Kapitel III

Programmanalyse und Typsysteme

Motivation

Verifikationsmethoden wie Model Checking untersuchen den vollständigen Zustandsraum eines Systems.

- Das beschränkt die Verifikation auf (Teil-)Systeme mit nicht zu großem Zustandsraum.
- Der Zustandsraum kann durch Abstraktion verkleinert werden; dann muss aber auch die Abstraktion verifiziert werden.

In diesem Kapitel befassen wir uns mit Methoden der Programmiersprachentheorie zur Analyse von Software realistischer Größe.

Motivation

Statische Analyse von Programmen realistischer Größe

Schwächere Eigenschaften können automatisch überprüft werden.

- Abwesenheit von Laufzeitfehlern wie Division durch Null, falscher Speicherzugriff, ungefangene Exceptions, Verletzung von Ressourcenschranken und -protokollen (z.B. Dateizugriff).
- Datenflusseigenschaften zur Programmoptimierung

Solche Eigenschaften sind z.B. im Compilerbau nützlich.

Kompliziertere Eigenschaften erfordern manuelle Hilfe durch

- Angabe von Typannotaten,
- Spezifikation von Zusicherungen und Invarianten,
- formale Beweise.

Inhalt Kapitel III

- Induktive Definitionen
- Programmanalyse für imperative Programme
  - Spezifikation einer While-Sprache
  - Datenflussanalyse
- Fixpunkttheorie
- Programmanalyse und Typsysteme
- Typ- und Effektsysteme
  - Funktionale Sprache
  - Typinferenz
  - Polymorphie
  - Kontrollflussanalyse

Induktive Definitionen

In der Programmiersprachentheorie werden viele Konzepte durch induktive Definitionen formalisiert.

Das übliche Format für induktive Definitionen sind *Inferenzregeln*.

Beispiele aus der Vorlesung:

$$\frac{e_1 \longrightarrow \mathsf{fn}_\pi\ x \Rightarrow e_1' \qquad e_2 \longrightarrow v_2 \qquad e_1'[x \mapsto v_2] \longrightarrow v}{e_1\ e_2 \longrightarrow v}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi_1} \tau_2\ \&\ \varphi_2 \qquad \Gamma \vdash e_2 : \tau_1\ \&\ \varphi_3}{\Gamma \vdash e_1\ e_2 : \tau_2\ \&\ \varphi_1 \cup \varphi_2 \cup \varphi_3}$$

Urteile

Ein *Urteil* ist ein formaler Ausdruck, der eine bestimme Aussage ausdrückt (meist eine Eigenschaft bestimmter Objekte).

Beispiele:

- n *even* — "Die natürliche Zahl n ist gerade."
- n *odd* — "Die natürliche Zahl n ist ungerade."
- $\mathit{sum}(m, n) = r$ — "Die Summe der nat. Zahlen m und n ist die nat. Zahl r."
- $\phi \vdash \psi$ — "Jede Belegung, welche die aussagenlogischen Formeln ϕ wahr macht, macht auch ψ wahr."
- $e : X$ — "Der Programmausdruck e hat Typ X."

Urteile sind als *rein syntaktische* Ausdrücke ohne inhärente Bedeutung zu verstehen (oben sind "*sum*" und "$\vdash$" nur Symbole).

Inferenzregeln

Die Bedeutung von Urteilen wird durch *Inferenzregeln* festgelegt.

Eine Inferenzregel hat die Form:

$$(\textsc{name})\ \frac{U_1 \qquad U_2 \qquad \ldots \qquad U_n}{U}$$

- Die Urteile $U_1, \ldots, U_n$ heißen *Prämissen*.
- Das Urteil U heißt *Konklusion*.
- Inferenzregeln ohne Prämissen (d.h. mit $n = 0$) heißen *Axiome*.

Bedeutung der Regel: Wenn die Prämissen alle zutreffen, dann auch die Konklusion.

Inferenzregeln: Beispiele

Gerade/ungerade:

$$\text{(EvenZ)}\ \frac{}{0\ \mathit{even}} \qquad \text{(OddS)}\ \frac{n\ \mathit{even}}{n+1\ \mathit{odd}} \qquad \text{(EvenS)}\ \frac{n\ \mathit{odd}}{n+1\ \mathit{even}}$$

Summen:

$$\text{(SumZ)}\ \frac{}{\mathit{sum}(n,0) = n} \qquad \text{(SumS)}\ \frac{\mathit{sum}(n,m) = r}{\mathit{sum}(n,m+1) = r+1}$$

Aussagenlogik:

$$\frac{}{\phi \vdash \phi} \qquad \frac{\phi \vdash \psi_1 \qquad \phi \vdash \psi_2}{\phi \vdash \psi_1 \wedge \psi_2} \qquad \frac{\phi \vdash \psi_1}{\phi \vdash \psi_1 \vee \psi_2} \qquad \frac{\phi \vdash \psi_2}{\phi \vdash \psi_1 \vee \psi_2}$$

$$\ldots$$

Inferenzregeln (genau)

Inferenzregeln enthalten meist Metavariablen und sind eigentlich als Regelschemata zu verstehen.

Gemeint sind alle Regeln, die man durch Einsetzung beliebiger passender(!) Ausdrücke für die Metavariablen erhält.

Beispiel:

$$\frac{\mathit{sum}(n, m) = r}{\mathit{sum}(n, m + 1) = r + 1}$$

steht für

$$\frac{\mathit{sum}(0, 0) = 0}{\mathit{sum}(0, 1) = 1}, \frac{\mathit{sum}(0, 0) = 1}{\mathit{sum}(0, 1) = 2}, \ldots, \frac{\mathit{sum}(5, 6) = 18}{\mathit{sum}(5, 7) = 19}, \ldots$$

Für m, n und r dürfen nur nat. Zahlen eingesetzt werden, da die Urteile in der Regel nur für nat. Zahlen definiert sind.

