Programmanalyse und Compileroptimierung
WS 2004/2005

Prof. Dr. Indermark
Mitgeschrieben von Timo Boettcher

Dieses Skript ist meine Mitschrift der Vorlesung Programmanalyse und Compileroptimierung im WS 2004/2005 an der RWTH Aachen (Dozent: Prof. Dr. K. Indermark). Es handelt sich nicht um eine offizielle Veröffentlichung des "Lehrstuhls Informatik II".
Ich übernehme keine Gewähr für die Fehlerfreiheit und Vollständigkeit des Skripts.
Korrekturen und Ergänzungen bitte ich an timo_boettcher@post.rwth-aachen.de zu mailen.
Timo Boettcher, 2. Februar 2005

Inhaltsverzeichnis

Kapitel 1

Einleitung

[13.10.2004]

Ziel der Programmanalyse: Berechnung von Laufzeiteigenschaften zur Übersetzungszeit, abhängig von der Programmsemantik, Beispiel: Terminierung, i.a. nicht entscheidbare Eigenschaften.
→ Programmanalyse berechnet Approximationen dieser Eigenschaften
Mittel: Abstrakte Interpretation Anwendungen der Programmanalyse:

- Compileroptimierung durch Programmtransformation
- Konstruktion von Software-Tools für Verfikation, Testen, Debugging

hier: Compileroptimierung
Verbesserung von Programmen mit Hilfe von Programmeigenschaften
Verschiedene Kostenmasse:

- Laufzeit
- Speicherbedarf der Laufzeitdaten (Grösse von Stack und Heap)
- Speicherbedarf des Codes (Programmgrösse)
  Kompakter Code wichtig bei Palmtops, Mobiltelefonen, Software für eingebettete Systeme

Programmiersprachen:

- imperative:
  - OO (Java: Klassenoptimierung)
  - FP (Striktheitsanalyse)
  - LP

[Folie: Struktur eines Compilers]

Möglichkeiten für Programmanalyse und Compileroptimierung:

1. Attributierte Abstrakter Syntaxbaum (Quellprogramm)
   erfordert Programmanalyse für strukturierte Programme

2. Zwischencode
   elementare Ausdrücke, 3-Adresscode
   keine Kontrollstrukturen bis auf Sprungbefehle
   keine Datenstrukturen, Adressberechnung

3. Maschinencode
   Registerzuteilung, Codeselection, Instruktionsanordnung

Inhalt PACO für Zwischencode

1. Straight-line Code (ohne: Verzweigung, Iteration, Rekursion, Datenstrukturen) Wichtige Teilklasse. Schleifenrümpfe

   90/10 Regel: Ein Programm benutzt 90% der Ausführungszeit für 10% seines Codes

   → lokale Analyse

2. Iterative Programme

   (a) Basisblock-Darstellung und Flussgraphen
       → globale Analyse, intra-prozedurale Analyse

       Technik: Datenflussanalyse in Abstrakter Interpretation

   (b) Kontrollflussanalyse und Schleifenoptimierung

3. Rekursive Programme

   → interprozedurale Analyse

4. Programme mit Datenstrukturen

   statische und dynamische Datenstrukturen, Shape Analyse

Optimierungen: dead code elimination, common subexpression elimination, constant propagation/constant folding, reduction in strength, code motion
Analysen: live variables, available expression, pointer, shape, alias

Kapitel 2

Analyse und Optimierung von Straight-Line Code

Programme ohne Verzweigung, Iteration, rekursion, Datenstrukturen, Ausdrücke entschachtelt
Bedeutung: wiederholte Ausführung in Schleifen
Ziel: Codeoptimierung (Laufzeit)
Themen:

- Live Variable (LV) Analyse - Dead Code (DC) Elimination
- Available Expression (AE) Analyse - Common Subexpression (CS) Elimination
- Reaching Definition (RD) Analyse - Konstantenpropagation/-faltung

Definition 2.0.1: Syntax von SLC-Programmen
$V = \{x, y, z, \ldots\}$ Variablen
$C = \{a, b, c, \ldots\}$ Konstantensymbole
$\Sigma = \bigcup_{r=1}^{\infty} \Sigma^{(r)}$ mit $|\Sigma| < \infty$ Operationssymbole
$\alpha \in \underline{Anw}$ (α ist eine Anweisung): $\curvearrowleft\curvearrowright$
$\alpha = x \leftarrow e$ und
$e \in \{f(u_1, \ldots, u_r) | f \in \Sigma^{(r)}, u_i \in V \cup C\}$
$V_\alpha := \{x\} \cup V_e$ (Variablen von α bzw. von e)
x wird in α definiert, $v \in V_e$ wird in α benutzt.
$\beta \in Blo$ (β ist ein Block) : $\curvearrowleft\curvearrowright$
$\beta = \alpha_1; \alpha_2; \ldots \alpha_n, \qquad \alpha_i \in \underline{Anw},\ n \geq 0$
$V_\beta := \bigcup_{i=1}^{n} V_{\alpha_i}$ (Variablen von β)

Folgerung 2.0.1:
$\pi \in SLC$ (π ist ein SLC-Programm):
$\curvearrowleft\curvearrowright\ \pi = (IV, \beta, OV)$, $\beta \in \underline{Blo}$

$IV \subseteq V$, endlich (Eingabevariablen, var β definiert)
$OV \subseteq V$, endlich, nicht leer (Ausgabevariablen, nach β benutzt)

[Folie: SLC-Beispielprogramm]

$V_\pi := \bigcup_{i=1}^{n} V_{\alpha_i} \cup IV \cup OV$

[15.10.2004]

Definition 2.0.2: Variablenbedingung
π erfüllt die Variablenbedingung
(π vollständig wohldefiniert) $:\curvearrowleft\curvearrowright$ Jede Programmvariable $v \in V_\pi$ wird vor ihrer Benutzung definiert

Generelle Vorraussetzung: SLC-Programme sollen die Variablenbedingung erfüllen
SLC-Programme (Semantik)
Sei $\pi = (IV, \beta, OV) \in SLC$.
Sei $\mathfrak{a} = \langle A; \varphi \rangle$ eine (Σ, C)- Algebra, d.h.,
A Menge, $f \in \Sigma^{(r)} \curvearrowright \varphi(f) = f_{\mathfrak{a}} : A^r \to A$
$c \in C \curvearrowright \varphi(c) = c_{\mathfrak{a}} \in A$
Zustandsraum $Z := \{\sigma | \sigma : V_\pi \to a\}$
$\alpha \in Anw \mapsto [\![\alpha]\!]_{\mathfrak{a}} : Z \to Z$ Zustandstransformation
$\alpha = x \leftarrow e$
$[\![x \leftarrow e]\!]_{\mathfrak{a}}(\sigma) = \sigma'$ mit $\sigma'(x) = [\![e]\!]_{\mathfrak{a}}(\sigma) \in A$
(Wert von e im Zustand σ, “Einsetzen und Ausrechnen“) und $\sigma'(y) = \sigma(y)$ für $y \neq x$
$\beta = \alpha_1; \ldots; \alpha_n \mapsto [\![\beta]\!]_{\mathfrak{a}} := [\![\alpha_n]\!] \circ [\![\alpha_{n-1}]\!] \circ \ldots \circ [\![\alpha_1]\!]$
$\beta = \Sigma \mapsto [\beta]_{\mathfrak{a}} := id_Z$
IV und OV bestimmen entsprechende Teile des Zustandsraumes Z.
$Z_{in} = \{\sigma_{in} | \sigma_{in} : IV \to A\}$ Eingabezustände
$Z_{out} = \{\sigma_{out} | \sigma_{out} : OV \to A\}$ Ausgabezustände
damit ergibt sich die Programmsemantik
$[\![\pi]\!]_{\mathfrak{a}} : Z_{in} \to Z_{out}$
$[\![\pi]\!]_{\mathfrak{a}}(\sigma_{in}) = \sigma_{out}$
$\sigma_{in} \mapsto \sigma_0(x) := \sigma_{in}(x)$ für $x \in IV$
$\sigma_0(y) = a \in A$ (beliebig) für $y \in IV$
$\sigma_{out}(z) = [\![\beta]\!]_{\mathfrak{a}}(\sigma_0)(z)$ für $z \in OV$
Beachte: π erfüllt die Variablenbedingung $\curvearrowright$ $[\![\pi_{\mathfrak{a}}]\!]$ unabhängig von Anfangswerten für $y \in V_\pi \backslash IV$

Definition 2.0.3: Programmäquivalenz
$\pi_1, \pi_2 \in SLC$. $\mathfrak{a}$ $\quad(\Sigma, C)$-Algebra

- π_1 und π_2 $\mathfrak{a}$-äquivalent ($\pi_1 \sim_{\mathfrak{a}} \pi_2$) $:\curvearrowleft\curvearrowright$ $[\![\pi_1]\!] = [\![\pi_2]\!]_{\mathfrak{a}}$
- π_1 und π_2 stark-äquivalent ($\pi_1 \approx \pi_2$) $:\curvearrowleft\curvearrowright$ $[\![\pi_1]\!] = [\![\pi_2]\!]$ für jede (Σ, C)-Algebra $\mathfrak{a}$

Optimierung von SLC-Programmen

Sei $c : SLC \to \mathbb{N}$ eine Kostenfunktion, z.B. Zahl der Anweisungen, Zahl der Operationen, Summe der Operations-Gewichte
Für $\pi_1, \pi_2 \in SLC$ mit $\pi_1 \sim_{\mathfrak{a}} \pi_2$ (bzw $\pi_1 \approx \pi_2$) ist $\pi_1 \leq_c \pi_2$ (c-besser):$\curvearrowleft\curvearrowright$ $c(\pi_1) \leq c(\pi_2)$

Definition 2.0.4: c-optimal

π_1 c-optimal (bzgl. $\mathfrak{a}$) :$\curvearrowleft\curvearrowright$ $\pi_1 \leq_c \pi$ für alle $\pi \in SLC$ mit $\pi_1 \approx \pi$ (bzw. $\pi_1 \sim_{\mathfrak{a}} \pi$)

Ziel: Programmtransformationen zur SLC-Optimierung [Folie: Beispiel: Common Subexpression Elimination]

2.1 AE-Analyse und CS-Elimination

Beispiel 2.1.1: Folie 2.2

$\pi \approx \pi'$ und $c(\pi') < c(\pi)$ bzgl. Op-Anzahl aber $c(\pi') > c(\pi)$ bzgl. Codelänge

AE-Analyse Bestimmung der für jede Anweisung verfügbaren Ausdrücke (AE = available expression)
Sei $\pi = (IV, \beta, OV) \in SLC$ mit $\beta = \alpha_1; \ldots; \alpha_n$
$\underline{OpExp}_\beta := \{e | x \leftarrow e = \alpha_i, e = f(u_1, \ldots, u_r)\} = \{e_1, \ldots, e_t\}$ Operationsausdrücke von β
Verfügbarkeit durch Bit-vektoren beschreiben:
$\mathbb{B}^t := \{(b_1, \ldots, b_t) | b_i \in \{0, 1\}\}$
Bedeutung: $b_i = 0$ heisst "e_i ist verfügbar"
$b_i = 1$ heisst "e_i ist nicht verfügbar"
$AE_i \in \mathbb{B}^t$: die für Ausführung von α_i verfügbaren Ausdrücke ($i = 1, \ldots, n$)
$AE_1 = (1, \ldots, 1)$
$\alpha \in Anw \mapsto t_\alpha : \mathbb{B}^t \to \mathbb{B}^t$ (information transformer)
t_α beschreibt die Veränderung der Verfügbarkeit von Ausdrücken durch Ausführung von α
$t_{x \leftarrow e} := kill_x \circ gen_e; gen_e, kill_x : \mathbb{B}^t \to \mathbb{B}^t$
$gen_e(b_1, \ldots, b_t) = (b'_1, \ldots, b'_t)$

$$b'_i = \begin{cases} 0, & \text{falls } e = e_i \\ b_i, & \text{sonst} \end{cases}$$

$kill_x(b_1, \ldots, b_t) = (b'_1, \ldots, b't)$

$$b'_i = \begin{cases} 1, & \text{falls } x \in V_{e_i} \\ b_i, & \text{sonst} \end{cases}$$

$AE_{i+1} = t_{\alpha_i}(AE_i)$ für $i = 1, \ldots, n-1$
Ergebnis: Analyseinformationen $AE_1, \ldots, AE_n$ [20.10.2004]
Beobachtung: $A_i = x \leftarrow x$
keine Veränderung von x, aber Verfügbarkeit von Ausdrücken wird i. a. reduziert.

Beispiel 2.1.2: Folie 2.2 Common Subexpression Elimination

$\underline{OpExp}_\beta = \{oe_1 = x + y, oe_2 = z * y, oe_3 = v - z, oe_4 = w + v\}$
$AE_1 = (1, 1, 1, 1)$
$AE_2 = (0, 1, 1, 1)$
$AE_3 = (0, 1, 1, 1)$
$AE_4 = (1, 1, 1, 1)$
$AE_5 = (0, 1, 1, 1)$
$AE_6 = (0, 1, 0, 1)$

<u>CS-Elimination</u>
<u>Ziel:</u> Vermeidung der wiederholten Berechnung eines Ausdrucks mit denselber Variablenwerten, weniger durchgeführte Rechenoperationen.
<u>Idee:</u> Ausdruckswerte auf temporären Variablen zwischenspeichern, statt wiederholter Berechnung temporäre Variable verwenden.
$T_{CS} : SLC \to SLC$
$T_{CS}(IV, \beta, OV) := (IV, T_{CS}(\beta), OV)$
$T_{CS}(\alpha_1; \ldots; \alpha_n) := T_{CS}(\alpha_1); \ldots; T_{CS}(\alpha_n)$
3 Fälle für $T_{CS}(\alpha_1)$

1. $\alpha_i = x \leftarrow e, e \notin \underline{OpExp}_\beta \curvearrowright T_{CS}(\alpha_i) := \alpha_i$

2. $\alpha_i = x \leftarrow oe_j, AE_i^{(j)} = 1 \curvearrowright T_{CS}(\alpha_i) := tv_j \leftarrow oe_j; x \leftarrow tv_j$

3. $\alpha_i = x \leftarrow oe_j, AE_i^{(j)} = 0 \curvearrowright T_{CS}(\alpha_i) := x \leftarrow tv_j$

mit temporären Variablen $tv_1, \ldots, tv_t$ für die Op-Ausdrücke $oe_1, \ldots, oe_t$

Beispiel 2.1.3:

Fall (1) tritt nicht auf. Die "relevanten" Verfügbarkeitsbits sind:
$AE_1^{(1)} = 1$
$AE_2^{(2)} = 1$
$AE_3^{(1)} = 0$
$AE_4^{(1)} = 1$
$AE_5^{(3)} = 1$
$AE_6^{(4)} = 1$

$$
\begin{aligned}
T_{CS}(\beta) = \quad tv_1 &\leftarrow x + y; \\
z &\leftarrow tv_1; \\
tv_2 &\leftarrow z * y; \\
z &\leftarrow tv_2; \\
\\
x &\leftarrow tv_1; \\
tv_1 &\leftarrow x + y; \\
v &\leftarrow tv_1; \\
tv_3 &\leftarrow v - z; \\
w &\leftarrow tv_3; \\
tv_4 &\leftarrow w + v; \\
w &\leftarrow tv_4;
\end{aligned}
$$

Korrektheit $\pi \approx T_{TC}(\pi)$
unabhängig von Interpretation
Klar für die Fälle 1 und 2
Fall 3: Der verfügbare Wert von oe_j liegt als Wert von tv_j vor.
Optimierung: $Op - Anzahl(T_{CS}(\pi)) \leq Op - Anzahl(\pi)$
aber: Codelänge($T_{CS}(\pi)$) > Codelänge(π)

2.2 LV-Analyse und DC-Elimination

LV-Analyse $\pi = (IV, \beta, OV) \in SLC$
Var. Bedingung: $v \in V_\pi$ wird vor ihrer Benutzung definiert.
Möglich: $v \in V_\pi$ wird nach ihrer Definition nicht benutzt.

Definition 2.2.1: lebendig
$v \in V_\pi$ heisst lebendig in i ($v \in LV_i$) für $i = 1, \ldots, n$ falls v in $\alpha_{i+1}, \alpha_{i+2}, \ldots, \alpha_n$ oder OV benutzt wird, ohne vor dieser Benutzung neu definiert zu werden.

