

Analyse und Verifikation

(SS 2009, 185.276, VU 2.0h, ECTS 3.0, MSE/W)

Jens Knopp

Institut für Computersprachen

knoop@compLang.tuwien.ac.at

<http://www.compLang.tuwien.ac.at/knoop/>

Dienstag, 13:30 Uhr bis 15:00 Uhr, FH Hörsaal 4 (Wiedner

Hauptstr. 8, 1040 Wien)

Kapitel 1 Grundlagen

Kap. 1 Grundlagen

1

Grundlagen

Syntax und Semantik von Programmiersprachen...

- *Syntax*: Regelwerk zur Spezifikation wohlgeformter Programme
- *Semantik*: Regelwerk zur Spezifikation der Bedeutung oder des Verhaltens wohlgeformter Programme oder Programmteile (aber auch von Hardware beispielsweise)

Kap. 1 Grundlagen

2

Literaturhinweise 1(2)

Als Textbücher...

- Hanne R. Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*, Springer, 2007.
- Hanne R. Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*, Wiley Professional Computing, Wiley, 1992.
(Siehe http://www.daimi.au.dk/~bra8130/W11eg_book/wiley.html für eine frei verfügbare (überarbeitete) Version.)

Kap. 1 Grundlagen

3

Literaturhinweise 2(2)

Ergänzend und weiterführend...

- Ernst-Rüdiger Olderog, Bernhard Steffen. *Formale Semantik und Programmvifikation*. In Informatik-Handbuch, P. Rechenberg, G. Pomberger (Hrsg.), Carl Hanser Verlag, 129 - 148, 1997.
- Krzysztof R. Apt, Ernst-Rüdiger Olderog. *Programmverifikation – Sequentielle, Parallele und verteilte Programme*. Springer, 1994.
- Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*, Wiley, 1984.
- Krzysztof R. Apt. *Ten Years of Hoare's Logic: A Survey – Part I*, ACM Transactions on Programming Languages and Systems 3, 431 - 483, 1981.

Kap. 1 Grundlagen

4

Kapitel 1.1 Motivation

Kap. 1.1 Motivation

5

Motivation

...*formale* Semantik von Programmiersprachen einzuführen:

(Mathematische) Rigorosität formaler Semantik...

- erlaubt Mehrdeutigkeiten, Über- und Unterspezifikationen in natürlichsprachlichen Dokumenten aufzudecken und aufzulösen
- bietet die Grundlage für Implementierungen der Programmiersprache, für Analyse, Verifikation und Transformation von Programmen

Kap. 1.1 Motivation

6

Unsere Modellsprache

- Die Programmiersprache WHILE
 - Syntax
 - Semantik
- Semantikdefinitionsstile (*...und wofür sie besonders geeignet sind und ihre Beziehungen zueinander*)
 - Operationelle Semantik
 - * Natürliche Semantik
 - * Strukturell operationelle Semantik
 - Denotationelle Semantik
 - Axiomatische Semantik
 - * Beweiskalküle für partielle & totale Korrektheit
 - * Korrektheit, Vollständigkeit

Überblick über Syntax & Semantik (1)

- **Syntax**

...Programme der Form:

$$\pi ::= x := a \mid \text{skip} \mid \text{abort} \mid$$
$$\text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \text{ fi} \mid$$
$$\text{while } b \text{ do } \pi_1 \text{ od} \mid$$
$$\pi_1; \pi_2$$

- **Semantik**

...in Form von *Zustandstransformationen*:

$$\llbracket \cdot \rrbracket : \text{Prg} \rightarrow (\Sigma \rightarrow \Sigma)$$

über

– $\Sigma =_{df} \{\sigma \mid \sigma : \text{Var} \rightarrow D\}$ Menge aller *Zustände* über der *Variablenmenge Var* und geeignetem *Datenbereich D*.
(In der Folge werden wir für D oft die Menge der ganzen Zahlen \mathbb{Z} betrachten.)

WHILE, der sog. "while"-Kern imperativer Programmiersprachen, besitzt

- Zuweisungen (einschließlich der leeren Anweisung und der Fehleranweisung)
- Fallunterscheidungen
- while-Schleifen
- Sequentielle Komposition

Beachte: WHILE ist "schlank", nichtsdestotrotz *Turingmächtig!*

Überblick über Syntax & Semantik (2)

Zahldarstellungen

$$z ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$$
$$n ::= z \mid nz$$

Arithmetische Ausdrücke

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

Boolesche Ausdrücke

$$b ::= \text{true} \mid \text{false} \mid$$
$$a_1 \neq a_2 \mid a_1 \neq a_2 \mid a_1 < a_2 \mid a_1 \leq a_2 \mid \dots \mid$$
$$b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b_1$$

Überblick über Syntax & Semantik (3)

In der Folge bezeichnen wir mit...

- **Num** die Menge der Zahldarstellungen, $n \in \text{Num}$
- **Var** die Menge der Variablen, $x \in \text{Var}$
- **Aexpr** die Menge arithmetischer Ausdrücke, $a \in \text{Aexpr}$
- **Bexpr** die Menge Boolescher Ausdrücke, $b \in \text{Bexpr}$
- **Prg** die Menge aller WHILE-Programme, $\pi \in \text{Prg}$

Überblick über Syntax & Semantik (4)

In der Folge werden wir im Detail betrachten...

- Operationelle Semantik
 - Natürliche Semantik: $\llbracket \cdot \rrbracket_{ns} : \text{Prg} \rightarrow (\Sigma \rightarrow \Sigma)$
 - Strukturell operationelle Semantik: $\llbracket \cdot \rrbracket_{sos} : \text{Prg} \rightarrow (\Sigma \rightarrow \Sigma)$
- Denotationelle Semantik: $\llbracket \cdot \rrbracket_{ds} : \text{Prg} \rightarrow (\Sigma \rightarrow \Sigma)$
- Axiomatische Semantik: ...*abweichender Fokus*
 - ...und deren Beziehungen zueinander, d.h. die Beziehungen zwischen $\llbracket \cdot \rrbracket_{sos}$, $\llbracket \cdot \rrbracket_{ns}$ und $\llbracket \cdot \rrbracket_{ds}$

Kapitel 1.3: Semantik von Ausdrücken

Semantik arithmetischer & Boolescher Ausdrücke

Die Semantik von WHILE stützt sich ab auf die...

Semantik

- arithmetischer Ausdrücke: $\llbracket \cdot \rrbracket_A : \text{Aexpr} \rightarrow (\Sigma \rightarrow \mathbb{Z})$
- Boolescher Ausdrücke: $\llbracket \cdot \rrbracket_B : \text{Bexpr} \rightarrow (\Sigma \rightarrow \mathbb{B})$

- $\llbracket \cdot \rrbracket_A : \mathbf{Aexpr} \rightarrow (\Sigma \rightarrow \mathbb{Z})$ induktiv definiert durch
- $\llbracket n \rrbracket_A(\sigma) =_{df} \llbracket n \rrbracket_N$
 - $\llbracket x \rrbracket_A(\sigma) =_{df} \sigma(x)$
 - $\llbracket a_1 + a_2 \rrbracket_A(\sigma) =_{df} plus(\llbracket a_1 \rrbracket_A(\sigma), \llbracket a_2 \rrbracket_A(\sigma))$
 - $\llbracket a_1 * a_2 \rrbracket_A(\sigma) =_{df} mul(\llbracket a_1 \rrbracket_A(\sigma), \llbracket a_2 \rrbracket_A(\sigma))$
 - $\llbracket a_1 - a_2 \rrbracket_A(\sigma) =_{df} minus(\llbracket a_1 \rrbracket_A(\sigma), \llbracket a_2 \rrbracket_A(\sigma))$
 - $\llbracket a_1 / a_2 \rrbracket_A(\sigma) =_{df} durch(\llbracket a_1 \rrbracket_A(\sigma), \llbracket a_2 \rrbracket_A(\sigma))$
 - ... (andere Operatoren analog)

wobei

- *plus*, *mul*, *minus*, *durch* : $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ die übliche Addition, Multiplikation, Subtraktion und (ganzzahlige) Division auf den ganzen Zahlen \mathbb{Z} bezeichnen.

- $\llbracket \cdot \rrbracket_N : \mathbf{Num} \rightarrow \mathbb{Z}$ induktiv definiert durch
- $\llbracket 0 \rrbracket_N =_{df} \mathbf{0}$, ..., $\llbracket 9 \rrbracket_N =_{df} \mathbf{9}$
 - $\llbracket ni \rrbracket_N =_{df} plus(mul(\mathbf{10}, \llbracket n \rrbracket_A), \llbracket i \rrbracket_N)$, $i \in \{0, \dots, 9\}$
 - $\llbracket -n \rrbracket_N =_{df} minus(\llbracket n \rrbracket_N)$
- Beachte*: $0, 1, 2, \dots$ bezeichnen *syntaktische* Entitäten, $\mathbf{0}, \mathbf{1}, \mathbf{2}, \dots$ bezeichnen *semantische* Entitäten, in diesem Fall die ganze Zahlen.

Kap. 1.3: Semantik von Ausdrücken

17

Semantik Boolescher Ausdrücke (1)

- $\llbracket \cdot \rrbracket_B : \mathbf{Bexpr} \rightarrow (\Sigma \rightarrow \mathbf{B})$ induktiv definiert durch
- $\llbracket true \rrbracket_B(\sigma) =_{df} \mathbf{tt}$
 - $\llbracket false \rrbracket_B(\sigma) =_{df} \mathbf{ff}$
 - $\llbracket a_1 = a_2 \rrbracket_B(\sigma) =_{df} \begin{cases} \mathbf{tt} & \text{falls } equal(\llbracket a_1 \rrbracket_A(\sigma), \llbracket a_2 \rrbracket_A(\sigma)) \\ \mathbf{ff} & \text{sonst} \end{cases}$
 - ... (andere Relatoren (z.B. $<$, \leq , \dots) analog)
 - $\llbracket \neg b \rrbracket_B(\sigma) =_{df} neg(\llbracket b \rrbracket_B(\sigma))$
 - $\llbracket b_1 \wedge b_2 \rrbracket_B(\sigma) =_{df} conj(\llbracket b_1 \rrbracket_B(\sigma), \llbracket b_2 \rrbracket_B(\sigma))$
 - $\llbracket b_1 \vee b_2 \rrbracket_B(\sigma) =_{df} disj(\llbracket b_1 \rrbracket_B(\sigma), \llbracket b_2 \rrbracket_B(\sigma))$

Kap. 1.3: Semantik von Ausdrücken

18

Semantik Boolescher Ausdrücke (2)

- ...wobei
- \mathbf{tt} und \mathbf{ff} die Wahrheitswertkonstanten "wahr" und "falsch" sowie
 - *conj*, *disj* : $\mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}$ und *neg* : $\mathbf{B} \rightarrow \mathbf{B}$ die übliche zweistellige logische Konjunktion und Disjunktion und einstellige Negation auf der Menge der Wahrheitswerte und
 - *equal* : $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbf{B}$ die übliche Gleichheitsrelation auf der Menge der ganzen Zahlen bezeichnen.

Beachte auch hier den Unterschied zwischen den *syntaktischen* Entitäten *true* und *false* und ihren *semantischen* Gegenstücken \mathbf{tt} und \mathbf{ff} .

Kap. 1.3: Semantik von Ausdrücken

19

Vereinbarung

In der Folge seien die

- *Semantik arithmetischer Ausdrücke*:

$$\llbracket \cdot \rrbracket_A : \mathbf{Aexpr} \rightarrow (\Sigma \rightarrow \mathbb{Z})$$
 - *Semantik Boolescher Ausdrücke*:

$$\llbracket \cdot \rrbracket_B : \mathbf{Bexpr} \rightarrow (\Sigma \rightarrow \mathbf{B})$$
- wie zuvor und die Menge der (Speicher-) Zustände wie folgt festgelegt:
- (*Speicher*-) *Zustände*: $\Sigma =_{df} \{ \sigma \mid \sigma : \mathbf{Var} \rightarrow \mathbb{Z} \}$

Kap. 1.3: Semantik von Ausdrücken

20

Kapitel 1.4 Syntaktische und semantische Substitution

Kap. 1.4 Syntaktische und semantische Substitution

21

Freie Variablen

...arithmetischer Ausdrücke:

$$\begin{aligned} FV(n) &= \emptyset \\ FV(x) &= \{x\} \\ FV(a_1 + a_2) &= FV(a_1) \cup FV(a_2) \\ &\dots \end{aligned}$$

...Boolescher Ausdrücke:

$$\begin{aligned} FV(true) &= \emptyset \\ FV(false) &= \emptyset \\ FV(a_1 = a_2) &= FV(a_1) \cup FV(a_2) \\ &\dots \\ FV(b_1 \wedge b_2) &= FV(b_1) \cup FV(b_2) \\ FV(b_1 \vee b_2) &= FV(b_1) \cup FV(b_2) \\ FV(\neg b_1) &= FV(b_1) \end{aligned}$$

Kap. 1.4 Syntaktische und semantische Substitution

22

Eigenschaften von $\llbracket \cdot \rrbracket_A$ und $\llbracket \cdot \rrbracket_B$

Lemma 1.4.1

Seien $a \in \mathbf{Aexpr}$ und $\sigma, \sigma' \in \Sigma$ mit $\sigma(x) = \sigma'(x)$ für alle $x \in FV(a)$. Dann gilt:

$$\llbracket a \rrbracket_A(\sigma) = \llbracket a \rrbracket_A(\sigma')$$

Lemma 1.4.2

Seien $b \in \mathbf{Bexpr}$ und $\sigma, \sigma' \in \Sigma$ mit $\sigma(x) = \sigma'(x)$ für alle $x \in FV(b)$. Dann gilt:

$$\llbracket b \rrbracket_B(\sigma) = \llbracket b \rrbracket_B(\sigma')$$

Kap. 1.4 Syntaktische und semantische Substitution

23

Syntaktische/Semantische Substitution

Von zentraler Bedeutung...

