Grundlagen des Compilerbaus

Prof. Dr. Rita Loogen

Fachbereich Mathematik und Informatik
Wintersemester 2009/10

Kapitel 6: Codeoptimierung

Inhaltsverzeichnis

6 Codeoptimierung

6.1 Gleitfenster-Optimierungen

Unter Gleitfenster-Optimierungen (engl. peephole - Schlüsselloch) versteht man eine sehr einfache, aber effektive Optimierungsform. Mit einem Gleitfenster wird jeweils ein Ausschnitt meist aufeinanderfolgender Instruktionen betrachtet. Dabei wird versucht, für die jeweils im Gleitfenster befindlichen Instruktionen eine kürzere oder schnellere Codefolge zu bestimmen. Typische Peephole-Optimierungen sind:

1. Elimination redundanten Codes

   - aufeinanderfolgende LOAD/STORE-Anweisungen `LOAD X; STORE X` kann vollständig eliminiert werden. In `LOAD R1, X; STORE R1, X` ist die `STORE`-Anweisung überflüssig, falls diese kein Sprungziel ist.

     Ist die Sequenz `STORE X; LOAD X` zu vereinfachen?

   - unerreichbarer Code

     Unerreichbar sind beispielsweise nicht markierte Befehle nach unbedingten Sprüngen: `goto Loop; OP`.

     Steht bei Verzweigungen das Ergebnis der Bedingung fest, so kann der Code für die nicht gewählte Alternative eliminiert werden:

     `LIT True;JPFALSE L2; <code> L2: ...` kann zu `<code>` vereinfacht werden.

2. Kontrollflussoptimierungen

   Eine typische Kontrollflussoptimierung ist die Komprimierung von Sprungfolgen:

   `JMP L1; ... L1: JMP L2; ...` kann modifiziert werden zu
   `JMP L2; ... L1: JMP L2; ...`. Die mit der Sprungmarke `L1` markierte Sprunganweisung ist dann eventuell nicht mehr erreichbar und kann eliminiert werden.

   Eine analoge Transformation ist möglich, falls der erste Sprungbefehl bedingt ist.

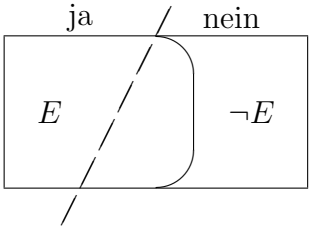
3. Algebraische Identitäten und „Reduction in Strength“

   Typische algebraische Identitäten, die bei der Codeoptimierung eingesetzt werden, sind etwa: $x + 0 = x$, $x * 0 = 0$ oder $x * 1 = x$.

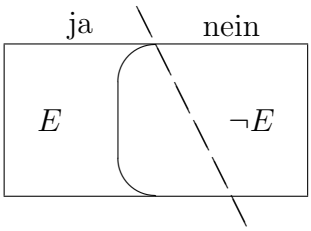
   Ersetzt man teure Operationen durch billigere bzw. aufwendige durch einfachere, so spricht man von einer Mächtigkeitsreduktion (engl: reduction in strength). Typische Beispiele sind die Verwendung einer speziellen Inkrementierungsoperation anstelle der allgemeinen Addition von Eins, das Ersetzen von Multiplikation mit Zweierpotenzen durch Shift-Anweisungen, Quadrieren statt Potenzierung mit 2, Gleitkommadivision durch Konstante durch Gleitkommamultiplikation mit Kehrwert.

6.2 Allgemeine Analysetechniken

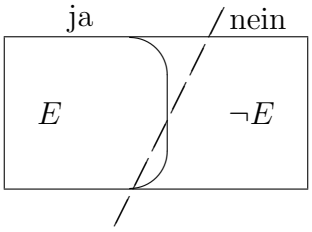
Allgemein basieren Codeoptimierungen auf der Analyse von Programmeigenschaften, mit denen man z.B. konstante Ausdrücke, die Benutzung von Variablen, Terminationseigenschaften oder Striktheitseigenschaften bestimmt. Allerdings sind solche Analysen in der Regel unentscheidbar, was unmittelbar aus der Unentscheidbarkeit des Halteproblems sowie aus dem Satz von Rice folgt. Man begnügt sich daher in der Regel mit einer approximativen Berechnung der gesuchten Eigenschaft E.
Man unterscheidet zwischen *konservativen* oder *sicheren* Approximationen und *spekulativen*. Konservative Approximationen nutzen die Semi-Entscheidbarkeit der meisten nicht entscheidbaren Eigenschaften oder ihres Komplements.
E ist semi-entscheidbar. Ein einfaches Approximationsverfahren erhält man, indem man das Semi-Entscheidungsverfahren nach einer fest vorgegebenen Schrittzahl stoppt. Falls das Verfahren vorher terminiert, gilt die Eigenschaft E auf jedem Fall. In diesem Fall stellt das Approximationsverfahren die Eigenschaft fest (Antwort ja). Falls das Verfahren abgebrochen wird, weil die Schrittzahl erreicht wird, so kann die Eigenschaft gelten (Entscheidung erfordert mehr Schritte) oder auch nicht (Verfahren terminiert nicht). Das Approximationsverfahren antwortet nein. Im Neinfall, kann die Eigenschaft also trotzdem gelten.



Ist $\neg E$ semi-entscheidbar, so kann eine entsprechende sichere Approximation von $\neg E$ erfolgen. Falls die Approximation nein ausgibt, gilt E definitiv nicht. Sonst kann keine Aussage gemacht werden.



Eine Approximation, die nie ein definitives Ergebnis liefert, ist trivialerweise sicher, aber natürlich ein sinnloser Grenzfall. Man sollte stets bemüht sein, möglichst genaue Approximationen zu entwickeln.
Spekulative Approximationen werden häufig bei Verifikationen eingesetzt. Hier gilt: „Nicht gemeldete Fehler sind schlimmer als gemeldete Nicht-Fehler“.



Für Compileroptimierungen kommen nur sichere Approximationen in Frage. Die Approximation setzt dabei Grenzen für die Leistungsfähigkeit von Optimierungen. Das heißt, dass optimierte Programme in der Regel nicht wirklich optimal sind, weil nicht alle Verbesserungsmöglichkeiten erkannt werden können. In der Praxis treten auch Fälle auf, in denen optimierte Programme nicht einmal besser sind.

Beispiel: Unter Codeverschiebung (engl.: code motion) versteht man die Verschiebung von Anweisungen, beispielsweise aus Schleifenrümpfen.

```
for i:=1 to n do                    h := d+3;
  h := d+3;                  ↷      for i:=1 to n do
  a[i] := h+i                         a[i] := h+i
  end                                 end
```

Weil die Anweisung `h := d+3` unabhängig von der Schleifenvariablen `i` ist, kann die Berechnung aus der Schleife gezogen werden.

Aber Vorsicht: Falls `n = 0` führt das optimierte Programm die herausgezogene Anweisung trotzdem durch. Dies kann unter Umständen sogar fehlerhaft sein (fälschliche Modifikation von `h`. ◁

Codeoptimierungen werden meist auf der Zwischencode-Ebene durchgeführt. Ein flexibler, häufig betrachteter Zwischencode ist der Drei-Adress-Code.

6.3 Drei-Adress-Code

Die meisten heutigen Maschinen sind Registermaschinen, so dass als Zwischencode meist sog. Drei-Adress-Code statt der Datenkellertechnik verwendet wird. In einem Drei-Adress-Code sind die elementaren Anweisungen Zuweisungen der Form

$$x := y \text{ op } z.$$

x, y und z sind drei Adressen bzw. Variablenbezeichner. Sie sind Exemplare eines unbegrenzten Registervorrats. Auf den rechten Zuweisungsseiten treten keine geschachtelten Ausdrücke auf. Alle Zwischenergebnisse von komplexen Ausdrücken sind im Code als Zuweisung erkennbar.