Bestimmung der LV_i durch Rückwärtsanalyse:
$LV_n = OV$
$\alpha \in \underline{Anw} \mapsto t_\alpha : P(V_\pi) \to P(V_\pi)$ “Rückwärtstransformation“
$\alpha = x \leftarrow e; t_\alpha := gen_e \circ kill_x$
$kill_x(M) := M/\{x\}$
$gen_e(M) := M \cup V_e$
$LV_{i-1} := t_{\alpha_i}(LV_i) \qquad i = 1, \ldots, n$

Beispiel 2.2.2:
$LV_4 = \{u, v\}$
$LV_3 = \{u, x, y\}$
$LV_2 = \{x, y\}$

$LV_1 = \{x, y, u, z\}$
$LV_0 = \{x, y, z\}$

<u>DC-Elimination</u> $T_{DC} : SLC \to SLC$
Entferne α_i aus $\beta = \alpha_1; \dots; \alpha_n$, falls $\alpha_i = x \leftarrow e$ und $x \notin LV_i (i = 1, \dots, n)$
Bsp: Entferne α_2, nicht jedoch α_1:
Möglichkeit: LV-Analyse wiederholen
<u>Verbesserte Analyse:</u> NV (needed Variable)-Analyse
Modifikation:
[22.10.2004] $\alpha = x \leftarrow e \qquad \tilde{t}_\alpha(M) :=$ <u>if</u> $x \in M$ <u>then</u> $M/x \cup V_e$ <u>else</u> M
$NV_n = OV$
$NV_{i-1} := \tilde{t}_{\alpha_i}(NV_i)$

Beispiel 2.2.3:
$NV_4 = \{u, v\}$
$NV_3 = \{u, x, y\}$
$NV_2 = \{x, y\}$
$NV_1 = \{x, y\}$
$NV_0 = \{x, y\}$

<u>Korrektheit:</u> $\pi \approx T_{DC}(\pi) \approx \tilde{T}_{DC}(\pi)$
<u>Optimierung:</u> $c(\tilde{T}_{DC}(\pi)) \le c(T_{DC}(\pi)) \le c(\pi)$ c: Programmlänge, Op-Anzahl

2.3 RD-Analyse und Konstantenfaltung

<u>Ziel:</u> partielle Auswertung von $\pi \leftarrow$ SLC-Programm unter Kenntnis von Konstanten (Einfluss der Semantik);
genauer: ersetze konstante Variablen und konstante Ausdrücke auf rechten Seiten von Wertzuweisungen durch ihre Werte (Propagation [Variablen] und Faltung [Ausdrücke])
<u>RD-Analyse:</u> $\pi = (IV, \beta, OV) \in SLC$ mit $\beta = \alpha_1; \dots; \alpha_n$
Interpretation $\mathfrak{a} = \langle A; \varphi \rangle$
Bestimme für $i = 1, \dots, n$ die dort gültigen, eingabeunabhängigen (konstanten) Variablenwerte.
$V_\pi = \{v_1, \dots, v_k\}$
$D := (A \cup \{\Box\})^K \; d = (d_1, \dots, d_k)$
$d_j \in A$ bedeutet: v_j hat den Wert d_j
$d_j \in \Box$ bedeutet: Wert von v_j unbekannt oder eingabeabhängig für $j = 1, ..., k$
Erweiterung der Ausdruckssemantik von e:
$[\![e]\!]_\mathfrak{a}(d) \in A \cup \{\Box\}$
$[\![e]\!]_\mathfrak{a}(d) = \Box :\curvearrowleft\curvearrowright \exists v_j \in V_e$ mit $d_j = \Box$
(Strikte Semantik: $0 * \Box = \Box$)
$t_{x \leftarrow e}(d_1, \dots, d_k) = d`_1, \dots, d`_k)$

$$d_j = \begin{cases} [\![e]\!]_\mathfrak{a}(d_1, \dots, d_k), & \text{falls } x = v_j \\ d_j, & \text{sonst} \end{cases}$$

$RD_1 = (\square, \ldots, \square) \in D$
$RD_{i+1} = t_{\alpha_i}(RD_i)$
$RD_i \in D = (A \cup \{\square\})^k$

Beispiel 2.3.1:
$V_\pi = \{\underbrace{x}_1, \underbrace{y}_2, \underbrace{z}_3\}$
$RD_1 = (\square, \square, \square)$
$RD_2 = (\square, 10, \square)$
$RD_3 = (\square, 10, 2)$
$RD_4 = (\square, 10, 20)$
$RD_5 = (\square, \square, 20)$
$RD_6 = (\square, \square, \square)$

<u>Konstantenfaltung</u>
$T_{CF} : SLC \to SLC$
$T_{CF}(IV, \beta, OV) := (IV, T_{CF}(\beta), OV)$
$T_{CF}(\alpha_1; \ldots; \alpha_n) := T_{CF}(\alpha_1); \ldots; T_{CF}(\alpha_n)$
$T_{CF}(\alpha_i) := x_i \leftarrow T_{CF}(e_i, RD_i)$
$\alpha_i = x_i \leftarrow e_i$
$T_{CF}(a, d) := a$
$T_{CF}(v_j, d) :=$ <u>if</u> $d_j \in A$ <u>then</u> d_j <u>else</u> v_j
$T_{CF}(f(u_1, \ldots, u_r), d) :=$ <u>if</u> $\exists j \in \{1, \ldots, k\} T_{CF}(u_j, d) \in V_\pi$ <u>then</u> $f(T_{CF}(u_1, d), \ldots, T_{CF}(u_r, d))$
<u>else</u> $f_{\mathfrak{a}}(T_{CF}(u_1, d), \ldots, T_{CF}(u_r, d))$

Beispiel 2.3.2:
$T_{CF}(\alpha_3) = z \leftarrow T_{CF}(y * z, (\square, 10, 2)) = z \leftarrow 20$
$T_{CF}(\alpha_5) = z \leftarrow T_{CF}(y * z, (\square, \square, z)) = z \leftarrow T_{CF}(y, (\square, \square, 20)) * T_{CF}(z, (\square, \square, 20)) =$
$Z \leftarrow y * 20)$

<u>Korrektheit von T_{CF}</u>
Offensichtlich gilt: $\pi \sim_{\mathfrak{a}} T_{CF}(\pi)$
<u>Optimierung:</u> Programmlänge unverändert, Op-Anzahl kann sinken
Besonderer Seiteneffekt: Entstehung von Dead Code
<u>Bsp:</u> $\alpha_2, \alpha_1, \alpha_3$

2.4 DAG-Darstellung von SLC-Programmen

DAG (directed acyclic graph) nützliche Datenstruktur zur Implementierung von Transformationen auf SLC-Programmen
Idee: Termdarstellung der Werte einer Berechnung

Definition 2.4.1: gerichteter (Σ, V, A)-Graph
$D = \langle K, lab, suc\rangle$ heisst gerichteter (Σ, V, A)-Graph, wenn

- k nicht-leere, endliche Menge von Knoten

- lab: $K \to (\Sigma \cup V \cup A)$ eine Markierungsfunktion
- suc: $K \times \mathbb{N} \to K$ eine Nachfolgerfunktion mit der Eigenschaft suc(k, i) ist def $\curvearrowleft\curvearrowright$ $\underline{lab}(k) = f \in \Sigma^{(r)}$ und $1 \leq i \leq r$

[25.10.2004]

Definition 2.4.2:
Ein gerichteter azyklischer (Σ, V, A)-Graph heisst (Σ, V, A)-DAG.

Beachte: Jeder Knoten k eines solchen Graphen repräsentiert einen Term $t_k \in T_\Sigma(V \cup A)$
Der DAG eines SLC-Programms
Sei $\pi = (IV, \beta, OV) \in SLC$ mit $\beta = \alpha_1; \ldots; \alpha_n$ Der DAG von π wird durch "Symbolische Programmausführung mit Konstantenfaltung" bestimmt.
Wir definieren den DAG zunächst ohne Bezug auf OV:
$D(IV, \beta) = \langle K; \underline{lab}; \underline{suc} \rangle$
zusammen mit einer Bewertungsfunktion
$\underline{val} : IV \cup V_\beta \to K$
Bei der $\underline{val}(x) = k$, falls k den Wert von x nach Durchführung von β darstellt
Idee:

- verschiedene Knoten für verschiedene Werte (sharing!)
  $\to$ CS-Elimination
- ersetzen konstanter Ausdrücke
  $\to$ CF-Transformation

Anschliessende Betrachtung von OV

- OV bestimmt Teilgraphen $\to D(\pi)$
  $\to$ DC-Elimination

  Konstruktion von $D(IV, \beta)$ und val durch Induktion über die Codelänge n.

  1. Induktionsanfang: n = 0
     $D(IV, \mathcal{E}) := \langle K; \underline{lab}; \underline{suc} \rangle$ mit
     $K := IV \underline{lab}(x) := x$ für alle $x \in IV$
     $\underline{Def}(\underline{suc}) := \emptyset$
     $\underline{val}(x) := x$
  2. Induktion: $n \to n + 1$
     (IV, β_{n+1}) mit $\beta_{n+1} = \alpha_1; \ldots \alpha_n; \alpha_{n+1}$
     Nach Induktionsvorraussetzung ist für $\beta_n := \alpha_1; \ldots; \alpha_n$ bekannt
     $D(IV, \beta_n) = \langle K; \underline{lab}; \underline{suc} \rangle$ und $\underline{val}(x) \in K$ für alle $x \in IV \cup V_{\beta_n}$
     Konstruktion von
     $D(IV, \beta_{n+1}) = \langle K'; \underline{lab'}; \underline{suc'} \rangle$ und $\underline{val'} : IV \cup V_{\beta_{n+1}} \to K'$ ist bestimmt durch $\alpha_{n+1} = x \leftarrow e$

(a) $e = y$
Nach Induktionsvorraussetzung (und Variablenbedingung) $\underline{val}(y) \in K$
$D(IV, \beta_{n+1}) := D(IV, \beta_n)$
$\underline{val}`(x) := \underline{val}(y)$ ("sharing")
$\underline{val}`(x`) := \underline{val}(x`)$ für $x` \neq x$

(b) $e = a \in A$
i. $a \in K$ $\quad D(IV, \beta_{n+1}) := D(IV, \beta_n)$ und $\underline{val}`(x) = a$ $\quad \underline{val}`(x`) = \underline{val}`(x`)$ für $x` \neq x$
ii. $a \notin K$ $\quad K` := K \cup \{a\}$ $\quad \underline{lab}`(a) = a$ und $\underline{suc}`(a, i)$ nicht definiert für alle i $\quad \underline{suc}`(k, i) = \underline{suc}(k, i)$ sonst
$\underline{val}`(x) := a \in K`$
$\underline{val}`(x') = \underline{val}`(x`)$ $\quad x` \neq x$

(c) $e = f(u_1, \ldots, u_f)$ mit $u_j \in V \cup A$
a) Für alle $j = 1, \ldots, r$ gilt:
$u_j \in A$ (falls $u_j \notin K$, neuen Knoten wie in 2b) oder $u_j \in V$ mit $\underline{val}(u_j) \in A$
Dann folgt: $f_{\mathfrak{a}}(u`_1, \ldots, u`_r) =: b \in A$

$$\text{mit } u`_j = \begin{cases} u_j, & \text{falls } u_j \in A \\ \underline{val}(u_j), & \text{falls } u_j \in V \end{cases}$$

a1) $b \in K$ $\quad D(IV, \beta_{n+1}) := D(IV, \beta_n)$
$\underline{val}`(x) := b$ $\quad \underline{val}`(x`) = \underline{val}(x`)$ für $x` \neq x$
a2) $b \notin K$ $\quad$ Erweiterung von $D(IV, \beta_n)$ um neuen Knoten für b
Bewertung $\underline{\text{val`}}$ wie in a1
b Es gibt $j \in \{1, \ldots, r\}$ mit $u_j \in V$ und $\underline{val}(u_j) \notin A$
b1 Es gibt $k \in k$ mit $\underline{lab}(b) = f$

$$\text{und } \underline{suc}(k, j) = \begin{cases} a, & \text{falls } u_j = a \in A \\ \underline{val}(u_j), & \text{falls } u_j \in V \end{cases}$$

$D(IV, \beta_{n+1}) := D(IV, \beta_n)$
$\underline{val}`(x) := k$ und $\underline{val}(x`) = \underline{val}(x`)$ für $x` \neq x$
b2) Es gibt kein $k \in K$, welches nach b1) den Wert e repräsentiert. $K` := K \dot\cup \{k`\}$ (neuer Knoten k`)
$\underline{lab}`(k`) := f$

$$\underline{suc}`(k`, j) := \begin{cases} a, & \text{falls } u_j = a \in A \\ \underline{val}(u_j), & \text{falls } u_j \in V \end{cases}$$

[Folie: Beispiel: DAG-Konstruktion]

[29.10.2004]

$\pi = (IV, \beta, OV) \in SLC$
$\beta = \alpha_1; \ldots; \alpha_n$
Eigenschaften von $D(IV, \beta)$:

1. Verschiedene Knoten repräsentieren verschiedene Terme:
   $k_1 \neq k_2 \curvearrowright t_{k_1} \neq t_{k_2}$
2. Kein Knoten repräsentiert einen konstanten Op-Term $t_k \in T_\Sigma(A) \curvearrowright t_k \in A$

<u>Folgerung:</u>

1. unterstützt CS-Elimination
2. unterstützt CF-Transformation

<u>Bsp:</u>

- gemeinsame Teilausdrücke: $\underline{val}(t) = \underline{val}(u)$
- Konstantenfaltung: $\underline{val}(z) = 4$, $\underline{val}(v) = \underline{val}(t) * 4$

$D(IV, \beta)$ unterstützt mit Hilfe von OV auch die DC-Elimination
$k \in K$ heisst <u>ausgaberelevant</u>, wenn $y \in OV$ existiert, so dass k von $\underline{val}(y)$ erreichbar ist.
Nicht ausgaberelevante Knoten repräsentieren Dead Code
<u>Bsp:</u> 2$\oplus$-Knoten und (2)
Durch entfernen dieser Knoten und Einschränkung von <u>val</u> auf OV erhalten wir:

1. den DAG von π: $D(\pi)$
2. die Bewertungsfunktion <u>val</u>: $OV \to K$

<u>Programmoptimierung mit $D(\pi)$ und val $OV \to K$</u>
Konstruktion eines DAG-optimierten Programms $T_D(\pi) := \pi_D := (IV, \beta_D)$

Fall 1 <u>val</u> injektiv, d.h. verschiedene Ausgabevariablen zeigen auf verschiedene Knoten

(a) $\underline{val}(OV) \cap (IV \cup A) = \emptyset$, d.h., Ausgabevariablen zeigen nur auf Op-Knoten
Idee: Op-Knoten als Hilfsvariablen verwenden und für $\underline{val}(y) = k \quad y$ statt k verwenden
Jeder Op-Knoten k bestimmt mit $\underline{lab}(k) = f \in \Sigma^{(r)}$ und den Nachfolgeknoten eine Anweisung:
$\alpha_k := K \leftarrow f(\underline{suc}(k, 1), \ldots, \underline{suc}(k, r))$
Beachte: $\underline{suc}(k, j) \in IV \cup A \cup K \cup OV$
Wähle eine Bottom-Up-Numerierung der Op-Knoten $k_1, k_2, \ldots, k_s$ (z.B schichtenweise von links nach rechts) so dass $k_j \leftarrow f(\ldots k_i \ldots) \curvearrowright i < j$.
Dann gilt $\pi_D := (IV, \beta_D, OV) \in SLC$ mit $\pi_D := \alpha_{k_1}; \ldots; \alpha_{k_s}$
Beachte: $OV \subseteq \{k_1, \ldots, k_i\}$ und π_D erfüllt die Variablenbedingung.
Ausserdem besitzt π_d die "single assignment"-Eigenschaft.

(b) $\underline{val}(y) \in IV \cup A$
π_D von a) um $y \leftarrow \underline{val}(y)$, es sei denn, dass $y = \underline{val}(y)$.

Fall 2 $\underline{val}(y_1) = \underline{val}(y_2) = \ldots = \underline{val}(y_n)$
π_D-Konstruktion von Fall 1 bezüglich y_1, durchführen auf $y_2 \leftarrow y_1, \ldots ; y_n \leftarrow y_1$ ergänzen

<u>Zusammenhang zwischen DAG-Optimierung und CS-, DC-, CF- und Cp-Optimierung</u>

Beispiel 2.4.3:

$T_{DC}(T_{CP}(T_{CS}(T_{CF}(\pi))))$

$T_{CF}(\pi)$:	$\underline{u \leftarrow 2;}$
$u \leftarrow 2;$	(1) $w \leftarrow x + y;$
$w \leftarrow x + y;$	$\cancel{z \leftarrow 4;}$
$z \leftarrow 4;$	(2) $u' \leftarrow x * w;$
$u \leftarrow x * w;$	$\cancel{u \leftarrow u';}$
$v \leftarrow 4 + w;$	$\cancel{v \leftarrow 4 + w;}$
$v \leftarrow u + 4;$	$\cancel{v \leftarrow u' + 4;}$
$t \leftarrow x * w;$	$\cancel{t \leftarrow u';}$
$v \leftarrow t * z;$	(3) $v \leftarrow u' * 4;$
OV: u, v	OV: u‘, v

Allgemein gilt:

1. Die DAG-Transformation ist korrekt:

   $\pi \approx \pi_D$

   <u>Beachte:</u> Die Konstantenfaltung kann bereits ohne Interpretation durchgeführt werden. (bzw. auf Termen; Grundterme als Konstantensymbole zulassen)

2. $T_D(\pi)$ ist optimal bzgl. T_{DC}, T_{CS}, T_{CF}, T_{CP}, d.h. keine dieser Transformationen kann $T_D(\pi)$ weiter optimieren.

3. $T_D(\pi)$ optimal bzgl. $\approx$ und Codelänge (?)

4. Vermutung: Für jedes $\pi \in SLC$ gibt es ein $n \in \mathbb{N}$ mit $T_D(\pi) = [T_{DC} \circ (T_{CP} \circ T_{CS})^n \circ T_{CF}](\pi)$ [03.11.2004]

5. T_D nicht auf iterative Programme übertragbar, wohl aber die anderen Transformationen T_{CS}, T_{CF}, T_{CP}, T_{DC}

2.5 Weitere Optimierung von SLC-Programmen

Berücksichtigung der Interpretation $\mathfrak{a} = \langle A_i; \varphi \rangle$ als (Σ, C)-Algebra
Algebraische Eigenschaften: Gleichungen

Beispiel 2.5.1:

$\langle \mathbb{Z}; +, -, *, / \rangle$

- neutrale Elemente:

$$\begin{aligned} 0 + a &= a \\ 1 * a &= a \\ a/1 &= 1 \end{aligned}$$

- kommutative Operationen:

$$\begin{aligned} a + b &= b + a \\ a * b &= b * a \end{aligned}$$

- assoziative Operationen: $(a + b) + c = a + (b + c)$
- distributive Operationen: $a * (b + c) = a * b + a * c$

Beispiel 2.5.2:
$\langle \mathbb{B}; \text{and}, \text{not}, \text{true}, \text{false} \rangle$

- true <u>or</u> b = true
- false <u>and</u> b = false
- <u>not</u> <u>not</u> b = b

Ziel algebraischer Transformationen:

- ersetze teure Anweisungen durch billigere
- lokale, kontextunabhängige Ersetzun
- keine Gesamtanalyse von π erforderlich

<u>Peep-hole-Optimierung</u>
Idee: schiebe ein kleines Fenster über das Programm und optimiere innerhalb des Fensters

Beispiel 2.5.3:

- $\left.\begin{array}{r} y \leftarrow x + a \\ z \leftarrow y - a; \end{array}\right\} \begin{array}{l} y \leftarrow x + a; \\ z \leftarrow x; \end{array}$
- $x \leftarrow x + 0;$ weglassen
- $u \leftarrow x + y;$
  $v \leftarrow y + z;$
  $w \leftarrow v + x;$
  v wird nicht mehr benötigt

$w = (y + z) + x = (x + y) + z$

$u \leftarrow x + y;$

$w \leftarrow u + z;$

Reduction in Strength
Ersetze $x \leftarrow y * *2$ durch $x \leftarrow y * y;$
Ersetze $x \leftarrow 2 * y$ durch $x \leftarrow y + y;$
Optimalität von SLC-Programmen
Spezialfall: $\mathfrak{a} = \langle \mathbb{Z}; +, * \rangle$
keine Konstanten
Kostenmass: Programmlänge
Dann gilt für $\pi \in SLC$:
es existiert ein optimales Programm π_{opt} für π
d.h. $\pi_{opt} \, \widetilde{\mathfrak{a}} \, \pi$ und $c(\pi_{opt}) \leq c(\pi‘)$ für alle $\pi‘$ mit $\pi‘ \, \widetilde{\mathfrak{a}} \, \pi$
Konstruktion von π_{opt}: NP-vollständig (?)
optimaler Code für arithmetische Ausdrücke
Polynomberechnung
Spezialfall: $g(x) = ax^{16} + x^8 + cx^4 + dx^2 + ex + f$