- Substitutionen
 - Syntaktische Substitution
 - Semantische Substitution
 - Substitutionslemma

Kap. 1.4 Syntaktische und semantische Substitution

24

Definition 1.4.3

Die *syntaktische Substitution* für arithmetische Terme ist eine dreistellige Abbildung

$$[./.] : \mathbf{Aexpr} \times \mathbf{Aexpr} \times \mathbf{Var} \rightarrow \mathbf{Aexpr}$$

die induktiv definiert ist durch

$$\begin{aligned} n[t/x] &=_{df} n & \text{für } n \in \mathbf{Num} \\ y[t/x] &=_{df} \begin{cases} t & \text{falls } y = x \\ y & \text{sonst} \end{cases} \\ (t_1 \text{ op } t_2)[t/x] &=_{df} (t_1[t/x] \text{ op } t_2[t/x]) & \text{für } \text{op} \in \{+, *, -, \dots\} \end{aligned}$$

Kap. 1.4 Syntaktische und semantische Substitution

25

Semantische Substitution

Definition 1.4.4

Die *semantische Substitution* ist eine dreistellige Abbildung

$$[./.] : \Sigma \times \mathbb{Z} \times \mathbf{Var} \rightarrow \Sigma$$

die definiert ist durch

$$\sigma[z/x](y) =_{df} \begin{cases} z & \text{falls } y = x \\ \sigma(y) & \text{sonst} \end{cases}$$

Kap. 1.4 Syntaktische und semantische Substitution

26

Substitutionslemma

Wichtig:

Lemma 1.4.5 (Substitutionslemma)

$$\llbracket e[t/x] \rrbracket_A(\sigma) = \llbracket e \rrbracket_A(\sigma[\llbracket t \rrbracket_A(\sigma)/x])$$

wobei

- $[t/x]$ die *syntaktische Substitution* und
- $\llbracket t \rrbracket_A(\sigma)/x$ die *semantische Substitution* bezeichnen.

Analog gilt ein entsprechendes Substitutionslemma für $\llbracket \cdot \rrbracket_B$.

Kap. 1.4 Syntaktische und semantische Substitution

27

Induktive Beweisprinzipien (1)

Zentral:

- Vollständige Induktion
 - Verallgemeinerte Induktion
 - Strukturelle Induktion
- ...zum Beweis einer Aussage A.

Kap. 1.5: Induktive Beweisprinzipien

29

Kapitel 1.5: Induktive Beweisprinzipien

Kap. 1.5: Induktive Beweisprinzipien

28

Induktive Beweisprinzipien (2)

Zur Erinnerung hier wiederholt:

Die Prinzipien der...

- *vollständigen Induktion*
 $(A(1) \wedge (\forall n \in \mathbb{N}. A(n) \rightarrow A(n+1))) \rightarrow \forall n \in \mathbb{N}. A(n)$
- *verallgemeinerten Induktion*
 $(\forall n \in \mathbb{N}. (\forall m < n. A(m)) \rightarrow A(n)) \rightarrow \forall n \in \mathbb{N}. A(n)$
- *strukturellen Induktion*
 $(\forall s \in S. \forall s' \in \text{Komp}(s). A(s')) \rightarrow A(s) \rightarrow \forall s \in S. A(s)$

Beachte: \rightarrow bezeichnet hier die logische Implikation.

Kap. 1.5: Induktive Beweisprinzipien

30

Beispiel: Beweis von Lemma 1.4.1 (1)

...durch strukturelle Induktion

Seien $a \in \mathbf{AExpr}$ und $\sigma, \sigma' \in \Sigma$ mit $\sigma(x) = \sigma'(x)$ für alle $x \in \text{FV}(a)$.

Induktionsanfang:

Fall 1: Sei $a \equiv n, n \in \mathbf{Num}$.

Mit den Definitionen von $\llbracket \cdot \rrbracket_A$ und $\llbracket \cdot \rrbracket_B$ erhalten wir unmittelbar wie gewünscht:

$$\llbracket a \rrbracket_A(\sigma) = \llbracket n \rrbracket_A(\sigma) = \llbracket n \rrbracket_B = \llbracket n \rrbracket_A(\sigma') = \llbracket a \rrbracket_A(\sigma')$$

Kap. 1.5: Induktive Beweisprinzipien

31

Beispiel: Beweis von Lemma 1.4.1 (2)

Fall 2: Sei $a \equiv x, x \in \text{Var}$.

Mit der Definition von $\llbracket _ \rrbracket_A$ erhalten wir auch hier wie gewünscht:

$$\llbracket a \rrbracket_A(\sigma) = \llbracket x \rrbracket_A(\sigma) = \sigma(x) = \sigma'(x) = \llbracket x \rrbracket_A(\sigma') = \llbracket a \rrbracket_A(\sigma')$$

Kap. 1.5: Induktive Beweisprinzipien

32

Induktionsschluss:

Fall 3: Sei $a \equiv a_1 + a_2, a_1, a_2 \in \mathbf{Aexpr}$

Dann erhalten wir:

$$\begin{aligned} \llbracket a \rrbracket_A(\sigma) &= \llbracket a_1 + a_2 \rrbracket_A(\sigma) \\ &= \llbracket a_1 \rrbracket_A(\sigma) + \llbracket a_2 \rrbracket_A(\sigma) \\ &= \llbracket a_1 \rrbracket_A(\sigma') + \llbracket a_2 \rrbracket_A(\sigma') \\ &= \llbracket a_1 + a_2 \rrbracket_A(\sigma') \\ &= \llbracket a \rrbracket_A(\sigma') \end{aligned}$$

Übrige Fälle: Analog.

q.e.d.

Kap. 1.5: Induktive Beweisprinzipien

33

Kapitel 1.6 Semantikdefinitionsstile

Kap. 1.6 Semantikdefinitionsstile

34

Semantikdefinitionsstile (2)

- *Sprachentwicklersicht*
 - Denotationelle Semantik
- *Sprach- und Anwendungsimplimentiersicht*
 - Operationelle Semantik
 - * Strukturell operationelle Semantik (small steps semantics)
 - * Natürliche Semantik (big steps semantics)
- *Programmierer- und Verifiziersicht*
 - Axiomatische Semantik

Kap. 1.6 Semantikdefinitionsstile

36

Kapitel 2.1 Strukturell Operationelle Semantik

Kap. 2.1 Strukturell Operationelle Semantik

38

Semantikdefinitionsstile (1)

Es gibt unterschiedliche Stile, die Semantik einer Programmiersprache festzulegen. Sie richten sich an unterschiedliche Adressaten und deren spezifische Sicht auf die Semantik...

Insbesondere unterscheiden wir den...

- *denotationellen*
- *operationellen*
- *axiomatischen*

Stil.

Kap. 1.6 Semantikdefinitionsstile

35

Kapitel 2 Operationelle Semantik von WHILE

Kap. 2 Operationelle Semantik von WHILE

37

Literaturhinweise

- Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Journal of Logic and Algebraic Programming 60-61, 17 - 139, 2004.
- Gordon D. Plotkin. *An Operational Semantics for CSP*. In Proceedings of TC-2 Working Conference on Formal Description of Programming Concepts II, Elsevier, 1982.

Kap. 2.1 Strukturell Operationelle Semantik

39

- Die *SO-Semantik von WHILE* ist gegeben durch ein Funktional:

$$\llbracket \cdot \rrbracket_{SOS} : \mathbf{Prg} \rightarrow (\Sigma \rightarrow \Sigma_\varepsilon)$$

das in der Folge von uns zu definieren ist...

Dabei gilt:

- $\Sigma_\varepsilon = \mathcal{d} \Sigma \cup \{error\}$, wobei *error* einen speziellen Fehlerzustand bezeichnet, $error \notin \Sigma$.

Konfigurationen

- Wir unterscheiden:
 - Nichtterminale* bzw. (*Zwischen-)* *Konfigurationen* γ der Form $\langle \pi, \sigma \rangle$:
 - ...(*Rest-*) Programm π ist auf den (*Zwischen-*) Zustand σ anzuwenden.
 - Terminale* bzw. *finale Konfigurationen* γ der Formen σ oder *error*
 - ...beschreiben das Resultat nach Ende der Berechnung, wobei Ende nach...
- regulärer* Terminierung: angezeigt durch gewöhnliche Zustände σ
 - irregulärer* Terminierung: angezeigt durch *error*-behaftete Konfiguration
- Γ bezeichne die Menge aller Konfigurationen, $\gamma \in \Gamma$

SOS-Regeln von WHILE (2)

$$\begin{array}{l} \llbracket t^H_{SOS} \rrbracket \\ \hline \langle \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \text{ fi}, \sigma \rangle \Rightarrow \langle \pi_1, \sigma \rangle \\ \llbracket t^f_{SOS} \rrbracket \\ \hline \langle \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \text{ fi}, \sigma \rangle \Rightarrow \langle \pi_2, \sigma \rangle \\ \llbracket \text{while}_{SOS} \rrbracket \\ \hline \langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \Rightarrow \langle \text{if } b \text{ then } \pi; \text{ while } b \text{ do } \pi \text{ od else skip fi}, \sigma \rangle \end{array} \quad \begin{array}{l} \llbracket b \rrbracket_B(\sigma) = \text{tt} \\ \llbracket b \rrbracket_B(\sigma) = \text{ff} \end{array}$$

Sprechweisen (2)

Im Fall der SO-Semantik von WHILE haben wir demnach

- 6 Axiome
 - ...für die leere Anweisung, Fehleranweisung, Zuweisung, Fallunterscheidung und while-Schleife.
- 2 Regeln
 - ...für die sequentielle Komposition.

Strukturell operationelle Semantik

Intitiv:

- Die SO-Semantik beschreibt den Berechnungsvorgang von Programmen $\pi \in \mathbf{Prg}$ als Folge elementarer Speicherzustandsübergänge.

Zentral:

- ...der Begriff der *Konfiguration!*

SOS-Regeln von WHILE (1)

$$\begin{array}{l} \llbracket \text{skip}_{SOS} \rrbracket \\ \hline \langle \text{skip}, \sigma \rangle \Rightarrow \sigma \\ \llbracket \text{abort}_{SOS} \rrbracket \\ \hline \langle \text{abort}, \sigma \rangle \Rightarrow error \\ \llbracket \text{ass}_{SOS} \rrbracket \\ \hline \langle x := L, \sigma \rangle \Rightarrow \sigma \llbracket \llbracket L \rrbracket_A(\sigma) / x \rrbracket \\ \llbracket \text{comp}^1_{SOS} \rrbracket \\ \hline \langle \pi_1, \sigma \rangle \Rightarrow \langle \pi_1', \sigma' \rangle \\ \langle \pi_1; \pi_2, \sigma \rangle \Rightarrow \langle \pi_1; \pi_2, \sigma' \rangle \\ \llbracket \text{comp}^2_{SOS} \rrbracket \\ \hline \langle \pi_1, \sigma \rangle \Rightarrow \sigma' \\ \langle \pi_1; \pi_2, \sigma \rangle \Rightarrow \langle \pi_2, \sigma' \rangle \end{array}$$

Sprechweisen (1)

Wir unterscheiden

- Prämissenlose Axiome der Form
 - Konklusion
- Prämissenbehaftete Regeln der Form
 - Prämissen
 - Konklusion

ggf. mit *Randbedingungen (Seitenbedingungen)* wie z.B. in Form von $\llbracket b \rrbracket_B(\sigma) = \text{ff}$ in der Regel $\llbracket t^H_{SOS} \rrbracket$.

Berechnungsschritt, Berechnungsfolge

- Ein *Berechnungsschritt* ... ist von der Form
 - $\langle \pi, \sigma \rangle \Rightarrow \gamma$ mit $\gamma \in (\mathbf{Prg} \times \Sigma_\varepsilon) \cup \Sigma_\varepsilon \cong \Gamma$
- Eine *Berechnungsfolge* zu einem Programm π angesetzt auf einen (Start-) Zustand $\sigma \in \Sigma$ ist
 - eine endliche Folge $\gamma_0, \dots, \gamma_k$ von Konfigurationen mit $\gamma_0 = \langle \pi, \sigma \rangle$ und $\gamma_i \Rightarrow \gamma_{i+1}$ für alle $i \in \{0, \dots, k-1\}$,
 - eine unendliche Folge von Konfigurationen mit $\gamma_0 = \langle \pi, \sigma \rangle$ und $\gamma_i \Rightarrow \gamma_{i+1}$ für alle $i \in \mathbb{N}$.

rechenungsfolgen

- Eine maximale (d.h. nicht mehr verlängerbare) Berechnungsfolge heißt
 - *regulär terminierend*, wenn sie endlich ist und die letzte Konfiguration aus Σ ist,
 - *irregulär terminierend*, wenn sie endlich ist und die letzte Konfiguration *error*-behaftet ist,
 - *divergierend*, falls sie unendlich ist.