6.1 Definition (Drei-Adress-Anweisungen und -Programme)

Seien *Var* eine Menge von Variablenbezeichnern, *Lab* eine Menge von Sprungmarken, *Proc* eine Menge von Prozedurbezeichnern und *Op* eine Menge von binären und unären Operationssymbolen.

Die Menge der Drei-Adress-Anweisungen *Instr* enthält

- Zuweisungen

$$\left.\begin{array}{lcl} x & := & y \text{ op } z \\ x & := & \text{op } y \\ x & := & y \end{array}\right\} (x, y, z \in \mathit{Var}, \text{op} \in \mathit{Op})$$

- Sprünge

$$\left.\begin{array}{l} \text{goto } l \\ \text{if } x \text{ op } y \text{ goto } l \end{array}\right\} (x, y \in \mathit{Var}, \text{op} \in \mathit{Op}, l \in \mathit{Lab})$$

- Prozeduraufrufe

$$\left.\begin{array}{l} \text{call } p, x_1, \ldots, x_n \\ \text{call } p, x_1, \ldots, x_n \to y \\ \text{return} \\ \text{return } x \end{array}\right\} (p \in \mathit{Proc}, x_1, \ldots, x_n, y, x \in \mathit{Var}$$

Sei $\mathit{LInstr} := \{l : i \mid l \in \mathit{Lab}, i \in \mathit{Instr}\}$ die Menge der Anweisungen mit Marke.

Die Menge *Prog* der Drei-Adress-Programme ist definiert durch:

$$\mathit{Prog} := \mathfrak{P}_{\text{fin}}(\{(p, w) \mid p \in \mathit{Proc}, w \in (\mathit{Instr} \cup \mathit{LInstr})^*\})$$

$M \in \mathit{Prog}$ mit $M = \{(p_1, w_1), \ldots, (p_n, w_n)\}$ heißt zulässig, falls gilt:

1. für jeden in w_j $(1 \leq j \leq n)$ auftretenden Prozedurbezeichner p existiert genau ein i mit $1 \leq i \leq n$ und $p = p_i$.
2. für jede in $w_j = i_1 \ldots i_m$ auftretende Sprungmarke l existiert genau ein k mit $1 \leq k \leq m$ und $i_k = l : i'_k$.

Zur Unterscheidung lokaler und globaler Variablen sowie Prozedurparametern werden die folgenden Bezeichnungskonventionen vereinbart:

- Prozedurparameter: a, a_0, a_1 ...
- globale Variablen: g, g_0, g_1 ...
- lokale Variablen: alle anderen Bezeichner

Op enthalte neben den arithmetischen und logischen Operationen auch Operationen für indizierten (Array-)Zugriff $a[i]$, Zeigerdereferenzierung $*y$ und Adressberechnung $\&x$.

Beispiel: Ein Drei-Adress-Programm zur Multiplikation von Zahlen ist etwa gegeben durch

$$\{(\mathit{add}, w_{\mathit{add}}), (\mathit{mult}, w_{\mathit{mult}})\},$$

wobei $w_{add} =$

	if $a_1 == 0$ goto L_1
	$x := a_1 - 1$
	call $add\ x, a_2 \to r$
	$r := r + 1$
	goto L_2
$L_1:$	$r := a_2$
$L_2:$	return r

und $w_{mult} =$

	$r := 0$
$L_1:$	if $a_1 > 0$ goto L_2
	goto L_3
$L_2:$	call $add\ r, a_2 \to r$
	$a_1 := a_1 - 1$
	goto L_1
$L_3:$	return r

◁

Eine Übersetzung von Drei-Adress-Code in MPP-Code ist sehr einfach. Die Übersetzung von PSPP (ohne Variablenparameter) in Drei-Adress-Code erfordert

- das Entschachteln von Ausdrücken

  E_1 op $E_2 \rightsquigarrow$ ⟨Code für E_1 mit Resultat in x_1⟩
  ⟨Code für E_2 mit Resultat in x_2⟩
  $x_3 := x_1$ op x_2

- das Flachklopfen von Prozeduraufrufen

  $p(E_1, \ldots, E_n) \rightsquigarrow$ ⟨Code für E_1 mit Resultat in x_1⟩
  $\vdots$
  ⟨Code für E_n mit Resultat in x_n⟩
  call $p, x_1, \ldots, x_n$

- die Übersetzung von Verzweigungen und Schleifen mit Sprungbefehlen.

Beispiel: Die Übersetzung der folgenden C-Prozedur *mult* ergibt den oben angegebenen Drei-Adress-Code:

```
int mult (int a2, int a2)
{ int r = 0;
  while (a1 > 0) {
    r = add(r, a2);
    a1--;
    };
  return r;
}
```

◁

6.4 Basisblockdarstellung und Datenflussgraphen

Ein Nachteil des Drei-Adress-Codes oder jedes Zwischencodes ist der implizite Kontrollfluss durch gotos. Für Analysen und Optimierungen ist die Kenntnis des Kontrollflusses essentiell. Daher werden Zwischencodeprogramme oft als Graphen dargestellt. Die Knoten werden mit Sequenzen unverzweigten Codes beschriftet, sogenannten Basisblöcken. Die Kanten beschreiben den expliziten Kontrollfluss.

6.2 Definition Sei $w \in (Instr \cup LInstr)^*$.

Die *Basisblockdarstellung* bzw. der *Datenflussgraph von* w ist ein beschrifteter, gerichteter Wurzelgraph

$$G(w) = (N, E, r, \sigma)$$

mit

- Knotenmenge $N \supseteq \{l \in Lab \mid l \text{ kommt in } w \text{ vor.}\}$
- Kantenmenge $E \subseteq N \times N$
- Wurzel $r \in N$
- Beschriftung $\sigma : N \to Instr^*$,

wobei gilt:

1. Das Abwickeln der Graphen ergibt w, bis auf das Vertauschen unabhängiger Blöcke und das Hinzufügen zusätzlicher Label:

$$r : \sigma(r); l_1 : \sigma(l_1); \ldots; l_n : \sigma(l_n) \cong w$$

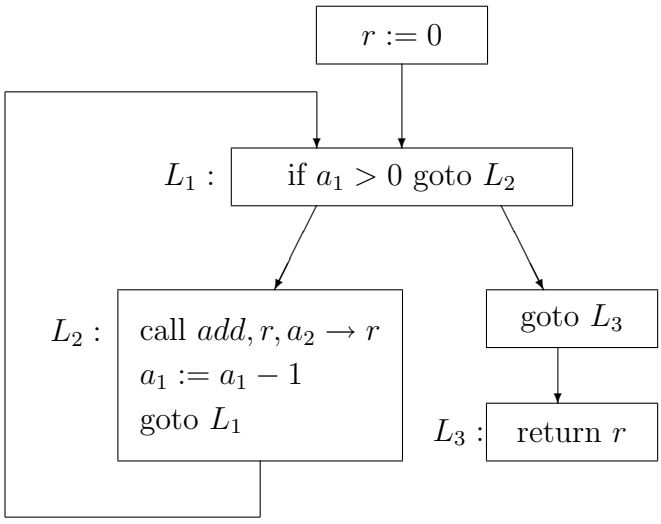
2. Jedes $\sigma(l) = i_1 \ldots i_n$ ist ein *Basisblock*, d.h. nur der letzte Befehl darf ein Sprungbefehl sein:

$$i_1, \ldots, i_{n-1} \in Instr \setminus \{\text{goto } l, \text{if } x \text{ op } y \text{ goto } l, \text{return }\}$$
$$i_n \in Instr$$

Sei $P \in Prog$. Die Basisblockdarstellung von P ist

$$G(P) = \{(p, G(w)) \mid (p, w) \in P\}.$$

Beispiel: Basisblockdarstellung von *mult*:



◁

Basisblöcke haben die folgenden wichtigen Eigenschaften:

- Der Kontrollfluss innerhalb eines Blocks ist linear.
- Es gibt keine Sprünge innerhalb von Basisblöcken. Nur am Ende ist ein Sprung zugelassen, aber nicht notwendig.
- Es gibt keine Sprünge von anderen Basisblöcken in das Innere eines Basisblocks, sondern nur an den Anfang.