[Folie: Optimierung von Polynomen]

- DAG-optimales π mit 36 Anweisungen, 31 Multiplikationen
- möglich: 14 Anweisungen, 9 Multiplikationen

Semantik iterativer und rekursiver Programme ebenso wie Datenflussanalyse beschreibbar als Lösung von Gleichungssystemen
dazu: vollständige Halbordnungen, vollständige Verbände
monoton und stetige Funnktionen
Fixpunktberechungen

Definition 2.5.4:
Halbordnung Sei D eine Menge und $\leq \subseteq D \times D$.
Dann heisst $\mathcal{D} = \langle D, \leq \rangle$ eine Halbordnung mit der Halbordnungrelation $\leq$, falls für alle $a, b, c, \in D$ gilt:

- $a \leq a$ (reflexiv)
- $a \leq b$, $b \leq c \curvearrowright a \leq c$ (transitiv)
- $a \leq b$, $b \leq a \curvearrowright a = b$ (antisymmetrisch)

Sei $a \in D$ und $T \subseteq D$. Dann heisst

- a obere Schranke von T, falls $T \leq a$
- a kleinstes Element von T, falls $a \in T$ und $a \leq T$
- a kleinste obere Schranke von T, falls $T \leq a$ und $(T \leq a' \curvearrowright a \leq a')$

  Bez: $a \bigsqcup T$ ("join", "lub" (least upperbound))

  Beachte: kleinste Elemente, insbesondere $\bigsqcup$, sind eindeutig bestimmt.
- Besitzt $\mathcal{D}$ ein kleinstes Element, so wird es durch $\perp$ ("bottom") bezeichnet
- $\gamma : \mathbb{N} \to D$ heisst Kette, falls $\gamma(i) \leq \gamma(i+1)$ für alle $i \in \mathbb{N}$ (aufsteigende ω Kette)

  Bez.: $(\gamma(i) | i \in \mathbb{N}) = \gamma$

  nicht verwechseln mit: $\{\gamma(i) | i \in \mathbb{N}\}$
- $\mathcal{D}$ heisst vollständige Halbordnung falls $\mathcal{D}$ ein kleinstes Element hat und jede Kette γ eine kleinste obere Schranke besitzt: $\bigsqcup \gamma := \bigsqcup \{\gamma(i) | i \in \mathbb{N}\}$
- $\mathcal{D}$ heisst vollständiger Verband, falls jede Teilmenge $T \subseteq D$ eine kleinste obere Schranke $\bigsqcup T$ besitzt (Beachte: $\bigsqcup = \perp \emptyset$)

Schreibweise: $a \bigsqcup b := \bigsqcup \{a, b\}$

[05.11.2004]

Beispiel 2.5.5:
Seien A und B Mengen.
Die Menge der $PF(A, B) := \{f | f : A \dashrightarrow B\}$ der partiellen Funktionen von A nach B ist halbgeordnet durch $\leq$ mit $f \leq g :\curvearrowleft\curvearrowright (f(a) \in B \curvearrowright f(a) = g(a)$ für alle $a \in A)$, oder: $graph(f) \subseteq graph(g)$.
Die "leere" Funktion $f_\emptyset$ mit $Def(f_\emptyset) = \emptyset$ ist kleinstes Element von $\langle PF(A, B); \leq \rangle$
Jede Kette $f_0 \leq f_1 \leq \ldots$ besitzt eine kleinste obere Schranke $f = \bigsqcup \{f_i | i \in \mathbb{N}\}$.

$f(a) = b \in B \curvearrowleft\curvearrowright \exists i \in \mathbb{N} : f_i(a) = b$
Also ist $\langle PF(A, B); \leq\rangle$ eine vollständige Halbordnung (cpo = complete partial order)
Spezialfall: $PF(A) := PF(A, A)$
Anwendung: Semantik iterativer Programme

Beispiel 2.5.6:
Sei A eine Menge
Die Potenzmenge $P(A) := \{T|T \subseteq A\}$ ist halbgeordnet durch $\subseteq$ und bildet mit $\subseteq$ einen vollständigen Verband:
für $\mathcal{T} \subseteq P(A)$ gilt $\bigsqcup \mathcal{T} = \cup\{T|T \in \mathcal{T}\}$
Anwendung: Datenflussanalyse (DFA)

Beachte: Ist $\mathcal{D} = \langle D, \leq\rangle$ eine Halbordnung, so auch $\mathcal{D}` := \langle D, \geq\rangle$, die duale Halbordnung von D

Definition und Lemma 2.5.7: Produkt von Halbordnungen
Sind $\mathcal{D}_i = \langle D_i; \leq_i\rangle$ für $i = 1, 2$ Halbordnung, so auch $\mathcal{D}_1 \times \mathcal{D}_2 := \langle D_1 \times D_2; \leq\rangle$ mit $(a_1, a_2) \leq (b_1, b_2) :\curvearrowleft\curvearrowright a_i \leq_i b_i$ für $i = 1, 2$

Definition und Lemma 2.5.8: Funktionenraum
Sei A eine Menge. Ist $\mathcal{D} = \langle D; \leq\rangle$ eine Halbordnung, so auch $F(A, \mathcal{D}) := \langle F(A, D); \leq\rangle$ mit $F(A; D) := \{f|f : A \to D\}$ und $f \leq g :\curvearrowleft\curvearrowright f(a) \leq g(a)$ für alle $a \in A$

Folgerung 2.5.8:
Sind $\mathcal{D}$, $\mathcal{D}_1$ und $\mathcal{D}_2$ vollständige Halbordnungen bzw vollständige Verbände, so auch $\mathcal{D}_1 \times \mathcal{D}_2$ und $F(A, \mathcal{D})$.
Insbesondere gilt für Ketten:

- $\bigsqcup\{(a_1^{(i)}, a_2^{(i)})|i \in \mathbb{N}\} = (\bigsqcup\{a_1^{(i)}|i \in \mathbb{N}\}, \bigsqcup\{a_2^{(i)}|i \in \mathbb{N}\})$
- $\bigsqcup(\{f^{(i)}|i \in \mathbb{N}\})(a) = \bigsqcup\{f^{(i)}(a)|i \in \mathbb{N}\}$

und für beliebige Teilmengen

- $\bigsqcup T = (\bigsqcup \underline{proj}_1(T), \bigsqcup \underline{proj}_2(T))$ mit $T \subseteq D_1 \times D_2$
- $(\bigsqcup T)(a) = \bigsqcup T(a)$ mit $T \subseteq F(A, D)$.

Definition 2.5.9: monotone und stetige Funktionen
Seien $\mathcal{D}_i = \langle D_i; \leq_i\rangle$ für $i = 1, 2$ Halbordnungen und f: $D_1 \to D_2$.

- f heisst monoton, falls $a \leq_1 b \curvearrowright f(a) \leq_2 f(b)$ für alle $a, b \in D_1$,
- f heisst stetig, falls $\mathcal{D}_1$ und $\mathcal{D}_2$ vollständig, f monoton und für jede Kette $(a_i|i \in \mathbb{N})$ um $\mathcal{D}_1$ gilt: $f(\bigsqcup\{a_i|i \in \mathbb{N}\}) = \bigsqcup\{f(a_i)|i \in \mathbb{N}\}$

Beachte: f monoton $\curvearrowright$ $(f(a_i)|i \in \mathbb{N})$ Kette.

Definition 2.5.10:
$\mathcal{D}$ besitzt die ACC-Eigenschaft (ACC = ascending chain condition), wenn gilt:
Für jede Kette $(a_i|i \in \mathbb{N})$ ist $\{a_i|i \in \mathbb{N}\}$ endlich

Folgerung 2.5.10:

1. Eine HO mit ACC ist vollständig
2. Monotone Abb. auf HO mit ACC sind stetig

Definition 2.5.11: Distributivität
Sind $\mathcal{D}_1$ und $\mathcal{D}_2$ vollständige Verbände, so heisst $f : D_1 \to D_2$ distributiv, falls $f(a \bigsqcup b) = f(a) \bigsqcup f(b)$ für alle $a, b \in D_1$

Lemma 2.5.12:
Seien $\mathcal{D}_1$ und $\mathcal{D}_2$ vollständige Verbände. Dann gilt: $f : D_1 \to D - 2$ distributiv $\curvearrowright$ f monoton.
Beweis
$a \leq b \curvearrowright b = a \bigsqcup b \curvearrowright \underline{f(b)} = f(a \bigsqcup b) \underline{= f(a) \bigsqcup f(b)} \curvearrowright f(a) \leq f(b)$

Folgerung 2.5.12:
Besitzen $\mathcal{D}_1$ und $\mathcal{D}_2$ auch ACC, so gilt: $f : D_1 \to D_2$ distributiv $\curvearrowright$ f monoton $\curvearrowright$ f stetig

Satz 2.5.13: Fixpunktsatz
Sei $\mathcal{D} = \langle D; \leq \rangle$ eine vollständige HO und $f : D \to D$ stetig.
Dann gilt:

1. $(f^i(\perp)|i \in \mathbb{N})$ ist eine Kette von $\mathcal{D}$
2. $a := \bigsqcup\{f^i(\perp)|i \in \mathbb{N}\}$ ist Fixpunkt von f, d.h.: $a = f(a)$.
3. a ist kleinster Fixpunkt von f, d.h. $a = f(a)$, Bez: fix(f).

Beweis:

1. $\perp \leq f(\perp) \leq f^2(\perp) \leq \dots$ (weil f monoton)
2. $f(a) = f(\bigsqcup\{f^i(\perp)|i \in \mathbb{N}\}) = \bigsqcup\{f^{i+1}(\perp)|i \in \mathbb{N}\} = \bigsqcup\{f(\perp)|i \in \mathbb{N}\} = a$
3. $b = f(b)$
   $\perp \leq b$, $f(\perp) \leq b$, $f^2(\perp) \leq b \dots \curvearrowright$ b ist obere Schranke von $\{f^i(\perp)|i \in \mathbb{N})$
   $\curvearrowright a \leq b$

$\square$

Folgerung 2.5.13:
Ist $\mathcal{D}$ eine HO mit ACC und $f : D \to D$ monoton. Dann gilt: f besitzt einen kleinsten Fixpunkt $\underline{fix}(f)$ und es existiert $m \in \mathbb{N}$, so dass $\underline{fix}(f) = f^m(\bot)$

[10.11.2004]

Kapitel 4

4 Datenflussanalyse und Optimierung iterativer Programme

Zwischencode für eine abstrakte Maschine: SLC und Verzweigung und Iteration
Bedingte und unbedingte Sprünge als einzige Kontrollstrukturen
Inhalt:

- Syntax und Semantik von IC (iterativer Code)
- Basisblock-Darstellung von IC-Programmen
- Abstrakte Interpretation und Datenflussanalyse (DFA)
- Verallgemeinerung der Analyse und Optimierung von SLC-Programmen auf IC-Programmen

4.1 Syntax von IC

Erweiterung von SLC
$V = \{x, y, \ldots\}$ Variablen
$C = \{c, d, \ldots\}$ Konstante Symbole
$\Sigma = \cup_{r \geq 1} \Sigma^{(r)} Op - Symbole$ (evtl.:$C = \Sigma^{(?)}$)
$\pi = \cup_{r \geq 0} \pi^{(r)}$ Prädikatssymbole
Σ und π endlich
$L = \{l_i | i \in \mathbb{N}\}$ Sprungmarken (“Label“)

Definition 4.1.1: Anw Anweisungen

- Wertzuweisungen:
  $v \leftarrow e$ mit $v \in V$ und $e \in V \cup C \cup \{f(u_1, \ldots, u_r) | f \in \Sigma^{(r)}, u_i \in V \cup C\}$

- Sprunganweisungen:

  goto l mit $l \in L$, if $p(u_1, \ldots, u_r)$ goto l mit $p \in \pi^{(r)}$, $u_i \in V \cup C$, $l \in L$

[Folie: IC-Beispielprogramm: Fakultätsberechnungen]

[Folie: Beispiel: Flussdiagramm und -graph]

IC-Programme
$\pi = (Vlist, Alist) \in IC :\Longleftrightarrow Vlist =$ in $x_1, \ldots, x_n$; out $y_1, \ldots, y_m$; loc $z_1, \ldots, z_p$; mit

- $IV := \{x_1, \ldots, x_n\}$ Eingabevariablen $(n \geq 0)$
- $OV := \{y_1, \ldots, y_m\}$ Ausgabevariablen $(m \geq 1)$
- $LV := \{z_1, \ldots, z_p\}$ lokale Variablen $(p \geq 0)$
- $V_\pi = IV \cup OV \cup LV$ Programmvariablen

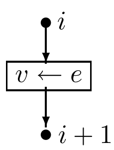
$Alist = \alpha_1; \ldots; \alpha_q$ (Anweisungsliste) mit

- $\alpha_i \in \underline{Anw}$, $V_{\alpha_i} \subseteq V_\pi$, $q \geq 0$
- eindeutige Sprungziele: $\alpha_i = l : \ldots$, $\alpha_j = l' : \ldots, i \neq j \curvearrowright l \neq l'$

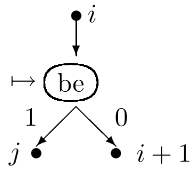
π erfüllt die Variablenbedingung: jede Programmvariable wird vor ihrer Benutzung definiert (Beachte: Variablen in bedingten Sprunganweisungen werden dort benutzt)
π heisst standardmarkiert, falls $\alpha_i = i : \alpha'_i$ für $i = 1, \ldots, q$ und $L_q \subseteq \{1, \ldots, q+1\}$ Marken von π
Bsp Fakultät (Folie) π_f berechnet die Fakultät

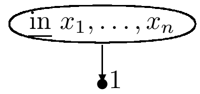
4.2 Semantik von IC

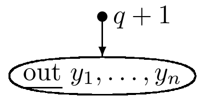
Sei $\pi = (Vlist, Alist) \in IC$, o.B.d.A. sei π standardmarkiert
Das Flussdiagram von π (Semantik der Kontrolle)
$Alist = 1 : \alpha_1; \ldots; q : \alpha_q$

$i : v \leftarrow e \mapsto$ 

i: goto j $\mapsto$ 

i: if be goto j $\mapsto$ 

Ferner: 

und: 

Das Flussdiagram von π ergibt sich durch Verkleben der entsprechenden Teilgraphen (lokale Variablen ergeben sich indirekt . . .)

Fixpunktsemantik

Sei $\mathfrak{a} = \langle A; \varphi\rangle$ eine Interpretation für π d.h. $\pi(f^{()}) = p_{\mathfrak{a}} : A^n \to \{0, 1\}$

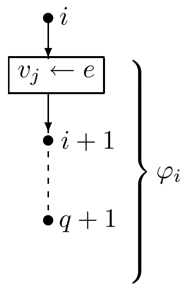
$V_\pi := \{v_1, \ldots, v_s\}$

Zustandsraum $Z := A^s\overline{a} = (a_1, \ldots, a_s)$ als Wert des Variablenvektor $\overline{v} = (v_1, \ldots, v_s)$

Für die Fortsetzungsfunktionen ("Continuations")

$\varphi_i : Z \to Z(i = 1, \ldots, q + 1)$ die von Programmmarke i bis zum Programmende berechnet Zustandstransformation, gelten folgende Gleichungen:

- $\alpha_i = v_j \leftarrow e$

  $i : v \leftarrow e \mapsto$ 

  $\curvearrowright \varphi_i(\sigma) = \varphi_{i+1}(\sigma[j/[e]\sigma])$

- $\alpha_i\ \underline{\text{goto}}\ l \curvearrowright \varphi_i(\sigma) = \varphi_l(\sigma)$

- $\alpha_i =\ \underline{\text{if}}\ be\ \underline{\text{goto}}\ l \curvearrowright \varphi(\sigma) =\ \underline{\text{if}}\ [be]\sigma\ \underline{\text{then}}\ \varphi_l(\sigma)\ \underline{\text{else}}\ \varphi_{i+1}(\sigma)$

[12.11.2004]

Fixpunktsemantik für $\pi \in IC$

$\pi = (Vlist, Alist)\ \mathfrak{a} = \langle A; \varphi\rangle$

$V_\varphi = \{v_1, \ldots, v_s\}\ Z := A^s$

$\overline{a} = (a_1, \ldots, a_s) \in Z$

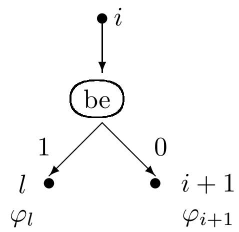
$Alist = 1 : \alpha_1; \ldots; q : \alpha_q$

Fortsetzungsfunktionen

- $\alpha_i = v_j \leftarrow e \qquad \varphi_i(\overline{a}) = \varphi_{i+1}(\overline{a}[j]/[e]\overline{a})$

- $\alpha_i =\ \underline{\text{goto}}\ l \qquad \varphi_i(\overline{a}) = \varphi_e(\overline{a})$

- $\alpha_i = \underline{\text{if}}\ be\ \underline{\text{goto}}\ l$



$\varphi_1(\overline{a}) = \underline{\text{if}}\ [be]\overline{a}\ \underline{\text{then}}\ \varphi_e(\overline{a})\ \underline{\text{else}}\ \varphi_{i+1}(\overline{a})$ und $\varphi_{q+1}(\overline{a}) = \overline{a}$

Die $\varphi_1, \ldots, \varphi_{q+1} : A^s \to A^s$ bilden daher die Lösung eines Gleichungssystems:

- $F_i(\overline{v} = \tau_i (i = 1, \ldots, q+1)$ mit

$$t_i = \begin{cases} F_{i+1}(\overline{v}[v_j/e], & \text{falls } \alpha_i = v_j \leftarrow e \\ F_l(\overline{v}, & \text{falls } \alpha_i = \underline{\text{goto}}\ l \\ \underline{\text{if}}\ be\ \underline{\text{then}}\ F_l(\overline{v})\ \underline{\text{else}}\ F_{i+1}(\overline{v}), & \text{falls } \alpha_i = \underline{\text{if}}\ be\ \underline{\text{goto}}\ l \end{cases}$$

$\tau_{q+1} = \overline{v}$

Berücksichtigung von Ein- / Ausgabe:

Sei $Vlist = \underline{in} x_1, \ldots, x_n; \underline{out} y_1, \ldots, y_n; \underline{loc} z_1, \ldots, z_p$

$\overline{x} = (x_1, \ldots, x_n)$ $(\overline{x}, \overline{a}_0) = \underbrace{(x_1, \ldots, x_n, a_0, \ldots, a_0)}_{s}$

$\overline{y} = (y_1, \ldots, y_m)$

Das Gleichungssystem (E_π) ist dann definiert durch

$$(E_\pi) \begin{cases} F(\overline{x}) = F_1(\overline{x}, \overline{a}_0)\ a_0 \in A \text{ beliebig} \\ F_i(\overline{v}) = \tau_i\ i = 1, \ldots, q \\ F_{q+1}(\overline{v}) = \overline{y} \end{cases}$$

Lösungstyp:

$f : A^n \to A^m f_i : A^s \to A^m$

Beispiel 4.2.1: Fakultätsprogramm

$$
\begin{aligned}
(E_\pi) F(x) &= F_1(x, 12, 12)\\
F_1(x,y,z) &= F_2(x,y,0)\\
F_2(x,y,z) &= F_3(x,1,z)\\
F_3(x,y,z) &= \underline{\text{if}}\ x = z\ \underline{\text{then}}\ F_7(x,y,z)\ \underline{\text{else}}\ F_4(x,y,z)\\
F_4(x,y,z) &= F_5(x,y,z+1)\\
F_5(x,y,z) &= F_6(x,y*z,z)\\
F_6(x,y,z) &= F_3(x,y,z)\\
F_7(x,y,z) &= y
\end{aligned}
$$

$\rightarrow F(x) = F_3(x,1,0)$
$F_3(x,y,z) = \underline{\text{if}}\ x = z\ \underline{\text{then}}\ y\ \underline{\text{else}}\ F_3(x, y*(z+1), z+1)$

$\rightarrow$ Endrekursive Funktionsgleichungen ("tail recursive")
Funktionsaufrufe nur in Termspitze, keine Rücksprünge

a) Reduktionssymantik (operationale Semantik)

$F(3) \rightarrow F_3(3,1,0) \rightarrow F_3(3,1,1) \rightarrow F_3(3,2,2) \rightarrow F_3(3,6,3) \rightarrow 6$

$\underline{fac}(3) = 1*2*3 = 6$

b) Fixpunktsemantik

Der Funktionenraum (Lösungsraum) $FR_\pi := PF(A^n, A^m) \times PF(A^s, A^m)^{q+1}$ ist eine vollständige Halbordnung.

