6.4 Lokale Optimierungen

In diesem Abschnitt werden einfache Blöcke isoliert betrachtet. Der Drei-Adress-Code in den Blöcken wird optimiert, dabei werden gemeinsame Teilausdrücke im Block entfernt, zur Übersetzungszeit bereits mögliche Berechnungen und einfache algebraische Umformungen durchgeführt.

6.4.1 Konstruktion eines DAGs für einen einfachen Block

Ausgehend von einem einfachen Block wird ein gerichteter azyklischer Graph (*directed acyclic graph* - **DAG**) aufgebaut, der die Berechnungen im Block darstellt. Dieser DAG kann dann als Eingabe für eine Maschinencode- Erzeugung verwendet werden oder aber in einen Block zurücktransformiert werden. Bei der Konstruktion werden gemeinsame Teilausdrücke erkannt und berücksichtigt.

Eingabe: Ein einfacher Block

Ausgabe: Ein DAG für den einfachen Block.

Die Knoten des DAGs enthalten die folgende Information:

- interne Knoten enthalten einen Operator als Markierung
- Blätter enthalten einen Variablennamen oder eine Konstante als Markierung
- den Knoten ist eine Liste angehängt, in der Null oder mehr Variablennamen auftreten.

Jeder Knoten des DAGs korrespondiert mit einem berechneten Zwischenergebnis und die in der Liste aufgeführten Variablen „enthalten“ diesen Wert.

Einige Knoten sind als Ausgabeknoten markiert. Die Werte dieser Knoten werden an andere Stelle im Programm (nicht in diesem Block!) benötigt. Die Variablen, die diese Werte enthalten sind lebendig am Ausgang dieses einfachen Blocks. Welche Variablen lebendig sind, kann man mit einer Datenfluss-Analyse berechnen.

Methode: Man benötigt eine Tabelle, in der ein Verweis auf den jeweils „neuesten“ Knoten zu finden ist, der den Wert einer Variablen bzw. einer Konstanten repräsentiert.
`node`(*identifier*) gibt jeweils diesen Knoten zurück.

Durchlaufe den einfachen Block von Anfang bis Ende.

Für jeden Drei-Adress-Befehl der Form `x := y op z` führe die folgenden Schritte aus:

1) Befindet sich `y` nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung `y` und vermerke dies in der Tabelle. (d.h. `node(y)` gibt diesen Knoten zurück!)
   Befindet sich `z` nicht in der Tabelle, so führe dieselben Schritte für `z` aus.
2) Bestimme, ob es im DAG einen Knoten α gibt, der als linken Nachfolger `node(y)`, als rechten Nachfolger `node(z)` und als Markierung `op` hat. Existiert ein derartiger Knoten nicht, so erzeuge einen Knoten α mit dieser Eigenschaft.
3) Ist die Variable `x` in der dem Knoten `node(x)` angehefteten Liste, so lösche `x` aus der Liste. Füge `x` in die Liste des Knotens α ein und ändere die Tabelle so, dass `node(x)` den Wert α hat.

Für jeden Drei-Adress-Befehl der Form `x := op y` führe die folgenden Schritte aus:

1) Befindet sich `y` nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung `y` und vermerke dies in der Tabelle.
2) Bestimme, ob es im DAG einen Knoten α gibt, der als Nachfolger `node(y)` und als Markierung `op` hat. Existiert ein derartiger Knoten nicht, so erzeuge einen Knoten α mit dieser Eigenschaft.
3) Ist die Variable `x` in der dem Knoten `node(x)` angehefteten Liste, so lösche `x` aus der Liste. Füge `x` in die Liste des Knotens α ein und ändere die Tabelle so, dass `node(x)` den Wert α hat.

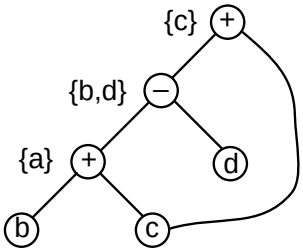
Für jeden Drei-Adress-Befehl der Form `x := y` führe die folgenden Schritte aus:

1) Befindet sich `y` nicht in der Tabelle, so erzeuge einen neuen Knoten mit Markierung `y` und vermerke dies in der Tabelle.
2) Sei $\alpha =$ `node(y)`.
3) Ist die Variable `x` in der dem Knoten `node(x)` angehefteten Liste, so lösche `x` aus der Liste. Füge `x` in die Liste des Knotens α ein und ändere die Tabelle so, dass `node(x)` den Wert α hat.