Inferenzregeln (genau)

Beispiel:

$$\frac{\phi \vdash \psi_1 \qquad \phi \vdash \psi_2}{\phi \vdash \psi_1 \wedge \psi_2}$$

steht für alle Regeln, die man durch Einsetzung von konkreten aussagenlogischen Formeln für ϕ, ψ_1 und ψ_2 erhält, z.B.

$$\frac{\top \vdash \top \qquad \top \vdash \top}{\top \vdash \top \wedge \top}, \frac{(A \Rightarrow B) \vdash (C \wedge B) \qquad (A \Rightarrow B) \vdash \bot}{(A \Rightarrow B) \vdash (C \wedge B) \wedge \bot}, \ldots$$

Hier dürfen nur aussagenlogische Formeln eingesetzt werden, da die Urteile nur dafür definiert sind.

Inferenzregeln: Seitenbedingungen

Die möglichen Werte der Metavariablen in schematischen Regeln werden manchmal durch Seitenbedingungen eingeschränkt.

Beispiel: Die Regel (SumS) könnte auch so geschrieben werden.

$$(\textsc{SumS})\ \frac{\mathit{sum}(n, m) = r}{\mathit{sum}(n, m') = r'}\ m' = m + 1,\ r' = r + 1$$

Die Metavariablen dürfen nur so instantiiert werden, dass alle Seitenbedingungen erfüllt sind.

(Seitenbedingungen werden manchmal auch wie Prämissen über den Strich geschrieben.)

Herleitungen

Der Begriff der *Herleitung für ein Urteil U* ist induktiv wie folgt definiert.

- Jede Regel der Form (d.h. ohne Prämisse)

  $$(\mathsf{R})\ \frac{}{U}$$

  ist eine Herleitung für ihre Konklusion U.
- Gibt es eine Regel

  $$(\mathsf{S})\ \frac{U_1 \qquad U_2 \qquad \ldots \qquad U_n}{U}$$

  und sind $\Pi_1, \ldots, \Pi_n$ jeweils Herleitungen für $U_1, \ldots, U_n$, dann ist

  $$(\mathsf{S})\ \frac{\Pi_1 \qquad \Pi_2 \qquad \ldots \qquad \Pi_n}{U}$$

  eine Herleitung für U.
- Nichts weiter ist eine Herleitung.

Herleitungen: Beispiele

Beispiele:

$$\dfrac{\dfrac{}{A \vdash A} \qquad \dfrac{\dfrac{}{A \vdash A} \qquad \dfrac{}{A \vdash B \vee A}}{A \vdash A \wedge (B \vee A)}}{A \vdash A \wedge (A \wedge (B \vee A))}$$

$$\begin{array}{l}(\textsc{SumZ})\ \dfrac{}{sum(3,0) = 3} \\ (\textsc{SumS})\ \dfrac{sum(3,0) = 3}{sum(3,1) = 4} \\ (\textsc{SumS})\ \dfrac{sum(3,1) = 4}{sum(3,2) = 5}\end{array}$$

Kein Beispiel:

$$\begin{array}{l}(\textsc{SumS})\ \dfrac{sum(3,0) = 4}{sum(3,1) = 5} \\ (\textsc{SumS})\ \dfrac{sum(3,1) = 5}{sum(3,2) = 6}\end{array}$$

(Regelnamen und Seitenbedingungen werden oft nicht ausgeschrieben.)

Induktive Definition

Inferenzregeln sind induktive Definitionen von Urteilen.

Induktionsprinzip: Um zu zeigen, dass alle herleitbaren Urteile eine bestimmte Eigenschaft haben, genügt es für alle Regeln zu zeigen: Wenn die Prämissen der Regel die Eigenschaft haben, dann auch die Konklusion.

Beispiel: Wenn $\mathsf{sum}(m, n) = r$ herleitbar ist, dann gilt $r = m + n$.

- Regel (SumZ)
$$\frac{}{\mathit{sum}(n, 0) = n}$$
Es gilt $n = n + 0$, also hat die Konklusion die Eigenschaft.
- Regel (SumS)
$$\frac{\mathit{sum}(n, m) = r}{\mathit{sum}(n, m + 1) = r + 1}$$
Angenommen die Prämisse hat die Eigenschaft, d.h. $r = m + n$. Dann gilt $r + 1 = m + n + 1 = (m + 1) + n$, also hat auch die Konklusion die Eigenschaft.

Induktive Definitionen

Inferenzregeln sind als reine Spezifikation von Urteilen zu verstehen.

- Inferenzregeln spezifizieren was herleitbar ist, aber nicht unbedingt wie.
- Die Frage, ob ein Urteil herleitbar ist, kann je nach Art der Regeln sehr leicht oder sehr schwer sein.
- Man verwendet Inferenzregeln, um möglichst einfach zu spezifizieren, was eine Analyse leisten soll, ohne sich dabei bereits auf einen Analysealgorithmus festzulegen.
- Beispiel: Das Urteil $\mathit{sum}(n, m) = r$ sagt, was eine Summe ist; verschiedene Implementierungen der Addition realisieren diese Spezifikation, z.B.
  - Ripple-Carry Adder
  - Lookahead-Carry-Adder

Einfache While-Sprache

Arithmetische Ausdrücke:

$$a, a_1, a_2 ::= x \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$$

Boolesche Ausdrücke:

$$b, b_1, b_2 ::= \texttt{true} \mid \texttt{false} \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid a_1 < a_2 \mid a_1 = a_2$$

Programmstücke (statements):

$$S, S_1, S_2 ::= [x := a]^\ell \mid [\texttt{skip}]^\ell \mid S_1; S_2 \mid \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } [b]^\ell \texttt{ do } S$$

Hierbei x läuft über Programmvariablen, n über Integerkonstanten, ℓ über Labels (konkret: natürliche Zahlen), welche elementare Programmteile eindeutig markieren sollen.
Konvention: $S_1; S_2; S_3$ steht für $S_1; (S_2; S_3)$.