Kap. 2.1 Strukturell Operationelle Semantik

48

Beispiel (2)

```

⟨y := 1; while x <> 1 do y := y * x; x := x - 1 od, σ⟩
⇒ ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
⇒ ⟨if x <> 1
   then y := y * x; x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od
   else skip fi, σ[1/y]⟩
⇒ ⟨y := y * x; x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
⇒ ⟨x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od, σ[1/y][3/y]⟩
(≡ ⟨x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od, σ[3/y]⟩ )
⇒ ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[3/y][2/x]⟩
    
```

Beispiel (4)

```

⇒ ⟨if x <> 1
   then y := y * x; x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od
   else skip fi, σ[6/y][1/x]⟩
⇒ ⟨skip fi, σ[6/y][1/x]⟩
⇒ ⟨σ[6/y][1/x]⟩
    
```

Kap. 2.1 Strukturell Operationelle Semantik

52

Beispiel (Detailbetrachtung) (6)

```

[whileseq] ⇒ ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
              ⟨if x <> 1
               then y := y * x; x := x - 1;
               while x <> 1 do y := y * x; x := x - 1 od
               else skip fi, σ[1/y]⟩
    
```

steht vereinfachend für...

```

[whileseq] ⇒ ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
              ⟨if x <> 1 then y := y * x; x := x - 1;
               while x <> 1 do y := y * x; x := x - 1 od
               else skip fi, σ[1/y]⟩
    
```

Kap. 2.1 Strukturell Operationelle Semantik

54

Sei

- $\sigma \in \Sigma$ mit $\sigma(x) = 3$
- $\pi \in \text{Prig}$ mit
 - $\pi \equiv y := 1; \text{while } x <> 1 \text{ do } y := y * x; x := x - 1 \text{ od}$

Betrachte

- die von π angesetzt auf σ , d.h. die von der Anfangskonfiguration
 - $\langle y := 1; \text{while } x <> 1 \text{ do } y := y * x; x := x - 1 \text{ od}, \sigma \rangle$
 induzierte Berechnungsfolge

Kap. 2.1 Strukturell Operationelle Semantik

49

Beispiel (3)

```

⇒ ⟨if x <> 1
   then y := y * x; x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od
   else skip fi, σ[3/y][2/x]⟩
⇒ ⟨y := y * x; x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od, σ[3/y][2/x]⟩
⇒ ⟨x := x - 1;
   while x <> 1 do y := y * x; x := x - 1 od, σ[6/y][2/x]⟩
⇒ ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[6/y][1/x]⟩
    
```

Kap. 2.1 Strukturell Operationelle Semantik

51

Beispiel (Detailbetrachtung) (5)

```

[lassseq] [comp2] ⇒ ⟨y := 1; while x <> 1 do y := y * x; x := x - 1 od, σ⟩
                    ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
steht vereinfachend für...
    
```

```

[comp2] [lassseq] ⇒ ⟨y := 1; while x <> 1 do y := y * x; x := x - 1 od, σ⟩
                    ⟨while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
    
```

Kap. 2.1 Strukturell Operationelle Semantik

53

Beispiel (Detailbetrachtung) (7)

```

[lassseq] [comp2] [comp1] ⇒ ⟨y := y * x; σ[1/y]⟩ ⇒ ⟨x := x - 1; σ[1/y][3/y]⟩
                             ⟨x := x - 1;
                             while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
                             while x <> 1 do y := y * x; x := x - 1 od,
                             (σ[1/y][3/y])
    
```

steht vereinfachend für...

```

[comp1] [lassseq] [comp2] ⇒ ⟨y := y * x; σ[1/y]⟩ ⇒ ⟨x := x - 1; σ[1/y][3/y]⟩
                             ⟨y := y * x; x := x - 1; σ[1/y]⟩
                             ⟨y := y * x; x := x - 1; while x <> 1 do y := y * x; x := x - 1 od, σ[1/y]⟩
                             x := x - 1; while x <> 1 do y := y * x; x := x - 1 od, (σ[1/y][3/y])
    
```

Kap. 2.1 Strukturell Operationelle Semantik

55

Lemma 2.1.1

$\forall \pi \in \mathbf{Prig}, \sigma \in \Sigma_{\varepsilon_1}, \gamma, \gamma' \in \Gamma. (\pi_1, \sigma) \Rightarrow \gamma \wedge (\pi, \sigma) \Rightarrow \gamma' \quad \succ \quad \gamma = \gamma'$

Erinnerung: \succ bezeichnet hier die logische Implikation.

Korollar 2.1.2

Die von den SOS-Regeln für eine Konfiguration induzierte Berechnungsfolge ist eindeutig bestimmt, d.h. *deterministisch*.

Salopper, wenn auch weniger präzise:

Die (SO-) Semantik von WHILE ist deterministisch!

Kap. 2.1 Strukturell Operationelle Semantik

56

Variante induktiver Beweisführung

Induktion über die Länge von Berechnungsfolgen:

- *Induktionsanfang*
 - Beweise, dass A für Berechnungsfolgen der Länge 0 gilt.
- *Induktionsschritt*
 - Beweise unter der Annahme, dass A für Berechnungsfolgen der Länge kleiner oder gleich k gilt (*Induktionshypothese*), dass A auch für Berechnungsfolgen der Länge $k + 1$ gilt.

Kap. 2.1 Strukturell Operationelle Semantik

58

Kap. 2.2 Natürliche Semantik

Kap. 2.2 Natürliche Semantik

60

$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = tt$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = tt$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = ff$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = tt$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = ff$

Natürliche Semantik (2)

$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = tt$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = tt$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = ff$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = tt$
$\llbracket r_{n,s}^k \rrbracket$	$\llbracket b \rrbracket_B(\sigma) = ff$

Kap. 2.2 Natürliche Semantik

62

Korollar 2.1.2 erlaubt uns jetzt festzulegen:

- Die strukturell operationelle Semantik von WHILE ist gegeben durch das Funktional

$$\llbracket \cdot \rrbracket_{SOS} : \mathbf{Prig} \rightarrow (\Sigma \rightarrow \Sigma_{\varepsilon})$$

welches definiert wird durch:

$$\forall \pi \in \mathbf{Prig}, \sigma \in \Sigma. \llbracket \pi \rrbracket_{SOS}(\sigma) =_{df} \begin{cases} \sigma' & \text{falls } (\pi, \sigma) \Rightarrow^* \sigma' \\ error & \text{falls } (\pi, \sigma) \Rightarrow^* error \text{ oder} \\ undef & \text{sonst} \end{cases}$$

Kap. 2.1 Strukturell Operationelle Semantik

57

Anwendung

- Induktive Beweisführung über die Länge von Berechnungsfolgen ist typisch zum Nachweis von Aussagen über Eigenschaften strukturell operationeller Semantik.

Ein Beispiel dafür ist der Beweis von ...

Lemma 2.1.3

$$\forall \pi, \pi' \in \mathbf{Prig}, \sigma, \sigma' \in \Sigma, k \in \mathbb{N}. ((\pi_1; \pi_2, \sigma) \Rightarrow^k \sigma') \succ$$

$$\exists \sigma' \in \Sigma, k_1, k_2 \in \mathbb{N}. (k_1 + k_2 = k \wedge (\pi_1, \sigma) \Rightarrow^{k_1} \sigma' \wedge (\pi_2, \sigma') \Rightarrow^{k_2} \sigma'')$$

Kap. 2.1 Strukturell Operationelle Semantik

59

Natürliche Semantik (1)

...ebenfalls für das Beispiel von WHILE:

$\llbracket skip_{n,s} \rrbracket$	$\llbracket skip_{n,s} \rrbracket \rightarrow \sigma$
$\llbracket abort_{n,s} \rrbracket$	$\llbracket abort_{n,s} \rrbracket \rightarrow error$
$\llbracket assign \rrbracket$	$\llbracket assign \rrbracket \rightarrow \sigma' \llbracket \llbracket r \rrbracket_A(\sigma) / x \rrbracket$
$\llbracket comb_{n,s} \rrbracket$	$\llbracket comb_{n,s} \rrbracket \rightarrow \sigma' \llbracket \llbracket r_1, r_2, \sigma' \rrbracket \rightarrow \sigma'' \rrbracket$
	$\llbracket comb_{n,s} \rrbracket \rightarrow \sigma' \llbracket \llbracket r_1, r_2, \sigma' \rrbracket \rightarrow \sigma'' \rrbracket$

Kap. 2.2 Natürliche Semantik

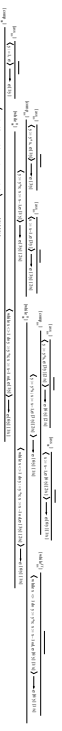
61

Beispiel zur natürlichen Semantik (1)

Sei $\sigma \in \Sigma$ mit $\sigma(x) = 3$.

Dann gilt:

$$\langle y := 1; \text{while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ od}, \sigma \rangle \longrightarrow \sigma[6/y][3/x]$$



Kap. 2.2 Natürliche Semantik

63

Kap. 3 Denotationelle Semantik von WHILE

Kap. 3 Denotationelle Semantik von WHILE

72

Denotationelle Semantik (2)

Es bezeichnen:

• $Id : \Sigma_\varepsilon \rightarrow \Sigma_\varepsilon$ die identische ZustandsTransformation:

$$\forall \sigma \in \Sigma_\varepsilon, Id(\sigma) =_{df} \sigma$$

• $Error : \Sigma_\varepsilon \rightarrow \Sigma_\varepsilon$ die konstante ZustandsTransformation mit:

$$\forall \sigma \in \Sigma_\varepsilon, Error(\sigma) =_{df} error$$

Kap. 3 Denotationelle Semantik von WHILE

74

Denotationelle Semantik (4)

Zur Hilfsfunktion FIX...

Funktionalität...

$$FIX : ((\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)) \rightarrow (\Sigma \rightarrow \Sigma)$$

Definiert durch...

$$F g = cond(\llbracket b \rrbracket_B, g \circ \llbracket \pi \rrbracket_{ds}, Id)$$

Daraus ergibt sich...

• FIX ist ein Funktional ("Zustandsstransformationsfunktion")

• Die denotationelle Semantik der while-Schleife ist ein Fixpunkt des Funktionals F (und zwar der kleinsten!)

Mehr Details zu FIX und Co. später!

Kap. 3 Denotationelle Semantik von WHILE

76

Denotationelle Semantik (6)

Zentral für denotationelle Semantiken: **Kompositionalität**:

Intuitiv:

- Für jedes Element der elementaren syntaktischen Konstrukts/Kategorien gibt es eine zugehörige semantische Funktion
- Für jedes Element eines zusammengesetzten syntaktischen Konstrukts/Kategorie gibt es eine semantische Funktion, die über die semantischen Funktionen der Komponenten des zusammengesetzten Konstrukts definiert ist.

Kap. 3 Denotationelle Semantik von WHILE

78

...auch für das Beispiel von WHILE:

$$\llbracket skip \rrbracket_{ds} = Id$$

$$\llbracket abort \rrbracket_{ds} = Error$$

$$\llbracket x := t \rrbracket_{ds}(\sigma) = \sigma[\llbracket t \rrbracket_A(\sigma)/x]$$

$$\llbracket \pi_1; \pi_2 \rrbracket_{ds} = \llbracket \pi_2 \rrbracket_{ds} \circ \llbracket \pi_1 \rrbracket_{ds}$$

$$\llbracket \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \rrbracket_{ds} = cond(\llbracket b \rrbracket_B, \llbracket \pi_1 \rrbracket_{ds}, \llbracket \pi_2 \rrbracket_{ds})$$

$$\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds} = FIX F$$

$$\text{where } F g = cond(\llbracket b \rrbracket_B, g \circ \llbracket \pi \rrbracket_{ds}, Id)$$

Kap. 3 Denotationelle Semantik von WHILE

73

Denotationelle Semantik (3)

Zur Hilfsfunktion *cond*...

Funktionalität...

$$cond : (\Sigma \rightarrow B) \times (\Sigma \rightarrow \Sigma) \times (\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)$$

Definiert durch...

$$cond(p, g_1, g_2) \sigma =_{df} \begin{cases} g_1 \sigma & \text{falls } p \sigma = \text{tt} \\ g_2 \sigma & \text{falls } p \sigma = \text{ff} \end{cases}$$

Zu den Argumenten und zum Resultat von *cond*...

