# Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

12. Januar 2016

# Datenfluss Analyse

Damit ein Programm sicher optimiert werden kann, müssen vor der Anwendung von Programmtransformationen Informationen über das gesamte Programm gesammelt werden. Dabei interessieren etwa Fragen

- nach der Lebendigkeit von Variablen
- an welchen Stellen der momentan Wert einer Variablen definiert wurde
- welchen Positionen Ausdrücke in Drei-Adress-Code erreichen.

### Definition

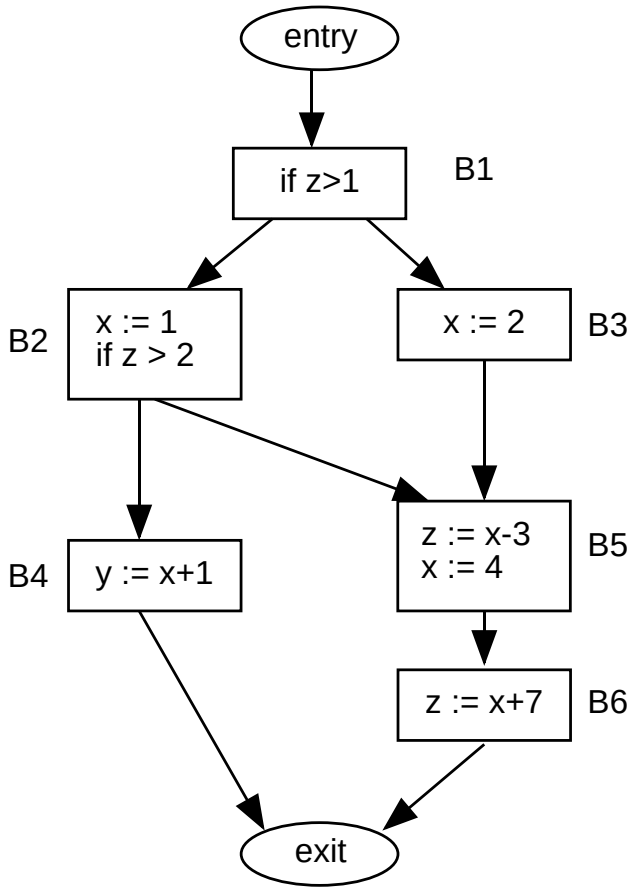
Sei $p$ eine Position im Drei-Adress-Code, an der eine Variable a definiert wird. Die du-Kette für $p$ und a ist eine Liste von Positionen im Drei-Adress-Code, an denen die Variable a gebraucht wird und auf dem Weg dorthin der in $p$ definierte Wert nicht überschrieben werden kann.

### 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

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.

# Transferfunktion für einen Befehl

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

# Transferfunktion für Blöcke

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. 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.

# Datenfluss-Informationen im Fluss-Graphen

Auch hier muss man zwischen den beiden Möglichkeiten unterscheiden.
Bei der Vorwärtsanalyse gilt

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

bei der Rückwärtsanlyse gilt dagegen

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

wobei $\mathrm{pred}(B)$ die Vorgängerblöcke und $\mathrm{succ}(B)$ die Nachfolgerblöcke von $B$ im Flussgraph bezeichnet.

# 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.

# Transferfunktion für einen Befehl

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. Also ist die Transferfunktion für diesen Befehl

$$f_p(x) = \texttt{gen}[p] \cup (x - \texttt{kill}[p]).$$

mit

- $\texttt{gen}[p] = \{p\}$, die von diesem Befehl erzeugte Definition und
- $\texttt{kill}[p]$ die Menge aller *anderen* Definitionen von `a` im Programm, also $\texttt{kill}[p] = \texttt{defs}(\texttt{a}) - \{p\}$, wobei $\texttt{defs}(x)$ die Menge aller Positionen ist, an denen $x$ definiert wird.

## Beispiel

Man betrachte folgendes Drei-Adress-Programm:

```
(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 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 |

# Transferfunktion für einen einfachen Block

Für zwei Funktionen $f_1(x) = \texttt{gen}[1] \cup (x - \texttt{kill}[1])$ und $f_2(x) = \texttt{gen}[2] \cup (x - \texttt{kill}[2])$ gilt,
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:

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

wobei

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

und

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

# 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 $\texttt{gen[B]} := \emptyset$ und $\texttt{kill[B]} := \emptyset$.
2. Durchlaufe die Instruktionen des Blocks B vom ersten bis zum letzten Befehl und führe für jeden Befehl $s$ auf Position $p$, der eine Variable definiert, die folgenden Schritte aus:

   2.1 $\texttt{gen[B]} := \texttt{gen}[s] \cup (\texttt{gen[B]} - \texttt{kill}[s])$
   2.2 $\texttt{kill[B]} := \texttt{kill[B]} \cup \texttt{kill}[s]$

   wobei $\texttt{gen}[s] = \{p\}$ ist, falls eine Variable definiert wird und $\texttt{kill}[s]$ die Menge der durch diesen Befehl überschriebenen Definitionen ist.

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

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

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 out[entry] $= \emptyset$.

# 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:
   - 2.1 `in[B]` $:= \bigcup_{\texttt{P} \in \text{pred}(\texttt{B})}$ `out[P]`
   - 2.2 `out[B]` $:=$ `gen[B]` $\cup$ (`in[B]` - `kill[B]`)

## Beispiel

Betrachten wir den ursprünglichen, nicht optimierten Flussgraphen aus dem Quicksort-Beispiel. Von Interesse seien nur die Definitionen der Variablen `i`, `j`, `v` und `x`.
Definitionen werden wie vorher durch die Zeilennummer des Drei-Adress-Befehls repräsentiert.

- `i` wird in Zeilen 1 (`B1`) und 5 (`B2`) definiert,
- `j` wird in Zeilen 2 (`B1`) und 9 (`B3`) definiert,
- `v` wird in Zeile 4 (`B1`) definiert,
- `x` wird in Zeilen 15 (`B5`) und 24 (`B6`) definiert.

Damit ergeben sich die Mengen `gen` und `kill` 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. 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 in Zeile 5 aktuell ist, während für `j` sowohl die Definition in Zeile 2 als auch die in Zeile 9 aktuell sein kann.

# ud - Ketten

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

# Algorithmus zur Bestimmung der ud-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

# 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.

## 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.*

# Transferfunktion für einen Befehle

Man betrachte einen Befehl $s$ 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 $f_s(x) = \texttt{use}[s] \cup (x - \texttt{def}[s])$.

# Transferfunktion für einen Block

Man erhält für einen einfachen Block $B$ mit $k$ Befehlen die Transferfunktion:

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

wobei

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

und

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

# 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 $\texttt{def}[\texttt{B}] := \emptyset$ und $\texttt{use}[\texttt{B}] := \emptyset$.
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:
   - 2.1 $\texttt{use}[\texttt{B}] := \texttt{use}[\texttt{B}] \cup (\texttt{use}[s] - \texttt{def}[\texttt{B}])$
   - 2.2 $\texttt{def}[\texttt{B}] := \texttt{def}[\texttt{B}] \cup \texttt{def}[s]$

# Datenfluss-Gleichungen

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

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

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 $\texttt{in}[\text{exit}] = \emptyset$.

# 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:
   2.1 `out[B]` $:= \bigcup_{\mathtt{S} \in \mathrm{succ}(\mathtt{B})}$ `in[S]`
   2.2 `in[B]` $:=$ `use[B]` $\cup$ (`out[B]` - `def[B]`)

## Beispiel

Betrachtet man den optimierten Flussgraphen des Quicksort - Beispiels aus dem vorigem Abschnitt, 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.