Beispiel 6.5:
Für den einfachen Block

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

erhielte man den DAG



Die Knotenmarkierungen sind in den Knoten, die Listen in den geschweiften Klammern daneben notiert.
Für die vierte Anweisung wird kein Knoten erzeugt, es wird nur die Ergebnisvariable `d` zu der Liste hinzugefügt. Diese Vorgehensweise entspricht dem Entfernen gemeinsamer Teilausdrücke. Ist die Variable `b` nicht lebendig am Ende des Blocks, könnte man den einfachen Block in

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

umschreiben. Sind `b` und `d` am Ende lebendig, müsste man noch den Befehl `b := d` anfügen.

In diese Konstruktion lassen sich leicht Optimierungen einbeziehen, die sich auf algebraische Identitäten beziehen. So wäre zum Beispiel bei der Konstruktion des DAGs bei kommutativen Operatoren die Reihenfolge der beiden Nachfolger irrelevant. Befehle der Art `y := 0 + x` lassen sich bei der Konstruktion leicht erkennen und die Identitäten können natürlich gleich berücksichtigt werden.

Außerdem kann man das Verfahren erweitern und in der Übersetzungsphase mögliche Berechnungen bereits hier durchführen.
So kann man etwa einen Befehl der Art `y := 5 + 7` durch `y := 12` ersetzen.

6.4.2 Darstellung von Feldzugriffen im DAG

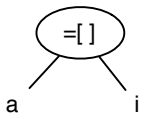
Auf den ersten Blick sieht es so aus, als könne man Zugriffe auf Feldelemente wir die anderen Operatoren bearbeiten. Aber es gibt da ein Problem:

Beispiel 6.6:
Betrachte die Drei-Adress-Befehle

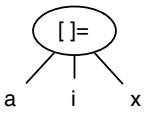
```
x := a[i]
a[j] := y
z := a[i]
```

Interpretiert man den Feldzugriff `a[i]` als eine Operation mit Operanden `a` und `i`, dann wäre `a[i]` im Beispiel ein gemeinsamer Teilausdruck. Allerdings könnten `i` und `j` den gleichen Wert haben und damit repräsentieren die beiden Auftreten von `a[i]` eventuell unterschiedliche Werte!

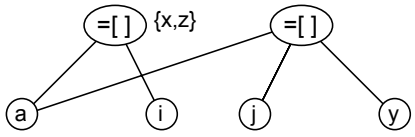
Also muss man bei einer Zuweisung zu einem Feldelement eine zusätzliche Regel einführen. Man übersetzt einen Feldzugriff der Art `x := a[i]` in einen Knoten mit zwei Nachfolgern der Art



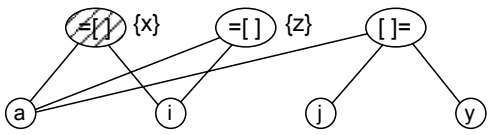
Eine Zuweisungen zu Feldelementen wie etwa `a[i] := x` wird durch einen Knoten mit einer dreifach-Verzweigung dargestellt:



Beim Einsetzen eines Knotens dieser Art in den entstehenden DAG werden alle Knoten, deren Wert vom Knoten `node(a)` abhängen, eingefroren, da angenommen werden muss, dass *jedes* Element des Feldes verändert wurde. Eingefrorene Knoten werden in den Abbildungen schraffiert gezeichnet. Dies bedeutet, dass ihre Listen nicht mehr verändert werden dürfen!
Ohne diese Regel würde man aus dem Block aus Beispiel 6.6 den (falschen) DAG

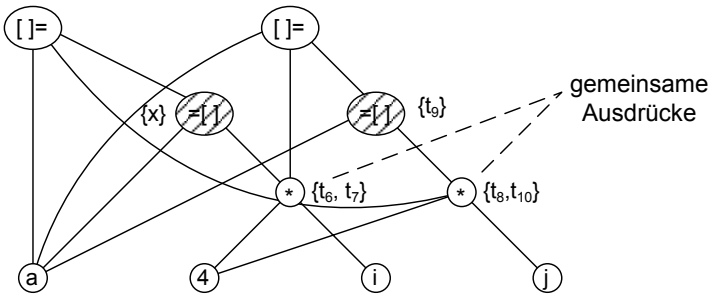


erhalten. Beachtet man jedoch die zusätzliche Regel, so erhält man:



Entsprechende Regeln müssen auch bei Zeiger-Operationen und bei Prozeduraufrufen beachtet werden. Im Zweifelsfall bleibt bei Sprachen, die etwa beliebige Zeigeroperationen ermöglichen nichts anderes über, als alle vorhandenen Knoten einzufrieren!

Beispiel 6.7:
Der ursprüngliche Block B_5 im Quicksort-Beispiel 6.4 ergäbe den folgenden DAG:



Was kann man nun mit dem so gewonnenen DAG machen?

1) Erkennen, welche Variablen und Konstanten in einem einfachen Block benötigt werden (Blätter) und welchen Variablen neue Werte zugewiesen werden (use/def-Werte für die Datenfluss-Analyse).

2) Rückwandlung in einen einfachen Block, wobei gemeinsame Teilausdrücke entfernt werden.

