

# Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

8. Dezember 2015

# Codeoptimierung

Ziel der Codeoptimierung ist die Umformung eines Zwischencode-Programms in ein kürzeres oder schnelleres, aber funktional äquivalentes Programm. Der Begriff „Codeoptimierung“ ist dabei eigentlich irreführend, denn Aho, Johnson und Ullman zeigten, dass eine optimale Code-Erzeugung für eine hypothetische Maschine mit einer noch einfacheren Struktur als die einer Drei-Adress-Code verarbeitende Maschine, bereits NP-vollständig ist.

Wichtigster Ansatzpunkt für die Optimierung sind die Rümpfe innerer Schleifen, denn nach der bekannten 80-20 Regel werden etwa 80% der Zeit in 20% des Codes verbraucht.

# Forderungen an die zur Codeoptimierung benutzten Programmtransformationen

1. Eine Programmtransformation darf die Funktion eines Programms nicht ändern.
2. Eine angewendete Programmtransformation sollte, zumindest im Mittel, die Laufzeit des Programms messbar verringern.
3. Eine Programmtransformation sollte kosteneffektiv sein, d.h. der Kostenaufwand sollte in vernünftiger Relation zur Verbesserung der Laufzeit des Programms stehen.

## Beispiel

Betrachten wir zunächst das Ausgangsprogramm:

```
subroutine tricky (a, b, n, m, k);  
    integer i, n, m, k;  
    int a[m], b[m];  
  
    for i := 1 to n do  
        a[i] := b[k] + a[i] + 100000;  
    end;
```

Wie man sofort sieht, gibt es eine „schleifeninvariante Berechnung“, d.h. eine Berechnung, die bei jedem Durchlauf durch die Schleife den gleichen Wert liefert.

Zieht man diese Berechnung vor die Schleife, so erhält man:

```
subroutine tricky (a, b, n, m, k);  
    integer i, n, m, k;  
    int a[m], b[m], C;  
  
    C := b[k] + 100000;  
    for i := 1 to n do  
        a[i] := a[i] + C;  
    end;
```

und man kann vermuten, dass dieses Programm schneller läuft, da der Term  $b[k] + 100000$  nur einmal ausgewertet werden muss.

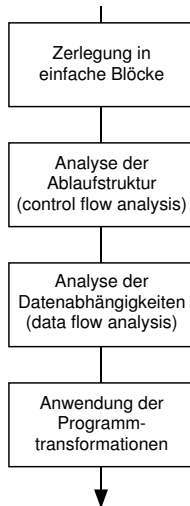
Diese harmlos aussehende Änderung hat nun aber folgende Konsequenzen:

- ▶ Es kann sein, dass im Originalprogramm alles richtig läuft, während im transformierten Programm ein Überlauf signalisiert wird!
- ▶ Ein Aufruf der Form

```
k := m+1;  
n := 0  
call tricky(a, b, n, m, k);
```

würde (hoffentlich) in der optimierten Version zu einer Fehlermeldung führen. Die Originalversion würde dagegen korrekt arbeiten.

# Möglicher Aufbau eines einfachen Codeoptimierers



# Einfache Blöcke und Flussgraphen

**Idee:** Partitionierung der Drei-Adress-Befehle des Zwischencode-Programms in maximale, zusammenhängende Teile, die bei Ablauf des Programms immer sequentiell vom ersten bis zum letzten Befehl abgearbeitet werden müssen. Diese Teil werden **einfache Blöcke** genannt.



## Beispiel

Betrachte das folgende Fragment eines Drei-Adress-Programms:

```
      t1 := a + t1
L1:   t2 := a + c
      t3 := t1 * t2
      t4 := t3 + t1
      if t2 < t4 goto L1
      t1 := 2 * t4
```

Die Folge der Drei-Adress-Befehle vom 2. bis zum 5. Befehl bilden einen einfachen Block. Der 1. Befehl gehört nicht dazu, da andernfalls über die Marke L1: in den Block eingetreten werden könnte. Der 6. Befehl gehört ebenfalls nicht zu diesem Block, da da nicht in jedem Fall nach dem 5. der 6. Befehl abgearbeitet wird.