- 1. Argument: Prädikat (in unserem Szenario total definiert; siehe Vorlesungsteil 1)
- 2. & 3. Argument: Je eine partiell definierte ZustandsTransformation
- Resultat: Wieder eine partiell definierte ZustandsTransformation

Kap. 3 Denotationelle Semantik von WHILE

75

Denotationelle Semantik (5)

• *Operationelle Semantik*

...der Fokus liegt darauf, wie ein Programm ausgeführt wird

• *Denotationelle Semantik*

...der Fokus liegt auf dem *Effekt*, den die Ausführung eines Programms hat: Für jedes *syntaktische* Konstrukt gibt es eine *semantische* Funktion, die erstem ein *mathematisches Objekt* zuweist, i.a. eine Funktion, die den Effekt der Ausführung des Konstrukts beschreibt (jedoch nicht, wie dieser Effekt erreicht wird).

Kap. 3 Denotationelle Semantik von WHILE

77

Denotationelle Semantik (7)

Lemma 3.1

Für alle $\pi \in \text{Prg}$ ist durch die Gleichungen von Folie "Denotationelle Semantik (1)" eine (partielle) Funktion $\llbracket \pi \rrbracket_{ds}$ definiert, die denotationelle Semantik von π .

Kap. 3 Denotationelle Semantik von WHILE

79

$$\forall \pi \in \mathbf{Prg}. \llbracket \pi \rrbracket_{\text{sos}} = \llbracket \pi \rrbracket_{\text{ns}} = \llbracket \pi \rrbracket_{\text{ds}}$$

Die Äquivalenz der strukturell operationellen, natürlichen und denotationellen Semantik von WHILE legt es nahe, den semantischen Index in der Folge fortzulassen und vereinfachend von $\llbracket \cdot \rrbracket$ als der Semantik der Sprache WHILE zu sprechen:

$$\llbracket \cdot \rrbracket : \mathbf{Prg} \rightarrow (\Sigma \rightarrow \Sigma^{\varepsilon})$$

definiert durch

$$\llbracket \cdot \rrbracket =_{df} \llbracket \cdot \rrbracket_{\text{sos}}$$

Kap. 3.1 Fixpunktfunktional

81

Kap. 3.1 Fixpunktfunktional

WHILE – Denotationelle Semantik (1)

- \mathbf{Prg} ...bezeichne die Menge aller Programme der Sprache **WHILE**

Denotationelle Semantik

$$\llbracket \cdot \rrbracket_{\text{ds}} : \mathbf{Prg} \rightarrow (\Sigma \rightarrow \Sigma^{\varepsilon})$$

Somit...

- Die *denotationelle Semantik* eines **WHILE**-Programms ist eine (partiell definierte) *Zustandstransformation*, wobei die Menge der Zustände gegeben ist durch

$$\Sigma =_{df} \{\sigma \mid \sigma : V \rightarrow D\}$$

Beachte...

- Auch die operationale (die strukturell operationelle wie auch die natürliche) Semantik eines **WHILE**-Programms ist eine (partiell definierte) *Zustandstransformation* auf Σ , nicht aber die axiomatische Semantik.

Kap. 3.1 Fixpunktfunktional

83

WHILE – Denotationelle Semantik (2)

Erinnerung:

$$\llbracket \text{skip} \rrbracket_{\text{ds}} = Id$$

$$\llbracket \text{abort} \rrbracket_{\text{ds}} = \text{Error}$$

$$\llbracket x := t \rrbracket_{\text{ds}}(\sigma) = \sigma[\llbracket t \rrbracket_{\text{ds}}(\sigma)/x]$$

$$\llbracket \pi_1; \pi_2 \rrbracket_{\text{ds}} = \llbracket \pi_2 \rrbracket_{\text{ds}} \circ \llbracket \pi_1 \rrbracket_{\text{ds}}$$

$$\llbracket \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \rrbracket_{\text{ds}} = \text{cond}(\llbracket b \rrbracket_{\text{ds}}, \llbracket \pi_1 \rrbracket_{\text{ds}}, \llbracket \pi_2 \rrbracket_{\text{ds}})$$

$$\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{\text{ds}} = \text{FIX } F$$

$$\text{where } F \cdot g = \text{cond}(\llbracket b \rrbracket_{\text{ds}}, g \circ \llbracket \pi \rrbracket_{\text{ds}}, Id)$$

Zur Bedeutung von cond

Hilfsfunktion *cond*...

Funktionalität...

$$\text{cond} : (\Sigma \rightarrow B) \times (\Sigma \rightarrow \Sigma) \times (\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)$$

Definiert durch...

$$\text{cond}(p, g_1, g_2) \sigma =_{df} \begin{cases} g_1 \sigma & \text{if } p \sigma = \text{tt} \\ g_2 \sigma & \text{if } p \sigma = \text{ff} \end{cases}$$

Zu den Argumenten und zum Resultat von *cond*...

- 1. Argument: Prädikat (in unserem Szenario total definiert; siehe Vorlesungsteil 1)
- 2.&3. Argument: Je eine partiell definierte Zustandstransformation
- Resultat: Wieder eine partiell definierte Zustandstransformation

Kap. 3.1 Fixpunktfunktional

85

Damit erhalten wir

...für die Bedeutung der Fallunterscheidung

$$\begin{aligned} & \llbracket \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \rrbracket_{\text{ds}} \sigma \\ &= \text{cond}(\llbracket b \rrbracket_{\text{ds}}, \llbracket \pi_1 \rrbracket_{\text{ds}}, \llbracket \pi_2 \rrbracket_{\text{ds}}) \sigma \\ &= \begin{cases} \sigma' & \text{falls } \llbracket b \rrbracket_{\text{ds}} \sigma = \text{tt} \wedge \llbracket \pi_1 \rrbracket_{\text{ds}} \sigma = \sigma' \\ \text{error} & \text{falls } \llbracket b \rrbracket_{\text{ds}} \sigma = \text{ff} \wedge \llbracket \pi_2 \rrbracket_{\text{ds}} \sigma = \sigma' \\ \text{undef} & \text{falls } \llbracket b \rrbracket_{\text{ds}} \sigma = \text{tt} \wedge \llbracket \pi_1 \rrbracket_{\text{ds}} \sigma = \text{error} \\ & \vee \llbracket b \rrbracket_{\text{ds}} \sigma = \text{ff} \wedge \llbracket \pi_2 \rrbracket_{\text{ds}} \sigma = \text{undef} \\ & \vee \llbracket b \rrbracket_{\text{ds}} \sigma = \text{tt} \wedge \llbracket \pi_1 \rrbracket_{\text{ds}} \sigma = \text{undef} \\ & \vee \llbracket b \rrbracket_{\text{ds}} \sigma = \text{ff} \wedge \llbracket \pi_2 \rrbracket_{\text{ds}} \sigma = \text{undef} \end{cases} \end{aligned}$$

Erinnerung:

- $\llbracket b \rrbracket_{\text{ds}}$ ist in unserem Szenario total definiert; $\llbracket b \rrbracket_{\text{ds}} \sigma$ ist daher stets von *undef* verschieden.

Kap. 3.1 Fixpunktfunktional

86

Zur Bedeutung von FIX F

Funktionalität...

$$\text{FIX} : ((\Sigma \rightarrow \Sigma) \rightarrow (\Sigma \rightarrow \Sigma)) \rightarrow (\Sigma \rightarrow \Sigma)$$

Definiert durch...

$$F \cdot g = \text{cond}(\llbracket b \rrbracket_{\text{ds}}, g \circ \llbracket \pi \rrbracket_{\text{ds}}, Id)$$

Daraus ergibt sich...

- *FIX* ist ein Funktional ("Zustandstransformationsfunktional")

- Die denotationelle Semantik der while-Schleife ist ein Fixpunkt des Funktionals *F* (und zwar der kleinste!)

Kap. 3.1 Fixpunktfunktional

87

Dazu folgende Beobachtung...

- $\text{while } b \text{ do } \pi \text{ od}$ muss dieselbe Bedeutung haben wie...
if b then $(\pi; \text{while } b \text{ do } \pi \text{ od})$ else skip fi

Daraus folgt...

- $\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds} = \text{cond}(\llbracket b \rrbracket_{ds}; \llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds} \circ \llbracket \pi \rrbracket_{ds}; Id)$
- Und daraus schließlich...

- $\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds}$ muss Fixpunkt des Funktionalis F sein, dass definiert ist durch

$$F g = \text{cond}(\llbracket b \rrbracket_{ds}; g \circ \llbracket \pi \rrbracket_{ds}; Id)$$

Oder anders ausgedrückt, es muss gelten:

$$\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds} = F(\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds})$$

...was uns wie gewünscht zu einer *kompositionellen* Definition von $\llbracket \text{while } b \text{ do } \pi \text{ od} \rrbracket_{ds}$ und damit von $\llbracket \rrbracket_{ds}$ insgesamt führen wird.

Erforderlich...

- Einige Resultate aus der *Fixpunkttheorie*

Zu tun...

- Nachzuweisen, dass diese Resultate auf unsere Situation anwendbar sind.

Anschließend bleibt nachzuholen...

- Der mathematische Hintergrund (Ordnungen, CPOs, Stetigkeit von Funktionen) und die benötigten Resultate (Fixpunktsatz)

Kap. 3.1 Fixpunktfunktional

Folgende drei Argumente...

...werden dafür entscheidend sein

1. $\llbracket \Sigma \rightarrow \Sigma \rrbracket$ kann vollständig partiell geordnet werden.
2. F im Anwendungskontext ist stetig
3. Fixpunktbildung im Anwendungskontext wird ausschließlich auf stetige Funktionen angewendet.

Insgesamt ergibt sich dann daraus die Wohldefiniertheit von

$$\llbracket \rrbracket_{ds} : \text{Prfg} \rightarrow (\Sigma \rightarrow \Sigma_\varepsilon)$$

Kap. 3.1 Fixpunktfunktional

Ordnung auf Zustandstransformationen

Sogar...

Lemma 3.1.2

Das Paar $(\llbracket \Sigma \rightarrow \Sigma \rrbracket, \subseteq)$ ist eine vollständige partielle Ordnung (CPO) mit kleinstem Element \perp .

Weiter gilt: Die kleinste obere Schranke $\sqcup Y$ einer Kette Y ist gegeben durch

$$\text{graph}(\sqcup Y) = \cup \{ \text{graph}(g) \mid g \in Y \}$$

Das heißt: $(\sqcup Y) \sigma = \sigma' \iff \exists g \in Y. g \sigma = \sigma'$

Kap. 3.1 Fixpunktfunktional

Stetigkeitsresultate (1)

Lemma 3.1.3

Sei $g_0 \in \llbracket \Sigma \rightarrow \Sigma \rrbracket$, sei $p \in \llbracket \Sigma \rightarrow \mathbf{B} \rrbracket$ und sei F definiert durch

$$F g = \text{cond}(p; g; g_0)$$

Dann gilt: F ist stetig.

Zur Erinnerung: Selen (C, \subseteq_C) und (D, \subseteq_D) zwei CPOs und sei $f : C \rightarrow D$ eine Funktion von C nach D .
Dann heißt f ...

- *monoton* gdw. $\forall c, c' \in C. c \subseteq_C c' \Rightarrow f(c) \subseteq_D f(c')$ (*Erhalt der Ordnung der Elemente*)
- *stetig* gdw. $\forall C' \subseteq C. f(\sqcup_{C'} C') = \sqcup_{D'} f(C')$ (*Erhalt der kleinsten oberen Schranken*)

Kap. 3.1 Fixpunktfunktional

Ordnung auf Zustandstransformationen

Bezeichne...

- $\llbracket \Sigma \rightarrow \Sigma \rrbracket$ die Menge der partiell definierten Zustandstransformationen.

Wir definieren...

$$g_1 \sqsubseteq g_2 \iff \forall \sigma \in \Sigma. g_1 \sigma \text{ definiert} = \sigma' \Rightarrow g_2 \sigma \text{ definiert} = \sigma' \text{ mit } g_1.g_2 \in \llbracket \Sigma \rightarrow \Sigma_\varepsilon \rrbracket$$

Lemma 3.1.1

1. $(\llbracket \Sigma \rightarrow \Sigma \rrbracket, \subseteq)$ ist eine partielle Ordnung.
2. Die *total undefinierte* (d.h. nirgends definierte) Funktion $\perp : \Sigma \rightarrow \Sigma$ mit $\perp \sigma = \text{undef}$ für alle $\sigma \in \Sigma$ ist *kleinstes* Element in $(\llbracket \Sigma \rightarrow \Sigma \rrbracket, \subseteq)$

Kap. 3.1 Fixpunktfunktional

Einschub: Graph einer Funktion

Der Graph einer totalen Funktion $f : M \rightarrow N$ ist definiert durch

$$\text{graph}(f) =_{df} \{ (m, n) \in M \times N \mid f m = n \}$$

Es gilt:

- $(m, n) \in \text{graph}(f) \wedge (m, n') \in \text{graph}(f) \Rightarrow n = n'$ (*rechtsindeutig*)
- $\forall m \in M. \exists n \in N. (m, n) \in \text{graph}(f)$ (*links-total*)

Der Graph einer partiellen Funktion $f : M \rightarrow N$ mit Definitionsbereich $M_f \subseteq M$ ist definiert durch

$$\text{graph}(f) =_{df} \{ (m, n) \in M \times N \mid f m = n \wedge m \in M_f \}$$

Vereinbarung...