3) Maschinencode-Erzeugung direkt vom DAG aus.

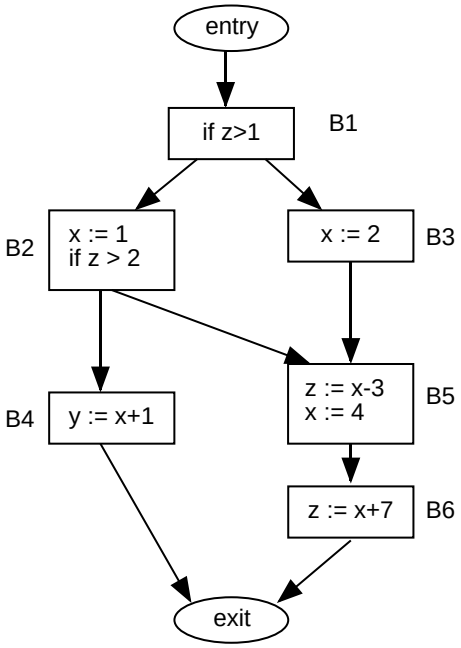
6.5 Datenfluss Analyse

Damit ein Programm sicher optimiert werden kann, müssen vor der Anwendung von Programmtransformationen Informationen über das gesamte Programm gesammelt werden. Dazu wird versucht, den „Zustand“ des Programms, d.h. die Werte der Variablen an einem Programm-Punkt zu bestimmen. Ein Programm-Punkt ist dabei eine Stelle im Drei-Adress-Programm vor oder nach einem Drei-Adress-Befehl. Jeder Drei-Adress-Befehl transformiert den Eingangszustand am Programm-Punkt *vor* dem Befehl in einen neuen Ausgangszustand am Programm-Punkt *nach* dem Befehl.
Häufig interessiert aber nur ein Teil der Variablen oder bestimmte Eigenschaften der Variablen, so dass man nur diese Datenfluss-Werte betrachtet und bestimmt. Man nennt die Menge der für eine Anwendung interessierenden Werte den *Datenfluss-Domain*.
So kann man zum Beispiel an einem Programm-Punkt die Frage stellen, an welchen Stellen der momentane Wert einer Variablen `x` im Programm definiert wurde. Dies ist das Problem der *reaching definitions*. Gibt es zum Beispiel an diesem Punkt nur genau eine Definition von `x` und wurde bei dieser Definition der Variaben `x` eine Konstante zugewiesen, so kann man diese Konstante statt der Variablen `x` verwenden und eventuell Berechnungen zur Übersetzungszeit statt zur Laufzeit durchführen (*constant propagation*).
Allgemeiner spricht man in diesem Kontext von der *use-definition-Verkettung* (ud-Kette) und der *definition-use-Verkettung* (du-Kette) von Variablen.

Definition: Sei p eine Position im Drei-Adress-Code, an der eine Variable `a` definiert wird. Die **du-Kette** für p und `a` ist eine Liste von Positionen im Drei-Adress-Code, an denen die Variable `a` gebraucht wird und auf dem Weg dorthin der in p definierte Wert nicht überschrieben werden kann.

Definition: Sei p eine Position im Drei-Adress-Code, an der eine Variable `a` gebraucht wird. Die **ud-Kette** für p und `a` ist eine Liste von Positionen im Drei-Adress-Code, an denen die Variable `a` definiert wird und der zugewiesene Wert auf jedem Weg bis p nicht überschrieben werden kann.

Beispiel 6.8:
Man betrachte den folgenden Flussgraphen:



Die du-Kette für die Definition von `x` im Block `B2` enthält die beiden Positionen in den Blöcken `B4` und `B5`, in denen `x` gebraucht wird. Der Gebrauch im Block `B6` ist nicht in der du-Kette, da `x` in `B5` neu definiert wird!
Die ud-Kette für den Gebrauch von `x` im Block `B4` enthält nur die Definition im Block `B2` während die ud-Kette für den Gebrauch von `x` in `B5` die beiden Definitionen in `B2` und `B3` enthält.

Zur Lösung dieser Fragen und zur Bestimmung von ud- bzw. du-Ketten werden Datenfluss-Gleichungen aufgestellt, die geeignete Informationen am Anfang eines einfachen Blocks (`in`-Mengen) mit den entsprechenden Informationen am Ende des Blocks (`out`-Mengen) und dem Inhalt des Blocks in Beziehung setzen.