Die Gleichungstransformation ("Einsetzungsfunktional") $T_\pi : FR_\pi \rightarrow FR_\pi$ ist wie folgt definiert:

für $\overline{\varphi} = (\varphi, \varphi_1, \dots, \varphi_{q+1}) \in FR_\pi$ und $\overline{F} = (F, F_1, \dots, F_{q+1})$ bezeichne $\tau_i[\overline{F}/\overline{\varphi}]$ die Ersetzung der Funktionsvariablen aus $\overline{F}$ durch die entsprechenden Funktionen aus $\overline{\varphi}$

Damit:

$T_\pi(\overline{\varphi}) := (\lambda\overline{x}, \varphi_1(\overline{x}, \overline{a}_0), \lambda\overline{v}.\tau_1[\overline{F}/\overline{\varphi}], \dots, \lambda\overline{v}.\tau_q[\overline{F}/\overline{\varphi}], \lambda, \overline{v}, \overline{y})$

Beachte: $\overline{\varphi}$ ist Lösung von E_π gdw. $\overline{\varphi} = T_\pi(\overline{\varphi})$

T_π ist stetig und besitzt daher einen kleinsten Fixpunkt $\underline{fix}(T_\pi)$. Seine 1. Komponente $[\pi]_{fix} := \underline{proj}_1(\underline{fix}(T_\pi)) : A^n \rightarrow A^m = \underline{proj}_1(\bigsqcup\{T_\pi^i(\perp) | i \in \mathbb{N}\})$ $\quad \perp = (f_\emptyset, f_{1_\emptyset}, \dots, f_{q+1_\emptyset})$ heisst Fixpunktsemantik von π

[17.11.2004]

Bsp Fakultätsprogramm
$F(x) = G(x,1,0)$
$G(x,y,z) = \underline{\text{if}}\ x = z\ \underline{\text{then}}\ y\ \underline{\text{else}}\ G(x, y*(z+1), z+1)$
$FR_{\pi_f} = PF(\mathbb{Z},\mathbb{Z}) \times PF(\mathbb{Z}^3, \mathbb{Z})$
$\perp = (f_\emptyset = \lambda x., g_\emptyset = \lambda(x,y,z).)$
$T(\perp) = (f_\emptyset = \lambda x., \lambda(x,y,z).\ \underline{\text{if}}\ x = z\ \underline{\text{then}}\ y)$
$T^2(\perp) = (\lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1, \lambda(x,y,z).\ \underline{\text{if}}\ x = z\ \underline{\text{then}}\ y \quad \underline{\text{if}}\ x = z+1\ \underline{\text{then}}\ y * (z+1))$

$$T^3(\bot) = (\lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1 \lambda(x,y,z). \quad \underline{\text{if}}\ x = z\ \underline{\text{then}}\ y$$
$$\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 1) \quad \underline{\text{if}}\ x = z+1\ \underline{\text{then}}\ y * (z+1)$$
$$\underline{\text{if}}\ x = z+2\ \underline{\text{then}}\ y * (z+1) * (z+2)$$

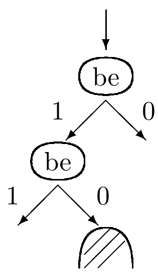
$$\begin{aligned}T^{n+1}(\bot) = (&\lambda x.\ \underline{\text{if}}\ x \leq n-1\ \underline{\text{then}}\ x`,\\ &(\lambda(x,y,z).\ \underline{\text{if}}\ x = z\ \underline{\text{then}}\ y\\ &\underline{\text{if}}\ z+1\ \underline{\text{then}}\ y*(z+1)\\ &\underline{\text{if}}\ z+n\ \underline{\text{then}}\ y*(z+1)*(z+2)*\ldots*(z+n)\end{aligned}$$

Also: $[\pi]_{fix}(x) = \underline{\text{if}}\ x \geq 0\ \underline{\text{then}}\ x!$

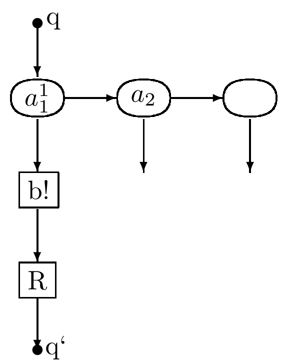
Folgerungen der Semantik von IC-Programmen

1. Eine Anweisung kann wiederholt werden, evtl. unendlich oft (Programmschleifen)
2. Ein Programmpfad (einen eingabeunabhängiger Pfad im Flussdiagramm vom eingabe- zum Ausgabeknoten) muss kein Berechnungspfad sein (eingabeunabhängiger Pfad einer Berechnung)

   Bsp



a! R, L | a?
q a b R q'



3. TM als IC-Programme
4. Programmäquivalenz nicht entscheidbar

→ Programmanalyse: Approximation von Programmeigenschaften, Prädikate ignorieren, Programmpfade betrachten

- Menge der Programmpfade von $\pi \in IC$ ist regulär.
- Menge der Berechnungspfade ist i. a. rekursiv aufzählbar, aber nicht entscheidbar

⇒ sichere, aber evtl. unvollständige Optimierungsinformationen.

4.3 Flussgraphen von IC-Programmen

Idee: deterministische Verzweigung durch nicht-deterministische Auswahl ersetzen; Programmpfade statt Berechnungspfade
Sei $\pi = \langle Vlist, Alist\rangle \in IC$ und $Alist = 1 : \alpha_1; \ldots; q : \alpha_q$

1. SI-Graph (SI = Single Instruction)

   $G_\pi^{SI} = \langle \underline{Kno}, \underline{Kan}, s\rangle$

   $\underline{Kno} := \{\alpha_i | \alpha_i = x \leftarrow e, 1 \leq i \leq q\} u \cup \{be | \alpha_i = \underline{\text{if}}\ be\ \underline{\text{goto}}\ j, 1 \leq i, j \leq q\}$

   $s = \alpha_1$ (o.B.d.A α_1 Wertzuweisung)

   Kan wie im Flussdiagramm

2. BB - Graph (BB = basic block)

   Ziel: Vereinfachung der Programmanalyse

   Idee: Zerlegung von π in maximale SLC-Blöcke

   Konstruktion von $G_\pi^{BB} = \langle \underline{Kno}, \underline{Kan}, s\rangle$:

   α_i heisst Blockanfang, falls eine der folgenden Bedingungen gilt:

   - $i = 1$
   - i ist Sprungziel von π (kommt als goto i in π vor)
   - α_{i-1} ist Sprunganweisung

   Ein Blockanfang α_i bestimmt $\alpha_i; \alpha_{i+1}; \ldots; \alpha_j$ als längste Anweisungsfolge ohne weiteren Blockanfang. Weglassen von goto k und ersetzen von if be goto k durch be ergibt den Basisblock $\alpha_{i_1}`; \ldots, \alpha_{i_k}`$

   Bsp π_f Blockanfänge $\alpha_1, \alpha_3, \alpha_4$

   $\beta_1 = \alpha_1; \alpha_2$

   $\beta_2 = \alpha_3$

   $\beta_3 = \alpha_4, \alpha_5$

[19.11.2004]

[Folie: ??? von π_f]

[Folie: Beispiel: BB- und SI-Graph]

Flussgraphen von IC-Programmen (Korrektur)

1. SI-Graphen

   $\pi = 1 : \alpha_1; \ldots; q : \alpha_q$

   $G_\pi^{SI} = \langle \underline{Kno}, \underline{Kan}, s\rangle$ und lab: Kno $\rightarrow$ Wertzuweisungen $\cup$ Bool. Exp. $\cup$ $\{START\}$

   $\underline{Kno} := \{i | 0 \leq i \leq q, \alpha_i \neq \underline{\text{goto}}\ j\}$

   $\underline{lab}(0) = START$

   $$\underline{lab}(i) = \begin{cases} \alpha_i, & \text{falls } \alpha_i = x \leftarrow e \\ be, & \text{falls } \alpha_i\ \underline{\text{if}}\ be\ \underline{\text{goto}}\ j \end{cases}$$

   $1 \leq i \leq q$; $s := 0$

   Kan folgen aus Flussdiagram von π $\{\boxed{\text{START}} \rightarrow \boxed{}$

2. BB-Regel
   $G_\pi^{BB} = \langle \underline{Kno}, \underline{Kan}, s \rangle$
   siehe letzte Vorlesung
   Kno = Basisblöcke + Startblöcke
3. Flussgraphen für die Rückwärtsanalyse
   $G_{\overleftarrow{\pi}}^{SI}$, $G_{\overleftarrow{\pi}}^{BB}$ Kanten invertieren
   STOP-Knoten des Flussdiagrams als Startknoten

4.4 Datenflussanalyse-Systeme

2 Komponenten:

- ein Flussgraph als Abstraktion eines Flussdiagrams (Programmpfade)
- eine abstrakte Interpretation:
  Analyseinformation als Element eines vollständigen Verbands mit ACC
  Informationstransformationen als monotone Abbildungen

Definition 4.4.1:
Sei Kno eine nicht-leere endliche Menge von Knoten;
Kan $\subseteq$ Kno $\times$ Kno eine Menge von Kanten
und $s \in$ Kno ein Startknoten.
Für $k \in$ Kno bezeichne Var $(k) := \{k‘ \in$ Kno $|(k‘, k) \in$ Kan$\}$ die Menge der (direkten) Vorgängerknoten.
Für $k, k‘ \in$ Kno bezeichne
$\underline{Pfad}[k, k‘) := \{(k_1, \ldots, k_r) | k_1 = k, k_r \in$ Var$(k‘)$ und $(k_i, k_{i+1}) \in$ Kan für $i = 1, \ldots, r-1\}$
die Menge der Pfade von k zu einem direkten Vorgängerknoten von k‘.
$G = \langle \underline{Kno}, \underline{Kan}, s \rangle$ heisst Flussgraph, falls
$\underline{Var}(s) = \emptyset$.

Folgerung 4.4.1:
Für $\pi \in IC$ sind G_π^{SI}, G_π^{BB}, $G_{\overleftarrow{\pi}}^{SI}$, $G_{\overleftarrow{\pi}}^{BB}$ Flussgraphen

Definition 4.4.2: Abstrakte Interpretation eines Flussgraphen
Sei $G = \langle \underline{Kno}, \underline{Kan}, s \rangle$ ein Flussgraph.
Sei $\mathcal{D} = \langle D; \leq \rangle$ ein vollständiger Verband mit ACC
$d_s \in D$ eine Startinformation
und $\varphi : \underline{Kno} \to \{f | f : D \to f$ monoton$\}$ mit $\varphi(s) = \underline{\mathrm{id}}_D$
Dann heisst $\mathfrak{I} = \langle \mathcal{D}, \varphi, d_s \rangle$ eine abstrakte Interpretation von G

Definition 4.4.3: DFA-System

Ist G ein Flussgraph und $\mathfrak{I}$ eine abstrakte Interpretation von G, so heisst $\Delta = \langle G, \mathfrak{I}\rangle$ ein DFA-System (bei Indermark, in der Literatur: "monotone framework") ein DFA-System bestimmt in natürlicher Weise eine MOP-Lösung (meet over all paths) sowie eine MFP-Lösung (maximal fixed-point)
Nachtrag: $\varphi_k(d) = \varphi(k)(d)$

Definition 4.4.4: Die MOP-Lösung eines DFA-Systems
Sei $\Delta = \langle G, \mathfrak{I}\rangle$ ein DFA-System
Für $k \in$ Kno und $p = (k_1, \ldots, k_r) \in$ Pfad $[s, k)$
definieren wir die Analyseinformation von k bzgl. p
$AI_k^p := \varphi_{k_r}(\varphi_{k_{r-1}} \ldots (\varphi_{k_1}(d_s) \ldots)$
Dann ist die MOP-Lösung für k die folgende Analyseinformation
$AI_R^{MOP} := \bigsqcup\{AI_k^p | p \in$ Pfad $[s,k)\}$ für $k \neq s$ und $AI_s^{MOP} := d_s$
$AI^{MOP} := (AI_k^{MOP} | k \in$ Kno) heisst MOP-Lösung von Δ

Bem Die MOP-Lösung AI^{MOP} ist im allgemeinen nicht berechenbar.

Definition 4.4.5: Die MFP-Lösung eines DFA-Systems
Sei $\Delta = \langle G, \mathfrak{I}\rangle$ ein DFA-System.
Δ bestimmt ein System E_Δ von Datenflussgleichungen

$$(E_\Delta) \begin{cases} X_s = d_s \\ X_k = \bigsqcup\{\varphi_i(X_i) | i \in \text{Var}\ (k)\}, \quad k \in \text{Kno}\ /\{s\} \end{cases}$$

[24.11.2004]

E_Δ bestimmt eine Gleichungstransformation
$T_\Delta : D^n \to D^n$
$T_\Delta(A_1, \ldots, a_n) := (a_s, \bigsqcup\{\varphi_i(a_i) | i \in \text{Var}\ (2), \ldots, \bigsqcup\{\varphi_i(a_i) | i \in \text{Var}\ (n))$
und die MFP-Lösung von Δ
$AI^{MFP} :=$ fix $(T_\Delta) \in D^n$
Beachte: $\mathcal{D} = \langle D, \leq\rangle$ vollständiger Verband mit ACC.
$\varphi_i : D \to D$ monoton $\curvearrowright \varphi_i$ stetig $\curvearrowright T_\Delta$ stetig und wegen ACC existiert $m \in \mathbb{N}$ mit fix $(T_\Delta) = T_\Delta^m(\perp) in D^n$
Also: AI^{MFP} in endlich vielen Schritten berechenbar.

Satz 4.4.6: Correctness-Theorem
In einem DFA-System Δ gilt:
$AI^{eigentl.} \leq AI^{MOP} \leq AI^{MFP}$
Beweis: Es genügt zu zeigen, dass für alle $1 < k \leq n$, $p \in$ Pfad $[1, k)$ und $a = (a_1, \ldots, a_n)$ mit $a = T_\Delta(a)$ gilt:
(*) $AI_r^p \leq a_k$
(*) $\curvearrowright$ Beh.: $AI_i^{MOP} = d_1 = AI_1^{MFP}$ $k \neq 1$ $AI_k^p \leq AI_k^{MFP}$ nach (*) wegen Fp-Eigenschaft von AI^{MFD}, also: $AI_k^{MOP} \leq AI_k^{MFD}$
Beweis von (*): Aus (E_Δ) und $a = T_\Delta(a)$
(**) $\varphi_i(a_i) \leq a_k$ für $1 < k \leq n$ und $i \in$ Var (k)
Wir zeigen (*) durch Induktion über die Länge r von p

$r = 1$: Dann muss $p = 1 \in \underline{\text{Var}}\,(k)$, so dass nach (**)

$AI_k^p = \varphi_1(a_1) \leq a_k$

$\tilde{r}r + 1$: Sei $p \in \underline{\text{Pfad}}\,[1, k)$ mit Länge r + 1.

$p = p` * i$ mit $i \in \underline{\text{Var}}\,(k)$, $i \neq 1$ (wegen $\underline{\text{Var}}\,(1) = \emptyset$) und $p` \in \underline{\text{Pfad}}\,[1, i)$

[Grafik]

Nach IV: $AI_i^{p`} \leq a_i$

und nach (**) $\varphi_i(a_i) \leq a_k$

Aus beiden Ungleichungen un der Monotonie von φ_i

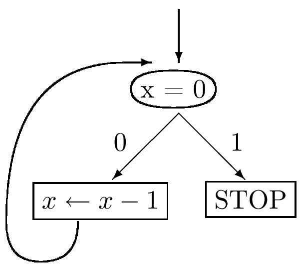
folgt: $AI_k^p = \varphi_i(AI_i^{p`}) \leq \varphi_i(a_i) \leq a_R$

[/Grafik]

□

Satz 4.4.7: Compleetness Theorem, Coincidence Theorem
Sind in einem DFA-System Δ die Funktion $\varphi_k : D \to D$ distributiv ($\varphi_k(a \bigsqcup b) = \varphi_k(a) \bigsqcup \varphi_k(b)$), so gilt:
$AI^{MOP} = AI^{MFP}$
Beweis: Es bleibt zu zeigen, dass $a = (a_i, \ldots, a_n) := (AI_1^{MOP}, \ldots AI_n^{MOP})$ ein Fixpunkt von T_Δ ist: $a = T_\Delta(a)$
$T_\Delta(a) = (d_s, \bigsqcup_{i \in \underline{\text{Var}}\,(2)} \varphi_i(a_i), \ldots)$
$= (d_s, \bigsqcup_{i \in \underline{\text{Var}}\,(2)} \varphi_i(AI_i^{MOP}), \ldots)$
$= (d_s, \bigsqcup_{i \in \underline{\text{Var}}\,(2)} \varphi_i(\bigsqcup_{p \in \underline{\text{Pfad}}\,[1,i)}), \ldots)$
[26.11.2004] $= ldots$ φ_i distr + ACC $\Rightarrow \varphi(\bigsqcup T) = \bigsqcup \varphi_i(T)$

Nachtrag: Kleinster Fixpunkt



$F(x) = \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ F(x-1)$

a) oper. Semantik $a \in \mathbb{Z}$:

$a \geq 0$: $F(a) \to F(a-1) \to F(a-2) \ldots F(0) \to 0$
$a < 0$: $F(a) \to F(a-1) \to \ldots$

$[\pi]_{op} = \lambda x.\ \underline{\text{if}}\ x \geq 0\ \underline{\text{then}}\ 0$
$[\pi]_{op}(x) = \underline{\text{if}}\ x \geq 0\ \underline{\text{then}}\ 0$

b) Fp-Semantik $\lambda x.$

$\lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 0$

$\lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}$

$\underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 0$

$[\pi]_{fix} = \lambda x.\ \underline{\text{if}}\ x \geq 0\ \underline{\text{then}}\ 0$

Weitere Fp:

$\lambda x.\ \underline{\text{if}}\ x \geq 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ A$

Nachtrag zu Kap 3:

Lemma 4.4.8:
Sei $\mathcal{D} = \langle D, \leq\rangle$ ein vollständiger Verband mit ACC und $f : D \to D$.
Dann gilt: f distributiv $(f(a \bigsqcup b) = f(a) \bigsqcup f(b)) \curvearrowright f(\bigsqcup T) = \bigsqcup f(T)$ für alle $T \subseteq D$, $T \neq \emptyset$
Beweis:
Sei $T \subseteq D$, $T \neq \emptyset$.