Beispielprogramm

$$[\texttt{y:=x}]^1; [\texttt{z:=1}]^2; \texttt{while } [\texttt{1<y}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6$$

Alternative Notation:

```
1: y:=x;
2: z:=1;
3: while 1<y do (
4:     z:=z*y;
5:     y:=y-1);
6: y:=0
```

Operationelle Semantik

Die operationelle Semantik spezifiziert wie sich While-Programme bei der Ausführung verhalten.

Ein *Programmzustand* (σ) ist eine endliche Abbildung von Programmvariablen auf ganze Zahlen.

Beispiel: $\sigma = [x \mapsto 3, y \mapsto 4]$.

Wenn σ alle Variablen in einem arithmetischen Ausdruck a auf Werte abbildet, dann schreiben wir $[\![a]\!]\sigma$ für den Wert des Ausdrucks mit den entsprechenden Belegungen. Sonst ist $[\![a]\!]\sigma$ undefiniert.

Beispiel: $[\![x + 4 * y]\!]\sigma = 19$

Analog für Boolesche Ausdrücke.

Beispiel: $[\![x + 4 * y < 18]\!]\sigma = \mathit{false}$

Operationelle Semantik

Die operationelle Semantik ist durch zwei Urteile gegeben.

- $\langle S, \sigma \rangle \to \sigma'$ "Die Ausführung von S im Startzustand σ ist nach einem Schritt beendet und endet im Zustand σ'."
- $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$ "Die Ausführung von S im Anfangszustand σ führt nach einem Schritt zum Zwischenzustand σ'; es bleibt danach noch die Anweisung S' abzuarbeiten."

Regeln für die operationelle Semantik

$$(\textsc{ass})\ \frac{}{\langle [x:=a]^\ell, \sigma\rangle \to \sigma[x \mapsto [\![a]\!]\sigma]}$$

$$(\textsc{skip})\ \frac{}{\langle \mathtt{skip}, \sigma\rangle \to \sigma}$$

$$(\textsc{seq1})\ \frac{\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \to \langle S_1'; S_2, \sigma'\rangle}$$

$$(\textsc{seq2})\ \frac{\langle S_1, \sigma\rangle \to \sigma'}{\langle S_1; S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle}$$

Beachte: (ASS) ist nur anwendbar, wenn $[\![a]\!]\sigma$ definiert ist.

Regeln für die operationelle Semantik, Forts.

$$(\textsc{IfT})\ \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle S_1, \sigma\rangle}\ [\![b]\!]\sigma = \mathit{true}$$

$$(\textsc{IfF})\ \frac{}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma\rangle \to \langle S_2, \sigma\rangle}\ [\![b]\!]\sigma = \mathit{false}$$

$$(\textsc{WhileT})\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \to \langle S; \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle}\ [\![b]\!]\sigma = \mathit{true}$$

$$(\textsc{WhileF})\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, \sigma\rangle \to \sigma}\ [\![b]\!]\sigma = \mathit{false}$$

Beispiel

Schreibe S als Abkürzung für folgendes Programm.

$$\texttt{while}\ [x < 1]^1\ \text{do}\ ([y := y - 1]^2;\ [x := x - 1]^3)$$

Dann kann man folgende Urteile herleiten:

- $\langle S, \rho_1 \rangle \to \langle ([y := y - 1]^2;\ [x := x - 1]^3); S, \rho_1 \rangle$
- $\langle ([y := y - 1]^2;\ [x := x - 1]^3); S, \rho_1 \rangle \to \langle [x := x - 1]^3; S, \rho_2 \rangle$
- $\langle [x := x - 1]^3; S, \rho_2 \rangle \to \langle S, \rho_3 \rangle$
- $\langle S, \rho_3 \rangle \to \rho_3$

wobei $\rho_1 := [x \mapsto 1, y \mapsto 5]$ und $\rho_2 := [x \mapsto 1, y \mapsto 4]$ und $\rho_3 := [x \mapsto 0, y \mapsto 4]$.

Herleitung für das zweite Urteil:

$$\text{(SEQ1)}\ \frac{\text{(SEQ2)}\ \dfrac{\text{(ASS)}\ \dfrac{}{\langle [y := y - 1]^2, \rho_1 \rangle \to \rho_2}}{\langle [y := y - 1]^2;\ [x := x - 1]^3, \rho_1 \rangle \to \langle [x := x - 1]^3, \rho_2 \rangle}}{\langle ([y := y - 1]^2;\ [x := x - 1]^3); S, \rho_1 \rangle \to \langle [x := x - 1]^3; S, \rho_2 \rangle}$$

Operationelle Semantik

Die operationelle Semantik ist eine kompakte Spezifikation der Programmausführung.

- Die Inferenzregeln selbst spezifizieren noch keinen Algorithmus zur Programmausführung.
- Ein Compiler/Interpreter implementiert sollte die operationelle Semantik möglichst effizient implementieren.
- Die operationelle Semantik ist Spezifikation des Compilers/Interpreters.
- Man kann einen einfachen Interpreter für die Sprache von der operationellen Semantik ableiten.
  Für gegebene S und σ versucht man einen Herleitunsbaum für $\langle S, \sigma \rangle \to X$ von unten nach oben aufzubauen und damit X zu bestimmen.

Small-Step- und Big-Step-Semantik

Die Urteile $\langle S, \sigma \rangle \to \sigma'$ und $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$ machen Aussagen über einen einzigen Schritt der Programmausführung.
Das nennt man eine *Small-Step-Semantik*.

Interessiert man sich nur für die Ausführung des gesamten Programms, kann man auch ein Urteil für eine *Big-Step-Semantik* definieren:

$$\text{(BIG1)}\ \frac{\langle S, \sigma \rangle \to \sigma'}{\langle S, \sigma \rangle \Downarrow \sigma'}$$

$$\text{(BIG2)}\ \frac{\langle S, \sigma \rangle \to \langle S', \sigma' \rangle \qquad \langle S', \sigma' \rangle \Downarrow \sigma''}{\langle S, \sigma \rangle \Downarrow \sigma''}$$

Das Urteil $\langle S, \sigma \rangle \Downarrow \sigma'$ sagt: Wenn man S mit Anfangszustand σ laufen lässt, dann terminiert die Programmausführung im Endzustand σ'.