# Algorithmus zur Partitionierung in einfache Blöcke

**Eingabe:** Eine Folge von Drei-Adress-Befehlen

**Ausgabe:** Eine Liste einfacher Blöcke; jede Anweisung befindet sich in genau einem einfachen Block.

**Verfahren:**

1. Zuerst wird die Menge der führenden Befehle in einem einfachen Block markiert.
  - 1.1 Die erste Anweisung des Drei-Adress-Programms wird markiert.
  - 1.2 Jede Anweisung, die das Ziel eines bestimmten oder unbestimmten Sprunges ist, wird markiert.
  - 1.3 Jede Anweisung, die einem bestimmten oder unbestimmten Sprung oder einem Prozeduraufruf folgt, wird markiert.
2. Für jeden führenden Befehl besteht der zugeordnete einfache Block aus dem führenden Befehl und allen nachfolgenden Befehlen ausschließlich des nächsten markierten Befehls oder dem Ende des Drei-Adress-Programms

# Gebrauch und Lebendigkeit von Variablen

An dem Beispiel kann man auch noch weitere wichtige Begriffe erläutern:

- ▶ Der 3. Befehl  $t3 := t1 * t2$  repräsentiert eine Position im Drei-Adress-Programm, an der die Variable  $t3$  **definiert** wird und an der die Variablen  $t1$  und  $t2$  **benötigt** (oder auch **gebraucht**) werden.
- ▶ Manchmal wird der Gebrauch einer speziellen Definition einer Variablen betrachtet. Im Beispiel wird im 3. Befehl die Definition der Variablen  $t1$  im 1. Befehl benötigt. Wichtig ist dabei, dass zwischen der Definition und dem Gebrauch keine Neudefinition von  $t1$  stattfindet.

- ▶ Ein Programm-Punkt im Drei-Adress-Programm charakterisiert eine Position vor oder nach einem Drei-Adress-Befehl.
- ▶ Am Programm-Punkt nach dem 4. Befehl ist die Variable  $t1$  **lebendig**, weil der Wert von  $t1$  an anderer, erreichbarer Stelle (in diesem Fall im 3. Befehl) im Programm benötigt wird. Die Erreichbarkeit ist nur hypothetisch und kann auch in einem anderen einfachen Block liegen. Man nimmt dabei also an, dass jede Alternative eines bedingten Sprungs auch möglich ist.

- ▶ Die Variable `t3` ist an diesem Programm-Punkt **nicht lebendig** oder **tot**, da kein weiterer Gebrauch von `t3` erreichbar ist.
- ▶ Auch hier kann man genauer von der Lebendigkeit einer speziellen Definition der Variablen sprechen. Am Programm-Punkt nach dem 4. Befehl ist die Definition der Variable `t1` im 1. Befehl lebendig. Die Definition von `t1` im letzten Befehl ist dagegen an diesem Programm-Punkt nicht lebendig

# Konstruktion des Flussgraphen

Aus der Liste der einfachen Blöcke wird ein gerichteter Graph erzeugt, der den Kontrollfluss innerhalb des Drei-Adress-Programms berücksichtigt. Die einfachen Blöcke bilden die Knoten des Flussgraphen. Der Knoten, der den ersten Drei-Adress-Befehl enthält, wird als **Startknoten** ausgezeichnet.

Eine gerichtete Kante führt vom Knoten  $B_1$  zum Knoten  $B_2$ , falls

1. es einen bedingten oder unbedingten Sprung vom letzten Befehl in  $B_1$  zum ersten Befehl in  $B_2$  gibt oder
2. der erste Befehl von  $B_2$  im Drei-Adress-Programm unmittelbar nach dem letzten Befehl von  $B_1$  folgt und der letzte Befehl in  $B_1$  kein unbedingter Sprung ist.

## Bemerkung

*In manchen Situationen ist es vorteilhaft, zwei zusätzliche, künstliche einfache Blöcke **entry** und **exit** dem Flussgraphen hinzuzufügen (**erweiterter Flussgraph**). Vom Block **entry** gibt es genau eine Kante zum ausgezeichneten Startknoten. Ferner führt eine Kante von jedem Block, über die der Flussgraph verlassen werden kann, zum Block **exit**.*

## Beispiel

Gegeben sei das folgende Programmfragment:

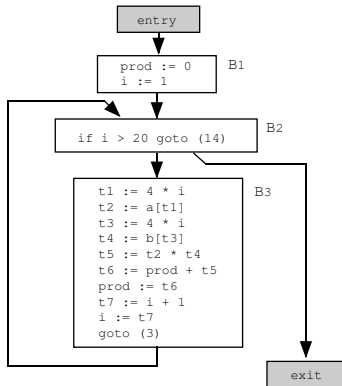
```
begin
  prod := 0;
  i := 1;
  while i <= 20 do
    begin
      prod := prod + a[i] * b[i];
      i := i+1;
    end
  end;
end;
```



Man erhält

```
(1)   prod := 0
(2)   i := 1
(3)   if i > 20 goto (14)
(4)   t1 := 4 * i
(5)   t2 := a[t1]
(6)   t3 := 4 * i
(7)   t4 := b[t3]
(8)   t5 := t2 * t4
(9)   t6 := prod + t5
(10)  prod := t6
(11)  t7 := i + 1
(12)  i := t7
(13)  goto (3)
```

Man erhält folgenden Flussgraphen:



An diesem Beispiel ist auch zu sehen, dass es in dieser Darstellung zweckmäßig ist, als Sprungziele bedingter und unbedingter Sprünge die entsprechenden einfachen Blöcke zu verwenden.