Sprünge sind bedingte oder unbedingte Sprungbefehle und Prozedurrücksprünge. Prozeduraufrufe werden nicht als Sprünge betrachtet, damit Basisblöcke möglichst groß und die Graphen entsprechend klein werden. Da Prozeduren in der Regel am Ende ihrer Ausführung die Kontrolle zurückgeben, können sie als komplexe Operationen gesehen werden. Dies ist natürlich nur legitim, wenn die Prozeduren seiteneffektfrei sind. Ansonsten müssen auch Prozeduraufrufe wie Sprünge behandelt werden.
Man bestimmt die Basisblockdarstellung zu einem Drei-Adress-Programm wie folgt:

1. Bestimme zunächst alle Instruktionen, die Anfänge von Basisblöcken sind. Man nennt diese Instruktionen *Leader*.

   Die Menge der *Leader* umfasst

   - die ersten Anweisungen von Prozeduren
   - alle Instruktionen, die Sprungziel sind
   - alle Anweisungen unmittelbar nach Sprüngen.

2. Zu jedem Leader wird der zugehörige Basisblock konstruiert. Dieser umfasst die Folge aller Anweisungen bis zum nächsten Leader oder bis zum Ende einer Prozedur.

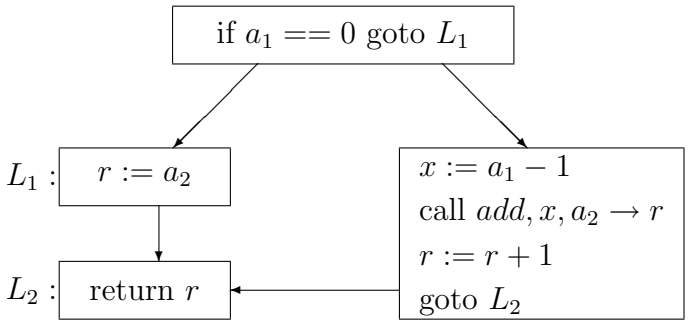
3. Die Kanten zwischen den Basisblöcken werden entlang der Sprünge und sequentiellen Fortsetzungen nach bedingten Sprüngen eingefügt.

Dieses Verfahren liefert einen Basisblockgraphen mit minimaler Knotenzahl. Jede Zerteilung von Blöcken liefert wieder einen Basisblockgraphen. Der minimale Graph ist bis auf Isomorphie eindeutig.

Beispiel: Im folgenden Code des Multiplikationsbeispielprogramms sind die Leader unterstrichen:

$$
\begin{array}{lll}
w_{add} = & & \\
 & \underline{\text{if } a_1 == 0 \text{ goto } L_1} & \\
 & \underline{x := a_1 - 1} & \\
 & \text{call } add\ x, a_2 \rightarrow r & \\
 & r := r + 1 & \\
 & \text{goto } L_2 & \\
L_1: & \underline{r := a_2} & \\
L_2: & \underline{\text{return } r} &
\end{array}
\qquad
\begin{array}{lll}
w_{mult} = & & \\
 & \underline{r := 0} & \\
L_1: & \underline{\text{if } a_1 > 0 \text{ goto } L_2} & \\
 & \underline{\text{goto } L_3} & \\
L_2: & \underline{\text{call } add\ r, a_2 \rightarrow r} & \\
 & a_1 := a_1 - 1 & \\
 & \text{goto } L_1 & \\
L_3: & \underline{\text{return } r} &
\end{array}
$$

Der Basisblockgraph zu w_{add} ergibt sich damit zu:



Der Basisblockgraph zu w_{mult} wurde zuvor bereits gezeigt. ◁

6.5 Lokale Analyse

Als lokale Analyse bezeichnet man die Betrachtung und Optimierung einzelner Basisblöcke. Dabei werden zwei Analysearten unterschieden:

1. *Vorwärtsanalyse:* Analyse des Verhaltens eines Programms vor einer Instruktion, d.h. es wird das vergangene Verhalten betrachtet.

   Beispiel: Ein typisches Beispiel für eine Vorwärtsanalyse ist die *Konstantenpropagation*. Ein Ausdruck heißt genau dann *konstant*, wenn er unabhängig von der Ausführung des Programms immer den gleichen Wert hat.
   Wir betrachten die Sequenz

   $$\begin{aligned} &\ldots \\ &x := 3 \\ &y := x + 4 \\ &\ldots \end{aligned}$$

   Offensichtlich ist x konstant. y ist konstant, falls die Zuweisung an y kein Sprungziel ist, sich also innerhalb eines Basisblocks befindet. ◁

2. *Rückwärtsanalyse:* Analyse des Programmverhaltens nach Ausführung einer Instruktion, d.h. es wird das zukünftige Verhalten analysiert.

   Beispiel: Ein Beispiel für eine Rückwärtsanalyse ist die Bestimmung nutzloser Zuweisungen (dead code elimination). Wir betrachten die Sequenz

   $$\begin{aligned} &\ldots \\ &x := y + z \\ &y := y + z \\ &x := y - z \\ &\ldots \end{aligned}$$

   Befindet sich diese Sequenz innerhalb eines Basisblocks, so ist die erste Zuweisung an x nutzlos, weil der zugewiesene Wert nicht verwendet wird, bevor er wieder überschrieben wird. ◁

Eine lokale Analyse kann wie folgt mathematisch modelliert werden. Zur Vereinfachung verzichten wir hier auf die Betrachtung von Prozeduraufrufen. Der Ausgangspunkt ist die Modellierung der zu analysierenden Eigenschaft als Menge M. Handelt es sich um eine Eigenschaft von Variablen, so wird häufig M als Menge aller Abbildungen von Variablen auf abstrakte Werte gewählt: $M = Var \to AbsVal$.
Ein Startwert $m_0 \in M$ beschreibt die Anfangssituation für die Analyse. Bei einer Vorwärtsanalyse ist dies die Situation vor der Ausführung des Basisblocks. Bei einer Rückwärtsanalyse beschreibt m_0 die Situation nach Ausführung des Basisblocks.
Die einzelnen Instruktionen modifizieren die Werte von M, also die Eigenschaften $m \in M$. Eine abstrakte Semantik

$$\mathfrak{A} : Instr \to M \to M$$

formalisiert dies. Die abstrakte Semantik muss nur für Zuweisungen definiert werden, denn Sprünge haben nur Auswirkungen auf den Kontrollfluss, nicht aber auf die Eigenschaften, d.h. die abstrakte Semantik entspricht der Identitätsfunktion id_M:

$$\mathfrak{A}[\![\text{goto } L]\!] = \mathfrak{A}[\![\text{if } x \text{ op } y \text{ goto } L]\!] = \mathfrak{A}[\![\text{return } x]\!] = \mathfrak{A}[\![\text{return}]\!] = id_M$$

Instruktionssequenzen $i_1 \dots i_n$ können einfach durch Komposition der abstrakten Semantik der einzelnen Instruktionen interpretiert werden, d.h. soll die Eigenschaft vor bzw. nach Auswertung der Instruktion i_k bestimmt werden, so gilt:

- *Vorwärtsanalyse:* $\mathfrak{A} : Instr^+ \to M \to M$ mit

  $$\mathfrak{A}[\![i_1 \dots i_{k-1}]\!] := \mathfrak{A}[\![i_{k-1}]\!] \circ \dots \circ \mathfrak{A}[\![i_1]\!].$$

  Dies entspricht der natürlichen Reihenfolge.