6.5.1 Transferfunktionen

Betrachten wir zunächst einen Drei-Adress Befehl s. $\texttt{in}[s]$ bezeichnet die Datenfluss-Werte am Programm-Punkt vor s, $\texttt{out}[s]$ die Werte am Programm-Punkt nach s. Dem Drei-Adress-Befehl zugeordnet ist eine Transferfunktion f_s. Da die Datenfluss-Information entweder in Richtung der Programmausführung (Vorwärts) oder aber in entgegengesetzter Richtung (Rückwärts) fließen kann, gibt es auch zwei mögliche Transferfunktionen:

1) für die **Vorwärtsanalyse**: $\texttt{out}[s] = f_s(\texttt{in}[s])$

2) für die **Rückwärtsanalyse**: $\texttt{in}[s] = f_s(\texttt{out}[s])$.

Bemerkung: Um die Notation nicht zu unübersichtlich werden zu lassen und da aus dem Kontext immer klar ist, ob es sich um eine Vorwärts– oder Rückwärtsanalyse handelt, werden die beiden Transferfunktionen gleich bezeichnet.

Innerhalb eines einfachen Blocks B, der aus den Befehlen $s_1, s_2, \ldots, s_k$ besteht, gilt immer $\texttt{out}[s_i] = \texttt{in}[s_{i+1}]$ für $1 \leq i < k$. Es ist daher sinnvoll, die Transferfunktion auf den einfachen Block B auszuweiten. Auf diese Weise verringert man die Zahl der Datenfluss-Gleichungen, ohne allzu viel Information zu verschenken. Man definiert dazu $\texttt{in}[B] = \texttt{in}[s_1]$ und $\texttt{out}[B] = \texttt{out}[s_k]$. Die Transferfunktion ist dann

1) für die Vorwärtsanalyse gilt $\texttt{out}[B] = f_B(\texttt{in}[B])$

2) für die Rückwärtsanalyse gilt $\texttt{in}[B] = f_B(\texttt{out}[B])$.

wobei $f_B = f_{s_k} \circ \ldots \circ f_{s_2} \circ f_{s_1}$ (für die Vorwärtsanalyse) bzw. $f_B = f_{s_1} \circ f_{s_2} \circ \ldots \circ f_{s_k}$ (für die Rückwärtsanlyse) gilt.
Jetzt bleibt nur noch, die Datenfluss-Informationen im Fluss-Graphen zu betrachten. Auch hier muss man zwischen den beiden Möglichkeiten unterscheiden.
Bei der Vorwärtsanalyse gilt

$$\texttt{in}[B] = \cup_{P \in \text{pred}(B)} \texttt{out}[P]$$

bei der Rückwärtsanlyse gilt dagegen

$$\texttt{out}[B] = \cup_{S \in \text{succ}(B)} \texttt{in}[S]$$

wobei $\text{pred}(B)$ die Vorgängerblöcke und $\text{succ}(B)$ die Nachfolgerblöcke von B im Flussgraph bezeichnet. Man erhält so zwei Datenfluss-Gleichungen für jeden Block des Fluss-Graphen. Das so erhaltenen Gleichungssystem wird dann üblicherweise iterativ gelöst.

6.5.2 Verfügbare Definitionen (Reaching Definitions)

Dies ist eine Analyse, die für eine oder auch mehrere Variablen alle Positionen im Drei-Adress-Code bestimmt, an denen diese Variable definiert wird und diese Definitionen bis zu einer bestimmten Stelle (Blockanfang, Blockende) im Flussgraphen nicht überschrieben wurden, also noch verfügbar sind. Diese Analyse ist offensichtlich eine Vorwärtsanalyse.
Man benötigt diese Information für die ud-Ketten (**use-definition-chaining**), bei dem man für jeden Gebrauch einer Variablen wissen möchte, an welchen Stellen im Programm der momentane Wert dieser Variablen definiert worden sein könnte. Die Datenfluss-Domain ist in diesem Fall die Menge aller Positionen von Drei-Adress-Befehlen.
Man kann diese Information auch nutzen, um einen möglichen Gebrauch einer Variablen `a` vor einer Definition von `a` zu bestimmen. Man ordnet dazu jeder Variablen im `entry`-Block einen dummy-Wert zu. Erreicht eine derartige Definition für `a` einen Punkt im Flussgraphen, an dem `a` benutzt wird, gibt einen Weg im Flussgraphen von `entry` zu dieser Position, in dem der Variablen `a` kein Wert zugewiesen wird.
Zunächst soll die Transferfunktion für einen Drei-Adress-Befehl beschrieben werden. Man betrachte einen Befehl der Art `a := b op c` auf Position p. Dieser Befehl erzeugt (*generates*) eine Definition p der Variablen `a` und löscht (*kills*) alle anderen Definitionen von `a` im Programm. Sämtliche Definitionen von anderen Variablen werden nicht verändert.
Also könnte man die Transferfunktion für diesen Befehl in der Form $f_p(x) = \texttt{gen}[p] \cup (x - \texttt{kill}[p])$ notieren, wobei $\texttt{gen}[p] = \{p\}$ die von diesem Befehl erzeugte Definition und $\texttt{kill}[p]$ die Menge aller *anderen* Definitionen von `a` im Programm ist, also $\texttt{kill}[p] = \texttt{defs}(\texttt{a}) - \{p\}$, wobei $\texttt{defs}(x)$ die Menge aller Positionen ist, an denen x definiert wird.
Bei einem Drei-Adress-Befehl der Form `a := b[c]` wird die Transferfunktion analog definiert. Für Befehle der Form `if a relop b` oder `a[b] := c` ist die Transferfunktion dagegen die Identität.