1. f distributiv $\curvearrowright$ f monoton $\curvearrowright f(T) \leq f(\bigsqcup T) \curvearrowright \bigsqcup f(T) \leq f(\bigsqcup T)$
2. Bleibt zu zeigen: $f(\bigsqcup T) \leq \bigsqcup f(T)$

   (*) Es existiert eine endliche Teilmenge $T_e \subseteq T$ mit $\bigsqcup(Z) \leq \bigsqcup T_e$

   Konstruktion einere Kette in D: $a_0 \leq a_1 \leq \ldots$

   $a_0 \in T$ beliebig

   $a_1 \to a_{n+1}$:

   (a) $T \leq a_n \quad a_{n+1} := a_n$ (also: $a_{n+k} = a_n$)
   (b) $T \not\leq a_n$ Dann existiert $b_n \in T$ mit $b_n \not\leq a_n$
   $a_{n+1} := a_n \bigsqcup b_n (a_n < a_{n+1})$

   ACC $\curvearrowright a_0 < a_1 < \ldots < a_n = a_{n+1} = a_{n+2}$

   $a_n = a_0 \bigsqcup a_0 \bigsqcup b_0 \bigsqcup b_1 \bigsqcup \ldots \bigsqcup b_n$

   und $T \leq a_n$, somit $\bigsqcup T \leq a_n = \bigsqcup\{\underbrace{a_0, b_n, \ldots, b_n}_{=T_e}\}$

   Damit folgt die Behauptung:

   $f(\bigsqcup T) \leq f(a_n) = f(a_0 \bigsqcup b_1 \bigsqcup b_2 \ldots \bigsqcup b_n)$

   $= f(a_0) \bigsqcup f(b_1) \bigsqcup \ldots f(b_n) = \bigsqcup\{f(a_0), f(b_1), \ldots f(b_n)\} \leq \bigsqcup f(T)$.

Lemma 4.4.9: Doppelt indizierte Mengen
$\mathcal{D} = \langle D; \leq\rangle$ vollständiger Verband
$T = \{a_{i_j} | i \in I, j \in J\} \subseteq D$
$\bigsqcup_{i \in I, j \in J} a_{i_j} = \bigsqcup_{i \in I} \bigsqcup_{j \in J} a_{i_j}$

Beweis: trivial

$AI_2^{p*i} = \varphi_i(AI_i^p)$

Satz 4.4.7: (Fortsetzung)
$= (d_s, \bigsqcup_{i \in \underline{\text{Var}}(2), p \in \underline{\text{Pfad}}[1,i]} AI_k^{p*i}, \ldots)$
$= (d_s, \bigsqcup_{p‘ \in \underline{\text{Pfad}}[1,2]} AI_k^{p‘}, \ldots, \bigsqcup_{p‘ \in \underline{\text{Pfad}}[1,n]} AI_n^{p‘})$
$= (a_1, \ldots, a_n) = a$
□

[Folie: Beispiel: Common Subexpression Elimination]

[01.12.2004]

4.5 AE-Analyse und CS-Elimination für IC-Programme

Ziel: Vermeidung wiederholter Auswertung von Ausdrücken
Beachte: Ein Ausdruck ist für eine Anweisung verfügbar, wenn er auf jedem Berechnungspfad zu diese Anweisung berechnet wird und die Werte dieser Variablen danach nicht verändert werden.

AE-Analyse Zerlegung der Analyse mit Hilfe von Basisblöcken in

1. Globale Analyse für BB-Graph
2. Lokale Analyse für Basisblöcke

Aufgabe (für 1): Konstruktion eines geeigneten DFA-Systems

$\Delta = \langle G, \mathfrak{I} \rangle$ zu $\pi \in IC$

[Folie: Beispiel: Common Subexpression Elimination]

G = BB-Graph für π für Vorwärtsanalyse

$\mathfrak{I} = \langle \mathcal{D}, \varphi, d_s \rangle$

Programmausdrücke:

$\underline{\text{OpExp}}_\pi := \{e | e = f(u_1, \ldots, u_r), \alpha_i = x \leftarrow e\} \cup \{be | be = p(u_1, \ldots, u_r), \alpha_i = \underline{\text{if}}\ be\ \underline{\text{goto}}\ \ldots, r \geq 1\} = \{e_1, \ldots e_t\}$

$D := \mathbb{B}$ zur Darstellung von Teilmengen von $\underline{OpExp}_\pi$

$\overline{b} = (b_1, \ldots, b_t)$

$$b_i = \begin{cases} 0, & \text{falls } e_i \text{ verfügbar} \\ 1, & \text{sonst} \end{cases}$$

$\overline{b} \leq \overline{b}‘ :\curvearrowleft\curvearrowright b_i \leq b‘_i\ i = 1, \ldots, t\ (0 < 1)$

$\perp = (0, \ldots, 0)$ “beste Information“

$\top = (1, \ldots, 1)\ 0 \sqcup 1 = 1$

$(b_1, \ldots, b_t) \sqcup (b‘_1, \ldots, b‘_t) = (b_1 \sqcup b‘_1, \ldots, b_t \sqcup b‘_t)$

—

$d_s = (1, \ldots, 1)$

—

Konstruktion von $\varphi: \underline{\text{Kno}} \rightarrow \{f : \mathbb{B}^t \rightarrow \mathbb{B}^t | \text{monoton}\}$

Zunächst für einzelne Anweisungen, dann für Blöcke: $\alpha \in \underline{\text{Anw}} \mapsto \varphi_\alpha : \mathbb{B}^t \rightarrow \mathbb{B}^t$

- $\varphi_{x \leftarrow e} := kill_x \circ gen_e$
  $gen_e(b_1, \ldots, b_t) := (b`_1, \ldots, b`_t)$
  $$b`_i = \begin{cases} 0, & \text{falls } e = e_i \\ b_i, & \text{sonst} \end{cases}$$
  $kill_x(b_1, \ldots, b_t)$
  $$b`_i = \begin{cases} 1, & \text{falls } x \in V_{e_i} \\ b_i, & \text{sonst} \end{cases}$$
- $\varphi_{gotok} := \underline{\text{id}}_{\mathbb{B}^t}$
- $\varphi_{\underline{\text{if}}\ be\ \underline{\text{goto}}\ k} := gen_{be}$

Beachte: φ_α monoton, weil alle gen- und kill-Funktionen monoton sind

Konstruktion der <u>Blocktransformationen</u>

$\beta = \alpha_1; \ldots; \alpha_k$ ($k > 0$, weil Sprungbefehle bleiben)

$\varphi_\beta := \varphi_{\alpha_k} \circ \ldots \circ \varphi_{\alpha_1}$ monoton

Für dieses DFA-System $\Delta = \langle G, \mathfrak{I} \rangle$ gilt: $AE_\beta^{MOP} = AE_\beta^{MFP}$ für alle $\beta \in \underline{\text{Kno}}$

(AE "available expression" statt AI "Analyseinformation")

$\rightarrow$ <u>distributive</u> Funktionen!

Grund: alle φ_β sind distributiv

$gen_{e_i}(b) \sqcup gen_{e_i}(b`) = (b_1, \ldots \underset{i}{0} \ldots b_t) \sqcup (b`_1, \ldots \underset{i}{0} \ldots b`_t)$

$gen_i(b \sqcup b`) = (b_1, \sqcup b`_1, \ldots \underset{i}{0} \ldots b_t \sqcup b`_t)$

$\underline{\text{OpExp}}_\pi := \{\underbrace{x+y}1, \underbrace{x*z}2, \underbrace{t*t}3, \underbrace{z*2}4, \underbrace{i+1}5, \underbrace{z>t}6, \underbrace{i>0}7\}$

$D = \mathbb{B}^7$

$\varphi_1(b_1, \ldots b_7) = (b_1, \ldots, b_7)$

$\varphi_2(b_1, \ldots b_7) = (0, 0, 0, 1, 1, 1, 1)$

$\varphi_3(b_1, \ldots b_7) = (0, 1, b_3, 1, b_5, 0, b_7)$

$\varphi_4(b_1, \ldots b_7) = (b_1, 0 b_3, \ldots b_7)$

$\varphi_5(b_1, \ldots b_7) = (b_1, b_2, 0, \ldots, b_7)$

$\varphi_6(b_1, \ldots b_7) = (b_1, \ldots b_4, 1, b_6, b_7)$

$$(E_\Delta)\begin{cases} X_1 = (1,\dots,1) \\ X_2 = X_1 \\ X_3 = (0,0,0,1,1,1,1) \sqcup \varphi_6(X_6) \\ X_4 = \varphi_3(X_3) \\ X_5 = \varphi_3(X_3) \\ X_6 = \varphi_4(X_4) \sqcup \varphi_5(X_5) \end{cases}$$

<u>Vereinfachung:</u> $X_3 = (0,0,0,1,1,1,1) \sqcup \varphi_6([\varphi_4 \sqcup \varphi_5]\varphi_3(X_3))$

$\varphi_6([\underline{\varphi_4 \sqcup \varphi_5}]\varphi_3(b_1,\dots,b_7) = (0,1,b_3,1,1,0,0)$

Fixpunkt-Iteration für (*) beginnt mit dem besten Wert $\perp = (0,\dots,0) \in \mathbb{B}^7$

$T^1_{(*)}(\perp) = (0,1,0,1,1,1,1) = T^2_{(*)}(\perp)$

[08.12.2004]

[Folie: Beispiel: Common Subexpression Elimination]

Fixpunkt-Iteration für (*):
$\perp = (0,\dots,0) \in \mathbb{B}^7$
$T_{(*)}(\perp) = (0,0,0,1,1,1,1) \sqcup (0,1,0,1,1,0,0) = (0,1,0,1,1,1,1)$
$T_{(*)}(\perp) = (0,0,0,1,1,1,1) \sqcup (0,1,0,1,1,0,0) = (0,1,0,1,1,1,1)$
$\underline{\text{fix}}(T_{(*)}) = (0,1,0,1,1,1,1)$
<u>Ergebnis:</u> $AE_1 = (1,\dots,1)$
$AE_2 = (1,\dots,1)$
$AE_3 = (0,1,0,1,1,1,1)$
$AE_4 = (0,1,0,1,1,0,1)$
$AE_5 = (0,1,0,1,1,0,1)$
$\varphi_3(b_1,\dots,b_7) = (0,1,b_3,1,b_5,0,b_7)$
$\varphi_4(b_1,\dots,b_7) = (b_1,0,b_3,\dots)$
$\varphi_5(b_1,\dots,b_7) = (b_1,b_2,0,b_4,\dots)$
$\rightarrow AE_6 = AE_4 = (0,1,0,1,1,0,1)$

Folgerung 4.5.0:
$\underline{x+y}$, $\underline{t*t}$ verfügbar in Block 3, 4, 5, 6
$z > t$ verfügbar in Block 4, 5, 6

<u>Beachte:</u>

1. Gabs schon... $x * z$ bereits füer B3 nicht verfügbar, weil B3 von B2 <u>und B6</u> erreichbar (Ohne B5 wäre $x * z$ für B3 verfügbar)

2. Lokale analysen für Basisblöcke

   Unterschied zur AE-Analyse an SLC-Programmen

   Neue Startinformationen: nicht mehr $(1,\dots,1)$

   sondern die durch globale Analyse gewonnenen Informationen AE_i für Block i ($i = 1,\dots,6$ im Beispiel)

- CS-Elimination
  $T_{CS} : SLC \rightarrow SLC$ übertragen
  Zwischenspeichern auf temporären Variablen

4.6 RD-Analyse und Konstantenfaltung

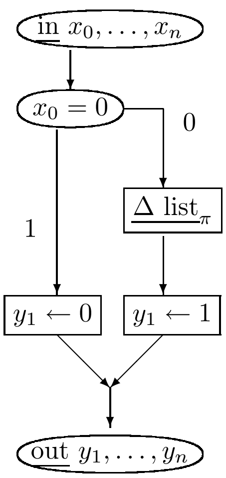
Ziel: Ersetze konstante Variablen und konstante Ausdrücke auf rechten Seiten von Wertzuweisungen und in Bedingungen durch ihre Werte (Propagation und Faltung)

Definition 4.6.1:
Sei $\pi \in IC$ mit Interpretation $\mathfrak{a} = \langle A, \varphi \rangle$, $v \in V_\pi$, $i \in L_\pi$ und $a \in A$.
Dann hat v bei i den konstanten Wert a (Bez.: $konst(v, i) = a$), falls dies auf jedem Berechnungspfad von i gilt.

Beachte i kann auf verschiedenen Berechnungspfaden (verschiedene Eingaben) und mehrfach auf einem Berechnungspfad erreicht werden.

Satz 4.6.2:
Die Konstanzeigenschaft von Variablen eines $\pi \in IC$ ist i.a. nicht entscheidbar (i.a. bei entsprechender Wahl der Interpretation)
Beweis Angenommen, die Konstanz-Eigenschaft wäre für alle Interpretationen entscheidbar.
Interpretation: $\mathfrak{a} = \langle \mathbb{N}, \underline{+1}, \underline{0}, \underline{0?} \rangle$
$IC(\mathfrak{a})$ universell (jede beliebige Funktion darstellbar)
Sei $\pi \in IC(\mathfrak{a})$, $IV_\pi = \{x_1, \ldots, x_n\}$, $OV_\pi = \{y_1, \ldots, y_m\}$

Neues Programm: $\pi_{const} \in IC(\mathfrak{a})$:



Dann gilt: $\underline{\text{const}}(y_1, i) = 0 \curvearrowright \pi$ divergiert [10.12.2004]

[Folie: Gleichungssystem????]

[Folie: Nicht-entscheidbarkeit der Konstanzeigenschaft]

$\sqcup \widehat{=} \sqcap$
$const(y_1, 0) \curvearrowleft\curvearrowright \pi$ divergiert, d.h. bei keiner Eingabe terminiert
Entscheidbarkeit der Konstanzeigenschaft $\curvearrowright$ Entscheidbarkeit der Divergenz von π $\curvearrowright$ Entscheidbarkeit des Halteproblems von π
(Modifikation von π durch Zuweisung der Eingabewerte an Eingabevariablen)

[Folie: Reaching Definitions Analyse und Konstantenfaltung]

RD-Analyse
Methode 2-stufige Berechnung

1. Bestimme für jeden Basisblock von $\pi \in IC$ die Länge der am Blockanfang konstanten Variablen mit ihrem Wert (globale Analyse)

2. Bestimme aus dieser Information die entsprechenden Informationen für die Anweisungen der Basisblöcke

Zu (1): Konstruktion eines DFA-Systems $\Delta = \langle G, \mathfrak{I} \rangle$
Konstruktion von $\mathfrak{I} = \langle \mathcal{D}, d_s, \varphi \rangle$
$v_\pi = \{v_1, \ldots, v_k\}$
Bsp: $\{\underbrace{u}_{1}, \underbrace{u}_{1}, \underbrace{v}_{2}, \underbrace{x}_{3}, \underbrace{y}_{4}, \underbrace{z}_{5}\}$
$D = (a \cup \{\bot, \top\}) \ldots A$
$d_j = a$ bedeutet: v_j hat den Wert d_j
$d_j = \top$ bedeutet v_j hat mehrere Werte
$d_j = \bot$ bedeutet ? (irgendein Wert, unbekannt)

Beispiel 4.6.3:
$D = (\mathbb{Z}, \cup\{\bot, \top\})^5$ $(\bot, 4, 5, 6, 7) \sqcup (1, 4, 2, 1, 7) = (1, 4, \top, \top, 7)$

$d_s = (\underbrace{\top, \top, \ldots, \top}_{Eingabevar.}, \underbrace{\bot, \ldots, \bot}_{brigeVar.})$ Startinformation

Konstruktion der monotonen Blocktransformation $\varphi_i : \widehat{A}^k \to \widehat{A}$
Zunächst: Anweisungstransformation $\varphi_\alpha : \widehat{A}^k \to \widehat{A}^k$
Dazu: erweitere Ausdrucksemantik (vgl. SLC)
$e \in V \cup C \cup \{f(u_1, \ldots, u_r) | f \in \Sigma^{(r)}, u_i \in V \cup C\}$
$d \in \widehat{A}^k$
$[e]_\mathfrak{a}(d) \in \widehat{A}$ wobei
$f_\mathfrak{a}(\ldots, \bot, \ldots) = \bot$ z.B. $3 + \bot = \bot$
$f_\mathfrak{a}(\ldots, \top, \ldots) = \top$ z.B. $4 * \top = \top$
$f_\mathfrak{a}(\ldots, \top, \ldots, \top, \ldots) = \top$
$\varphi_{x \leftarrow e}(d_1, \ldots, d_k) = (d`_1, \ldots, d`_k)$

$$d'_j = \begin{cases} [e]_\mathfrak{a}(d_1, \ldots, d_k), & \text{falls } x = v_j \\ d_j, & \text{sonst} \end{cases}$$

Blocktransformation: Komposition der Anweisungstransformationen

Beispiel 4.6.4:

$V_\pi = \{\underbrace{u}_{1}, \underbrace{u}_{1}, \underbrace{v}_{2}, \underbrace{x}_{3}, \underbrace{y}_{4}, \underbrace{z}_{5}\}$
$\varphi_1(d_1, \ldots d_5) = (d_1, \ldots, d_5)$
$\varphi_2(d_1, \ldots d_5) = (d_1, d_2, 1, 1, 1)$
$\varphi_3(d_1, \ldots d_5) = (d_1, \ldots, d_5)$
$\varphi_4(d_1, \ldots d_5) = (d_3 + d_4 + d_5, d_2 \ldots d_5)$
$\varphi_5(d_1, \ldots d_5) = (d_1, d_2, d_4 + 2, d_4, d_5)$
$\varphi_6(d_1, \ldots d_5) = (d_1, \ldots, d_5)$
φ_i monoton, weil alle erweiterten Grundfunktionen $\widehat{f_{\mathfrak{a}}}$ monoton

Bestimmung der <u>MFP-Lösung</u> von $\Delta = \langle G, \mathfrak{I} \rangle$
Datenflussgleichungen:

Beispiel 4.6.5:

$$(E_\Delta) \begin{cases} X_1 = d_1 \\ X_2 = \varphi_1(X_1) \\ X_3 = \varphi_2(X_2) \sqcup \varphi_6(X_6) \\ X_4 = \varphi_3(X_3) \\ X_5 = \varphi_4(X_4) \\ X_6 = \varphi_4(X_4) \sqcup \varphi_5(X_5) \end{cases}$$

$$\begin{aligned} X_1 &= (\top, \top, \top, \top, \top) \\ X_2 &= (\top, \top, \top, \top, \top) \\ (*) X_3 &= (\top, \top, 1, 1, 1) \sqcup \varphi_6([\varphi_4 \sqcup \varphi_5 \circ \varphi_4](X_3) \\ X_4 &= X_3 \\ X_5 &= \varpi_4(X_3) \\ X_6 &= [\varphi_4 \sqcup \varphi_5 \circ \varphi_4](X_3) \end{aligned}$$

$[\varphi_5 \circ \varphi_4](d_1, \ldots, d_5) = (d_3 + d_4 + d_5, d_2, d_4 + 2, d_4, d_5)$
$\varphi_4 \sqcup [\varphi_5 \circ \varphi_4](d_1, \ldots, d_5) = (d_3 + d_4 + d_5, d_2, d_3 \sqcup (d_4 + 2), d_4, d_5)$
$\varphi_6(\varphi_4 \sqcup [\varphi_5 \circ \varphi_4])(d_1, \ldots, d_5) = (d_3 + d_4 + d_5, (d_3 \sqcup (d_4 + 2)) + d_4, , d_3 \sqcup (d_4 + 2), d_4, d_5)$

[08.12.2004]

[Folie: Reaching Definitions Analyse und Konstantenfaltung]

[Folie: RD^{MFP}: Transferfunktionen und Gleichungssystem]

Fp-Iteration für Gleichung (*)
$T_{(*)}(\bot, \ldots, \bot) = (\bot, \ldots, \bot)$
$T^1_{(*)}(\bot, \ldots, \bot) = (\top, \top, 1, 1, 1)$
$T^2_{(*)}(\bot, \ldots, \bot) = (\top, \top, \top, 1, 1)$
$T^3_{(*)}(\bot, \ldots, \bot) = (\top, \top, \top, 1, 1)$
Also: $\underline{\mathrm{fix}}(T_{(*)}) = (\top, \top, \top, 1, 1)$
MFP-Lösung von Δ
$RD_1^{MFP} = (\top, \top, \ldots, \top)$
$RD_2^{MFP} = (\top, \top, \ldots, \top)$
$RD_3^{MFP} = (\top, \top, \top, 1, 1)$
$RD_4^{MFP} = (\top, \top, \top, 1, 1)$
$RD_5^{MFP} = (\top, \top, \top, 1, 1)$
$RD_6^{MFP} = (\top, \top, \top, 1, 1)$

Die Variablen y und z haben bei Eintritt in die Blöcke 3, 4, 5, 6 den konstanten Wert 1.
Lokale Analyse
SLC-Analyse von Block i mit der Startinformation $RD_i^{MFP} (i = 1, \ldots, k)$ hier $k = 6$
Optimierung
Konstantenfaltung wie bei SLC
Frage $RD^{MOD} = RD^{MFP}$?
Knotentransformationen φ_i distributiv?

Satz 4.6.6:
Es gibt ein $\pi \in IC$ mit $RD^{MOP} \neq RD^{MFP}$

[Folie: $RD^{MOP} \neq RD^{MFP}$]

Beweis:
$V_\pi = \{x, y, z\} \quad D = \widehat{\mathbb{Z}}^3$
$\varphi_1(d_1, d_2, d_3) = (d_1, d_2, 0)$
$\varphi_2(d_1, d_2, d_3) = (2, 3, d_3)$
$\varphi_3(d_1, d_2, d_3) = (3, 2, d_2)$
$\varphi_4(d_1, d_2, d_3) = (d_1, d_2, d_1 + d_2)$
$RD_5^{MOP} = (2, 3, 5) \sqcup (3, 2, 5) = (\top, \top, 5)$

$$\begin{aligned} RD^{MFP} X_1 &= (\top, \top, \top) \\ X_2 &= \varphi_1(X_1) = (\top, \top, 0) \\ X_3 &= \varphi_1(X_1) = (\top, \top, 0) \\ X_4 &= \varphi_2(X_2) \sqcup \varphi_3(X_3) = (2, 3, 0) \sqcup (3, 2, 0) = (\top, \top, 0) \\ X_5 &= \varphi_4(X_4) = (\top, \top, \top) \end{aligned}$$

Also: $RD_5^{MFP} = (\top, \top, \top) \overset{>}{\neq} RD_5^{MOP}$
$\square$
Grund: φ_4 nicht distributiv
$\varphi_4(d \sqcup d') \neq \varphi_4(d) \sqcup \varphi_4(d')$ für $d = (2, 3, z) \quad d' = (3, 2, z)$
Insgesamt gilt:
falls Block 2 bei $z = 0$ Sprungziel $RD_5 = (2, 3, 5) < RD_5^{MOP} < RD_5^{MFP}$

[Folie: Nicht-Entscheidbarkeit der Konstanteninformationen Rl]

Satz 4.6.7:
Die Pfaddefinierten Konstanteninformationen $RD_i^{MOP} (i = 1, \ldots, k)$ sind i.a. nicht entscheidbar.
Beweis:
Wären die Mengen RD_i^{MOP} immer entscheidbar, so auch das MPCP ("modifiziertes Postsches Korrespondenzproblem") und damit das PCP.
MPCP: sei Σ Alphabet, $n \in \mathbb{N}$ und $u_1, v_1, \ldots, u_n, v_n \in \Sigma^*$
Gibt es $i_1, \ldots, i_m \in \{1, \ldots, n\}$, $m \geq 1$ mit $i_1 = 1$ ("modifiziertes PCP"), so dass $u_{i_1} u_{i_2} u_{i_3} \ldots v_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_m}$
Es gilt (z.B. Schöning, Theoretische Informatik): Das MPCP ist nicht entscheidbar

[Folie: Nicht-Entscheidbarkeit der Knonstanteninformationen RL]

mit der Interpretation $\langle A; \varphi \rangle$

$$A := \Sigma^* \quad \varphi(++)(u,v) := uv$$
$$\varphi(=)(u,v) := \begin{cases} \mathcal{E} \text{ falls } u \neq v \\ a \text{ falls } u = v \quad (a \in \Sigma) \end{cases}$$

Dann folgt: Jeder Pfad von Startknoten nach k entspricht einer Indexfolge $i_1, \ldots, i_m$ mit $i_1 = 1$ und umgekehrt
Also: $RD_k^{MOP}(Z) = \mathcal{E} \curvearrowleft\curvearrowright MPCP(u_1, v_1, \ldots u_n, v_n)$ hat keine Lösung.
Somit: Entscheidbarkeit der $RD_i^{MOP} \curvearrowright$ Entscheidbarkeit von $MPCP(u_1, v_1, \ldots u_n, v_n)$
Widerspruch $\lightning$
$\square$

Zusammenfassung:

1. $RD \leq RD^{MOP} \leq RD^{MFP}$
2. $RD < RD^{MOP} < RD^{MFP}$
3. RD und RD^{MOP} i.a. nicht entscheidbar

4.7 Effiziente Fp-Berechnung

Δ DFA-System $T_\Delta : D^n \to D^n$
MFP-Lösung von Δ
$AI^{MFP} = \underline{\text{fix}}(T_\Delta) \in D^n$
T_Δ stetig $\curvearrowright \underline{\text{fix}}(T_\Delta) = \bigsqcup_{i=0}^{\infty} T_\Delta^i(\perp)$
$\mathcal{D} = \langle D; \leq \rangle$ mit acc $\curvearrowright \underline{\text{fix}}(T_\Delta) = \bigsqcup_{i=0}^{m} T_\Delta^i(\perp) = T_\Delta^m(\perp)$ [17.12.2004]
Statt gleichmässiger Fp-Iteration komponentenweise Iteration

Definition 4.7.1:
Eine Folge $I = (i_1, i_2, \ldots,)$ mit $i_j \in \{1, \ldots, n\}$ heisst fair, wenn für jedes $j \in \mathbb{N}$ und jedes $i \in \{1, \ldots, n\}$ ein $m \in \mathbb{N}$ existiert mit $i = i_{j+m}$.
Mit anderen Worten: In einer fairen folge kommt jeder Index unendlich oft vor.

Beispiel 4.7.2:
$(1, \ldots, n, 1, \ldots, n, \ldots)$

Ein Index i bestimmt für $T_\Delta : D^n \to D^n$ die Komponentenfunktion $g_i : D^n \to D^n$
mit $g_i(d_1, \ldots, d_n) = (d`_1, \ldots, d`_n)$
wobei $d`_i := \underline{\text{proj}_i}(\Delta(d_1, \ldots, d_n))$
und $d`_j := d_j$ für $j \neq i$(Berechnung durch i-te Gleichung)

Satz 4.7.3: Chaotische Fp-Iteration
Sei $\mathcal{D} = \langle D, \leq \rangle$ein vollständiger Verband und $T : D^n \to D^n$ stetig
Dann gilt für eine faire Indexfolge $I = (i_1, i_2, \ldots)$ mit $i_j \in \{1, \ldots, n\}$
$\underline{\text{fix}}(T) = \bigsqcup \{ g_{i_j}(g_{i_j-1}, \ldots (g_{i_1}(\perp)) \ldots) | j \in \mathbb{N} \}$

Frage: Ist diese Folge monoton wachsend?
Beweis des Fp-Satzes: $\perp \leq T(\perp) \leq T^2(\perp) \leq \dots$
kann hier nicht angewandt werden: $\perp \leq g_{i_1}(\perp)?g_{i_2}(g_{i_1}(\perp))$
Beweisskizze:
$g_{i_j}(\dots g_{i_1}(\perp)\dots) \leq T^j(\perp)$
$\curvearrowright \bigsqcup\{g_{i_j}(\dots g_{i_1}(\perp)\dots) | j \in \mathbb{N}\} \leq \bigsqcup\{T^j(\perp) | j \in \mathbb{N}\} = \underline{\text{fix}}(T)$
Wegen Fairness: Zu jedem $j \in \mathbb{N}$ existiert k_j, so dass in $(i_1, \dots, i_{k_j})$ jeder Index mindestens j-mal vorkommt.
$\curvearrowright T^j(\perp) \leq g_{i_{k_j}}(\dots(g_{i_1}(\perp)\dots)$
$\underline{\text{fix}}(T) = \bigsqcup\{T^j(\perp) | j \in \mathbb{N}\} \leq \bigsqcup g_{i_{k_j}}(\dots g_{i_1}(\perp)\dots)$
$\square$

Folgerung 4.7.3:

Besitzt $\mathcal{D}$ die arc-Eigenschaft, dann existiert eine endliche Indexfolge $i_1, \dots, i_k$, so dass $\underline{\text{fix}}(T) = g_{i_k}(\dots g_{i_1}(\perp)\dots)$

Problem: Bestimmung einer möglichst kurzen Indexfolge
Worklist-Algorithmus
Komponentenfunktionen $g_i : D^n \to D^n$ weiter zerlegen: $g_i(d_1, \dots, d_n) = (d`_1, \dots, d`_n)$
$d`_i = \underline{\text{proj}_i}(T_\Delta(d_1, \dots, d_k)) = \bigsqcup\{\varphi_j(d_j) | j \in \underline{\text{Var}}(i)\}$
$d`_k = d_k$ für $k \neq i$
Rechenschritte: $\varphi_j(d_j)$ durch Flussregeln steuern
Eingabe: $\Delta = \langle G, \mathfrak{I}\rangle$ mit $G = \langle \underline{\text{Kno}}, \underline{\text{Kan}}, s\rangle$ und $\mathfrak{I} = \langle \mathcal{D}, \varphi, d_s\rangle$ $\underline{\text{Kno}} = \{s = 1, 2, \dots, n\}$
Verfahren:

```
var k, k‘: Kno;
var AI: array[1 .. n] of D
var W: list of Kan %Worklist für Rechenschritt
Start AI[1] := d_s
AI[j] := ⊥_D für j = 2, ..., n;
W := nil;
for all (j, l) ∈ Kan do
   W := cons((j, l), W);
end for
Iteration %φ_j als Rechenschritt
while W ≠ nil do
   k := fst(head(W));
   k' := snd(head(W));
   W := tail(W);
   if φ_k(AI[k]) ⋢ AI[k‘] then
      AI[k‘] := AI[k‘] ⊔ φ_k(AI[k]);
      for all k“ with (k‘, k“) ∈ Kon, but not(k‘, k“) ∈ W  do
         W := cons((k‘, k“), W);
      end for
   end if
```

end while
Ausgabe: AI = (AI[i], . . . , AI[n])

Beispiel 4.7.4: AE-Analyse
$\underline{\text{OpExp}}_\pi = \{\underbrace{x+y}_1, \underbrace{x*z}_2, \underbrace{t*t}_3, \underbrace{z*2}_4, \underbrace{i+1}_5\}$
$D = \mathbb{B}^5$
$W = (1,2)(2,3)(3,4)(3,5)(4,6)(5,6)(6,3)$

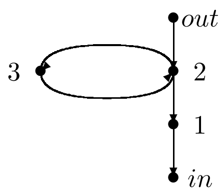
W	RD[1]	RD[2]	RD[3]	RD[4]	RD[5]	RD[6]
(1, 2) . . .	(1, . . . , 1)	(0, . . . , 0)	(0, . . . , 0)	(0, . . . , 0)	(0, . . . , 0)	(0, . . . , 0)
(2, 3) . . .		(1, . . . , 1)	(0, 0, 0, 1, 1)	(0, 1, 0, 1, 1)	(0, 1, 0, 1, 1)	(0, 0, 0, 1, 1)
(3, 4) . . .			(0, 1, 0, 1, 1)	(0, 1, 0, 1, 1)	(0, 1, 0, 1, 1)	(0, 1, 0, 1, 1)
(3, 5) . . .						
(4, 6) . . .						
(5, 6) . . .						
(1, 2) . . .						
(1, 2) . . .						

[22.12.2004]

4.8 Live Variable Analyse und Dead Code Elimination für IC-Programm

Definition 4.8.1:
Sei $\pi \in IC$, $x \in V_\pi$, $i \in L_\pi$.
Dann heisst x lebendig in i, falls $p \in$ Pfad$[i, q+1]$ existiert, so dass x auf p benutzt wird (rechte Seite einer Zuweisung oder Test oder Ausgabevariable), ohne vorher neu definiert zu werden.

Aufgabe: Bestimme für jeden Programmpunkt $i \in L_\pi$ die Menge LV_i der lebendigen Variablen.
Hier: Verbesserung (s. SLC): Bestimmung der NV_i (needed Variables)
Beachte Wertzuwesiungen mit toter linker Seite bilden Dead Code. [Folie: Dead Code Elimination]
Konstruktion eines DFA-Systems $\Delta = \langle G, \mathfrak{I} \rangle$
Rückwärtsanalyse:
Datenflussgraph G (“Rückwärts-BB-Graph“)



$\mathcal{D} = \langle P(V_\pi); \subseteq \rangle$
Blocktransformationen:
$\varphi_i : P(V_\pi) \to P(V_\pi)$

in Block i: $\alpha_1; \ldots; \alpha_k$
$\varphi_i := \varphi_{\alpha_1} \circ \ldots \circ \varphi_{\alpha_k}$
wegen Rückwärtsanalyse.
Anweisungstransformation $t_\alpha : P(V_\pi) \rightarrow P(V_\pi)$

- $\alpha = v \leftarrow e$

  $t_{v\leftarrow e}(M) :=$ <u>if</u> $v \in M$ <u>then</u> $M/\{v\} \cup V_e$ <u>else</u> M

- $\alpha =$ <u>if</u> be <u>goto</u> l

  $t_\alpha(M) := M \cup V_{be}$

- $\alpha =$ <u>goto</u> l

  $t_\alpha(M) = M$

Zusätzlich für Ausgabeknoten:
$\varphi_{out}(M) := ov_\pi$
Startinformation: $d_s = \emptyset$

$$
\begin{aligned}
E_\Delta X_{out} &= d_s \\
X_2 &= \varphi_(out)(X_{out}) \sqcup \varphi_3(X_3) \\
X_3 &= \varphi_2(X_2) \\
X_1 &= \varphi_2(X_2) \\
X_{in} &= \varphi_1(X_1)
\end{aligned}
$$

$$
\begin{aligned}
\varphi_1(M) &= M - w \\
&\quad - u + y \qquad \text{falls } u \in M \\
&\quad + x \qquad \text{falls } v \in M \\
&\quad - v + (x, y) \qquad \text{falls } v \in M \\
&= M - w \\
&\quad - u + y \qquad \text{falls } u \in M \\
&\quad - v + (x, y) \qquad \text{falls } v \in M \\
\varphi_2(M) &= M - z + (u, v) \\
\varphi_3(n) &= M - z + (u, v) \qquad \text{falls } z \in M \\
&\quad - v + w \qquad \text{falls } v \in M
\end{aligned}
$$

Berechnung von NV^{MFP} nach Worklist-Algorithmus

W	NV_{out}	NV_1	NV_2	NV_3	NV_{in}
(out, 2), (2, 3), (3, 2), (2, 1), (1, in)	$d_s = \emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
(2, 3), (3, 2), (2, 1), (1, in)			z		
(3, 2), (2, 1), (1, in)				u, v	
(2, 1), (1, in)			u, w, z		
(3, 2), (2, 1), (1, in)				u, v, v	
(3, 2), (2, 1), (1, in)					
(2, 1), (1, in)		u, v, w			
(1, in)					x, y

Ergebnis der globalen NV-Analyse
$NV_1^{MFP} = \{u, v, w\}$
$NV_2^{MFP} = \{u, w, z\}$
$NV_3^{MFP} = \{u, v, w\}$
$NV_4^{MFP} = \{x, y\}$
φ_i distributiv: $\varphi(M \cup M') = \varphi_i(M) \cup \varphi_i(M')$
$\curvearrowright NV^{MOP} = NV^{MFP} \curvearrowright$ Dead Code Optimierung (vgl. SLC) [12.01.2005]

Kapitel 5

Kontrollflussanalyse und Schleifenoptimierung

Sei $\pi \in IC$
Ausnutzung der Schleifenstruktur des Flussgraphen.
G_π (SI / vorwärts)

- schnellere Fp-Berechnung (Schleifen)
- Schleifenoptimierung (grosser Einfluss)
  a) Code Motion (verschieben von schleifeninvarianten Anweisungen vor die Schleife)
  b) Elimination von Induktionsvariablen (gleiche Wertprogression von mehreren Variablen nur auf einer Variablen durchführen)

5.1 Schleifenanalyse von Flussgraphen

Flussgraphen eines strukturierten Quellprograms
Baum mit Rücksprüngen:

[Folie: Beispiele für Flussgraphen]

- while-Schleifen: 1 Eingang = Ausgang
- repeat-exit i-Schleifen: 1 Eingang - mehrere Ausgänge (Java: break, continue Anweisungen)

Flussgraph G_1
Schleifen: {1, 2, 3}, {5, 7}, (aber nicht: {1, 5, 7, 6, 8}), {1, 5, 7, 6, 8}, {1, 2, 3, 5, 6, 7, 8}
Beachte: Jede Schleife hat genau einen Anfangsknoten, welcher vom Startknoten ausserhalb der Schleife erreichbar ist. hier: 1 bzw. 5.
Flussgraph G_2

Schleifen: keine, weil {2, 3} 2 Anfangsknoten besitzt

Definition 5.1.1:
Sei $G = \langle \underline{Kno}, \underline{Kan}, s\rangle$ ein Flussgraph ($\underline{\text{Vor}}$(s) = $\emptyset$, und $\underline{\text{Pfad}}$[s, k] $\neq \emptyset$ für alle $k \in$ $\underline{\text{Kno}}$) falls gilt:

1. S ist stark zusammenhängend, d.h. für alle $k, k' \in S$ mit $k \neq k'$ existiert $p \in \underline{\text{Kno}}(p) \subseteq S$, und

2. Es existiert genau ein $k \in S$, so dass $\underline{\text{Vor}}(k) \backslash S \neq \emptyset$. k heisst Anfang von S.