- *Rückwärtsanalyse:* $\mathfrak{A} : Instr^+ \to M \to M$ mit

  $$\mathfrak{A}[\![i_{k+1} \dots i_n]\!] := \mathfrak{A}[\![i_{k+1}]\!] \circ \dots \circ \mathfrak{A}[\![i_n]\!].$$

  Dies entspricht der umgekehrten Reihenfolge.

Damit folgt, dass für die Modellierung eines Analyseproblems nur die folgenden Festlegungen getroffen werden müssen:

1. die Menge M
2. der Startwert $m_0 \in M$
3. die abstrakte Zuweisungssemantik

In den folgenden Beispielen werden wir eine abkürzende Notation zur argumentweisen Modifikation von Funktionen verwenden:
Sei $A \to B := \{f \mid f : A \to B\}$. Für $f : A \to B$, $a \in A$ und $b \in B$ sei

$$f[a \mapsto b] := a' \mapsto \begin{cases} b & \text{falls } a' = a \\ f(a') & \text{sonst} \end{cases}$$

Desweiteren sei $f[a_1 \mapsto b_1, \dots, a_n \mapsto b_n] := (\dots((f[a_1 \mapsto b_1])[a_2 \mapsto b_2])\dots)[a_n \mapsto b_n]$.
Dabei ist zu beachten, dass im allgemeinen $f[a_1 \mapsto b_1, a_2 \mapsto b_2] \neq f[a_2 \mapsto b_2, a_1 \mapsto b_1]$, insbesondere, wenn etwa $a_1 = a_2$.

6.5.1 Lokale Konstantenpropagation — Vorwärtsanalyse

Sei C eine Menge von Konstanten und $? \notin C$ ein spezieller Wert. Wir definieren

1. $M := Var \to C \cup \{?\}$ und legen
2. $m_0 \in M$ mit $m_0(x) =?$ für alle $x \in Var$ als Startwert fest.

Der Wert ? repräsentiert einen unbekannten Wert. Es wird somit davon ausgegangen, dass alle Variablen zu Beginn einen unbekannten Wert haben. Wir verwenden die Hilfsfunktion $val : (Val \cup C) \times M \to C \cup \{?\}$ mit

$$val(y, m) := \begin{cases} m(y) & \text{falls } y \in Var \\ y & \text{falls } y \in C \end{cases}$$

3. Die abstrakte Zuweisungssemantik wird definiert durch

$$\mathfrak{A}[\![x := y \text{ op } z]\!]m := \begin{cases} m[x \mapsto c] & \text{falls} \quad val(y,m) = c_y \neq ?, \\ & \quad\; val(z,m) = c_z \neq ?, \\ & \quad\; c = c_y \text{ op } c_z \\ m[x \mapsto ?] & \text{sonst.} \end{cases}$$

Beispiel: Wir betrachten die Sequenz

$$\begin{array}{ll} i_1 & x := 3 \\ i_2 & y := x + 4 \\ i_3 & z := z + y \end{array}$$

Es folgt

- $\mathfrak{A}[\![i_1]\!]m_0 = m_0[x \mapsto 3]$
- $\mathfrak{A}[\![i_1; i_2]\!]m_0 = m_0[x \mapsto 3, y \mapsto 7]$
- $\mathfrak{A}[\![i_1; i_2; i_3]\!]m_0 = m_0[x \mapsto 3, y \mapsto 7]$

Damit kann der Code wie folgt optimiert werden:

$$\begin{array}{l} x := 3 \\ y := 7 \\ z := z + y \end{array}$$

◁

6.5.2 Nutzlose Anweisungen — Rückwärtsanalyse

Wir definieren

1. $M := Var \to \{0, 1\}$ und legen
2. $m_0 \in M$ mit $m_0(x) = 0$ für alle $x \in Var$ als Startwert fest.

$m(x) = 1$ bedeutet für eine Zuweisung $x := y$ op z, dass die Zuweisung nutzlos ist, weil der Wert von x bis zur nächsten Zuweisung an x nicht verwendet wird. Der Startwert bedeutet, dass am Ende eines Blocks davon ausgegangen wird, dass alle Variablen verwendet werden, also nützlich sind.

3. Die abstrakte Zuweisungssemantik wird wie folgt definiert:

$$\mathfrak{A}[\![x := y \text{ op } z]\!]m := m[x \mapsto 1, y \mapsto 0, z \mapsto 0].$$

Dies bedeutet, dass vorherige x-Werte nutzlos werden, während y und z gebraucht werden. Beachten Sie, dass die Reihenfolge relevant ist, da x, y und z nicht verschieden sein müssen.

Beispiel: Wir betrachten die Sequenz

$$\begin{aligned} i_1 &\quad x := y + z \\ i_2 &\quad y := y + z \\ i_3 &\quad x := y - z \end{aligned}$$

Es folgt

- $\mathfrak{A}[\![i_3]\!]m_0 = m_0[x \mapsto 1, y \mapsto 0, z \mapsto 0] = m_0[x \mapsto 1]$
- $\mathfrak{A}[\![i_2; i_3]\!]m_0 = m_0[x \mapsto 1][y \mapsto 1, y \mapsto 0, z \mapsto 0] = m_0[x \mapsto 1]$
- $\mathfrak{A}[\![i_1; i_2; i_3]\!]m_0 = m_0[x \mapsto 1][x \mapsto 1, y \mapsto 0, z \mapsto 0] = m_0[x \mapsto 1]$

Wegen $\mathfrak{A}[\![i_2; i_3]\!]m_0 = m_0[x \mapsto 1]$ kann die erste Zuweisung an x (Instruktion i_1) entfernt werden. ◁

6.5.3 Elimination gemeinsamer Teilausdrücke — Vorwärtsanalyse

Beispiel: Die Beispielsequenz

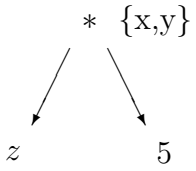
$$\begin{aligned} i_1 &\quad a := b * c \\ i_2 &\quad d := d - a \\ i_3 &\quad e := b * c \\ i_4 &\quad f := d + e \\ i_5 &\quad i := b * c \\ i_6 &\quad c := d + i \end{aligned}$$

enthält mehrere identische (gemeinsame) Teilausdrücke, deren Mehrfachauswertung durch Kopieranweisungen verhindert werden kann. Der Ausdruck $b * c$ tritt in den Instruktionen i_1, i_3 und i_5 auf. Die rechten Seiten der Instruktionen i_4 und i_6 sind ebenfalls identisch, obwohl sie syntaktisch verschieden sind. ◁

1. Als Menge M verwenden wir eine Menge von Abhängigkeitsgraphen, das sind geordnete, gerichtete, beschriftete und azyklische Graphen. Die Beschriftung besteht an den Blattknoten aus einer Konstanten oder Variablen. Innere Knoten sind mit Operationen und Mengen von Variablen beschriftet.

   Anstelle einer formalen Definition von Abhängigkeitsgraphen begnügen wir uns mit einem Beispiel:

Beispiel: Der Graph



beschreibt, dass die Variablen x und y den Wert $z + 5$ erhalten. ◁

2. Als Startwert m_0 wird der leere Graph festgelegt. Ziel der abstrakten Semantik ist der Abhängigkeitsgraph des gesamten Blocks.

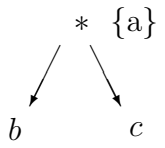
Es wird eine Vorwärtsanalyse durchgeführt. Im folgenden verwenden wir für einen Abhängigkeitsgraphen m die Notation $m[x]$ für den ersten Knoten von der Wurzel aus mit Beschriftung x, falls ein solcher existiert.