Datenflussanalyse

Analyse der möglichen Werte der Variablen in einem Programm

- statisch, d.h. zur Compilierzeit
- approximativ; allgemein ist die Frage welche Werte eine Variable annehmen kann unentscheidbar (Satz von Rice)

Wir betrachten:

- Liveness
- Reaching Definitions
- Available Expressions

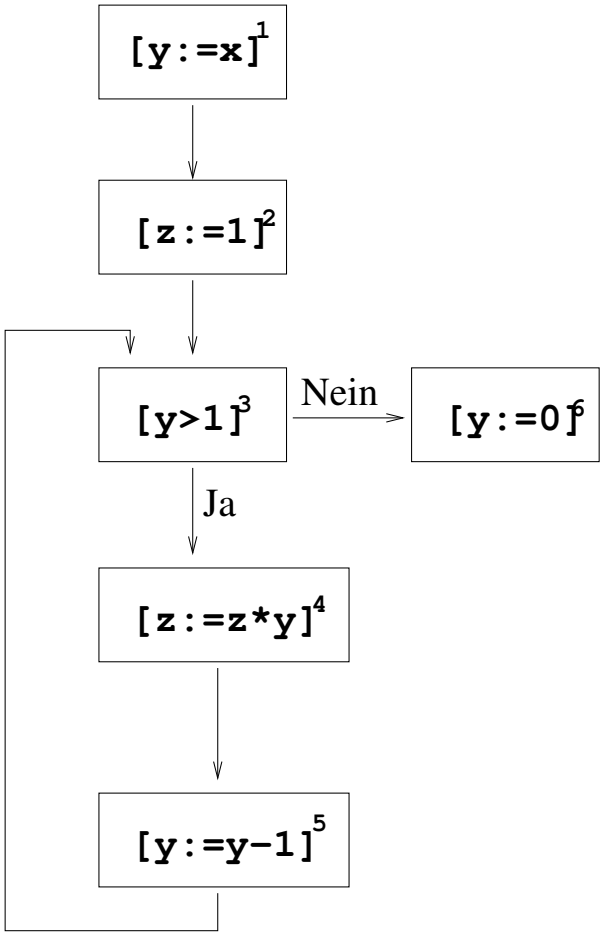
Typische Anwendungen:

- Programmoptimierung in Compilern
- Verifikation einfacher Korrektheitseigenschaften: keine `NullPointerExceptions`, vollständige Bereinigung von CGI-Parametern, ...

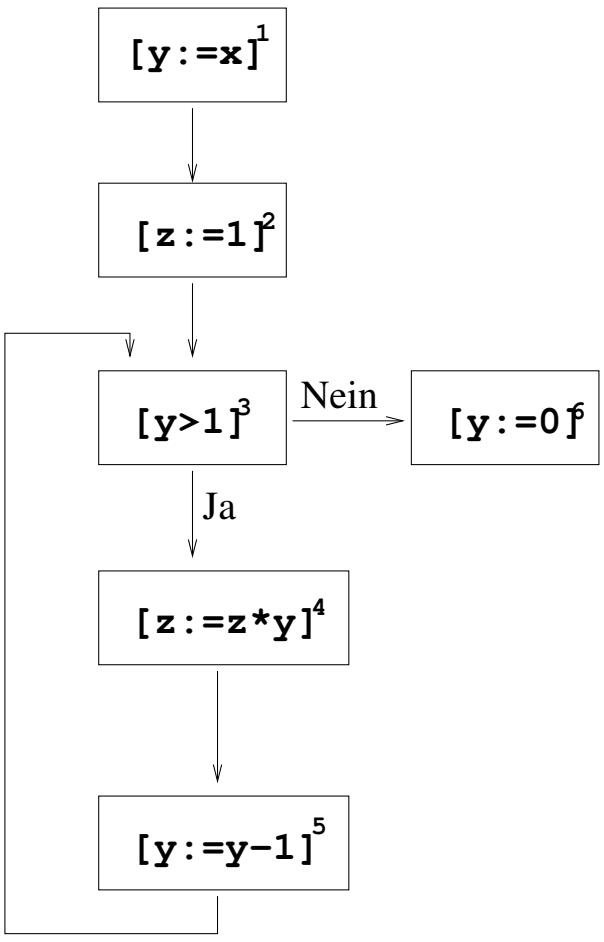
Kontrollflussgraph

Der *Kontrollflussgraph* eines Programms enthält alle potentiellen Ausführungsfolgen der elementaren Anweisungen eines Programms.

$$
\begin{aligned}
&[\mathtt{y:=x}]^1;\\
&[\mathtt{z:=1}]^2;\\
&\mathtt{while}\ [\mathtt{1<y}]^3\ \mathtt{do}\ (\\
&\quad [\mathtt{z:=z*y}]^4;\\
&\quad [\mathtt{y:=y-1}]^5\\
&);\ [\mathtt{y:=0}]^6
\end{aligned}
$$



Kontrollflussgraph



- Knoten des Kontrollflussgraphen sind die elementaren Anweisungen, also die, die ein Label tragen;
- Knoten, die bei der Ausführung unmittelbar aufeinanderfolgen *können*, werden durch Kanten verbunden.
- Bei Fallunterscheidungen (in `if` und `while`) geht man unabhängig von der Bedingung davon aus, dass beide Fälle auftreten können.
- Bei Fallunterscheidungen können die Kanten mit Antworten beschriftet werden.

Liveness von Variablen

Eine Programmvariable ist an einem bestimmten Programmpunkt in einer Programmausführung *live*, wenn ihr Wert später noch gelesen wird.

Anders gesagt, kann man eine Variable, die *nicht* live ist, ungestraft überschreiben.

Anwendungen:

- Zuweisungen zu Variablen, die nach der Zuweisung nicht live sind, sind redundant und können eliminiert werden.
- Zwei Variablen, die nie gleichzeitig live sind, können zu einer einzigen verschmolzen werden.