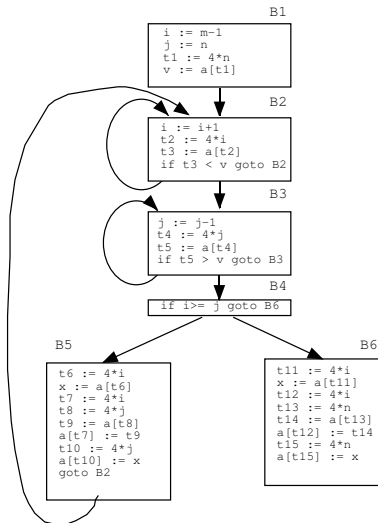
# Möglichkeiten der Codeoptimierung

Betrachtet wird eine Implementation einer Quicksort-Variante.

```
void quicksort(m,n)
int m,n;
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* ab hier beginnt das betrachtete Codefragment */
    i = m-1;  j = n;  v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* hier endet das betrachtete Codefragment */
    quicksort(m,j); quicksort(i+1,n);
```

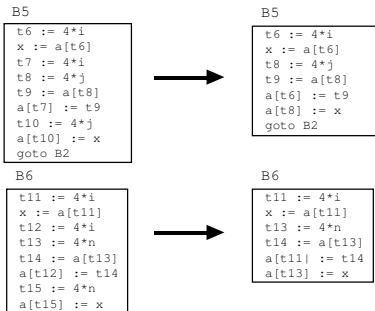
(1)	i := m-1	(16)	t7 := 4*i
(2)	j := n	(17)	t8 := 4*j
(3)	t1 := 4*n	(18)	t9 := a[t8]
(4)	v := a[t1]	(19)	a[t7] := t9
(5)	i := i+1	(20)	t10 := 4*j
(6)	t2 := 4*i	(21)	a[t10] := x
(7)	t3 := a[t2]	(22)	goto (5)
(8)	if t3 < v goto (5)	(23)	t11 := 4*i
(9)	j := j-1	(24)	x := a[t11]
(10)	t4 := 4*j	(25)	t12 := 4*i
(11)	t5 := a[t4]	(26)	t13 := 4*n
(12)	if t5 > v goto (9)	(27)	t14 := a[t13]
(13)	if i >= j goto (23)	(28)	a[t12] := t14
(14)	t6 := 4*i	(29)	t15 := 4*n
(15)	x := a[t6]	(30)	a[t15] := x

# Der zugehörige Flussgraph



In B5 werden die Ausdrücke  $4*i$  und  $4*j$  zweimal berechnet, in B6 werden die Ausdrücke  $4*i$  und  $4*n$  ebenfalls zweimal berechnet. Wir haben also sogenannte **gemeinsame Teilausdrücke**, die natürlich nur einmal berechnet werden müssen. Da diese Optimierung sich nur auf jeweils einen einfachen Block bezieht, bezeichnet man diesen Vorgang auch als **lokale Entfernung gemeinsamer Teilausdrücke**.

Die beiden Blöcke werden also wie folgt umgeformt:



Nach der lokalen Entfernung gemeinsamer Teilausdrücke stellt man fest, dass die Berechnung von  $4*j$  in B5 überflüssig ist, da dieser Term bereits in B3 berechnet wurde und es keine andere Möglichkeit gibt, als über B3 nach B5 zu gelangen. Also kann man die drei Befehle  $t8 := 4*j$ ,  $t9 := a[t8]$  und  $a[t8] := x$  durch die beiden Befehle  $t9 := a[t4]$  und  $a[t4] := x$  ersetzen. Damit ergibt sich aber der gemeinsame Teilausdruck  $a[t4]$  in B3 und B5, also kann man  $t9 := a[t4]$  und  $a[t6] := t9$  ersetzen durch  $a[t6] := t5$ .

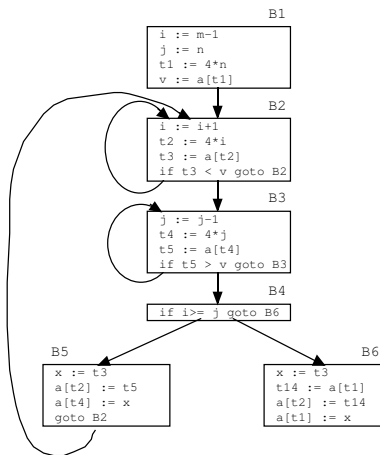
Der Block B5 enthält jetzt die Befehle:

```
t6 := 4*i  
x := a[t6]  
a[t6] := t5  
a[t4] := x  
goto B2
```

Analog erkennt man jetzt den gemeinsamen Teilausdruck  $4*i$  in B5 und B2 und ersetzt die drei Befehle  $t6 := 4*i$ ,  $x := a[t6]$  und  $a[t6] := t5$  durch  $x := a[t2]$  und  $a[t2] := t5$ . Danach wird wiederum  $a[t2]$  als gemeinsamer Teilausdruck erkannt und  $x := a[t2]$  durch  $x := t3$  ersetzt.