3. Damit legen wir die abstrakte Zuweisungssemantik wie folgt fest:

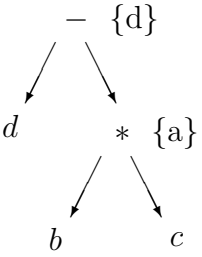
$\mathfrak{A}[\![x := y \text{ op } z]\!]m :=$
1. $\nexists m[y] \curvearrowright$ erzeuge neues Blatt mit Beschriftung y
2. $\nexists m[z] \curvearrowright$ erzeuge neues Blatt mit Beschriftung z
3. $\nexists$ Knoten mit Beschriftung op, linkem Kind $m[y]$ und rechtem Kind $m[z]$ $\curvearrowright$ erzeuge solchen Knoten
4. Füge Beschriftung x zu Knoten aus Schritt 3 hinzu.

Beispiel: Zu obiger Sequenz ergibt sich die folgende Graphfolge:

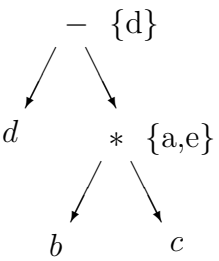
$\mathfrak{A}[\![i_1]\!]m_0 =$



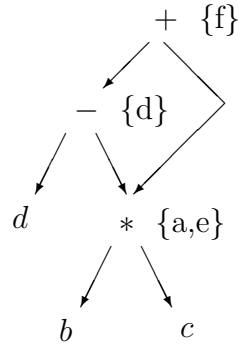
$\mathfrak{A}[\![i_1; i_2]\!]m_0 =$



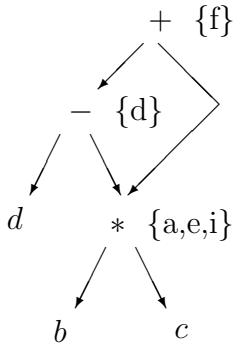
$\mathfrak{A}[\![i_1; i_2; i_3]\!]m_0 =$



$\mathfrak{A}[\![i_1; i_2; i_3; i_4]\!]m_0 =$

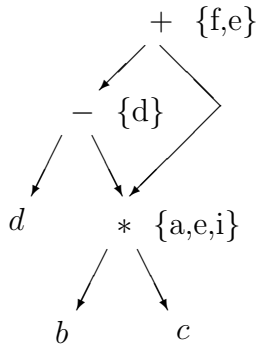


$\mathfrak{A}[\![i_1; i_2; i_3; i_4; i_5]\!]m_0 =$



Final ergibt sich der folgende Graph

$\mathfrak{A}[\![i_1; i_2; i_3; i_4]\!]m_0 =$



Mehrere Variablenbeschriftungen an inneren Knoten zeigen gemeinsame Teilausdrücke an. Diese werden meist dadurch eliminiert, dass sie zunächst in Hilfsvariablen berechnet werden und dann nur noch aus der Hilfsvariable in die Zielvariablen kopiert werden. Im Beispiel ergibt sich die folgende optimierte Codesequenz:

$$
\begin{aligned}
&temp := b * c\\
&a := temp\\
&d := d - a\\
&e := temp\\
&temp2 := d + e\\
&f := temp2\\
&i := temp\\
&c := temp2
\end{aligned}
$$

◁

6.6 Fallstudie: Code-Optimierung am Beispiel von Quicksort

Bevor wir die Analyse von Datenflussgraphen betrachten, stellen wir eine Reihe von gängigen globalen Optimierungen am bekannten, in Abbildung 22 gezeigten Quicksort-Beispielprogramm vor.
Aus diesem Programm wird nur die While-Schleife mit den drei vorherigen und den drei nachfolgenden Zuweisungen betrachtet. Für dieses Fragment erhält man den folgenden Drei-Adress-Code:

```
void quicksort(int m, int n)
   {  int i,j; int v,x;
      if (n <= m) return;
      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;
      quicksort(m,j); quicksort(i+1,n);
   }
```

Abbildung 22: Quicksort-Beispielprogramm

```
    i  = m-1
    j  = n
    t1 = 4*n
    v  = a[t1]
L1: i  = i+1
    t2 = 4*i
    t3 = a[t2]
    if t3<v goto L1
L2: j  = j-1
    t4 = 4*j
    t5 = a[t4]
    if t5>v goto L2
    if i>=j goto L3
    t6 = 4*i
    x  = a[t6]
    t7 = 4*i
    t8 = 4*j
    t9 = a[t8]
    a[t7] = t9
    t10 = 4*j
    a[t10] = x
    goto L1
L3: t11 = 4*i
    x = a[t11]
    t12 = 4*i
    t13 = 4*n
    t14 = a[t13]
    a[t12] = t14
    t15 = 4*n
    a[t15] = x
```

In diesem Code wird angenommen, dass die einzelnen Array-Werte als Worte gespeichert werden. Die indizierte Adressierung erfolgt allerdings byteweise. Daher werden die Array-Indizes jeweils durch Multiplikation mit 4 in Byteadressen überführt. Zu diesem Code erhält man die in Abbildung 23 gezeigte Basisblockdarstellung.

6.6.1 Elimination lokaler gemeinsamer Teilausdrücke

In den beiden längeren Basisblöcken in Abbildung 23 können die Indexberechnungen als gemeinsame Teilausdrücke (wie in Abschnitt 6.5.3 gezeigt) erkannt und teilweise eliminiert werden:

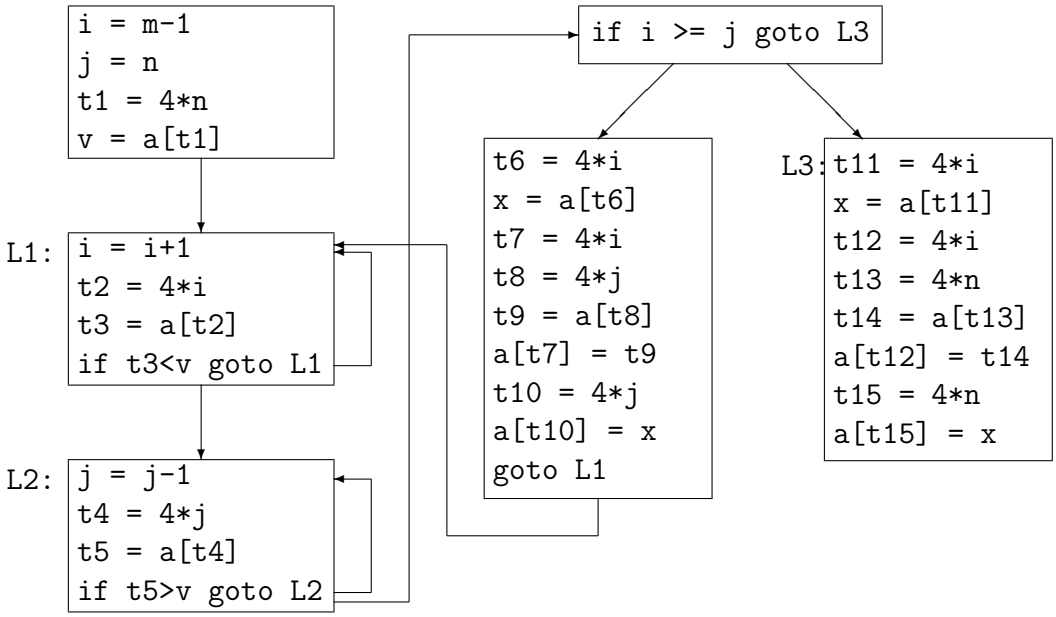
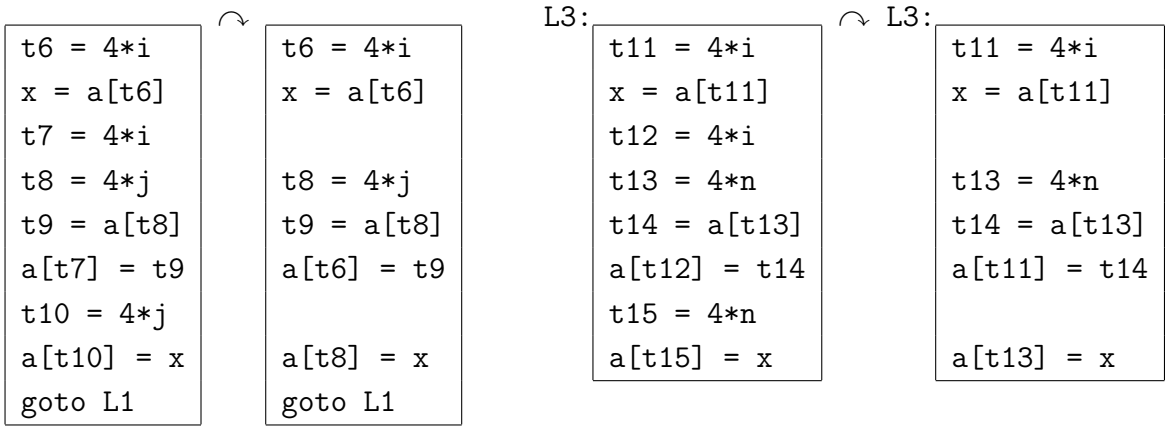