Liveness von Variablen: Anwendungen

Beispiel: Im linken Programm sind x und y niemals gleichzeitig live. Die beiden Variablen können zu einer verschmolzen werden (siehe rechts).

```
[y:=0]^1;
while [x<10]^2 do (
   [y:=x+1]^3;
   [z:=z+y]^4;
   [x:=2*y]^5
); [r:=z]^6
```

```
while [x<10]^2 do (
   [x:=x+1]^3;
   [z:=z+x]^4;
   [x:=2*x]^5
); [r:=z]^6
```

Solches Verschmelzen ist für Compiler wichtig, da Prozessoren nur wenige Register haben (z.B. 8 Register auf x86).

Liveness-Analyse

Wir wollen für jeden Programmpunkt ℓ zwei Mengen bestimmen:

- $LV_{entry}(\ell)$: Variablen, die am Eingang von ℓ live sind oder möglicherweise live sein könnten.
- $LV_{exit}(\ell)$: Variablen, die am Ausgang von ℓ live sind oder möglicherweise live sein könnten.

Beispiel:

$[\mathtt{y:=0}]^1; \mathtt{while}\ [\mathtt{x<10}]^2\ \mathtt{do}\ ([\mathtt{y:=x+1}]^3; [\mathtt{z:=z+y}]^4; [\mathtt{x:=2*y}]^5); [\mathtt{r:=z}]^6$

ℓ	LV_{entry}	LV_{exit}
1	$\{\mathtt{x},\mathtt{z}\}$	$\{\mathtt{x},\mathtt{z}\}$
2	$\{\mathtt{x},\mathtt{z}\}$	$\{\mathtt{x},\mathtt{z}\}$
3	$\{\mathtt{x},\mathtt{z}\}$	$\{\mathtt{y},\mathtt{z}\}$
4	$\{\mathtt{y},\mathtt{z}\}$	$\{\mathtt{y},\mathtt{z}\}$
5	$\{\mathtt{y},\mathtt{z}\}$	$\{\mathtt{x},\mathtt{z}\}$
6	$\{\mathtt{z}\}$	$\emptyset$

Liveness-Analyse

Vom Kontrollflussgraphen des gegebenen Programms können wir hinreichende Gleichungen für die *LV*-Mengen aufstellen.

- Wir interessieren uns für eine möglichst genaue Analyse, in der die *LV*-Mengen möglichst klein sind.
  Es wäre korrekt (aber nicht interessant) für jede *LV*-Menge die Menge aller Variablen zu wählen.
- Für jedes ℓ liefert die Anweisung mit Label ℓ eine Gleichung für $LV_{entry}(\ell)$.

  Die Anweisung $[\mathsf{y:=x+1}]^3$ liefert zum Beispiel

$$LV_{entry}(3) = \{\mathsf{x}\} \cup (LV_{exit}(3) \setminus \{\mathsf{y}\})$$

- Die Menge $LV_{exit}(\ell)$ sollte gleich der Vereinigung der Entry-Mengen aller Nachfolgerknoten von ℓ im Kontrollflussgraphen sein.

Gleichungssystem für das Beispielprogramm

$$[\texttt{y:=0}]^1; \texttt{while}\,[\texttt{x<10}]^2\,\texttt{do}\,([\texttt{y:=x+1}]^3; [\texttt{z:=z+y}]^4; [\texttt{x:=2*y}]^5); [\texttt{r:=z}]^6$$

$$\begin{aligned}
LV_{entry}(1) &= LV_{exit}(1)\backslash\{\texttt{y}\} \\
LV_{entry}(2) &= LV_{exit}(2) \cup \{\texttt{x}\} \\
LV_{entry}(3) &= (LV_{exit}(3)\backslash\{\texttt{y}\}) \cup \{\texttt{x}\} \\
LV_{entry}(4) &= (LV_{exit}(4)\backslash\{\texttt{z}\}) \cup \{\texttt{z},\texttt{y}\} \\
LV_{entry}(5) &= (LV_{exit}(5)\backslash\{\texttt{x}\}) \cup \{\texttt{y}\} \\
LV_{entry}(6) &= (LV_{exit}(6)\backslash\{\texttt{r}\}) \cup \{\texttt{z}\} \\
LV_{exit}(1) &= LV_{entry}(2) \\
LV_{exit}(2) &= LV_{entry}(3) \cup LV_{entry}(6) \\
LV_{exit}(3) &= LV_{entry}(4) \\
LV_{exit}(4) &= LV_{entry}(5) \\
LV_{exit}(5) &= LV_{entry}(2) \cup LV_{entry}(6)
\end{aligned}$$

Kleinste Lösung

- Wir interessieren uns für die kleinste Lösung dieser Gleichungen. (d.h. die Lösung, die alle unbekannten Mengen möglichst klein macht)
- Jede Lösung des Gleichungssystems ist korrekt in dem Sinne, dass alle live Variablen erfasst sind.
- Es gibt Lösungen, die mehr live Variablen aufführen, als tatsächlich vorhanden sind, z.B. bleiben die Gleichungen gültig, wenn man die Variable r zu allen Mengen außer denen für Label 6 hinzunimmt.
- Auch die kleinste Lösung kann überflüssige Einträge enthalten. Beispiel:

$$(\texttt{while}\ [\texttt{true}]^1\ \texttt{do}\ [\texttt{skip}]^2); [\texttt{r:=x}]^3$$

Hier gilt selbst für die kleinste Lösung $LV_{entry}(1) = \{\texttt{x}\}$, obwohl tatsächlich nie wieder von x gelesen wird.

Berechnung der kleinsten Lösung

Benutzte eine Tabelle mit einem Eintrag für jede gesuchte Menge.

- Initalisiere alle Einträge mit $\varnothing$
- Betrachte dann jeden Tabelleneintrag, berechne die rechte Seite der Gleichung für diesen Eintrag mit den aktuellen Werten in der Tabelle und aktualisiere den Tabelleneintrag mit dem Ergebnis.
- Wiederhole bis sich nichts mehr ändert.