Beispiel 6.9:
Man betrachte folgendes Drei-Adress-Programm, wobei Gebrauch und Definition der beiden Variablen `a` und `c` interessieren:

```
(1)      a := 7
(2)      c := 2
(3) L:   if c > a goto L1
(4)      c := c + a
(5)      goto L
(6) L1:  a := c - a
(7)      c := 0
```

Dann berechnen sich die `gen`- und `kill`-Mengen für die uns interessierenden Variablen `a` und `c` wie folgt:

s	gen[s]	kill[s]
(1)	1	6
(2)	2	4,7
(3)		
(4)	4	2,7
(5)		
(6)	6	1
(7)	7	2,4

Die Transferfunktion für einen einfachen Block kann man nun einfach durch Komposition der Transferfunktionen für die einzelnen Befehle bestimmen. Dabei gilt für zwei Funktionen $f_1(x) = \texttt{gen}[1] \cup (x - \texttt{kill}[1])$ und $f_2(x) = \texttt{gen}[2] \cup (x - \texttt{kill}[2])$, dass die Komposition die Form

$$\begin{aligned} f_2(f_1(x)) &= \texttt{gen}[2] \cup (\texttt{gen}[1] \cup (x - \texttt{kill}[1]) - \texttt{kill}[2]) \\ &= (\texttt{gen}[2] \cup (\texttt{gen}[1] - \texttt{kill}[2])) \cup (x - (\texttt{kill}[1] \cup \texttt{kill}[2])) \\ &= \texttt{gen}[2] \cup (\texttt{gen}[1] - \texttt{kill}[2]) \cup (x - \texttt{kill}[1] - \texttt{kill}[2]) \end{aligned}$$

hat.
Also erhält man für einen einfachen Block B mit k Befehlen die Transferfunktion:

$$f_B(x) = \texttt{gen}[B] \cup (x - \texttt{kill}[B])$$

wobei

$$\texttt{kill}[B] = \texttt{kill}[1] \cup \texttt{kill}[2] \cup \ldots \cup \texttt{kill}[k]$$

und

$$\begin{aligned} \texttt{gen}[B] &= \texttt{gen}[k] \cup (\texttt{gen}[k-1] - \texttt{kill}[k]) \cup (\texttt{gen}[k-2] - (\texttt{kill}[k-1] \cup \texttt{kill}[k])) \cup \\ &\quad \ldots \cup (\texttt{gen}[1] - (\texttt{kill}[2] \cup \texttt{kill}[3] - \cdots \cup \texttt{kill}[k])) \\ &= \texttt{gen}[k] \cup (\texttt{gen}[k-1] - \texttt{kill}[k]) \cup (\texttt{gen}[k-2] - \texttt{kill}[k-1] - \texttt{kill}[k]) \cup \\ &\quad \ldots \cup (\texttt{gen}[1] - \texttt{kill}[2] - \texttt{kill}[3] - \ldots - \texttt{kill}[k]) \end{aligned}$$

$\texttt{gen}[B]$ enthält also alle Definitionen im Block, die am Ende des Blocks noch gültig, d.h. nicht durch andere Definitionen überschrieben sind. Die Menge $\texttt{kill}[B]$ ist einfach die Vereinigung der `kill`-Mengen aller in B vorhandenen Befehle.

Algorithmus zur Bestimmung der `gen`- und `kill`-Menge eines Blocks

Eingabe: Ein einfacher Block `B`
Ausgabe: Die `gen` und `kill`-Menge für den Block `B`
Verfahren :

1) Setze `gen[B]` $:= \emptyset$ und `kill[B]` $:= \emptyset$.

2) Durchlaufe die Instruktionen des Blocks `B` vom ersten bis zum letzten Befehl und führe für jeden Befehl s an Position p, der eine Variable definiert, die folgenden Schritte aus:

   i) `gen[B]` $:= \texttt{gen}[s] \cup (\texttt{gen[B]} - \texttt{kill}[s])$

   ii) `kill[B]` $:= \texttt{kill[B]} \cup \texttt{kill}[s]$

   wobei $\texttt{kill}[s]$ die Menge der Definitionen ist, die durch diesen Befehl überschrieben werden und $\texttt{gen}[s] = \{p\}$ gilt, sofern auf Position p eine Variable definiert wird.