Für $f : M \rightarrow N$ partiell definierte Funktion auf $M_f \subseteq M$ schreiben wir

- $f m = n$, falls $(m, n) \in \text{graph}(f)$
- $f m = \text{undef}$, falls $m \notin M_f$

Kap. 3.1 Fixpunktfunktional

Stetigkeitsresultate (2)

Lemma 3.1.4

Sei $g_0 \in \llbracket \Sigma \rightarrow \Sigma \rrbracket$ und sei F definiert durch

$$F g = g \circ g_0$$

Dann gilt: F ist stetig.

Kap. 3.1 Fixpunktfunktional

Lemma 3.1.5

Die Gleichungen zur Festlegung der denotationellen Semantik von WHILE (vgl. Folie 15 von heute) definieren eine totale Funktion

$$\llbracket _ \rrbracket_{ds} \in [\mathbf{Prg} \rightarrow (\Sigma \rightarrow \Sigma_\varepsilon)]$$

...sind wir durch! Wir können beweisen:

$$\llbracket _ \rrbracket_{ds} : \mathbf{Prg} \rightarrow (\Sigma \rightarrow \Sigma_\varepsilon)$$

ist wohldefiniert!

Kap. 3.1 Fixpunktfunktional

96

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

98

Wichtig insbesondere...

- Mengen, Relationen, Verbände
- Partielle und vollständige partielle Ordnungen
- Schranken, Fixpunkte und Fixpunktheoreme

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

100

Mengen und Relationen 2(2)

Eine Relation R auf M heißt

- *Quasiordnung* gdw. R ist reflexiv und transitiv
- *partielle Ordnung* gdw. R ist reflexiv, transitiv und antisymmetrisch

Zur Vollständigkeit sei ergänzt...

- *Äquivalenzrelation* gdw. R ist reflexiv, transitiv und symmetrisch

...: eine partielle Ordnung ist also eine antisymmetrische Quasiordnung, eine Äquivalenzrelation eine symmetrische Quasiordnung.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

102

Aus...

1. Die Menge $[\Sigma \rightarrow \Sigma]$ der partiell definierten Zustandsstransformationen bildet zusammen mit der Ordnung \sqsubseteq eine CPO.
2. Funktional F mit " $F g = \text{cond}(p, g, g_0)$ " und " $g \circ g_0$ " ist stetig
3. In der Definition von $\llbracket _ \rrbracket_{ds}$ wird die Fixpunktbildung ausschließlich auf stetige Funktionen angewendet.

...ergibt sich wie gewünscht:

$$\llbracket _ \rrbracket_{ds} : \mathbf{Prg} \rightarrow (\Sigma \rightarrow \Sigma_\varepsilon)$$

...ist wohldefiniert!

Kap. 3.1 Fixpunktfunktional

97

Mathematische Grundlagen

Im Zusammenhang mit der...

1. Definition abstrakter Semantiken für Programmanalysen
2. Definition der denotationellen Semantik von WHILE im Detail

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

99

Mengen und Relationen 1(2)

Sei M eine Menge und R eine Relation auf M , d.h. $R \subseteq M \times M$.

Dann heißt R ...

- *reflexiv* gdw. $\forall m \in M. m R m$
- *transitiv* gdw. $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow m R p$
- *antisymmetrisch* gdw. $\forall m, n \in M. m R n \wedge n R m \Rightarrow m = n$

Darüberhinaus... (in der Folge allerdings weniger wichtig)

- *symmetrisch* gdw. $\forall m, n \in M. m R n \Leftrightarrow n R m$
- *total* gdw. $\forall m, n \in M. m R n \vee n R m$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

101

Schranken, kleinste, größte Elemente

Sei (Q, \sqsubseteq) eine Quasiordnung, sei $q \in Q$ und $Q' \sqsubseteq Q$.

Dann heißt q ...

- *obere (untere) Schranke* von Q' , in Zeichen: $Q' \sqsubseteq q$ ($q \sqsubseteq Q'$), wenn für alle $q' \in Q'$ gilt: $q' \sqsubseteq q$ ($q \sqsubseteq q'$)
- *kleinste obere (größte untere) Schranke* von Q' , wenn q obere (untere) Schranke von Q' ist und für jede andere obere (untere) Schranke \hat{q} von Q' gilt: $q \sqsubseteq \hat{q}$ ($\hat{q} \sqsubseteq q$)
- *größtes (kleinstes) Element* von Q , wenn gilt: $Q \sqsubseteq q$ ($q \sqsubseteq Q$)

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

103

- In partiellen Ordnungen sind kleinste obere und größte untere Schranken eindeutig bestimmt, wenn sie existieren.
- Existenz (und damit Eindeutigkeit) vorausgesetzt, wird die kleinste obere (größte untere) Schranke einer Menge $P' \subseteq P$ der Grundmenge einer partiellen Ordnung (P, \sqsubseteq) mit $\sqcup P'$ ($\sqcap P'$) bezeichnet. Man spricht dann auch vom *Supremum* und *Infimum* von P' .
- Analog für kleinste und größte Elemente: Existenz vorausgesetzt, werden sie üblicherweise mit \perp und \top bezeichnet.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

104

Vollständige partielle Ordnungen

...ein etwas schwächerer, aber in der Informatik oft ausreichender und daher angemessener Begriff.

Sei (P, \sqsubseteq) eine partielle Ordnung.

Dann heißt (P, \sqsubseteq) ...

- *vollständig*, kurz *CPO* (von engl. complete partial order), wenn jede aufsteigende Kette $K \subseteq P$ eine kleinste obere Schranke in P besitzt.

Es gilt:

- Eine CPO (C, \sqsubseteq) (genauer wäre: "kettenvollständige partielle Ordnung (engl. chain complete partial order (CCPO)")) besitzt stets ein kleinstes Element, eindeutig bestimmt als Supremum der leeren Kette und üblicherweise mit \perp bezeichnet; $\perp =_{df} \sqcup \emptyset$.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

106

Kettenendlichkeit, endliche Elemente

Eine partielle Ordnung (P, \sqsubseteq) heißt

- *kettenendlich* gdw. P enthält keine unendlichen Ketten

Ein Element $p \in P$ heißt

- *endlich* gdw. die Menge $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ keine unendliche Kette enthält
- *endlich relativ zu* $r \in P$ gdw. die Menge $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ keine unendliche Kette enthält

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

108

(Standard-) CPO-Konstruktionen 2(4)

Produktkonstruktionen...

Seien $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ CPOs. Dann sind auch...

- das *nichtstrikte (direkte) Produkt* $(\times P_i, \sqsubseteq)$ mit
 - $(\times P_i, \sqsubseteq) = (P_1 \times P_2 \times \dots \times P_n, \sqsubseteq)$ mit $\forall (p_1, p_2, \dots, p_n); (q_1, q_2, \dots, q_n) \in \times P_i, (p_1, p_2, \dots, p_n) \sqsubseteq (q_1, q_2, \dots, q_n) \Rightarrow \forall i \in \{1, \dots, n\}: p_i \sqsubseteq_i q_i$
- und das *strikte (direkte) Produkt (smash Produkt)* mit
 - $(\otimes P_i, \sqsubseteq) = (P_1 \otimes P_2 \otimes \dots \otimes P_n, \sqsubseteq)$, wobei \sqsubseteq wie oben definiert ist, jedoch zusätzlich gesetzt wird:
 - $(p_1, p_2, \dots, p_n) = \perp \Rightarrow \exists i \in \{1, \dots, n\}: p_i = \perp_i$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

110

Sei (P, \sqsubseteq) eine partielle Ordnung.
Dann heißt (P, \sqsubseteq) ...

- *Verband*, wenn jede *endliche* Teilmenge P' von P eine kleinste obere und eine größte untere Schranke in P besitzt
 - *vollständiger Verband*, wenn *jede* Teilmenge P' von P eine kleinste obere und eine größte untere Schranke in P besitzt
- ... (vollständige) Verbände sind also spezielle partielle Ordnungen.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

105

Ketten

Sei (P, \sqsubseteq) eine partielle Ordnung.

Eine Teilmenge $K \subseteq P$ heißt...

- *Kette* in P , wenn die Elemente in K total geordnet sind. Für $K = \{k_0 \sqsubseteq k_1 \sqsubseteq k_2 \sqsubseteq \dots\}$ ($\{k_0 \supseteq k_1 \supseteq k_2 \supseteq \dots\}$) spricht man auch genauer von einer *aufsteigenden (absteigenden)* Kette in P .

Eine Kette K heißt...

- *endlich*, wenn K endlich ist, sonst *unendlich*.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

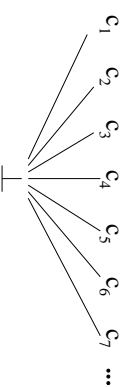
107

(Standard-) CPO-Konstruktionen 1(4)

Flache CPOs...

Sei (C, \sqsubseteq) eine CPO. Dann heißt (C, \sqsubseteq) ...

- *flach*, wenn für alle $c, d \in C$ gilt: $c \sqsubseteq d \Leftrightarrow c = \perp \vee c = d$



Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

109

(Standard-) CPO-Konstruktionen 3(4)

Summenkonstruktion...

Seien $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ CPOs. Dann ist auch...

- die *direkte Summe* $(\oplus P_i, \sqsubseteq)$ mit...
 - $(\oplus P_i, \sqsubseteq) = (P_1 \dot{\cup} P_2 \dot{\cup} \dots \dot{\cup} P_n, \sqsubseteq)$ disjunkte Vereinigung der $P_i, i \in \{1, \dots, n\}$ und $\forall p, q \in \oplus P_i, p \sqsubseteq q \Rightarrow \exists i \in \{1, \dots, n\}: p, q \in P_i \wedge p \sqsubseteq_i q$ und der Identifikation der kleinsten Elemente der $(P_i, \sqsubseteq_i), i \in \{1, \dots, n\}$, d.h. $\perp =_{df} \perp_{P_i}, i \in \{1, \dots, n\}$
- eine CPO.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

111

Funktionsraum...

Seien (C, \sqsubseteq_C) und (D, \sqsubseteq_D) zwei CPOs und $[C \rightarrow D] = \{f : C \rightarrow D \mid f \text{ stetig}\}$ die Menge der stetigen Funktionen von C nach D .

Dann ist auch...

- der stetige Funktionsraum $([C \rightarrow D], \sqsubseteq)$ eine CPO mit
 - $\forall f, g \in [C \rightarrow D]. f \sqsubseteq g \iff \forall c \in C. f(c) \sqsubseteq_D g(c)$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

112

Funktionen auf CPOs / Resultate

Mit den vorigen Bezeichnungen gilt...

Lemma

f ist monoton gdw. $\forall C' \subseteq C. f(\bigsqcup_{C'} C') \supseteq_D \bigsqcup_{D'} f(C')$

Korollar

Eine stetige Funktion ist stets monoton, d.h. f stetig $\Rightarrow f$ monoton.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

114

(Kleinste und größte) Fixpunkte 2(2)

Seien $d, c_d \in C$. Dann heißt $c_d \dots$

- *bedingter kleinster Fixpunkt* von f bezüglich d gdw. c_d ist der kleinste Fixpunkt von C mit $d \sqsubseteq c_d$, d.h. für alle anderen Fixpunkte x von f mit $d \sqsubseteq x$ gilt: $c_d \sqsubseteq x$.

Bezeichnungen:

Der kleinste bzw. größte Fixpunkt einer Funktion f wird oft mit μf bzw. νf bezeichnet.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

116

Beweis des Fixpunktsatzes 3.2.1 1(4)

Zu zeigen: $\mu f \dots$

1. existiert
2. ist Fixpunkt
3. ist kleinster Fixpunkt

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

118

Seien (C, \sqsubseteq_C) und (D, \sqsubseteq_D) zwei CPOs und sei $f : C \rightarrow D$ eine Funktion von C nach D .

Dann heißt $f \dots$

- *monoton* gdw. $\forall c, d \in C. c \sqsubseteq_C d \Rightarrow f(c) \sqsubseteq_D f(d)$
(*Erhalt der Ordnung der Elemente*)
- *stetig* gdw. $\forall C' \subseteq C. f(\bigsqcup_{C'} C') =_D \bigsqcup_{D'} f(C')$
(*Erhalt der kleinsten oberen Schranken*)

Sei (C, \sqsubseteq) eine CPO und sei $f : C \rightarrow C$ eine Funktion auf C . Dann heißt $f \dots$

- *inflationär* (*Vergrößernd*) gdw. $\forall c \in C. c \sqsubseteq f(c)$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

113

(Kleinste und größte) Fixpunkte 1(2)

Sei (C, \sqsubseteq) eine CPO, $f : C \rightarrow C$ eine Funktion auf C und sei c ein Element von C , also $c \in C$.

Dann heißt $c \dots$

- *Fixpunkt* von f gdw. $f(c) = c$
- *Ein Fixpunkt* c von f heißt...
- *kleinster Fixpunkt* von f gdw. $\forall d \in C. f(d) = d \Rightarrow c \sqsubseteq d$
- *größter Fixpunkt* von f gdw. $\forall d \in C. f(d) = d \Rightarrow d \sqsubseteq c$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

115

Fixpunktsatz

Theorem 3.2.1 (Knaster/Tarski, Kleene)

Sei (C, \sqsubseteq) eine CPO und sei $f : C \rightarrow C$ eine stetige Funktion auf C .