Abbildung 23: Basisblockdarstellung zu Quicksort-Fragment



Die doppelten Berechnungen werden jeweils ersatzlos gestrichen. Referenzen auf die entsprechenden Variablen werden ersetzt durch die Variable, die den Wert bereits enthält, d.h. `t6` wird z.B. für `t7` eingesetzt.

Andere lokale Optimierungen sind in diesem Beispiel nicht möglich. Als Ausgangspunkt für die globale Optimierung erhält man entsprechend den in Abbildung 24 angegebenen lokal optimierten Datenflussgraphen.

An diesem Beispiel erläutern wir im folgenden eine Reihe typischer globaler Optimierungen.

6.6.2 Global gemeinsame Teilausdrücke

Wenn sich ein Teilausdruck auf keinem möglichen Pfad durch den Datenflussgraphen ändert, kann die Eliminierung gemeinsamer Teilausdrücke auch global durchgeführt

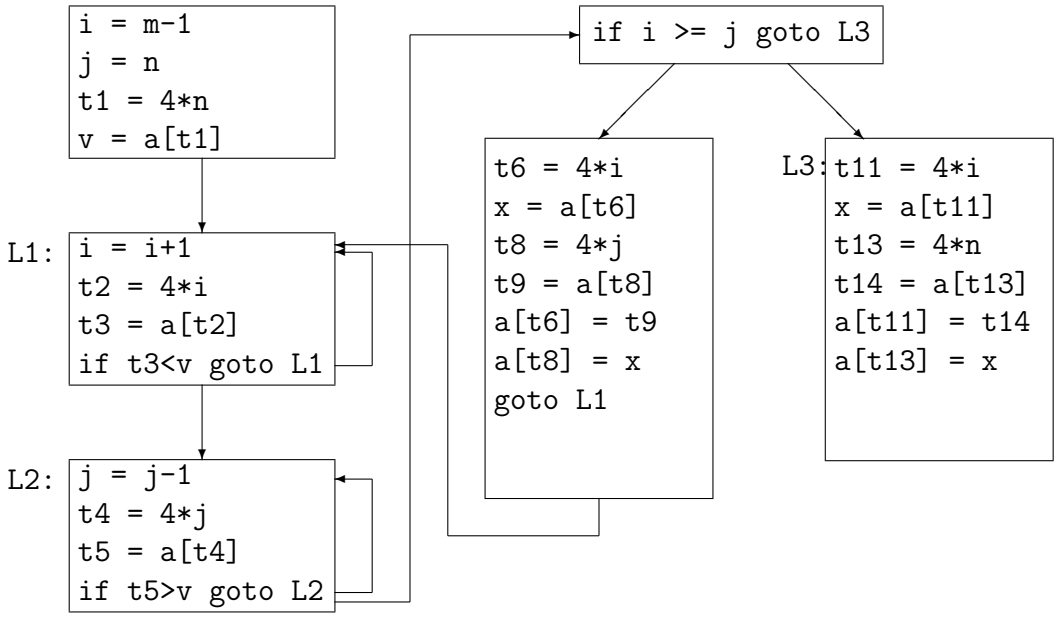


Abbildung 24: Lokal optimierte Basisblockdarstellung zu Quicksort-Fragment

werden. Dabei werden neue temporäre Variablen zur Zwischenspeicherung gemeinsamer Teilausdrücke verwendet, falls die Ursprungsvariable überschrieben wird.

Im Quicksort-Beispiel kann man feststellen, dass die Indexberechnungen auch global gemeinsam sind. In den Blöcken `L1` und `L2` werden in den Variablen `t2` und `t4` die Werte `4*i` und `4*j` berechnet. Werden die lokalen `L1`- und `L2`-Schleifen verlassen, so werden die beiden längeren (bereits lokal optimierten) Blöcke erreicht, in denen `4*i` und `4*j` nochmals berechnet werden. Stattdessen können `t6` und `t8` einfach durch `t2` und `t4` ersetzt werden. Auch die Neuberechnung von `4*n` in `t13` ist überflüssig. Der Wert `4*n` liegt noch in `t1` vor. Nach diesen Optimierungen kann man desweiteren feststellen, dass die Array-Zugriffe in den längeren Basisblöcken eliminiert werden können, weil die entsprechenden Werte bereits in Registern vorliegen. So liegt der Wert `a[t2]` in der Variablen `t3` vor, so dass die beiden Zuweisungen `x = a[t2]` durch einfache Kopieranweisungen `x = t3` ersetzt werden können. Ebenso kann die Zuweisung `t9 = a[t4]` durch `t9 = t5` ersetzt werden.

Obwohl der Wert von `t1 = 4*n` sich bei der Ausführung des Codes nicht ändert, darf man hingegen die Zuweisung `t14 = a[t1]` nicht durch `t14 = v` ersetzen, obwohl der Wert von `v` während der gesamten Ausführung unverändert bleibt. Der Grund hierfür ist, dass es möglich ist, dass das Array `a` zwischen dem ersten Zugriff auf `a[t1]` in der Zuweisung `v = a[t1]` und dem späteren Zugriff in der Zuweisung `t14 = a[t1]` modifiziert wurde. Dies ist bei den oben beschriebenen Ersetzungen nicht möglich. Es ergibt sich der in Abbildung 25 gezeigte Datenflussgraph.

