Versionsnummer)
- Prozeduraufrufe wie Zuweisungen behandeln
 - LHS: **var**-Parameter, **geschriebene** nicht-lokale und globale Variablen
 - RHS: Parameter (**var** und Wert), **gelesene** nicht-lokale und globale Variablen

```
let
  var f : Integer;
  var g : Integer;
  var n : Integer;
  proc p() ~ begin f := 2*f; g := g+1 end
in begin
  n := 1; f := 2; g := 3;
  while n < 10 do begin
    p();
    n := n + 1
  end;
  putint(f); puteol(); putint(g)
end
```

A. Koch

- Sehe $p()$ an als $\{f, g\} = p \{f, g\}$
- RHS: Operator p , angewandt auf Werte f und g
- LHS: Erzeuge neue Versionen von f und g

- $\{f3, g3\} = p() \{f2, g2\}$ in Schleife
- Im Kopfknoten nun:
 $f2 = \text{Phi}(f1, f3)$
 $g2 = \text{Phi}(g1, g3)$
 $n2 = \text{Phi}(n1, n3)$
 $n2 < 10$
- Details in Cytron, Abschnitt 3.1
 - Arrays, Records, Prozeduren und Funktionen
 - Besser als nachlesen: Idee verstanden haben :-)

Rückwandlung aus SSA-Form

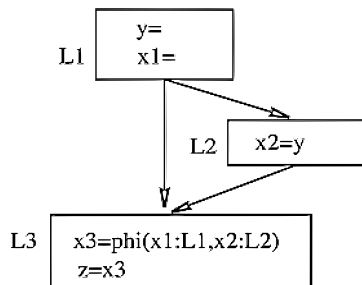
- Normale Prozessoren haben keine Phi-Instruktion
- Phi-Instruktionen müssen entfernt werden

Rückwandlung aus SSA-Form 2

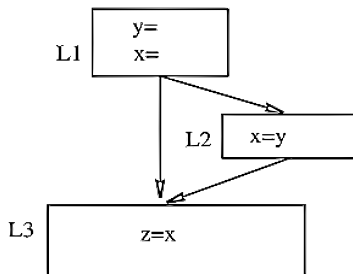
Naive Idee: Phi einfach löschen und Wertnummern entfernen

A. Koch

Vorher:



Nachher:



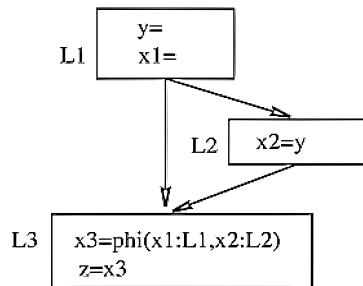
... so weit, so gut.

Rückwandlung aus SSA-Form 3

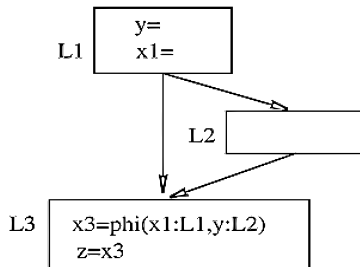
Jetzt Annahme: Einfache Optimierung hat stattgefunden

A. Koch

Vor Copy-Propagation



Nach Copy-Propagation



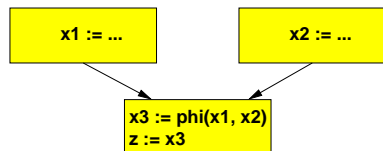
Rückwandlung durch einfaches Löschen ... geht schief:

- Phi-Funktion auflösen nach x oder y?

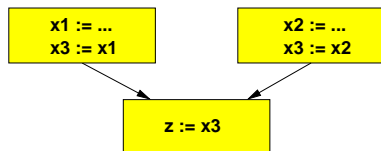
Rückwandlung aus SSA-Form 4

Besserer Ansatz: Füge Kopieroperationen in
Vorgängerblöcke der Phi-Funktion ein

Vorher



Nachher



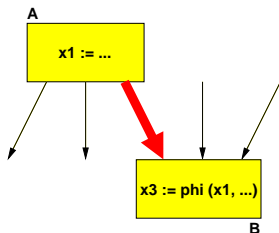
Zielführender als naives Löschen!

Problemfall: Kritische Kanten

A. Koch

Kritische Kontrollflusskante

Eine kritische Kante im CFG verläuft von einem Block mit mehreren Nachfolgern zu einem Block mit mehreren Vorgängern.

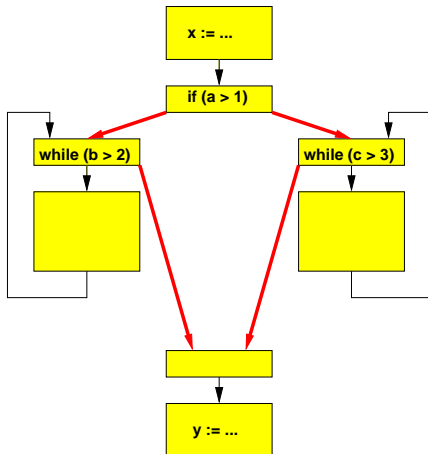


Rückwandlung aus der SSA-Form 6

Praktisch: Können kritische Kanten in strukturierten CFGs à la Triangle auftreten?