Beachte: Es muss $p \in \underline{\text{Pfad}}[s, k)$ existieren mit $\underline{\text{Kno}}(p) \cap S = \emptyset$
Aufgabe: Bestimmung der Schleifen von Flussgraphen
Hilfsmittel: Dominatorrelation auf Knoten

Definition 5.1.2:
$G = \langle \underline{Kno}, \underline{Kan}, s\rangle$ ein Flussgraph und $k, k' \in \underline{\text{Kno}}$.

- k' dominiert k (Bezeichnung: k' dom k) $:\curvearrowleft\curvearrowright$ für jedes $p \in \underline{\text{Pfad}}[s, k]$ gilt: $k' \in \underline{\text{Kno}}(p)$

- k' dominiert k direkt $\curvearrowleft\curvearrowright$ k' dom k, $k' \neq k$, für alle k" gilt: k" dom k, $k'' \neq k \curvearrowright$ k" dom k'

Beispiel 5.1.3:
Flussgraph G_3

- s dom k, k dom k für alle $k \in \underline{\text{Kno}}$

- 2 dom k $\curvearrowright$

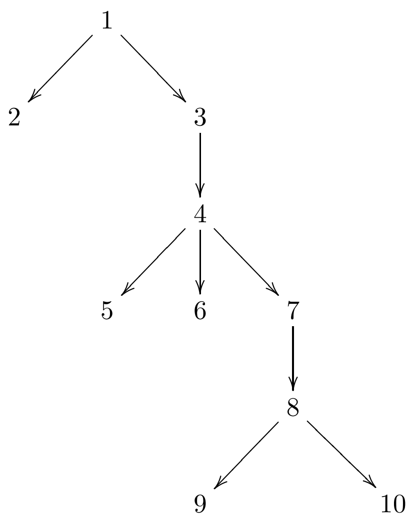
- Dominatoren von 8: (s), 1, 3, 4, 7, (8)

- 4 dom 7 direkt

Folgerung 5.1.3:
Jeder Knoten k, $k \neq s$ besitzt genau 1 direkten Dominator.
Grund: Die Dominatoren von k sind linear geordnet

Folgerung 5.1.3:
Baumdarstellung der Dominatorenrelation:

Beispiel 5.1.4:
Dominatorbaum von G_3:



Definition 5.1.5:
$(b, a) \in$ Kan heisst Rückwärtskante, falls a dom b

Lemma 5.1.6:
Sei (b, a) eine Rückwärtskante und $S := \{k \in \text{Kno} \mid \text{es existiert } p \in \text{Pfad}[k, b] \text{ mit } a \notin \text{Kno}(p)\}$
Dann ist $S \cup \{a\}$ eine Schleife mit Anfang a.
Beweis: [Grafik] Sei $k \in S$ und $p \in \text{Pfad}[a, b]$ mit $a \neq \text{Kno}(p)$
Sei $p' \in \text{Pfad}[s, k]$. Also $p'p \in \text{Pfad}[s, b]$. a dom b $\curvearrowright a \in \text{Kno}(p'p) \curvearrowright a \in \text{Kno}(p')$
$\curvearrowright$ es existiert $p'' \in \text{Pfad}(a, k]$ mit $a \notin \text{Kno}(p'')$ so dass $\text{Kno}(p'') \subseteq S$
Es folgt:

1. $S \cup \{a\}$ist stark zusammenhängend: $k, k' \in S \cup \{a\}$

   $k \to a \to k'$

2. a ist Anfang von $S \cup \{a\}$: Da $\text{Var}(s) = \emptyset$, so muss $a \neq a$

   Sei $p \in \text{Pfad}[s, a)$ mit $a \notin \text{Kno}(p)$. Da a dom k für alle $k \in S$, so folgt $\text{Kno}(p) \cap S = \emptyset$ und auch die Eindeutigkeit von a, weil S unter Vorgängerabschluss abgeschlossen.

□

[14.01.2005]

Bemmerkung $S \cup \{a\}$ heisst natürliche Schleife von (b, a)

[Folie: Beispiele für Flussgraphen]

Beispiel 5.1.7:
G_3 Die natürliche Schleife um (4,3) ist $\{3, 4, 5, 6, 7, 8, 10\}$

Aufgabe: Bestimmung der Rückwärtskanten und ihrer natürlichen Schleifen

Idee: dom-Relation $\curvearrowright$ Rückwärtskanten
Vorgängerrelation $\curvearrowright$ natürliche Schleife
Flussgraphen von strukturienten Programmen besitzen nur natürliche Schleifen

Definition 5.1.8: Reduzierbarer Flussgraph
Ein Flussgraph $G = \langle \underline{Kno}, \underline{Kan}, s\rangle$ heisst reduzierbar, falls $G' = \langle \underline{Kno}, \underline{Kan}, s\rangle$ mit Kan' := Kan$\backslash\{(a, b)|(a, b)$ ist Rückwärtskante$\}$ azyklisch ist.

Beispiel 5.1.9:
G_1 und G_3 sind reduzierbar, G_2 nicht.

5.2 ud- und du-chains

Weitere Programminformation zur Bestimmung schleifeninvarianter Berechnungen

- use-definition-chain (ud-chain): Menge der Definitionen für die Benutzung einer Variablen
- definition-use-chain (du-chain): Menge der Benutzungen für die Definition einer Variablen

Definition 5.2.1:
Sei $\pi \in IC$, $v \in V_\pi$, $l, l_1, l_2 \in L_\pi$ und $p \in \underline{\text{Pfad}}_\pi(l_1, l_2)$

- v wird in l definiert: (def(v, l)): $\curvearrowright$ $\alpha_l = l : v \leftarrow e$
- v wird in l benutzt: (use(v, l)): $\curvearrowright$ entweder $\alpha_e = l : v' \leftarrow e$ mit $v \in V_e$ oder $\alpha_e = l$: if be ... mit $v \in V_{be}$
- v wird auf p definiert: $\curvearrowleft\curvearrowright$ def(v, l) für ein $l \in \underline{\text{Kno}}(p)$
- clear(v, p): $\curvearrowright$ v wird nicht auf p definiert
- ud(v, l) := $\{l'|use(v, l), def(v, l'),$ es existiert $p \in \underline{\text{Pfad}}(l', l) : clear(v, p)\}$
- du(v, l) := $\{l'|l \in ud(v, l')\}$

Berechnung dieser Information durch Variante der Reading-Definition-Analyse (eigentlich RD-Analyse) DFA-System $\Delta = \langle G, \mathfrak{I}\rangle$ zu $\pi \in IC$.
G SI/vorwärts-Graph von π
$\mathfrak{I} = \langle \mathcal{D}, \varphi, d_s\rangle$
Analyseinformation für $l \in \underline{\text{Kno}} = l_\pi : RD_l = \{(v, l')|def(v, l')$ und $\exists p \in \underline{\text{Pfad}}(l', l)$ mit $clear(v, p)\}$
Es folgt: $l' \in ud(v, l) \curvearrowleft\curvearrowright (v, l') \in RD_l$ und $use(v, l)$
Abstrakte Interpretation $\mathfrak{I}$:

$D := P(V_\pi \times (L_\pi \cup \{0\}))$ 0 ist Definitionsmarke für $v \in IV$
$d'_s = \{(v,0)|v \in IV\}$
$\varphi_\alpha : D \to D$

- $\alpha = i : v \leftarrow e$

$\varphi_\alpha(M) := M \backslash \{(v,l)|l \in L_\pi \cup \{0\}\} \cup \{(v,i)\}$

- andere α

$\varphi_\alpha = id_D$
φ_α distributiv, so dass $RD_l^{MFP} = RD_l^{MOP}$

[Folie: Beispiel: RD-Analyse und ud-chains]

5.3 Schleifeninvariante Berechnungen und Code Motion

$\pi \in IC$ mit SI/vorwärts-Graph $G_\pi = \langle \underline{Kno}, \underline{Kan}, s\rangle$ Schleife $S \subseteq$ Kno mit Anfang $k \in S$
ud-chains für $(v,l) \in V_\pi \times L_\pi$

a) schleifeninvariante Berechnungen

   Idee: Rechenausdrücke mit festem Wert in S vor Schleifeneintritt berechnen

   Beachte: mögliche Propagation fester Werte innerhalb S

   Definition 5.3.1:

   $l \in S$ heisst s-invariant, wenn in $\alpha_l : x \leftarrow e$ oder $\alpha_l = l :$ if $be \ldots$ Schleifenberechnung stets den selben Wert haben.

   Bestimmung S-invarianter Knoten

   Es gilt: $l \in S$ ist S-invariant, falls $\alpha_l = l : x \leftarrow e$ oder $\alpha_l = l :$ if be goto l' und für alle $y \in V_l$ bzw. $y \in V_{be}$ gilt:

   - $ud(y,l) \cap S = \emptyset$
   - $ud(y,l) = \{l'\}$ und l' ist S-invariant

   [19.01.2005]

   [Folie: Beispiel: Illegale Codeverschiebung (Bedingung 1)]

   Algorithmus (Analyse): induktive Markierung S-invarianter Knoten.

   Algorithmus (Optimierung): Berechnung der Ausdrücke S-invarianter Knoten vor Schleifeneintritt auf temporären Variablen (SLC)

b) Code Motion

   Verschiebung einer Wertzuweisung vor den Schleifenanfang.

   Sei $l \in S$ S-invariant und $\alpha_e = l : x \leftarrow e$

   Dann gilt: α_l kann direkt vor Schleifenanfang ausgeführt werden, falls :

[Folie: Bedingungen 2 und 3 für Codeverscheibung]

(a) Für jeden Ausgangsknotem $k' \in S$, d.h. es existiert $k'' \notin S$ mit $(k', k``) \in \underline{Kan}$ ("Ein Ausgangsknoten ist ein Knoten, der eine Stelle erreicht, die nicht in der Schleife liegt"), gilt:
$ldomk'$

(b) Es gibt nur 1 Definition von x in S

(c) Wird x in $l' \in S$ benutzt, so folgt: $ud(x, l') = \{l\}$
Analysealgorithmus:

i. S-invariante Knoten
ii. Eigenschaften 1-3 mit ud- und dom-Information prüfen

Opt: Ja

5.4 Induktionsvariablen

Induktionsvariablen: Variablen einer Schleife mit knónstanter Wertprogression

- $i \leftarrow i + 1$ direkte Induktion
- $j \leftarrow 3 * i$ indirekte Induktion

2 Optimierungen:

a) Strengthreduktion (Addition statt Multiplikation)
Bsp:$j \leftarrow j + 3$

b) Elimination von Induktionsvariablen

Entstehung von Induktionsvariablen: z.B. Zählschleifen

Definition 5.4.1: Induktionsvariablen
Sei $\pi \in IC(\mathbb{Z}, +, *, <, >)$ und S eine Schleife im (SI/Vorwärts)-Flussgraphen von π

1. $x \leftarrow V_s$ heisst Basisinduktionsvariable ($x \in BN(S)$), falls gilt:
   - es existiert $i \in S$ mit $\alpha_i = i : x \leftarrow e$
   - für alle $i \in S$ mit $\alpha_i 0 i : x \leftarrow e$ gilt: $e \in \{x + c, c + x | c \in \mathbb{Z}\}$
2. $y \in V_s$ heisst abhängige Induktionsvariable ($y \in AIV(S)$) falls gilt:
   - es existiert $i \in S$ mit $\alpha_i = i : y \leftarrow e$
   - für alle $i \in S$ mit $\alpha_i = i : y \leftarrow e$ gilt:
     $e \in \{x * x, x * c, c + x, x + c | c \in \mathbb{Z}, x \in BIV(S)\}$

Bemerkung: Vereallgemeinerung durch verschachtelte Abhängigkeiten

Folgerung 5.4.1:
Für eine Schleifenberechnung gilt: konstante Wertprogression von $x \in BN(S)$ überträgt sich mit einem Faktor auf die von x abhängigen $y \in AN(S)$

[Folie: Strength reduction für Induktionsvariablen]

Beachte: verschiedene Definitionen mit unterschiedlichen Progressionen

a) "Strength reduction" für Induktionsvariablen

Idee: Ersetze Multiplikationen durch Additionen (Schleife!)

Sei $y \in AIV(S)$ und bei $i \in S$ eine Definiton von y:

$\alpha_i = i : y \leftarrow e$ mit $V_e = \{x\}$, $x \in BN(S)$

Ist e ein Produkt, so ist dies wie folgt vermeidbar:

Wähle neue Variable t_i und füge hinter jeder Definition von x eine Definition von t_i ein. Diese ist bestimmt durch e und die Definition von x.

Falls $e = d * x$ und $x \leftarrow x + c$, so ist $t_i \leftarrow t_i + d * c$

Initialisierung von t_i vor Schleifenanfang: $t_i \leftarrow d * x$

b) Elimination von Induktionsvariablen

Sei $y \in AIV(S)$ mit genau einer Definition. $i : y \leftarrow d * x$, also $x \in PN(S)$

x werde nur benutzt in ihren Definitionen, in der Definition von y und in Bedingungen der Form if *be* goto ...

Dann kann x eliminiert werden:

(a) Transformation durch Strength reduction

(b) Simulation von x in be durch t_i

[Folie: Beispiel: Elimination von Induktionsvariablen]

[21.01.2005]

[Folie: ICP-Beispielprogramm: Fibonaccifunktion]

Kapitel 6

Interprozedurale Analyse

bisher: intra-prozedural (π als Prozedur-rumpf)
jetzt: Erweiterung von IC um rekursive Prozeduren
Probleme: Abhängigkeit zwischen Aufruf und Rücksprung, Parameterbehandlung
hier: keine geschachtelten Prozedurdeklarationen, nur Wert- und Ergebnisparameter

6.1 Syntax von ICP (Intermediate Code with Procedures

$\pi = (Plist, Vlist, Alist) \in ICP$
$Plist = Pdecl_1, \ldots, Pdecl_r$ Liste von Prozedurdeklarationen
$Pdecl_i = (pid_i, Vlist_i, Alist_i)$, $i = 1, \ldots, r$
pid_i: Prozedurbezeichner
$Vlist_i$; $\underline{\text{in}}$ $x_1, \ldots, x_{n_i}$; $\underline{\text{out}}$ $y_1, \ldots y_m$; $\underline{\text{loc}}$ $z_1, \ldots, z_{p_i}$ Liste der formalen Parameter
$Alist_i$: Anweisungsliste
$\underline{Anw} := \underline{Anw}(IC) \cup \{pid_i(e_1, \ldots, e_{n_i}, v_1, \ldots, v_{m_i}) | e_j$einfache Rechenausdrücke als aktuelle Parameter, $v_k \in V$ (paarweise verschieden), $i = 1, \ldots, r\}$
Prozeduraufrufe
in $Alist_0$ und $Alist_1, \ldots, Alist_r$
$\rightarrow$ verschränkte Rekursion
Variablen müssen deklariert sein:
$V_{Alist_i} \subseteq V_{Vlist_i}$ $(i = 0, \ldots, r)$

6.2 Semantik in ICP

Eingabe als Wertparameter (call-by-value)
Jeder Prozeduraufruf mit eigenem Zustandsraum.
Werte der formalen Ausgabeparameter in aktuellen Ausgabeparameter übertragen
Laufzeitkeller mit Aufrufframes.