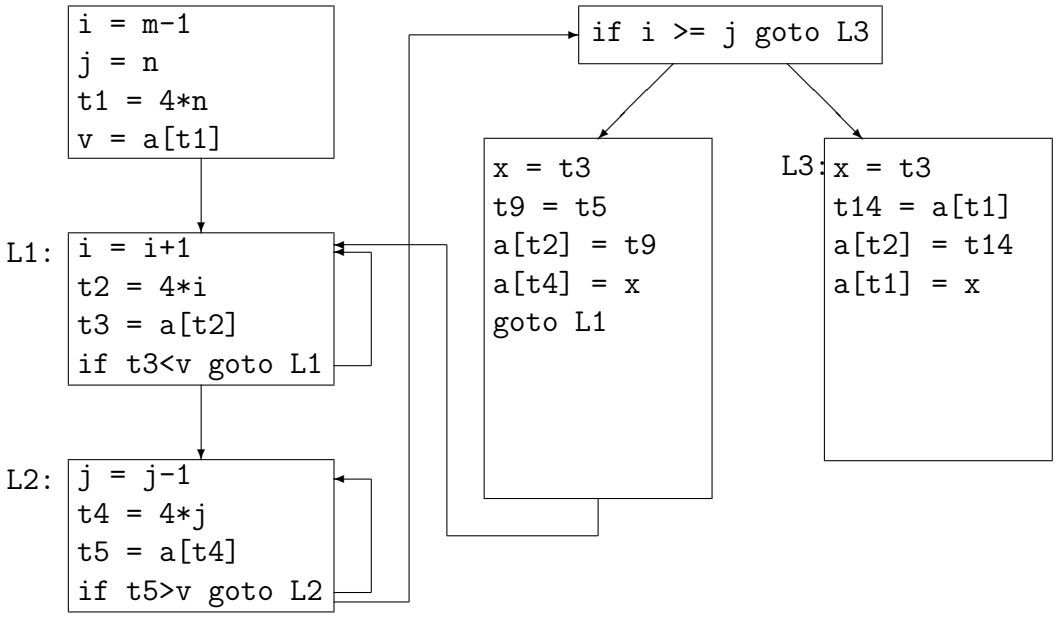


Abbildung 25: Basisblockdarstellung nach Eliminierung globaler gemeinsamer Teilausdrücke

6.6.3 Kopienpropagation und Elimination nutzloser Anweisungen

Kopieranweisungen der Form `u = v` werden häufig bei der Elimination gemeinsamer Teilausdrücke eingeführt. Häufig kann man Kopieranweisungen völlig eliminieren, indem man `v` so oft wie möglich statt `u` verwendet.
Generell heißt eine Variable *lebendig*, falls ihr Wert in einem Programmlauf benutzt werden kann. Ansonsten ist eine Variable tot. Nutzlose Anweisungen sind solche, deren Wert nie benutzt wird. Solche nutzlosen Anweisungen werden häufig durch vorherige Optimierungsphasen erzeugt. Im Quicksort-Beispiel können die Kopieranweisungen, die beim Ersetzen der wiederholten Array-Zugriffe eingeführt wurden, wieder eliminiert werden. Dies führt zu der in Abbildung 26 gezeigten Basisblockdarstellung.

6.6.4 Codeverschiebung

Codeverschiebung (engl. code motion) wird vor allem durchgeführt, um schleifeninvariante Berechnungen aus Schleifen herauszuziehen. Im Quicksort-Beispiel gibt es keine solchen schleifeninvarianten Berechnungen. Daher betrachten wir ein allgemeines Beispiel.

Beispiel: Wir betrachten die Sequenz

```
L : ⋮
    x := y op z
    ⋮
    if ...goto L
```

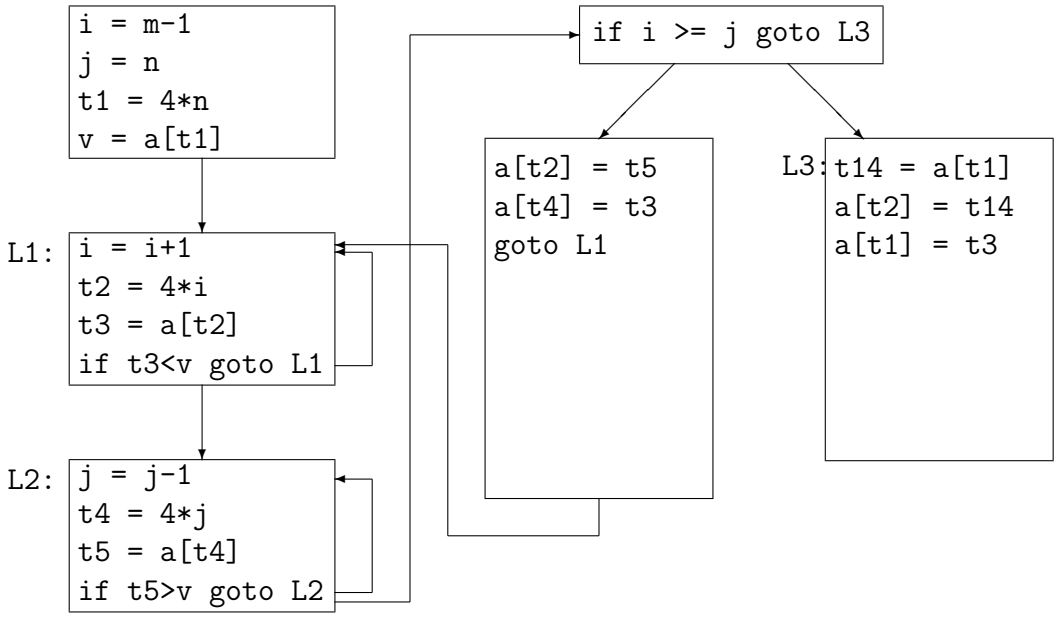


Abbildung 26: Basisblockdarstellung nach Kopienpropagation und Eliminierung nutzloser Anweisungen

Werden weder `y` noch `z` in den nicht gezeigten Code-Sequenzen modifiziert, so ist die Wertzuweisung `x := y op z` schleifeninvariant. Um die mehrfache Ausführung bei jedem Schleifendurchlauf zu verhindern, zieht man die Anweisung vor die Schleife. Dies führt zu folgendem modifizierten Code:

```
      x := y op z
L :   ⋮
      ⋮
      if ...goto L
```

Dabei ist zu beachten, dass die Zuweisung `x := y op z` auf allen Pfaden, die zu `L` führen, eingefügt werden muss. ◁

6.6.5 Induktionsvariablen und Operatorreduktion

Eine Variable `x` heißt *Induktionsvariable* einer Schleife, falls sich der Wert von `x` bei Zuweisung um eine (positive oder negative) Konstante `c` erhöht. Dies muss in einer Inkrementierung pro Schleifendurchlauf geschehen. Häufig existieren mehrere Induktionsvariablen per Schleife. Manchmal ist es möglich, mehrere Induktionsvariablen durch eine zu ersetzen.
Unter Operatorreduktion (engl. strength reduction) versteht man das Ersetzen von teuren Operationen durch billigere. Standardbeispiele sind die Verwendung von Additionen oder Shift-Operationen statt Multiplikationen, falls möglich, oder die Verwendung von speziellen Inkrementierungsbefehlen statt allgemeiner Addition.