A. Koch

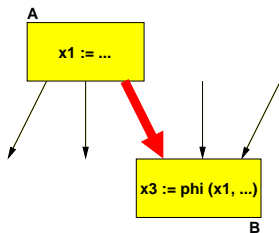
```
x := ...  
if (a > 1) then {  
  while (b > 2) do {  
  }  
} else {  
  while (c > 3) do {  
  }  
}  
y := ...
```



Problem bei kritischen Kanten

- Wo Kopierzuweisungen von A bei Auflösen der Phi-Funktion in B unterbringen?
- Am Ende von A?
 - Nicht effizient, da dann alle Nachfolger von A die Kopie für B bekommen!
- Am Anfang von B?
 - Geht nicht, da dann alle Vorgänger von B die Kopie von A bekommen!

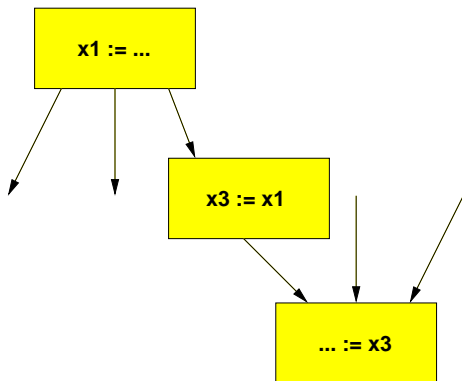
A. Koch



Rückwandlung aus der SSA-Form 8

Einfache Lösung:
Kante aufspalten und neuen Block einfügen!

A. Koch



Funktioniert immer!

Nachteil: Verlangsamt möglicherweise Programm

- Beispiel: Zusätzliche Sprunganweisung bei REPEAT/UNTIL

Abhilfe: Gezielteres Einfügen von Kopien

- Briggs 1998 oder Sreedhar 1999

Kommt noch in eigener Vorlesung!

- Aber nicht alle kritischen Kanten sind relevant
- Nur solche **vor** Blöcken mit phi-Funktionen

Damit einfache Vorgehensweise zur Rückwandlung

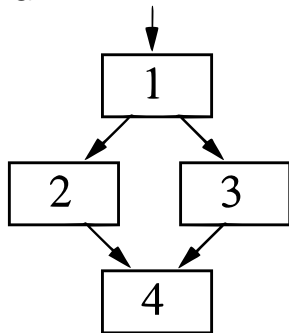
- Teile phi-Funktion in Kopieranweisungen auf
- Lege Kopieranweisung am Ende des entsprechenden Vorgängerknotens ab
- **Es sei denn**, dass Kante zum Vorgänger kritisch ist
- **Dann** Kante aufspalten, Kopieranweisung in eingefügten Knoten legen

Berechnung von Dominatoren

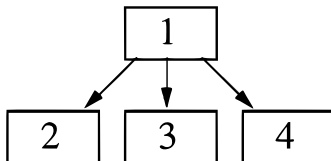
- Muß bei Cytron et al. bei der SSA-Umformung gemacht werden
- War hier nicht nötig
- Dominatoren sind aber nach wie vor nützlich
- Wie sind sie hier berechenbar?
- Viel einfacher als im allgemeinen Fall!

- Auch hier Berechnung in einem Pass möglich
 - Über Quelltext oder AST
- Dominatorbaum
 - Vater eines Blocks ist dessen unmittelbarer Dominator
IDOM
- Idee hier: Sub-CFGs der Konstrukte
IF/WHILE/FOR/REPEAT/CASE
- ... haben **einen** Eintrittspunkt und **einen** Austrittspunkt
- Der Eintrittspunkt dominiert **alle** Knoten des Konstrukts
- Unmittelbare Dominatoren können immer nach dem gleichen Schema bestimmt werden
- Dann Hochhangeln für weiter entfernte Dominatoren

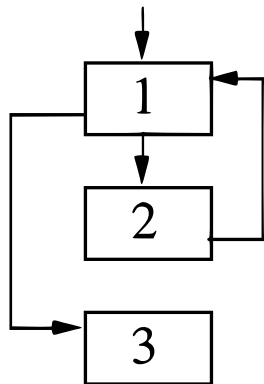
CFG



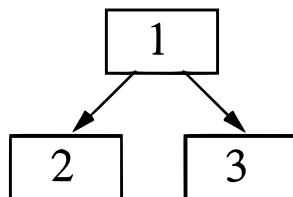
Dominatorbaum



CFG

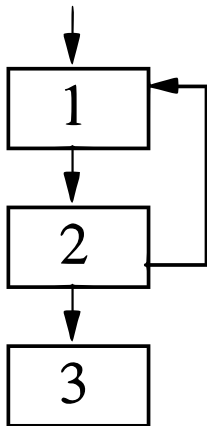


Dominatorbaum

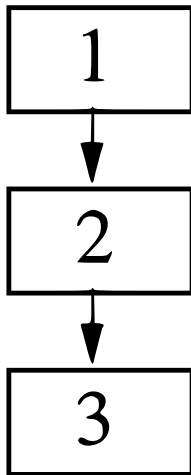


Berechnung von Dominatoren für REPEAT

CFG



Dominatorbaum



- Kontrollflussgraphen
- Versionsnummern für Variablen
- Aufbau der SSA-Form
- Transformation in SSA-Form
- Allgemeiner Fall (aus dem Orbit)
- Sonderfall: Strukturierte Programmiersprachen
- Rückwandlung aus der SSA-Form (einfaches Verfahren!)
- Berechnung von Dominatoren