Im Beispiel:

ℓ	$LV_{exit}(\ell)$	$LV_{entry}(\ell)$
6	$\varnothing$	$\varnothing$
5	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
2	$\varnothing$	$\varnothing$
1	$\varnothing$	$\varnothing$

$\rightsquigarrow$

ℓ	$LV_{exit}(\ell)$	$LV_{entry}(\ell)$
6	$\varnothing$	$\{z\}$
5	$\{z\}$	$\{y, z\}$
4	$\{y, z\}$	$\{y, z\}$
3	$\{y, z\}$	$\{x, z\}$
2	$\{x, z\}$	$\{x, z\}$
1	$\{x, z\}$	$\{x, z\}$

$\rightsquigarrow$

Berechnung der kleinsten Lösung

ℓ	$LV_{exit}(\ell)$	$LV_{entry}(\ell)$
6	$\varnothing$	$\{z\}$
5	$\{x, z\}$	$\{y, z\}$
4	$\{y, z\}$	$\{y, z\}$
3	$\{y, z\}$	$\{x, z\}$
2	$\{x, z\}$	$\{x, z\}$
1	$\{x, z\}$	$\{x, z\}$

$\rightsquigarrow$

ℓ	$LV_{exit}(\ell)$	$LV_{entry}(\ell)$
6	$\varnothing$	$\{z\}$
5	$\{x, z\}$	$\{y, z\}$
4	$\{y, z\}$	$\{y, z\}$
3	$\{y, z\}$	$\{x, z\}$
2	$\{x, z\}$	$\{x, z\}$
1	$\{x, z\}$	$\{x, z\}$

Ein $\rightsquigarrow$-Schritt entspricht hier einem Durchgang aller Einträge von links nach rechts und von oben nach unten.

Berechnung der kleinsten Lösung

Andere Durchlaufreihenfolgen liefern das gleiche Ergebnis, können aber mehr (oder auch weniger) Iterationen brauchen.

Bei der Liveness-Analysis sollte man möglichst rückwärts, also entgegen des Programmflusses vorgehen.

Live-Variablen Gleichungen

Allgemein können die Liveness-Datenflussgleichen für ein gegegebenes Programm nach folgendem Schema aufgestellt werden.

$$\begin{aligned} LV_{exit}(\ell) &= \bigcup_{\ell':\ell\to\ell'} LV_{entry}(\ell') \\ LV_{entry}(\ell) &= (LV_{exit}(\ell) \backslash kill_{LV}(B^\ell)) \cup gen_{LV}(B^\ell)) \end{aligned}$$

Hierbei ist B^ℓ das durch ℓ beschriftete Programmstück (“Block”), sowie:

$$\begin{aligned} kill_{LV}([x:=a]^\ell) &= \{x\} \\ kill_{LV}([\texttt{skip}]^\ell) &= \varnothing \\ kill_{LV}([b]^\ell) &= \varnothing \\ gen_{LV}([\texttt{skip}]^\ell) &= \varnothing \\ gen_{LV}([x:=a]^\ell) &= FV(a) \\ gen_{LV}([b]^\ell) &= FV(b) \end{aligned}$$

Es bezeichnet $FV(a)$ die Menge aller Variablen, die im Ausdruck a vorkommen.

Live-Variablen Gleichungen

Die Gleichungen drücken aus

- an welche Programmpunkten Variablen gelesen werden und somit direkt live sind; und
- welche Variablen in den Vorgängern einer elementaren Anweisung live sein können, wenn sie in einem Nachfolger live sind.

Für die *LV*-Mengen gilt “kleiner = genauer”; wir berechnen also die kleinste Lösung des Gleichungssystems.

Die kleinste Lösung kann auch allgemein wie im Beispiel berechnet werden: Tabelle mit $\varnothing$ initialisieren, Gleichungen iterieren bis sich nichts mehr ändert

Reaching Definitions

Definition

Eine Zuweisung $[\mathtt{x:=a}]^\ell$ *erreicht* einen bestimmten Programmpunkt p, wenn es eine Programmausführung gibt, die p über die Wertzuweisung erreicht, so dass die Variable x zwischen der Wertzuweisung und p nicht verändert wird.

Beispiel:

$$[\mathtt{y:=x}]^1; [\mathtt{z:=1}]^2; \mathtt{while}\ [\mathtt{y>1}]^3\ \mathtt{do}\ ([\mathtt{z:=z*y}]^4; [\mathtt{y:=y-1}]^5); [\mathtt{y:=0}]^6$$

- Die Zuweisung 1 erreicht 3.
- Die Zuweisung 4 erreicht 3.
- Die Zuweisung 2 erreicht 5 nicht.

Reaching Definitions: Anwendungen

- Verifikation: Wird Variable x stets initialisiert?
  Füge eine neue Zuweisung $[x:=?]^\ell$ an den Anfang des Programms an. Diese Zuweiung darf keine Programmstelle erreichen, in der x gelesen wird.
- Optimierung: Konstantenpropagation
  Wenn die einzige Zuweisung für x, die einen Programmpunkt erreicht, eine Konstanten-Zuweisung $[x:=c]^\ell$ ist, dann können wir an diesem Programmpunkt x durch c ersetzen.
- Optimierung: Hoisting
  Wenn die Anweisung $[x:=y*z]$ nur von Zuweisungen für y und z erreicht wird, die vor der `while`-Schleife stehen, dann können wir sie aus der Schleife herausziehen.

```
while b do (          x:=y*z;
  x:=y*z;             while b do (
  ...)                  ...)
```

Reaching Definitions

Für jeden Programmpunkt ℓ bestimmen wir zwei Mengen:

- $RD_{entry}(\ell)$: Zuweisungen, die den Eingang von ℓ erreichen oder möglicherweise erreichen könnten.
- $RD_{exit}(\ell)$: Zuweisungen, die den Ausgang von ℓ erreichen oder möglicherweise erreichen könnten.