Dann hat f einen kleinsten Fixpunkt μf und dieser Fixpunkt ergibt sich als kleinste obere Schranke der Kette (sog. *Kleene-Kette*) $\{\perp, f(\perp), f^2(\perp), \dots\}$, d.h.

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{\perp, f(\perp), f^2(\perp), \dots\}$$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

117

Beweis des Fixpunktsatzes 3.2.1 2(4)

1. *Existenz*

- Es gilt $f^0 \perp = \perp$ und $\perp \sqsubseteq c$ für alle $c \in C$.
- Durch vollständige Induktion lässt sich damit zeigen: $f^n \perp \sqsubseteq f^m c$ für alle $c \in C$.
- Somit gilt $f^n \perp \sqsubseteq f^m \perp$ für alle n, m mit $n \leq m$. Somit ist $\{f^n \perp \mid n \geq 0\}$ eine (nichtleere) Kette in C .
- Damit folgt die Existenz von $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ aus der CPO-Eigenschaft von (C, \sqsubseteq) .

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

119

2. Fixpunkteigenschaft

$$\begin{aligned} f(\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)) &= \bigsqcup_{i \in \mathbb{N}_0} f(f^i(\perp)) \\ (f \text{ stetig}) &= \bigsqcup_{i \in \mathbb{N}_0} f^{i+1}(\perp) \\ &= \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp) \\ (K \text{ Kette } \Rightarrow \bigsqcup K = \perp \sqcup \bigsqcup K) &= \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp) \sqcup \perp \\ (f^0 = \perp) &= \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \\ &= \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \end{aligned}$$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

120

3. Kleinstes Fixpunkt

- Sei c beliebig gewählter Fixpunkt von f . Dann gilt $\perp \sqsubseteq c$ und somit auch $f^n \perp \sqsubseteq f^n c$ für alle $n \geq 0$.
- Folglich gilt $f^n \perp \sqsubseteq c$ wg. der Wahl von c als Fixpunkt von f .
- Somit gilt auch, dass c eine obere Schranke von $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$ ist.
- Da $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ nach Definition die kleinste obere Schranke dieser Kette ist, gilt wie gewünscht $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c$.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

121

Bedingte Fixpunkte

Theorem 3.2.2 (Endliche Fixpunkte)

Sei (C, \sqsubseteq) eine CPO, sei $f : C \rightarrow C$ eine stetige, inflationäre Funktion auf C und sei $d \in C$.

Dann hat f einen kleinsten bedingten Fixpunkt μf_d und dieser Fixpunkt ergibt sich als kleinste obere Schranke der Kette $\{d, f(d), f^2(d), \dots\}$, d.h.

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \dots\}$$

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

122

Endliche Fixpunkte

Theorem 3.3.3 (Endliche Fixpunkte)

Sei (C, \sqsubseteq) eine CPO und sei $f : C \rightarrow C$ eine stetige Funktion auf C .

Dann gilt: Sind in der Kleine-Kette von f zwei aufeinanderfolgende Glieder gleich, etwa $f^i(\perp) = f^{i+1}(\perp)$, so gilt $\mu f = f^i(\perp)$.

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

123

Existenz endlicher Fixpunkte

Hinreichende Bedingungen für die Existenz endlicher Fixpunkte sind ...

- Endlichkeit von Definitions- und Wertebereich von f
- f ist von der Form $f(c) = c \sqcup g(c)$ für monotonen g über kettenendlichem Wertebereich

Kap. 3.2 Mengen, Relationen, Ordnungen und Verbände

124

Axiomatische Semantik

Insbesondere: ...Korrektheit und Vollständigkeit der axiomatischen Semantik

Erinnerung:

- *Hoare-Tripel* (syntaktische Sicht) bzw. *Korrektheitsformeln* (semantische Sicht) der Form $\{p\} \pi \{q\}$ bzw. $[p] \pi [q]$
- Gültigkeit einer Korrektheitsformel im Sinne
 - *partieller* Korrektheit
 - *totaler* Korrektheit

Kap. 4 Axiomatische Semantik von WHILE

126

Kapitel 4 Axiomatische Semantik von WHILE

Kap. 4 Axiomatische Semantik von WHILE

125

Kapitel 4.1 Partielle und totale Korrektheit

Kap. 4.1 Partielle und totale Korrektheit

127

Sei $\pi \in \text{Prig}$ ein WHILE-Programm:

Eine Hoaresche Zusicherung $\{p\} \pi \{q\}$ heißt

- *gültig* (im Sinne der **partiellen Korrektheit**) oder kurz (*partiell*) **korrekt** gdw. für jeden Anfangszustand σ gilt: ist die Vorbedingung p in σ erfüllt **und** terminiert die zugehörige Berechnung von π angesetzt auf σ regulär in einem Endzustand σ' , **dann** ist auch die Nachbedingung q in σ' erfüllt.

Kap. 4.1 Partielle und totale Korrektheit

128

Definition totaler Korrektheit

Sei $\pi \in \text{Prig}$ ein WHILE-Programm:

Eine Hoaresche Zusicherung $[p] \pi [q]$ heißt

- *gültig* (im Sinne der **totalen Korrektheit**) oder kurz (*total*) **korrekt** gdw. für jeden Anfangszustand σ gilt: ist die Vorbedingung p in σ erfüllt, **dann** terminiert die zugehörige Berechnung von π angesetzt auf σ regulär mit einem Endzustand σ' **und** die Nachbedingung q ist in σ' erfüllt.

Kap. 4.1 Partielle und totale Korrektheit

129

Partielle und totale Korrektheit

- Die Zustandsmenge

$$Ch(p) =_{df} \{\sigma \in \Sigma \mid \llbracket p \rrbracket_B(\sigma) = \text{tt}\}$$

heißt *Charakterisierung* von p in **Bexp**.

- *Semantik von Korrektheitsformeln*:

Eine Korrektheitsformel $\{p\} \pi \{q\}$ heißt

- *partiell korrekt* (in Zeichen: $\models_{pk} \{p\} \pi \{q\}$), falls $\llbracket \pi \rrbracket(Ch(p)) \subseteq Ch(q)$
- *total korrekt* (in Zeichen: $\models_{tk} \{p\} \pi \{q\}$), falls $\{p\} \pi \{q\}$ partiell korrekt ist und $Def(\llbracket \pi \rrbracket) \supseteq Ch(p)$ gilt. Dabei bezeichnet $Def(\llbracket \pi \rrbracket)$ die Menge aller Zustände, für die π regulär terminiert.

Konvention: $\llbracket \pi \rrbracket(Ch(p)) =_{df} \{\llbracket \pi \rrbracket(\sigma) \mid \sigma \in Ch(p)\}$

Kap. 4.1 Partielle und totale Korrektheit

131

Intuitiv

“Totale Korrektheit = Partielle Korrektheit + Terminierung”

Kap. 4.1 Partielle und totale Korrektheit

130

Erinnerung

...an einige Sprechweisen:

Ein (deterministisches) Programm π

- angesetzt auf einen Anfangszustand σ **terminiert regulär** gdw. π nach endlich vielen Schritten in einem Zustand σ' $\in \Sigma$ endet.
- angesetzt auf einen Anfangszustand σ **terminiert irregulär** gdw. π nach endlich vielen Schritten zur Konfiguration *undef* führt.
- Ein Programm π heißt **divergent** gdw. π terminiert für keinen Anfangszustand regulär.

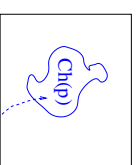
Kap. 4.1 Partielle und totale Korrektheit

132

Veranschaulichung (1)

...der Charakterisierung $Ch(p)$ einer logischen Formel p :

Menge aller Zustände Σ



Charakterisierung von p : $Ch(p) \subseteq \Sigma$

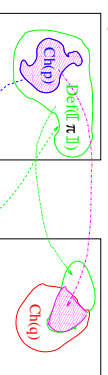
Kap. 4.1 Partielle und totale Korrektheit

133

Veranschaulichung (3)

...der Gültigkeit eine Hoareschen Zusicherung $[p] \pi [q]$ im Sinne totaler Korrektheit:

Menge aller Zustände Σ



Charakterisierung von p : $Ch(p) \subseteq \Sigma$

Definitionsbereich von π : $Def(\llbracket \pi \rrbracket) \subseteq \Sigma$

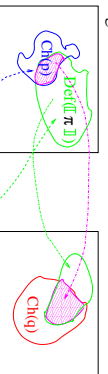
Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket)$

Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket) \cap Ch(p)$

Veranschaulichung (2)

...der Gültigkeit eine Hoareschen Zusicherung $\{p\} \pi \{q\}$ im Sinne partieller Korrektheit:

Menge aller Zustände Σ



Charakterisierung von p : $Ch(p) \subseteq \Sigma$

Definitionsbereich von π : $Def(\llbracket \pi \rrbracket) \subseteq \Sigma$

Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket)$

Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket) \cap Ch(p)$

Sei $\pi \in \text{Prig}$ ein WHILE-Programm:

Eine Hoaresche Zusicherung $[p] \pi [q]$ heißt

- *gültig* (im Sinne der **totalen Korrektheit**) oder kurz (*total*) **korrekt** gdw. für jeden Anfangszustand σ gilt: ist die Vorbedingung p in σ erfüllt, **dann** terminiert die zugehörige Berechnung von π angesetzt auf σ regulär mit einem Endzustand σ' **und** die Nachbedingung q ist in σ' erfüllt.

Kap. 4.1 Partielle und totale Korrektheit

129

Partielle und totale Korrektheit

- Die Zustandsmenge

$$Ch(p) =_{df} \{\sigma \in \Sigma \mid \llbracket p \rrbracket_B(\sigma) = \text{tt}\}$$

heißt *Charakterisierung* von p in **Bexp**.

- *Semantik von Korrektheitsformeln*:

Eine Korrektheitsformel $\{p\} \pi \{q\}$ heißt

- *partiell korrekt* (in Zeichen: $\models_{pk} \{p\} \pi \{q\}$), falls $\llbracket \pi \rrbracket(Ch(p)) \subseteq Ch(q)$
- *total korrekt* (in Zeichen: $\models_{tk} \{p\} \pi \{q\}$), falls $\{p\} \pi \{q\}$ partiell korrekt ist und $Def(\llbracket \pi \rrbracket) \supseteq Ch(p)$ gilt. Dabei bezeichnet $Def(\llbracket \pi \rrbracket)$ die Menge aller Zustände, für die π regulär terminiert.

Konvention: $\llbracket \pi \rrbracket(Ch(p)) =_{df} \{\llbracket \pi \rrbracket(\sigma) \mid \sigma \in Ch(p)\}$

Kap. 4.1 Partielle und totale Korrektheit

131

Erinnerung

...an einige Sprechweisen:

Ein (deterministisches) Programm π

- angesetzt auf einen Anfangszustand σ **terminiert regulär** gdw. π nach endlich vielen Schritten in einem Zustand σ' $\in \Sigma$ endet.
- angesetzt auf einen Anfangszustand σ **terminiert irregulär** gdw. π nach endlich vielen Schritten zur Konfiguration *undef* führt.
- Ein Programm π heißt **divergent** gdw. π terminiert für keinen Anfangszustand regulär.

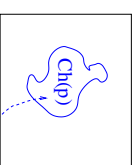
Kap. 4.1 Partielle und totale Korrektheit

132

Veranschaulichung (1)

...der Charakterisierung $Ch(p)$ einer logischen Formel p :

Menge aller Zustände Σ



Charakterisierung von p : $Ch(p) \subseteq \Sigma$

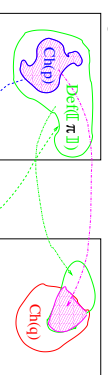
Kap. 4.1 Partielle und totale Korrektheit

133

Veranschaulichung (3)

...der Gültigkeit eine Hoareschen Zusicherung $[p] \pi [q]$ im Sinne totaler Korrektheit:

Menge aller Zustände Σ



Charakterisierung von p : $Ch(p) \subseteq \Sigma$

Definitionsbereich von π : $Def(\llbracket \pi \rrbracket) \subseteq \Sigma$

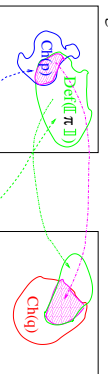
Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket)$

Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket) \cap Ch(p)$

Veranschaulichung (2)

...der Gültigkeit eine Hoareschen Zusicherung $\{p\} \pi \{q\}$ im Sinne partieller Korrektheit:

Menge aller Zustände Σ



Charakterisierung von p : $Ch(p) \subseteq \Sigma$

Definitionsbereich von π : $Def(\llbracket \pi \rrbracket) \subseteq \Sigma$

Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket)$

Bild von $\llbracket \pi \rrbracket$ für $Def(\llbracket \pi \rrbracket) \cap Ch(p)$

Stärkste Nachbedingungen, schwächste Vorbedingungen

In der Folge:

Präzisierung von...