Bemerkung: Prozeduraufrufe und Zeigerzugriffe der Art `*p := x` müssen wegen ihrer eventuellen Seiteneffekte gesondert betrachtet werden!

Beispiel 6.10:
Für den einfachen Block `B`:

```
(1)   a := 3
(2)   a := 4
```

würde `gen[B]` $= \{2\}$ und $\{1, 2\} \subseteq$ `kill[B]` gelten.

Sei pred(B) die Menge der Vorgängerblöcke von B im Flussgraphen. Dann gelten für jeden Block B die Datenfluss-Gleichungen:

$$\begin{aligned} \texttt{out}[\texttt{B}] &= \texttt{gen}[\texttt{B}] \cup (\texttt{in}[\texttt{B}] - \texttt{kill}[\texttt{B}]) \\ \texttt{in}[\texttt{B}] &= \bigcup_{\texttt{B}' \in \text{pred}(\texttt{B})} \texttt{out}[\texttt{B}'] \end{aligned}$$

Da sich der Wert von in[B] aus Werten der Vorgängerknoten berechnet, haben wir hier eine Vorwärtsanalyse.
Für den Eingangsblock „entry“ gilt offensichtlich die Randbedingung $\texttt{out}[\texttt{entry}] = \emptyset$.
Wie löst man nun so ein Gleichungssystem? Man setzt zunächst $\texttt{out}[\texttt{B}] = \emptyset$ für alle Blöcke B und iteriert solange, bis es keine Änderungen an den Mengen mehr gibt! Man berechnet also einen kleinsten Fixpunkt dieses Gleichungssystems. Es lässt sich zeigen, dass die die iterativ berechneten in- und out-Mengen monoton wachsen und das System somit einen Fixpunkt besitzt. Die Anzahl der Iterationen ist also kleiner als die Knotenzahl im Flussgraphen. Bei „günstiger“ Auswahl der Blöcke hat sich gezeigt, dass die Anzahl der der Iterationen meist kleiner gleich 5 ist.

Algorithmus zur Analyse der verfügbaren Definitionen

Eingabe: Ein Flussgraph, für dessen Blöcke die jeweiligen gen- und kill-Mengen bestimmt worden sind.
Ausgabe: Die in und out-Mengen für jeden Block im Flussgraphen.
Verfahren :
pred(B) sei die Menge der Vorgängerblöcke von B im Flussgraphen.

1) Setze out[B] := $\emptyset$.

2) Solange sich eine der out-Mengen ändert, führe man die folgenden Schritte für jeden Block B aus:

   i) in[B] : = $\bigcup_{\text{P} \in \text{pred(B)}}$ out[P]

   ii) out[B] := gen[B] $\cup$ (in[B] - kill[B])

Beispiel 6.11:
Betrachten wir zunächst das Beispiel 6.9 oben. In diesem Beispiel wird die Analyse nicht auf Blöcken, sondern auf einzelnen Befehlen durchgeführt. Man erhält somit:

s	in[s]	out[s]
(1)		1
(2)	1	1,2
(3)	1,2,4	1,2,4
(4)	1,2,4	1,4
(5)	1,4	1,4
(6)	1,2,4	2,4,6
(7)	2,4,6	6,7

Also könnte man z.B. das a im Befehl (3) und im Befehl (6) durch 7 ersetzen! (Copy Propagation)

Beispiel 6.12:
Betrachten wir den ursprünglichen, nicht optimierten Flussgraphen aus dem Quicksort-Beispiel auf Seite 83. Von Interesse seien nur die Definitionen der Variablen `i, j, v` und `x`.
Wie oben erwähnt sollen Definitionen durch Positionen des Drei-Adress-Befehls repräsentiert werden.

`i` wird in Position 1 (`B1`) und 5 (`B2`) definiert,

`j` wird in Position 2 (`B1`) und 9 (`B3`) definiert,

`v` wird in Position 4 (`B1`) definiert,

`x` wird in Position 15 (`B5`) und 24 (`B6`) definiert.

Damit ergeben sich die Mengen `gen` und `kill` für die einfachen Blöcke wie folgt:

	gen	kill
B1	1, 2, 4	5, 9
B2	5	1
B3	9	2
B4	-	-
B5	15	24
B6	24	15

Man setzt zunächst $\texttt{in}[\texttt{B}] = \texttt{out}[\texttt{B}] = \emptyset$ für alle Blöcke `B` und iteriert solange, bis es keine Änderungen an den Mengen mehr gibt.
Nach einigen Iterationen bleiben die `in`- und `out`-Mengen konstant und man erhält das folgende Ergebnis:

	in	out
B1	-	1, 2, 4
B2	1, 2, 4, 5, 9, 15	2, 4, 5, 9, 15
B3	2, 4, 5, 9, 15	4, 5, 9, 15
B4	4, 5, 9, 15	4, 5, 9, 15
B5	4, 5, 9, 15	4, 5, 9, 15
B6	4, 5, 9, 15	4, 5, 9, 24

Man sieht zum Beispiel, dass am Anfang von Block `B3` für `i` nur die Definition auf Position 5 aktuell ist, während für `j` sowohl die Definition in Position 2 als auch die in Position 9 aktuell sein kann.