Je kleiner diese Mengen gewählt werden können, desto genauer ist die Analyse.

Notation: Wir schreiben kurz ℓ für eine Zuweisung $[\mathsf{x}:=\mathsf{a}]^{\ell}$.

Reaching Definitions: Datenflussgleichungen

Allgemein können die RD-Gleichungen für ein gegegebenes Programm nach folgendem Schema aufgestellt werden.

$$
\begin{aligned}
RD_{entry}(\ell) &= \bigcup_{\ell':\ell' \to \ell} RD_{exit}(\ell') \\
RD_{exit}(\ell) &= (RD_{entry}(\ell) \setminus kill_{RD}(B^\ell)) \cup gen_{RD}(B^\ell)
\end{aligned}
$$

$$
\begin{aligned}
kill_{RD}([x := a]^\ell) &= \{\ell' \mid \ell' \text{ ist eine Zuweisung an } x\} \\
kill_{RD}([\texttt{skip}]^\ell) &= \emptyset \\
kill_{RD}([b]^\ell) &= \emptyset \\
gen_{RD}([\texttt{skip}]^\ell) &= \emptyset \\
gen_{RD}([x := a]^\ell) &= \{\ell\} \\
gen_{RD}([b]^\ell) &= \emptyset
\end{aligned}
$$

Annahme: Keine Gleichung erreicht den Anfangspunkt des Programms.

Reaching Definitions: Datenflussgleichungen

Die Gleichungen drücken aus

- welche Definitionen im Programm gemacht werden; und
- wie Definitionen von den Vorgängern zu den Nachfolgern einer elementaren Anweisung propagiert werden.

Auch für die *RD*-Mengen gilt “kleiner = genauer”.

Die kleinste Lösung des Gleichungssystems kann mit dem gleichen Verfahren wie für Liveness bestimmt werden.

Information wird hier vorwärts und nicht rückwärts propagiert. Die Berechnung geht am schnellsten, wenn man die Gleichungen in Reihenfolge des Programmflusses durchgeht.

Reaching Definitions: Beispiel

$$[\texttt{y:=x}]^1; [\texttt{z:=1}]^2; \texttt{while } [\texttt{y>1}]^3 \texttt{ do } ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6$$

$$RD_{entry}(1) = \varnothing \qquad RD_{exit}(1) = (RD_{entry}(1) \setminus \{1,5,6\}) \cup \{1\}$$

$$RD_{entry}(2) = RD_{exit}(1) \qquad RD_{exit}(2) = (RD_{entry}(2) \setminus \{2,4\}) \cup \{2\}$$

$$RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5) \qquad RD_{exit}(3) = RD_{entry}(3)$$

$$RD_{entry}(4) = RD_{exit}(3) \qquad RD_{exit}(4) = (RD_{entry}(4) \setminus \{2,4\}) \cup \{4\}$$

$$RD_{entry}(5) = RD_{exit}(4) \qquad RD_{exit}(5) = (RD_{entry}(5) \setminus \{1,5,6\}) \cup \{5\}$$

$$RD_{entry}(6) = RD_{exit}(3) \qquad RD_{exit}(6) = (RD_{entry}(6) \setminus \{1,5,6\}) \cup \{6\}$$

Berechnung der kleinsten Lösung

$$[\texttt{y:=x}]^1; [\texttt{z:=1}]^2; \texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6$$

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\varnothing$	$\varnothing$
2	$\varnothing$	$\varnothing$
3	$\varnothing$	$\varnothing$
4	$\varnothing$	$\varnothing$
5	$\varnothing$	$\varnothing$
6	$\varnothing$	$\varnothing$

$\rightsquigarrow$

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\varnothing$	$\{1\}$
2	$\{1\}$	$\{1,2\}$
3	$\{1,2\}$	$\{1,2\}$
4	$\{1,2\}$	$\{1,4\}$
5	$\{1,4\}$	$\{4,5\}$
6	$\{1,2,4,5\}$	$\{2,4,6\}$

$\rightsquigarrow$

Berechnung der kleinsten Lösung, Forts.

$$[\texttt{y:=x}]^1; [\texttt{z:=1}]^2; \texttt{while}\ [\texttt{y>1}]^3\ \texttt{do}\ ([\texttt{z:=z*y}]^4; [\texttt{y:=y-1}]^5); [\texttt{y:=0}]^6$$

	ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
	1	$\emptyset$	$\{1\}$
	2	$\{1\}$	$\{1,2\}$
$\rightsquigarrow$	3	$\{1,2,4,5\}$	$\{1,2,4,5\}$
	4	$\{1,2,4,5\}$	$\{1,4,5\}$
	5	$\{1,4,5\}$	$\{4,5\}$
	6	$\{1,2,4,5\}$	$\{2,4,6\}$

$\rightsquigarrow$

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\emptyset$	$\{1\}$
2	$\{1\}$	$\{1,2\}$
3	$\{1,2,4,5\}$	$\{1,2,4,5\}$
4	$\{1,2,4,5\}$	$\{1,4,5\}$
5	$\{1,4,5\}$	$\{4,5\}$
6	$\{1,2,4,5\}$	$\{2,4,6\}$

Available Expressions

Wir interessieren uns jetzt dafür, welche arithmetischen Ausdrücke an einem bestimmten Programmpunkt *verfügbar* (available) sind, also nicht neu berechnet werden müssen, wenn man bereit ist, berechnete Ausdrücke in einer Tabelle abzuspeichern.

$$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$$

Beim Test $[\texttt{y>a+b}]^3$ ist der Ausdruck a+b verfügbar, müsste also dort nicht neu berechnet werden. Man könnte das Programm also durch folgende effizientere Version ersetzen:

$$[\texttt{mem:=a+b}]^{1a}; [\texttt{x:=mem}]^1; [\texttt{y:=a*b}]^2;$$
$$\texttt{while } [\texttt{y>mem}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{mem:=a+b}]^{5a}; [\texttt{x:=mem}]^5)$$

Available Expressions

Gegeben sei ein Programm, in dem wir die Verfügbarkeit von Ausdrücken analysieren wollen.