- Stärkste Nachbedingungen
- Schwächste Vorbedingungen

Kap. 4.1 Partielle und totale Korrektheit

136

Stärkste Nach- und schwächste Vorbedingungen (2)

Lemma 4.1.1

Ist $\llbracket \pi \rrbracket$ total definiert, d.h. gilt $Def(\llbracket \pi \rrbracket) = \Sigma$, dann gilt für alle Formeln p und q :

$$\llbracket \pi \rrbracket(Ch(p)) \subseteq Ch(q) \iff \llbracket \pi \rrbracket^{-1}(Ch(q)) \supseteq Ch(p)$$

Beweis: Übungsaufgabe

Kap. 4.1 Partielle und totale Korrektheit

138

Schwächste Vor- und stärkste Nachbedingungen

...noch einmal anders betrachtet:

Definition

Seien A, B, A_1, A_2, \dots (logische) Formeln

- A heißt *schwächer* als B , wenn gilt: $B \Rightarrow A$
- A_1 heißt *schwächste* Formel in $\{A_1, A_2, \dots\}$, wenn gilt: $A_j \Rightarrow A_1$ für alle j .

Kap. 4.1 Partielle und totale Korrektheit

140

Stärkste Nachbedingungen (1)

Analog zu A ist *schwächer* als B lässt sich definieren:

- A heißt *stärker* als B , wenn gilt: B ist schwächer als A , d.h. wenn gilt: $A \Rightarrow B$
- A_1 heißt *stärkste* Formel in $\{A_1, A_2, \dots\}$, wenn gilt: $A_i \Rightarrow A_1$ für alle j .

Zum Überlegen:

Ist es sinnvoll, den Begriff der stärksten (liberalen) Nachbedingung $spo(p, \pi)$ bzw. $slpo(p, \pi)$ "in genau gleicher Weise" zum Begriff der schwächsten (liberalen) Vorbedingung $wvp(\pi, q)$ bzw. $wlp(\pi, q)$ zu gegebenem Program π und Vorbedingung p zu betrachten?

Kap. 4.1 Partielle und totale Korrektheit

142

umgungen (1)

In der Situation der vorigen Abbildungen gilt:

- $\llbracket \pi \rrbracket(Ch(p))$ heißt *stärkste Nachbedingung* von π bezüglich p .
- $\llbracket \pi \rrbracket^{-1}(Ch(q))$ heißt *schwächste Vorbedingung* von π bezüglich q , wobei $\llbracket \pi \rrbracket^{-1}(\Sigma') =_{df} \{\sigma \in \Sigma \mid \llbracket \pi \rrbracket(\sigma) \in \Sigma'\}$
- $\llbracket \pi \rrbracket^{-1}(Ch(q)) \cup C(Def(\llbracket \pi \rrbracket))$ heißt *schwächste liberale Vorbedingung* von π bezüglich q , wobei C den Mengenkomplementoperator (bzgl. der Grundmenge Σ) bezeichnet.

Kap. 4.1 Partielle und totale Korrektheit

137

Partielle vs. totale Korrektheit

Lemma 4.1.2

Für deterministische Programme π gilt:

$$[p] \pi [q] \Rightarrow \{p\} \pi \{q\}$$

d.h. für deterministische Programme impliziert totale Korrektheit bzgl. eines Paares aus Vor- und Nachbedingung auch partielle Korrektheit bzgl. dieses Paares aus Vor- und Nachbedingung.

Kap. 4.1 Partielle und totale Korrektheit

139

Schwächste Vorbedingungen

Definition

Sei π ein Programm und q eine Formel.

Dann heißt

- $wvp(\pi, q)$ *schwächste Vorbedingung* für totale Korrektheit von π bezüglich (der Nachbedingung) q , wenn
$$[wvp(\pi, q)] \pi [q]$$
 total korrekt ist und $wvp(\pi, q)$ die schwächste Formel mit dieser Eigenschaft ist.
- $wlp(\pi, q)$ *schwächste liberale Vorbedingung* für partielle Korrektheit von π bezüglich (der Nachbedingung) q , wenn
$$\{wlp(\pi, q)\} \pi \{q\}$$
 partiell korrekt ist und $wlp(\pi, q)$ die schwächste Formel mit dieser Eigenschaft ist.

Stärkste Nachbedingungen (2)

Betrachte...

Definition(sversuch)

Sei π ein Programm und p eine Formel.

Dann heißt

- $spo(p, \pi)$ *stärkste Nachbedingung* für totale Korrektheit von π bezüglich (der Vorbedingung) p , wenn
$$[p] \pi [spo(p, \pi)]$$
 total korrekt ist und $spo(p, \pi)$ die stärkste Formel mit dieser Eigenschaft ist.
- $slpo(p, \pi)$ *stärkste liberale Nachbedingung* für partielle Korrektheit von π bezüglich (der Vorbedingung) p , wenn
$$\{p\} \pi \{slpo(p, \pi)\}$$
 partiell korrekt ist und $slpo(p, \pi)$ die stärkste Formel mit dieser Eigenschaft ist.

Fragen

- Gibt es Programme π und Formeln p derart, dass
 - $spo(p, \pi)$
 - $slpo(p, \pi)$unterscheidbar, d.h. logisch nicht äquivalent sind?
- Wie passen die hier betrachteten Begriffe von schwächsten Vor- und stärksten Nachbedingungen mit denen auf Folie 13 von diesem Vorlesungsteil betrachteten zusammen?

Kap. 4.1 Partielle und totale Korrektheit

144

Kapitel 4.2 Beweiskalkül für partielle Korrektheit

Kap. 4.2 Beweiskalkül für partielle Korrektheit

145

Hoare-Kalkül HK_{PK} für partielle Korrektheit

- [skip] $\frac{}{\overline{\{p\} skip \{q\}}}$
- [abort] $\frac{}{\overline{\{p\} abort \{q\}}}$
- [ass] $\frac{\overline{\{p\} \{x:=t\}}}{\overline{\{p\} \{x:=t\} \{q\}}}$
- [comp] $\frac{\overline{\{p\} \pi_1 \{q\}} \quad \overline{\{p\} \pi_2 \{q\}}}{\overline{\{p\} \pi_1; \pi_2 \{q\}}}$
- [ite] $\frac{\overline{\{p \wedge b\} \pi_1 \{q\}} \quad \overline{\{p \wedge \neg b\} \pi_2 \{q\}}}{\overline{\{p\} \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \text{ fi } \{q\}}}$
- [while] $\frac{\overline{\{I \wedge b\} \pi \{I\}}}{\overline{\{I\} \text{while } b \text{ do } \pi \text{ od } \{I \wedge \neg b\}}}$
- [cons] $\frac{p \Rightarrow p_1. \{p_1\} \pi \{q\}_1. q_1 \Rightarrow q}{\overline{\{p\} \pi \{q\}}}$

Kap. 4.2 Beweiskalkül für partielle Korrektheit

146

- ### Diskussion von Vorwärtszuweisungsregel(n)
- Eine Vorwärtsregel für die Zuweisung wie
$$[\text{ass } fwd] \quad \frac{}{\overline{\{p\} x:=t \{p[t/x]\}}}$$
mag natürlich erscheinen, ist aber beweistechnisch unangenehm durch das Mitschleppen quantifizierter Formeln.
 - Beachte: Folgende scheinbar naheliegende quantorfreie Realisierung der Vorwärtszuweisungsregel ist nicht korrekt:
$$[\text{ass } nativ] \quad \frac{}{\overline{\{p\} x:=t \{p[t/x]\}}}$$
Beweis: Übungsaufgabe

Kap. 4.2 Beweiskalkül für partielle Korrektheit

147

Kapitel 4.3 Beweiskalkül für totale Korrektheit

Kap. 4.3 Beweiskalkül für totale Korrektheit

148

Zur Vollständigkeit

...sien die übrigen Regeln des Hoare-Kalkül HK_{TK} für totale Korrektheit hier ebenfalls angeben:

- [skip] $\frac{}{\overline{\{p\} skip \{p\}}}$
- [ass] $\frac{\overline{\{p\} \{x:=t\}}}{\overline{\{p\} \{x:=t\} \{q\}}}$
- [comp] $\frac{\overline{\{p\} \pi_1 \{q\}} \quad \overline{\{p\} \pi_2 \{q\}}}{\overline{\{p\} \pi_1; \pi_2 \{q\}}}$
- [ite] $\frac{\overline{\{p \wedge b\} \pi_1 \{q\}} \quad \overline{\{p \wedge \neg b\} \pi_2 \{q\}}}{\overline{\{p\} \text{if } b \text{ then } \pi_1 \text{ else } \pi_2 \text{ fi } \{q\}}}$
- [cons] $\frac{p \Rightarrow p_1. \{p_1\} \pi \{q\}_1. q_1 \Rightarrow q}{\overline{\{p\} \pi \{q\}}}$

Zum Überlegen: Warum fehlt eine Regel für abort?

Kap. 4.3 Beweiskalkül für totale Korrektheit

150

Hoare-Kalkül HK_{TK} für totale Korrektheit

...identisch mit HK_{PK} , wobei aber Regel [while] ersetzt ist durch:

$$[\text{while}_{TK}] \quad \frac{I \wedge b \Rightarrow u[t/a]. \{I \wedge b \wedge I = w\} \pi \{I \wedge K = w\}}{\overline{\{I\} \text{while } b \text{ do } \pi \text{ od } \{I \wedge \neg b\}}}$$

wobei

- u Boolescher Ausdruck über der Variablen v_i ,
- t Term,
- w Variable, die in I, b, π und t nicht frei vorkommt,
- $M =_{df} \{\sigma(v) \mid \sigma \in \Sigma \wedge \llbracket u \rrbracket_{\mathcal{B}}(\sigma) = \text{tt}\}$ noethersch geordnete Menge (sog. noethersche Halbordnung).
 \rightsquigarrow *Terminationsordnung!*

Kap. 4.3 Beweiskalkül für totale Korrektheit

149

Bemerkung

In den vorigen Regeln verwenden wir geschweifte statt eckige Klammern für zugesicherte Eigenschaften, um einen Bezeichnungskonflikt mit der ebenfalls durch eckige Klammern bezeichneten *syntaktischen Substitution* zu vermeiden.

Kap. 4.3 Beweiskalkül für totale Korrektheit

151

Wohlfundierte oder Noethersche Ordnungen (1)

Definition

Sei P eine Menge und sei $<$ eine irreflexive und transitive Relation auf P .

Dann ist das Paar $(P, <)$ eine *irreflexive partielle Ordnung*.

Beispiele: $(\mathbb{Z}, <)$, $(\mathbb{Z}, >)$, $(\mathbb{N}, <)$, $(\mathbb{N}, >)$

Wohlfundierte oder Noethersche Ordnungen (3)

Konstruktionsprinzipien für wohlfundierte Ordnungen aus gegebenen wohlfundierten Ordnungen...

Lemma 4.3.1

Seien $(W_1, <_1)$ und $(W_2, <_2)$ zwei wohlfundierte Ordnungen. Dann sind auch

- $(W_1 \times W_2, <_{com})$ mit *kompONENTENWEISER* Ordnung definiert durch

$$(m_1, m_2) <_{com} (n_1, n_2) \text{ gdw. } m_1 <_1 n_1 \wedge m_2 <_2 n_2$$
- $(W_1 \times W_2, <_{lex})$ mit *LEXIKOGRAPHISCHER* Ordnung def. durch

$$(m_1, m_2) <_{lex} (n_1, n_2) \text{ gdw. } (m_1 <_1 n_1) \vee (m_1 = n_1 \wedge m_2 <_2 n_2)$$

wohlfundierte Ordnungen.

Zur Konsequenzregel (1)

$$[cons] \frac{p \Rightarrow p_1, \{p_1\} \pi \{q_1\}, q_1 \Rightarrow q}{\{p\} \pi \{q\}}$$

Intuitiv:

Die Konsequenzregel

- ...stellt die Schnittstelle zwischen Programmverifikation und den logischen Formeln der Zuspicherungssprache dar
- ...erlaubt es,
 - Vorbedingungen zu *verstärken* (Übergang von p_1 zu p möglich, falls $p \Rightarrow p_1$ ($\Leftrightarrow Ch(p) \subseteq Ch(p_1)$))
 - Nachbedingungen *abzuschwächen* (Übergang von q_1 zu q möglich, falls $q_1 \Rightarrow q$ ($\Leftrightarrow Ch(q_1) \subseteq Ch(q)$))
- ...um so die Anwendung anderer Beweisregeln zu ermöglichen.

Zur while-Regel in HKPK

$$[while] \frac{\{I \wedge b\} \pi \{I\}}{\{I\} while \ b \ \text{do} \ \pi \ \text{od} \ \{I \wedge \neg b\}}$$

Intuitiv:

- Das durch I beschriebene Prädikat gilt...
 - vor und nach jeder Ausführung des Rumpfes der while-Schleife
 - und wird deswegen als *Invariante* der while-Schleife bezeichnet.
- Die while-Regel besagt weiter, dass
 - wenn zusätzlich (zur Invarianten) auch b vor jeder Ausführung des Schleifenrumpfs gilt, dass nach Beendigung der while-Schleife $\neg b$ wahr ist.

Definition

Sei $(P, <)$ eine irreflexive partielle Ordnung und sei W eine Teilmenge von P .