Fp-Semantik als Erweiterung der Fortsetzungssemantik von IC

6.3 MOP-Analyse für ICP-Programme

[Folie: Flussdiagramm von π_{fib}]

Problem 1 Pfade von $\pi \in ICP$

(a) Die reguläre Pfadmenge von G_{Fib} (naiver Ansatz)

$G_{fib} \mapsto G_{fib}^{iter}$ entsteht aus G_{fib} durch Hinzunahme von Aufrufkanten: $(1,3)$, $(4,3)$, $(5,3)$ und Rücksprungkanten $(7,2)$, $(7,5)$, $(7,7)$

Ferner: Weglassen der (auf der Folie gestrichelten) Kanten hinter Prozeduraufrufen $(1,2)$, $(4,5)$, $(5,7)$

$\underline{\text{Pfad}}[1,2] \ni 13672, 136753672$

Vorteil: DFA-Technik ist direkt übertragbar

Nachteil: zu viele Pfade, Bedingungen für Analyse zu scharf

Grund: kenine Bindung zu Rücksprung und Aufruf

(b) Die kontextfreie Pfadmenge von G_{fib}:

$$
\begin{aligned}
G =& \langle N, \Sigma, P, S \rangle \\
\Sigma =& \{1, 2, \ldots, 7\} \\
N =& \{P[1,2], P[2,2], P[3,7], P[4,7], \ldots P[7,7]\} \\
P :& P[1,2] \to 1P[3,7]P[2,2] \\
& P[2,2] \to 2 \\
& P[3,7] \to 3P[4,7] | 3P[6,7] \\
& P[4,7] \to 4P[3,7]P[5,7] \\
& P[5,7] \to 5P[3,7]P[7,7] \\
& P[6,7] \to 6P[7,7] \\
& P[1,2] 136753672
\end{aligned}
$$

$\to \underline{\text{RPFad}}[s,k)$

Problem 2 Knotentransformationen für Aufruf- und Rücksprungknoten

Abstrakte Interpretation von G_π

$\mathcal{D} = \langle D; \leq \rangle$ vollständiger Verband mit ACC

$d_s \in D$ Startinformation (Hauptprogramm)

Knotentransformationen:

(a) Bei Zuweisungs- und Verzweigungsknoten wie bekannt aus IC: $\varphi_{ij} : D \to D$

(b) Bei Stop- und Goto-Knoten $\varphi_{ij} = id_D$

(c) Bei Aufruf und Rücksprung?
Idee: Pfadberechnung mit Kellerspeicher (analog Laufzeitkeller)
Beachte: Rücksprungadressen unnötig, da Pfad vorgegeben
$Stack_D := D^+$ (leerer Keller unnötig) Spitze rechts
$\varphi_{ij} : D \to D$ für Aufrufknoten ij zur Berechnung der Startinformation für den Aufruf aus oberstem Kellerelement
$\varphi_{i(q_i+1)} : D \times D \to D$ für den Rücksprungknoten zur Berechnung der Information beim Rücksprung, bestimmt durch Information von Aufruf und bei Aufrufende (die beiden obersten Kellereinträge)
Jedes φ_{ij} bestimmt eine Stacktransformation $\varphi_{ij} : \underline{Kno} \to \{f | f_S stack_D \to Stack_D\}$

[26.01.2005]

[Folie: Flussgraph von $\pi - fib$]

für Fall (1): $\varphi_{ij}^{st}(wd) := w\varphi_{ij}(d)$
für Fall (2): $\varphi_{ij}^{st}(wd) := wd\varphi_{ij}(d)$ ("PUSH")
für Fall (3): $\varphi_{k(q_k+1)}^{st}(wdd') := w\varphi_{ij}(d)$
Für $k \in$ Kno $\setminus \{s\}$ (s = 01) und $p = (k_1, \ldots, k_q) \in RPfad[s,k)$ definieren wir die Analyseinformation für k bezüglich p
$AI\langle p\rangle := top(\varphi_{kq}(\ldots\varphi_{k_1}(d_1)\ldots)) \in D$ mit $top : Stack_D \to D$ $(wd \mapsto d)$
Die MOP-Lösung für k

- $AI_s^{MOP} := d_s$
- $AI_k^{MOP} := \bigsqcup\{AI\langle p\rangle | p \in$ R Pfad$[s,k)\}$

6.4 MFP-Analyse für ICP-Programme

$\pi = (Plist, Vlist_0, Alist_0) \in ICP$ bestimmt die Flussgraphen $G_0, \ldots, G_r$ und den daraus kombinierten Flussgraphen G_π
Unterschiedliche Kantenstruktur: $\alpha_{ij} = pid_k(\ldots)$:

1. $(ij, i(j+1)) \in \underline{\text{Kan}}(G_i)$
2. $(ij, k1)$, $(k(q_k+1), i(j+1)) \in \underline{\text{Kan}}(G_\pi)$

MFP-Lösung für iterative Flussgraphen:

$$E \begin{cases} AI_s = d_s \\ AI_k = \bigsqcup_{l \in \underline{\text{Var}}(k)} \varphi_l(AI_l) \end{cases}$$

Übertragung auf den rekursiven Fall
Betrachtung von $G_0, \ldots, G_r$ statt G_π
Grund: Gleichungen aus E beschreiben lokale Zusammenhänge
Aufrufknoten ij mit $a_{ij} = pid_k(\ldots)$:
volle Transformationen $\Phi_k : D \to D$ und G_k verwenden
Neben den Gleichungen für die Analyseinformationen zusätzliche Gleichungen für die $\Phi_k (k = 1, \ldots, r)$

Dazu. duale Sicht zur Berechnung der Φ_k
$ij \mapsto$ nicht die von dort bis $i(q_i+1)$, sondern die von i1 bis ij bestimmte Transformation.
[Grafik]

- $i1\ \Phi_{ij}$
- ij
- $i(q_i+1)$

[/Grafik]

$$\begin{tikzcd} \bullet il \arrow[dr] & & \bullet im \arrow[dl] \\ & \bullet ij & \end{tikzcd} \qquad \Phi_{ij}(d) = \tilde{\varphi}_{il}(\Phi_{il}(a)) \sqcup \tilde{\varphi}_{im}(\Phi_{im}(a))$$

$\tilde{\varphi}_{il}$, $\tilde{\varphi}_{im}$ lokale Anweisungstransformationen
$\tilde{\varphi}_{il} := \varphi_{il}$ für "iterative" Knoten (Anweisung, Verzweigung, Sprung)
il Aufrufknoten mit $\alpha_{il} = pid_R(\ldots)$
[Grafik] [/Grafik]
$\tilde{\varphi}_{il}(d) := \varphi_{k(q_k+1)}(d, \Phi_k(\varphi_{il}(d)))$
$\rightarrow$ rekursive Funktionsgleichungssysteme zur Bestimmung der Transformationen $\Phi_1, \ldots, \Phi_r$
$F_i(X) = F_{i(q_i+q)}(X)$
$F_{i1}(X) = X$
$F_{ij}(X) = \bigsqcup_{il \in \underline{\text{Var}}(ij) in G_i} \tilde{\varphi}_{il}(F_{il}(X))$

$$\text{wobei } \tilde{\varphi}_{il}(X) = \begin{cases} \varphi_{il}(X) \text{ bei "iter. Knoten"} \\ \varphi_{k(q_k+1)}(X, F_k(\varphi_{il}(X)) \end{cases}$$

Die eigentlichen Gleichungssysteme $E_0, \ldots, E_r$ zur Berechnung der Analyseinformation AI_{ij}:
$AI_{01} = d_s$
$AI_{i1} = \bigsqcup_{kj \text{ mit } a_{k_j} = pid_i()} \varphi_{ka}(AI_{kj})$
$AI_{ij} = \bigsqcup_{il \in \underline{\text{Var}}(ij) \text{ im } G_i} \tilde{\varphi}_{il}(AI_{il})$
$A_\pi := \bigcup_{i=0} E_i \cup \bigcup_{i=1}^{r} FE_i$
Fp-Technik bestimmt die MFP-LKösung
$AI^{MFP} := (AI_{ij}^{MFP} | i = 0, \ldots, r, j = 1, \ldots, q_i)$

Satz 6.4.1: Knoop, Steffen

a) Korrektheit $AI_{ij}^{MOP} \leq AI_{ij}^{MFP}$

b) Vollständigkeit $AI_{ij}^{MOP} = AI_{ij}^{MFP}$ falls alle φ_{ij} distributiv

Kapitel 7

Analyse dynamischer Datenstrukturen

<u>Bisher:</u> statische Datenstrukturen (Programmvariablen $\widehat{=}$ Speicherzellen)
<u>Jetzt:</u> Zeiger(-variablen), dynamische Speicherverwaltung im Heap
<u>Ziel:</u> Shape-Analyse = approximative Analyse der Datenstrukturen im Heap
Gewonnene Informationen:

- Datentypen ($\rightarrow$ vermeidung von Typfehlern, z.B. Dereferenzierung von nil-Zeigern)
- Sharing (Zeigervariablen mit gleichem Adressinhalt $\rightarrow$ Aliasing Effekte)
- Erreichbarkeit ($\rightarrow$ Garbage Collection)
- Disjunkte Heapbereiche ($\rightarrow$ Parallelisierung)
- Shape (Listen, Bäume, Zykelfreiheit etc.)

<u>Literatur:</u> [Nielson, Nielson, Hankin99, Set 2.6]

7.1 Erweiterung der Sprache IC

Strukturierte Heapzellen (Werte oder Zeiger)
Zugriff über Selektoren: $sel \in \underline{\mathrm{Sel}}$
<u>Zeigerausdrücke:</u>
$p \in \underline{\mathrm{PExp}} = V \cup \{x.sel | x \in V, sel \in \underline{\mathrm{Sel}}\}$
<u>Wertzuweisungen:</u>
$p \leftarrow e$ mit $p \in \underline{\mathrm{PExp}}$ und $e \in \ldots \cup \underline{\mathrm{PExp}} \cup \underline{\mathrm{nil}}$
<u>Allokierungsanweisungen:</u>
<u>malloc</u> p mit $p \in \underline{\mathrm{PExp}}$ (Sprunganweisungen, Markierungen wie gehabt)

Beispiel 7.1.1:
<u>in</u> x; <u>out</u> y; <u>loc</u> z;

1. $y \leftarrow$ nil ;
2. if $x =$ nil goto 8;
3. $z \leftarrow y$;
4. $y \leftarrow x$;
5. $x \leftarrow x.next$;
6. $y.next \leftarrow z$;
7. $tgoto2$;
8. $z \leftarrow$ nil ;

Operationelle Semantik: siehe [NNH99, Set 2.6.1]
Beispiel: siehe Folien 7.1 - 7.3

7.2 Shape-Graphen

Ziel: Abstraktion des (unbeschränkten) Heaps durch eine endliche Datenstruktur

Definition 7.2.1:
Ein Shape-Graph $G = (S, H)$ besteht aus einer Menge $S \subseteq$ Aloc abstrakter Adressen und einem abstrakten Heap $H \subseteq S \times Sel \times S$ (Notation: $\overline{X^{Sel} y}$ für $(X, sel, Y) \in H$)
Hierbei sein *Aloc* $:= p(V)$

Bemerkung: [Interpretation von $G = (S, H)$]

- Für $X \in S$ bedeutet $x, y \in X$, dass die Variablen x und y auf die gleiche Heapadresse verweisen. Somit repräsentiert $\emptyset$ alle Heapadressen, die nicht direkt durch Variablen dereferenziert werden.
- Verweisen $x \in sel$ und y auf die gleichen Heapadresse ($x, y \in V, sel \in$ Sel) und sind $X, Y \in S$ mit $x \in X$ und $y \in Y$, so gilt $X \stackrel{sel}{\rightarrow} Y$ in H

Beispiel 7.2.2: Folien 7.1 - 7.3
(beachte: Shape-Graph a (für x) $\widehat{=}$ Shape-Graph e (für y) $\rightarrow$ Ein-/Ausgabe strukturell "ähnlich")

Obiger Interpretation entsprechend sollte jeder Shape-Graph $G = (S, H)$ den folgenden Invarianten genügen:

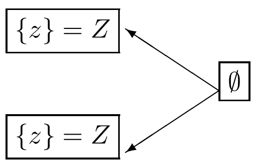
1. Disjunktheit: $X, Y \in S \rightarrow X = Y$ oder $X \cap Y = \emptyset$

   (d.h. eine Variable kann nur auf eine (abstrakte) Heapadresse verweisen)

2. Determiniertheit: $X \stackrel{sel}{\to} Y$ und $X \stackrel{sel}{\to} Z$ in $H \to X = \emptyset$ oder $Y = Z$ (d.h. die über einen Selektor referenzierte Heapadresse ist eindeutig, wenn die Ausgangsadresse nicht $\emptyset$ ist)

Bemerkung: Das folgende Beispiel zeigt, dass in (2) im Fall $X = \emptyset$ Determiniertheit im allgemeinen nicht vorliegt:

Beispiel 7.2.3:
Konkret
$y \to \circ \stackrel{sel}{\leftarrow} \circ$
$z \to \circ \stackrel{sel}{\leftarrow} \circ$
Abstrakt:



Definition 7.2.4:
Ein Shape-Graph mit (1) und (2) heisst kompatibel. Die Menge aller kompatiblen Shape-Graphen wird mit SG bezeichnet.

Bemerkung: IN [NNH99] besitzen Shape-Graphen eine weitere Komponente, die sogenannte Sharing-Information.

7.3 Die Analyse

Ansatz: Vorwärtsanalyse zur Bestimmung aller Shape-Graphen, die die mögliche Heapstruktur im entsprechenden Programmpunkt abstrahieren.
Vollständiger Verband: $\mathcal{D} = \langle p(SG), S \rangle$
Klar:
V, Sel endlich
→ SG endlich
→ p(SG) endlich
→ ACC-Eigenschaft
Abstrakte Interpretation: $\mathfrak{I} = \langle D, \varphi, d_s \rangle$
Startinformationen: d_s = Menge aller Shape-Graphen zu den möglichen Anfangswerten aller $x \in IV$

Beispiel 7.3.1: Listenspiegelung
$IV = \{x\}$, x (endliche) Liste → $d_s = \{\underbrace{(\emptyset, \emptyset)}_{\text{leere}}, \underbrace{\{x\}}_{\text{ein El.}}, \underbrace{\{x\} \stackrel{next}{\to} \emptyset}_{\text{zwei El.}}, \underbrace{\{x\} \stackrel{next}{\to} \emptyset \circlearrowleft next}_{\geq \text{ drei El. Eingabeliste}}\}$

[02.02.2005]

Transferfunktion: $\varphi_\alpha : p(SG) \to p(SG)$ (monoton) für $\alpha \in \underline{Anw}$

Diese transformieren jeden einzelnen Shape-graphen in eine Menge solcher Graphen, d.h. $\varphi_\alpha(\{G_1,\ldots,g_n\}) = \cup_{i=1}^n \Psi_\alpha(G_i)$ wobei $\Psi_\alpha : SG \to p(SG)$ durch Fallunterscheidung bezgl. α definiert ist.

a) $\alpha =$ goto l/ if $p(u_1,\ldots,u_r)$ goto $l/x \leftarrow x$
   Keine Modifikation des Heaps $\to \Psi_\alpha(G) = \{G\}$

b) $\alpha = x \leftarrow c/x \leftarrow f(u_1,\ldots,u_r)/x \leftarrow$ nil
   evtl Verweise von x auf den Heap werden aufgehoben $\to \Psi_\alpha(b) = \{kill_x(G)\}$
   Wobei $kill_x(S,H) = (S',H')$
   mit $S' = \{X\backslash\{x\} | X \in S\}$
   und $H' = \{(X\backslash\{x\}, sel, Y\backslash\{x\} | (X, selY) \in H\}$
   Bsp Folie 7.3, Schritt $d \to e$ ($\alpha = z \leftarrow$ nil)

c) $\alpha = x \leftarrow y$ (mit $x \neq y$)
   Effekt:

   (a) Aufhebung der Bindung von x
   (b) Weitergabe evtl. Bindungen von y an x
   $\to \Psi_\alpha(G| = \{gen_x^y(kill_x(G))\}$
   wobei $gen_x^y(S,H) = (S',H')$
   mit $S' = \{H' = \{(g_x^y(X), sel, g_x^y(Y) | (X, sel, Y) \in H\}$
   für
   $$g_x^y = \begin{cases} X \cup \{x\}, & \text{falls } y \in X \\ X, & \text{sonst} \end{cases}$$

   Bsp Folie 7.1/7.2, Schritt $a3 \to a4$ ($\alpha = y \leftarrow x$)

d) $\alpha = x \leftarrow x.sel$:
   wird ersetzt durch die (äquivalente) Anweisungsfolge
   $t \leftarrow x.sel; x \leftarrow t; t \leftarrow$ nil ($t \in LV$ neu)
   $\to \varphi_\alpha = \varphi_{t \leftarrow \underline{\text{nil}}} \circ \varphi_{x \leftarrow t} \circ \varphi_{t \leftarrow x.sel}$

e) $x \leftarrow y.sel$ ($x \neq y$):
   Sei zunächst $(S',H') = kill_x(G)$
   Mögliche Fälle:

   (a) $y \notin X$ für alle $X \in S'$ oder
   $y \in Y$ für ein $Y \in S'$, aber es existiert kein $Z \in S'$ mit $Y \stackrel{sel}{\to} Z$ (d.h. y oder y.sel ist eine Zahl, nil, oder undefiniert)
   (b) es existiert $Y, Z \in S'$ mit $y \in Y, Z \neq \emptyset$ und $Y \stackrel{sel}{\to} Z$
   (d.h. die Adresse y.sel wird auch durch andere Variablen referenziert)

(c) es existiert $y \in S'$ mit $y \in Y$ und $Y \overset{sel}{\to} \emptyset$
(d.h. keine andere Variable verweist auf die gleiche Adresse wie y.sel)

Entsprechende Modifikation dees Shape-Graphen:

1. Nur x entfernen:

   $\Psi_\alpha(G) = \{(S', H')\}$

2. In z einen x-Verweis ergänzen

   $\Psi_\alpha(G) = \{(S'', H'')\}$

   mit $S'' = \{h_x^z(X) | X \in S'\}$

   und $H'' = \{h_x^z(X), sel, h_x^z(Y) | (X, sel, Y) \in H'\}$

   $$\text{für } h_x^z(X) = \begin{cases} Z \cup \{x\}, & \text{falls } X = Z \\ X, & \text{sonst} \end{cases}$$

3. Abstrakte Adresse für x einführen und alle möglichen Heapstrukturen berücksichtigen.

   $\Psi_\alpha(G) = \{(S'', H'') \in SG | kill_x(S'', H'') = (S', H'), \{x\} \in S'', (y, sel, \{x\}) \in H''\}$

   Bsp (Fall 3) Folie 7.4

   Bem: Das Eintreten von Fall 1 bedeutet, dass im konkreten Programm möglicherweise ein nil-Zeiger dereferenziert wird

4. $\alpha = x.sel \leftarrow x / x.sel \leftarrow f(u_1, \ldots, u_r) / x.sel \leftarrow$ nil : ähnlich zu (e) (Fallunterscheidung bzgl. x/x.sel)

5. $\alpha = x.sel \leftarrow y$:

   Bsp Folie 7.2, Schritt $a5 \to b$ ($\alpha = x.next \leftarrow z$)

6. $\alpha = x.sel \leftarrow y.sel'$:

   wird ersetzt durch $t \leftarrow y.sel', x.sel; x.sel \leftarrow t; t \leftarrow$ nil

7. $\alpha =$ malloc$x$:

   $\Psi_\alpha(S, H) = (S' \cup \{x\})$ für $(S', H') = kill_x(S, H)$

8. $\alpha =$ malloc$x.sel$:

   wird ersetzt durch malloc$t$; $x.sel \leftarrow t; t \leftarrow$ nil