Schreibe *AExp* für die Menge aller arithmethischer Ausdrücke, die im gegebenen Program vorkommen (evtl. als Teilausdruck eines größeren arithmetischen Ausdrucks), abgesehen Konstanten und Variablen.

Für jeden Programmpunkt ℓ berechnen zwei Mengen:

- $AE_{entry}(\ell) \subseteq AExp$: Eine Menge arithmetischer Ausdrücke, die sicher am Eingang von ℓ verfügbar sind.
- $AE_{exit}(\ell) \subseteq AExp$: Eine Menge arithmetischer Ausdrücke, die sicher am Ausgang von ℓ verfügbar sind.

Dieses Mal entsprechen größere Mengen genauerer Information.

Datenflussgleichungen

$$
\begin{array}{lcl}
AE_{entry}(\ell) & = & \begin{cases} \varnothing & \ell \text{ ist Einstiegslabel} \\ \bigcap_{\ell':\ell' \to \ell} AE_{exit}(\ell') & \text{sonst} \end{cases} \\
AE_{exit}(\ell) & = & (AE_{entry}(\ell) \setminus kill_{AE}(B^\ell)) \cup gen_{AE}(B^\ell)
\end{array}
$$

wobei

$$
\begin{array}{lcl}
kill_{AE}([x:=a]^\ell) & = & \{a' \in AExp \mid x \in FV(a')\} \\
kill_{AE}([\texttt{skip}]^\ell) & = & \varnothing \\
kill_{AE}([b]^\ell) & = & \varnothing \\
gen_{AE}([x:=a]^\ell) & = & \{a' \in AExp \mid a' \text{ Teilausdruck von } a \text{ und } x \notin FV(a')\} \\
gen_{AE}([\texttt{skip}]^\ell) & = & \varnothing \\
gen_{AE}([b]^\ell) & = & \{a' \in AExp \mid a' \text{ Teilausdruck von } b\}
\end{array}
$$

(Ein Teilausdruck entspricht einem Teilbaum des Syntaxbaums.)

Datenflussgleichungen im Beispiel

$[\mathsf{x:=a+b}]^1; [\mathsf{y:=a*b}]^2; \mathtt{while}\ [\mathsf{y>a+b}]^3\ \mathsf{do}\ ([\mathsf{a:=a+1}]^4; [\mathsf{x:=a+b}]^5)$

ℓ	$kill_{AE}$	gen_{AE}
1	$\varnothing$	{a+b}
2	$\varnothing$	{a*b}
3	$\varnothing$	{a+b}
4	{a+b, a*b, a+1}	$\varnothing$
5	$\varnothing$	{a+b}

$$
\begin{array}{lcl}
AE_{entry}(1) & = & \varnothing \\
AE_{entry}(2) & = & AE_{exit}(1) \\
AE_{entry}(3) & = & AE_{exit}(2) \cap AE_{exit}(5) \\
AE_{entry}(4) & = & AE_{exit}(3) \\
AE_{entry}(5) & = & AE_{exit}(4)
\end{array}
$$

$$
\begin{array}{lcl}
AE_{exit}(1) & = & AE_{entry}(1) \cup \{\mathsf{a+b}\} \\
AE_{exit}(2) & = & AE_{entry}(2) \cup \{\mathsf{a*b}\} \\
AE_{exit}(3) & = & AE_{entry}(3) \cup \{\mathsf{a+b}\} \\
AE_{exit}(4) & = & AE_{entry}(4) \backslash \{\mathsf{a+b}, \mathsf{a*b}, \mathsf{a+1}\} \\
AE_{exit}(5) & = & AE_{entry}(5) \cup \{\mathsf{a+b}\}
\end{array}
$$

$$AExp = \{\mathsf{a+b}, \mathsf{a*b}, \mathsf{a+1}\}$$

Größte Lösung

Auch hier ist jede Lösung der Datenflussgleichung gerechtfertigt, allerdings sind wir an möglichst großen *AE*-Mengen interessiert. Wir berechnen daher die *größte* Lösung. Man erhält sie wie die kleinste Lösung, initialisiert allerdings die dynamischen Mengen jeweils mit der Menge *aller Ausdrücke* (die im Programm vorkommen) anstatt mit $\varnothing$.

Man erhält:

ℓ	$AE_{entry}(\ell)$	$AE_{exit}(\ell)$
1	*AExp*	*AExp*
2	*AExp*	*AExp*
3	*AExp*	*AExp*
4	*AExp*	*AExp*
5	*AExp*	*AExp*

$\rightsquigarrow \cdots \rightsquigarrow$

ℓ	$AE_{entry}(\ell)$	$AE_{exit}(\ell)$
1	$\varnothing$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\varnothing$
5	$\varnothing$	{a+b}

Größte vs. kleinste Lösung

Im Beispiel stimmen größte und kleinste Lösung zufällig überein. Das liegt daran, dass die Schleife alle Ausdrücke “killt”.

Anders bei diesem Beispiel:

$$[\mathsf{x:=a+b}]^1; \mathtt{while}\ [\mathsf{x>y}]^2\ \mathsf{do}\ [\mathsf{x:=x-1}]^3$$

Hier gelten die Gleichungen

$$\begin{array}{|l} AE_{entry}(1) = \emptyset \\ AE_{entry}(2) = AE_{exit}(1) \cap AE_{exit}(3) \\ AE_{entry}(3) = AE_{exit}(2) \end{array}$$

$$\begin{array}{|l} AE_{exit}(1) = (AE_{entry}(1) \setminus \{\mathsf{x-1}\}) \cup \{\mathsf{a+b}\} \\ AE_{exit}(2) = AE_{entry}(2) \\ AE_{exit}(3) = AE_{entry}(3) \end{array}$$

Die größte Lösung liefert $AE_{exit}(3) = \{\mathsf{a+b}\}$ während bei der kleinsten Lösung $AE_{exit}(3) = \emptyset$ gilt.