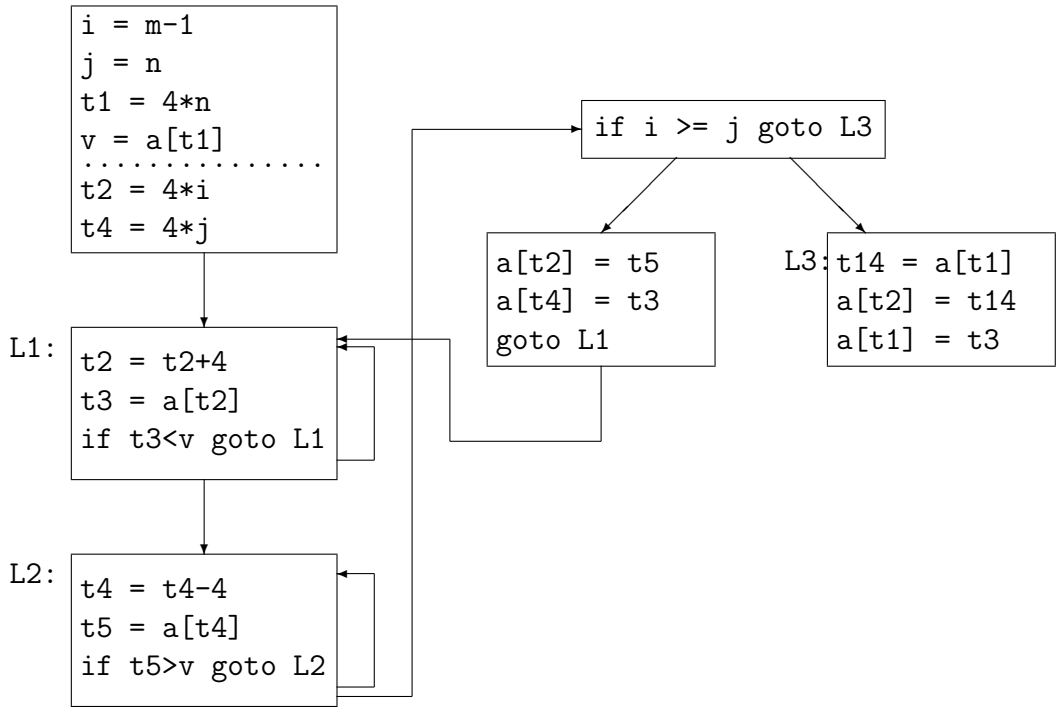


Abbildung 27: Basisblockdarstellung nach Elimination von Induktionsvariablen

Im Laufbeispiel (siehe Abbildung 26) erkennt man schnell, dass die Variablen `i` und `j` in den Blöcken `L1` und `L2` Induktionsvariablen sind. Deswegen können die wiederholten Multiplikationen von `i` und `j` mit 4 durch billigere Additionen `t2 := t2+4` bzw. `t4 := t4-4` ersetzt werden. Hierfür ist es allerdings erforderlich, die Variablen `t2` und `t4` im Anfangsblock zu initialisieren. Jetzt sind sowohl `i` und `j` als auch `t2` und `t4` Induktionsvariablen. Nach diesen Änderungen kommen `i` und `j` nur noch im Vergleich des bedingten Sprungbefehls nach Block `L2` vor. Nutzt man aus, dass `i >= j` genau dann gilt, wenn `t2 >= t4`, können `i` und `j` vollständig aus den Blöcken `L1` und `L2` eliminiert werden. Dies führt zu dem in Abbildung 27 gezeigten Endergebnis unserer Folge von Optimierungen.
Diese Fallstudie hat gezeigt, dass Optimierungen meistens von innen nach außen durchgeführt werden sollten und dass Optimierungen weitere Optimierungsmöglichkeiten nach sich ziehen können.

6.7 Globale Analyse: Datenflussanalyse

Die grundsätzliche Vorgehensweise bei einer Datenflussanalyse besteht aus den folgenden drei Schritten:

1. Festlegung der Modellierung und Analyse des lokalen Verhaltens der Basisblöcke

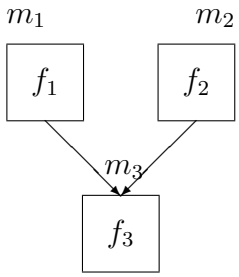
   Ergebnis dieses Schrittes sind die

   - Menge M und

- zu jedem Basisblock B die Transferfunktion $f_B : M \to M$.

2. Aufstellen eines Gleichungssystems zur Ermittlung des Gesamtverhaltens

   - Für jeden Basisblock BB_i wird eine Variable m_i eingeführt.
   - Der Startwert (Randbedingung) $m_0 \in M$ wird für die Wurzel (Vorwärtsanalyse) bzw. die Blätter (Rückwärtsanalyse) festgelegt.
   - Es muss eine geeignete Zusammenfassung des Gesamtverhaltens entlang aller Pfade von der Wurzel/den Blättern zum Knoten BB_i erfolgen. Dies geschieht durch lokale Zusammenfassung aller Vorgänger mittels einer Zusammenführungsfunktion (engl. meet function):



$$m_3 = f_1(m_1) \diamond f_2(m_2)$$

$\diamond : M \times M \to M$ ist die „meet“-Funktion, die über verschiedene Pfade im Datenflussgraphen bestimmte Informationen zusammenführt.

Beispiel: Eine mögliche Zusammenführungsfunktion für die Konstantenpropagation ist

$$m_1 \diamond m_2 := x \mapsto \begin{cases} c & \text{falls } m_1(x) = m_2(x) = c \neq ? \\ ? & \text{sonst.} \end{cases}$$

◁

Beispiel: Wir stellen ein Gleichungssystem für eine Vorwärtsanalyse des Datenflussgraphen zum Quicksort-Programm (siehe Abbildung 27) auf. Zur Repräsentation der abstrakten Werte zu Beginn der Basisblöcke führen wir die folgenden Variablen ein:

Basisblock	Anfang	L1	L2	Sprung	äußere Schleife	L3	Ende
Variable	m_0	m_1	m_2	m_S	m_B	m_3	m_{EXIT}

Das Gleichungssystem in 6 Unbekannten hat dann die folgende Struktur:

$$\begin{aligned} m_1 &= f_A(m_0) \diamond m_2 \diamond f_B(m_B) \\ m_2 &= f_{L1}(m_1) \diamond m_S \\ m_S &= f_{L2}(m_2) \\ m_B &= f_S(m_S) \\ m_3 &= f_S(m_S) \\ m_{\text{EXIT}} &= f_{L3}(m_3) \end{aligned}$$

◁

3. Lösen des Gleichungssystems

   Zur Lösung des Gleichungssystems ist eine Fixpunktbestimmung notwendig. Hierzu muss der Grundbereich M zu einem Verband, d.h. einer vollständigen Halbordnung mit existierendem Supremum und Infimum für je zwei Werte. Letzteres ist wichtig bei zusammenlaufenden Kanten. In Verbänden endlicher Höhe ist die Existenz von Fixpunkten für monotone Funktionen garantiert. Wir werden dies hier nicht weiter ausführen.

6.8 Taxonomie von Datenflussproblemen

Wir beenden dieses Kapitel mit einer Taxonomie von Datenflussproblemen. Man unterscheidet, wie bereits bei der Diskussion lokaler Analysen, Vorwärts- und Rückwärtsanalysen:

Vorwärts:
- Datenfluss entlang der Pfeile
- Variablen am Anfang von Basisblöcken
- Randbedingungen an der Wurzel

Es werden Aussagen über die Vergangenheit der Berechnung gemacht.

Rückwärts:
- Datenfluss entgegen den Kanten
- Variablen am Ende von Basisblöcken
- Randbedingungen an den Blättern

Es werden Aussagen über die Zukunft der Berechnung gemacht.

Außerdem werden All- und Existenzprobleme unterschieden:

Allprobleme: Eigenschaften müssen auf allen Pfaden erhalten bleiben.

$\curvearrowright$ Meet-Funktion entspricht logischem Und bzw. einem Durchschnitt.

Existenzprobleme: Eigenschaften müssen auf mindestens einem Pfad gelten.

$\curvearrowright$ Meet-Funktion entspricht logischem Oder bzw. einer Vereinigung.

Beispiel:

	$\forall$	$\exists$
vorwärts	• Konstantenpropagation • verfügbare Ausdrücke • gemeinsame Teilausdrücke	• Sichtbare Definitionen
rückwärts	• wichtige Ausdrücke	• lebendige Variablen

Bei sichtbaren Definitionen fragt man nach der Existenz eines Pfades, auf dem ein Zuweisungswert erhalten bleibt. Wichtige Ausdrücke sind solche, die auf jeden Fall, d.h. bei jedem Programmlauf berechnet werden. Bei lebendigen Variablen fragt man, ob ein Pfad existiert, auf dem eine Variable benutzt wird. $\triangleleft$