Die gleichen Überlegungen können für den Block B6 angestellt werden und man erhält einen neuen Flussgraphen:



## Bemerkung

*Der Ausdruck  $a[t1]$  in B1 und B6 ist **kein** gemeinsamer Ausdruck!  
Auf dem Weg von B1 nach B6 kann der Block B5 durchlaufen werden, in dem eine Wertzuweisung an das Feld  $a$  steht. Sofern keine weiteren Informationen zur Verfügung stehen, muss angenommen werden, dass **jedes** Element von  $a$  dadurch verändert wurde, also auch  $a[t1]$ !*

# Copy-Propagation

Durch die Entfernung gemeinsamer Teilausdrücke werden häufig Anweisungen der Form  $x := y$ , sogenannte „Kopien“, eingeführt. Ziel einer weiteren Programmtransformation ist es, nach einer derartigen Anweisung soweit wie möglich ein auftretendes  $x$  durch  $y$  zu ersetzen. Man nennt diese Transformation auch das **Weiterreichen von Kopien (Copy Propagation)**.

In unserem Beispiel kann man in B5  $a[t4] := x$  durch  $a[t4] := t3$  und in B6  $a[t1] := x$  durch  $a[t1] := t3$  ersetzen.

# Ersetzen von Run-Time– durch Compile-Time– Berechnungen

Eine weitere Möglichkeit zur Codeverbesserung besteht darin, Befehle mit konstanten Operanden bereits zu diesem Zeitpunkt auszuwerten und die Ergebnisse ähnlich wie oben weiterzureichen.

Beide Transformationen hinterlassen meist überflüssige Befehle. Ein Befehl ist überflüssig, wenn der einfache Block, in dem er auftritt, nicht vom Startknoten erreichbar ist oder aber wenn der Befehl von der Form  $a := \dots$  ist und  $a$  nach diesem Befehl nicht lebendig ist. Überflüssige Befehle können ohne weiteres entfernt werden.

# Erkennen und Verschieben schleifeninvarianter Berechnungen

Das erste Problem hierbei ist das Erkennen von Schleifen und deren Eintrittspunkt im Flussgraph. Als nächstes sind die Befehle im Rumpf der Schleife zu identifizieren, die bei jedem Durchlauf durch die Schleife den selben Wert berechnen. Dann müssen diese Befehle in einen einfachen Block vor dem Schleifenanfang ausgelagert werden.

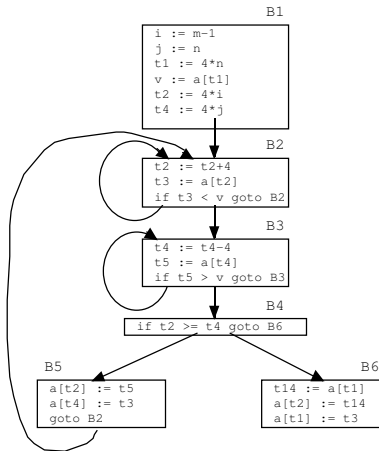
# Induktionsvariable

Ein anderer Ansatz zur Schleifenoptimierung führt über das Erkennen von sogenannten **Induktionsvariablen**, das sind Variable, die sich bei jedem Schleifendurchlauf um einen konstanten Wert verändern.

Häufig kann man diese Induktionsvariablen bis auf eine einsparen und dabei eventuell sogar eine „teure“ Operation (etwa eine Multiplikation) durch eine „billige“ (etwa eine Addition) ersetzen.

In unserem Beispiel werden im Block B3 die Werte von  $j$  und  $t4$  synchron bei jedem Schleifendurchlauf verändert. Wird  $j$  um 1 vermindert, so wird  $t4$  um 4 vermindert. Entsprechendes gilt für die Werte von  $i$  und  $t2$  im Block B2. Letztlich ist es sogar möglich, im Block B4 die Abfrage `if i >= j goto B6` durch `if t2 >= t4 goto B6` zu ersetzen, da  $t2 = 4*i$  und  $t4 = 4*j$  an dieser Stelle gilt.

Damit werden die Variablen  $i$  und  $j$  in den Blöcken B2 und B3 überflüssig und man erhält:



Es wären noch weitere Optimierungen in B1 möglich, jedoch ist es fraglich, ob sich der Aufwand lohnt, da B1 nur einmal durchlaufen wird.