Bemerkung: Bei einer realen Implementation dieses Algorithmus würde man mit entsprechenden Bit-Vektoren arbeiten und die Mengen-Operationen durch Boolesche Operationen ersetzen.

6.5.3 ud - Ketten

Zur Bestimmung von ud-Ketten benutzt man die Analyse der verfügbaren Definitionen aus dem vorigen Abschnitt. Es wird für eine Benutzung einer Variablen in einem Drei-Adress-Befehl eine Liste (**Kette**) von Positionen von Befehlen konstruiert, an denen diese Variable definiert wird und der dort definierte Wert unverändert bleibt.

Algorithmus zur Bestimmung der Verwendungs-Definitions-Ketten

Eingabe: Ein Flussgraph, für dessen Blöcke die jeweiligen `in`–Mengen der Analyse der verfügbaren Definitionen bestimmt worden sind.
Ausgabe: Für jede Verwendung einer Variablen eine Liste mit den Definitionen dieser Variablen, die den Verwendungspunkt erreichen
Verfahren :
Durchlaufe jeden Block `B` im Flussgraphen und führe für jede interessierende Verwendung einer Variablen `a` die folgenden Schritte aus.

1) Wenn vor der Verwendung von `a` keine Definition von `a` in `B` erfolgt, so ist die Definionskette für diese Verwendung von `a` die Menge der Definitionen für `a`, die in `in[B]` enthalten sind.

2) Existieren Definitionen von `a` im Block `B`, die vor der Verwendung von `a` liegen, so wird nur die letzte dieser Definitionen in die Definitionskette aufgenommen (also `in[B]` wird nicht aufgenommen!)

6.5.4 Lebendigkeit von Variablen

Es soll für eine Position im Drei-Adress-Programm festgestellt werden, ob der aktuelle Wert einer Variable entlang eines Pfades durch den Flussgraphen nochmal genutzt wird. Ist dem so, wird die Variable als **lebendig** an dieser Stelle bezeichnet, im anderen Fall als **tot**. Bei der Generierung des Maschinencodes für ein Programm kann so z. B. entschieden werden, ob der berechnete Wert einer Variablen, der sich am Ende eines Blocks in einem Register befindet, abgespeichert werden muss, da er später noch gebraucht wird.
Bei der Registerzuordnung kann eine Situation auftreten, in der alle Register vergeben sind und noch ein Register benötigt wird. In diesem Fall sollten die Register, die Werte toter Variablen enthalten, zuerst wiederverwendet werden.
Die Datenfluss-Domain ist in diesem Fall die Menge der verwendeten Variablen.

Bemerkung: Man kann die folgenden Verfahren auch statt auf Blöcken auf einzelnen Befehlen arbeiten lassen. Dadurch erhöht sich die Anzahl der Datenfluß-Gleichungen, dagegen ist die Bestimmung der Transferfunktionen einfacher.

Zunächst wird wieder die Transferfunktion für einen Drei-Adress-Befehle s bestimmt. Man betrachte einen Befehl der Form `a := b op c`. Dieser Befehl benötigt die Variablen aus der Menge $\texttt{use}[s] = \{\texttt{b}, \texttt{c}\}$ und definiert wird die Menge $\texttt{def}[s] = \{\texttt{a}\}$. Also sind am Programm-Punkt vor diesem Befehl die Variablen in $\texttt{use}[s]$ sowie alle anderen Variablen lebendig, die am Programm-Punkt nach diesem Befehl lebendig sind abzüglich der Varibalen in $\texttt{def}[s]$. Wir haben damit eine Rückwärtsanalyse vor uns. Die Transferfunktion wäre $f_s(x) = \texttt{use}[s] \cup (x - \texttt{def}[s])$.
Wie oben beschrieben kann man aus diesen Transferfunktionen die Transferfunktion für einen Block zusammensetzen. Man erhält für einen einfachen Block B mit k Befehlen die Transferfunktion:

$$f_B(x) = \texttt{use}[B] \cup (x - \texttt{def}[B])$$

wobei

$$\texttt{def}[B] = \texttt{def}[1] \cup \texttt{def}[2] \cup \ldots \cup \texttt{def}[k]$$

und

$$\begin{aligned} \texttt{use}[B] \quad = \quad & \texttt{use}[1] \cup (\texttt{use}[2] - \texttt{def}[1]) \cup (\texttt{use}[3] - \texttt{def}[1] - \texttt{def}[2]) \cup \\ & \ldots \cup (\texttt{use}[k] - \texttt{def}[1] - \texttt{def}[2] - \cdots - \texttt{def}[k-1]) \end{aligned}$$

Algorithmus zur Bestimmung der `def`- und `use`-Menge eines Blocks

Eingabe: Ein einfacher Block `B`
Ausgabe: Die `def` und `use`-Menge für den Block `B`.
Verfahren :

1) Setze `def[B]` $:= \emptyset$ und `use[B]` $:= \emptyset$.
2) Durchlaufe die Drei-Adress-Befehle des Blocks `B` vom ersten bis zum letzten Befehl und führe für jeden Befehl s die folgenden Schritte nacheinander aus:
   i) `use[B]` $:=$ `use[B]` $\cup$ (`use`$[s]$ $-$ `def[B]`)
   ii) `def[B]` $:=$ `def[B]` $\cup$ `def`$[s]$

Nachdem diese Mengen bestimmt worden sind, kann man nun die lebendigen Variablen am Eintritt in einen Block bzw. beim Austritt aus einem Block bestimmen. Sei succ(`B`) die Menge der Nachfolgerblöcke von `B` im Flussgraphen. Dann gelten für jeden Block `B` die Datenfluss-Gleichungen:

$$\begin{aligned} \texttt{in}[\texttt{B}] \quad &= \quad \texttt{use}[\texttt{B}] \cup (\texttt{out}[\texttt{B}] - \texttt{def}[\texttt{B}]) \\ \texttt{out}[\texttt{B}] \quad &= \quad \bigcup_{\texttt{S} \in \text{succ}(\texttt{B})} \texttt{in}[\texttt{S}] \end{aligned}$$

Da sich der Wert von `out[B]` aus Werten der Nachfolgerknoten berechnet, haben wir hier eine Rückwärtsanalyse.
Für den Ausgangsblock „`exit`“ gilt offensichtlich die Randbedingung `in`$[$`exit`$] = \emptyset$.
Man setzt zunächst für jeden Block `in[B]` $= \emptyset$. Gelöst wird das Gleichungssystem wiederum mit fortgesetzten Iterationen bis sich die `in`-Mengen nicht mehr verändern.

Algorithmus zur Analyse der lebendigen Variablen

Eingabe: Ein Flussgraph, für dessen Blöcke die jeweiligen `def`- und `use`-Mengen bestimmt worden sind.
Ausgabe: Die `out` und `in`-Mengen für jeden Block im Flussgraphen.
Verfahren :
succ(`B`) sei die Menge der Nachfolgerblöcke von `B` im Flussgraphen.

1) Setze in[B] $:= \emptyset$.
2) Solange sich eine der `in`-Mengen ändert, führe man die folgenden Schritte für jeden Block `B` aus:
   i) `out[B]` $: = \bigcup_{\texttt{S} \in \text{succ}(\texttt{B})}$ `in[S]`
   ii) `in[B]` $:=$ `use[B]` $\cup$ (`out[B]` - `def[B]`)

Beispiel 6.13:
Betrachtet man den optimierten Flussgraphen des Quicksort-Beispiels aus dem vorigem Abschnitt auf Seite 86, so erhält man:

	use	def
B1	m, n	i, j, t1, v, t2, t4
B2	t2, v	t3, t2
B3	t4, v	t5, t4
B4	t2, t4	-
B5	t2, t3, t4, t5	-
B6	t1, t2, t3	t14

Nach 5 Iterationen gibt es keine Änderungen an den `in`- und `out`-Mengen mehr (Die Anzahl der Iterationen hängt auch von der Reihenfolge ab, in der die Blöcke betrachtet werden!). Es ergibt sich:

	in	out
B1	m, n	t1, t2, t4, v
B2	t1, t2, t4, v	t1, t2, t3, t4, v
B3	t1, t2, t3, t4, v	t1, t2, t3, t4, t5, v
B4	t1, t2, t3, t4, t5, v	t1, t2, t3, t4, t5, v
B5	t1, t2, t3, t4, t5, v	t1, t2, t4, v
B6	t1, t2, t3	-

Man erkennt, dass zum Beispiel `i` und `j` nicht lebendig am Ende von Block `B1` sind.

Bemerkung: Treten in einem Drei-Adress-Programm Prozeduraufrufe und Zeigerzugriffe auf, so muss man auch dafür `def`- und `use`-Mengen bestimmen. Ein konservativer Ansatz wäre es, anzunehmen, dass ein Prozeduraufruf keine Variablen definiert und dass alle nicht-temporären Variablen benutzt werden. Für Speicherzugriffe der Form `*p := a` gilt, dass nur die Variablen `p` und `a` verwendet werden. Ein Zugriff der Form `a := *p` definiert jedoch `a` und verwendet neben `p` auch jede Variable, auf die `p` verweisen könnte!