Dann heißt die Relation $<$ auf W *wohlfundiert*, wenn es keine unendlich absteigende Kette

$$\dots < w_2 < w_1 < w_0$$

von Elementen $w_i \in W$ gibt.

Das Paar $(W, <)$ heißt dann eine *wohlfundierte Struktur* oder auch eine *wohlfundierte* oder *Noethersche Ordnung*.

Sprechweise: Gilt $w < w'$ für $w, w' \in W$, sagen wir, w ist *kleiner* als w' oder w' ist *größer* als w .

Beispiele: $(\mathbb{N}, <)$, aber nicht $(\mathbb{Z}, <)$, $(\mathbb{Z}, >)$ oder $(\mathbb{N}, >)$

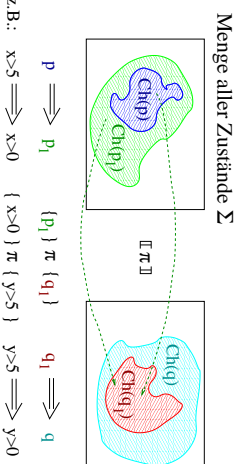
Anmerkungen zu...

...den der

- Konsequenzregel [cons] und der
 - Schleifenregeln [whilePK] und [whileTK]
- von HKPK bzw. HKTK zugrundeliegenden Intuitionen.

Zur Konsequenzregel (2)

Veranschaulichung von Verstärkung und Abschwächung:



Zur while-Regel in HKTK (1)

Erinnerung:

$$[whileTK] \frac{I \wedge b \Rightarrow \alpha \ll [b] \pi \{I \wedge b \wedge \alpha = w\} \pi \{I \wedge k < w\}}{\{I\} while \ b \ \text{do} \ \pi \ \text{od} \ \{I \wedge \neg b\}}$$

wobei

- α Boolescher Ausdruck über der Variablen v_i ,
- t arithmetischer Term,
- w Variable, die in I, b, π und t nicht frei vorkommt,
- $M = \# \{ \sigma \in \Sigma \mid \sigma \in \Sigma \wedge \ll [u] \# \{ \sigma \} = \text{tt} \}$ noethersch geordnete Menge (sog. noethersche Halbordnung),
 \rightsquigarrow *Terminationsordnung!*

• **Prämisse 1:** $I \wedge b \Rightarrow u \{I/a\}$
 Wann immer der Schleifenrumpf noch einmal ausgeführt wird (d.h. $I \wedge b$ ist wahr), gilt, dass $u \{I/a\}$ wahr ist, voraus aufgrund der Definition von M folgt, dass der Wert von t Element einer noethersch geordneten Menge ist.

• **Prämisse 2:** $\{I \wedge b \wedge t = w\} \pi \{I \wedge t < w\}$

- w speichert den initialen Wert von t (w ist sog. *logische Variable*), d.h. den Wert, den t vor Eintritt in die Schleife hat (gilt, da w als logische Variable insbesondere nicht in π vorkommt)
- Zusammen damit, dass der Wert von w (als logische Variable) invariant unter der Ausführung des Schleifenrumpfs ist, garantiert $t < w$ in der Nachbedingung von Prämisse 2, dass der Wert von t nach jeder Ausführung des Schleifenrumpfs bzgl. der noetherschen Ordnung abgenommen hat.

• Zusammen implizieren die obigen beiden Punkte die Terminierung der while-Schleife, da es in einer noethersch geordneten Menge keine unendlich absteigenden Ketten gibt. Folglich kann die Bedingung $I \wedge b$ in Prämisse 1 nicht unendlich oft wahr sein, da dies zusammen mit Prämisse 2 ein unendliches Absteigen erforderte.)

HKTK versus HKPK

Beachte:

$HKTK$ und $HKPK$ sind bis auf die Schleifenregel (und die Regel für *abort*) identisch...

- **Totale Korrektheit:** $[whileTK] \frac{I \wedge b \Rightarrow u \{I/a\}, \{I \wedge b \wedge t = w\} \pi \{I \wedge t < w\}}{\{I\} \text{ while } b \text{ do } \pi \text{ od } \{I \wedge \neg b\}}$
- **Partielle Korrektheit:** $[whilePK] \frac{\{I \wedge b\} \pi \{I\}}{\{I\} \text{ while } b \text{ do } \pi \text{ od } \{I \wedge \neg b\}}$

Nachtrag zur totalen Korrektheit (2)

Beweistechnische Anmerkung:

"Zerlegt" man $[whileTK]$ wie folgt:

$$[whileTK] \frac{I \Rightarrow t \geq 0, \{I \wedge b\} \pi \{I\}, \{I \wedge b \wedge t = w\} \pi \{t < w\}}{\{I\} \text{ while } b \text{ do } \pi \text{ od } \{I \wedge \neg b\}}$$

wird deutlich, dass der Nachweis totaler Korrektheit einer Hoarschen Zusicherung besteht aus

- dem Nachweis ihrer partiellen Korrektheit
- dem Nachweis der Termination

Diese Trennung kann im Beweis explizit vollzogen werden. Der Gesamtbeweis wird dadurch modular. Oft gilt, dass der Terminationssatz einfach ist.

Randbemerkung: Die obige Trennung kann für $[whilePK]$ analog vorgenommen werden.

Linearer vs. baumartiger Beweisstil

Vorteil linearen gegenüber baumartigen Beweisnotationsstils:

- wenig Redundanz
- daher insgesamt knappere Beweise

- **Programmvariablen**
 ... Variablen, die in π vorkommen
- **logischen Variablen**
 ... Variablen, die in π nicht vorkommen

Logische Variablen erlauben...

- sich *initiale* Werte von Programmvariablen zu "merken", um in Nachbedingungen geeignet darauf Bezug zu nehmen.

Beispiel:

- $\{x = n\} y := 1; \text{ while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ od } \{y = n! \wedge n > 0\}$
 ...die Nachbedingung macht eine Aussage über den Zusammenhang des Anfangswertes von x (gespeichert in n) und des schließlichen Wertes von y .
- $\{x = n\} y := 1; \text{ while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ od } \{y = x! \wedge x > 0\}$
 ...die Nachbedingung macht eine Aussage über den Zusammenhang der schließlichen Werte von x und y . (*Beachte:* nur mit Programmvariablen keine Aussage über die Fakultätsberechnung in diesem Bsp.)

Nachtrag zur totalen Korrektheit (1)

Oft, insbesondere für die von uns betrachteten Beispiele, reicht folgende, weniger allgemeine Regel für while-Schleifen, um Terminierung und insgesamt totale Korrektheit zu zeigen.

$$[whileTK] \frac{I \Rightarrow t \geq 0, \{I \wedge b \wedge t = w\} \pi \{I \wedge t < w\}}{\{I\} \text{ while } b \text{ do } \pi \text{ od } \{I \wedge \neg b\}}$$

wobei

- t arithmetischer Term über ganzen Zahlen,
- w ganzzahlige Variable, die in I , b , π und t nicht frei vorkommt.

Beachte: Statt beliebiger Terminationsordnungen hier Festlegung auf eine spezielle Noethersche Ordnung als Terminationsordnung, nämlich $(\mathbb{N}, <)$.

Kapitel 4.5 Ergänzungen, Sprechweisen

Sprechweisen im Zshg. mit Hoare-Tripeln (1)

Hoarsche Zusicherungen sind von einer der zwei Formen

- $\{p\} \pi \{q\}$ und
- $[p] \pi [q]$

wobei

- p, q logische Formeln sind (meist prädikatenlogische Formeln 1. Stufe) und
- π ein Programm ist.

Sprechweisen im Zshg. mit Hoare-Tripeln (2)

In einer Hoareschen Zusicherung von einer der Formen

- $\{p\} \pi \{q\}$ und
 - $[p] \pi [q]$
- heißen
- p und q Vor- bzw. Nachbedingung.

Kap. 4.5 Ergänzungen, Sprechweisen

168

Sprechweisen (3)

In einer Hoareschen Zusicherung werden üblicherweise

- geschweifte Klammern wie in $\{p\} \pi \{q\}$ für Tripel im Sinne *partieller Korrektheit* und
- eckige Klammern wie in $[p] \pi [q]$ für Tripel im Sinne *totaler Korrektheit* benutzt.

Kap. 4.5 Ergänzungen, Sprechweisen

169

Sprechweisen im Zshg. mit Hoare-Tripeln (4)

Zwei Beispiele Hoarescher Zusicherungen:

$$\{a > 0\} \\ x := a; y := 1; \text{ while } x > 1 \text{ do } y := y * x; x := x - 1 \text{ od} \\ \{y = a!\}$$

...zum Ausdruck *partieller Korrektheit* von π bzgl. der Vorbedingung $a > 0$ und der Nachbedingung $y = a!$

$$[a > 0] \\ x := a; y := 1; \text{ while } x > 1 \text{ do } y := y * x; x := x - 1 \text{ od} \\ [y = a!]$$

...zum Ausdruck *totaler Korrektheit* von π bzgl. der Vorbedingung $a > 0$ und der Nachbedingung $y = a!$

Kap. 4.5 Ergänzungen, Sprechweisen

170

Sprechweisen im Zshg. mit Hoare-Tripeln (5)

Die Wortwahl

- *Hoaresches Tripel* oder kurz *Hoare-Tripel* bzw.
- *Hoaresche Zusicherung* oder kurz *Korrektheitsformel*

betont jeweils die

- syntaktische bzw.
- semantische Sicht auf
- $\{p\} \pi \{q\}$ bzw. $[p] \pi [q]$

Kap. 4.5 Ergänzungen, Sprechweisen

171

Kapitel 4.3 Korrektheit und Vollständigkeit

Kap. 4.3 Korrektheit und Vollständigkeit

172

Korrektheit und Vollständigkeit von HK_{PK} und HK_{TK}

Sei K ein Kalkül für partielle bzw. totale Korrektheit

Zentral sind dann die Fragen der...

- *Korrektheit*: ...ist jede mithilfe von K ableitbare Korrektheitsformel partiell bzw. total korrekt?
- *Vollständigkeit*: ...ist jede partiell bzw. total korrekte Korrektheitsformel mithilfe von K ableitbar?

Speziell:

- Sind HK_{PK} und HK_{TK} korrekt und vollständig?

Kap. 4.3 Korrektheit und Vollständigkeit

173

Hauptresultate

Zur Korrektheit:

Theorem [Korrektheit von HK_{PK} und HK_{TK}]

1. HK_{PK} ist korrekt, d.h. jede mit HK_{PK} ableitbare Korrektheitsformel ist gültig im Sinne partieller Korrektheit.
2. HK_{TK} ist korrekt, d.h. jede mit HK_{TK} ableitbare Korrektheitsformel ist gültig im Sinne totaler Korrektheit.

Beweis ...durch Induktion über die Anzahl der Regelanwendungen im Beweisbaum zur Ableitung der Korrektheitsformel.

Zur Vollständigkeit:

Für Korrektheitskalküle ist i.a. nur sog. *relative* Vollständigkeit möglich. Das gilt auch für HK_{PK} und HK_{TK} . Details dazu in der Folge.

Kap. 4.3 Korrektheit und Vollständigkeit

174

Zur Korrektheit und Vollständigkeit Hoarescher Beweiskalküle

Sei K ein Hoarescher Beweiskalkül (z.B. HK_{PK} und HK_{TK}).

Dann heißt K ...

- *korrekt* (engl. *sound*), falls gilt: Ist eine Korrektheitsformel mit K herleitbar/beweisbar, dann ist sie auch semantisch gültig. In Zeichen:
$$\vdash \{p\} \pi \{q\} \Rightarrow \models \{p\} \pi \{q\}$$
- *vollständig* (engl. *complete*), falls gilt: Ist eine Korrektheitsformel semantisch gültig, dann ist sie auch mit K herleitbar/beweisbar.
$$\models \{p\} \pi \{q\} \Rightarrow \vdash \{p\} \pi \{q\}$$

Kap. 4.3 Korrektheit und Vollständigkeit

175

Theorem [Korrektheit von HK_{PK} und HK_{TK}]

1. HK_{PK} ist korrekt, d.h. jede mit HK_{PK} ableitbare Korrektheitsformel ist gültig im Sinne partieller Korrektheit:
$$\vdash_{HK} \{p\} \pi \{q\} \Rightarrow \models_{HK} \{p\} \pi \{q\}$$
2. HK_{TK} ist korrekt, d.h. jede mit HK_{TK} ableitbare Korrektheitsformel ist gültig im Sinne totaler Korrektheit:
$$\vdash_{HK} [p] \pi [q] \Rightarrow \models_{HK} [p] \pi [q]$$

Beweis ... durch Induktion über die Anzahl der Regelanwendungen im Beweisbaum zur Ableitung der Korrektheitsformel.

Kap. 4.3 Korrektheit und Vollständigkeit

176

Zur Vollständigkeit Hoarescher Beweiskalküle

Generell müssen wir unterscheiden zwischen Vollständigkeit

- *extensionaler* und
- *intensionaler*

Ansätze.

Kap. 4.3 Korrektheit und Vollständigkeit

177

Extensionale vs. intensionale Ansätze

- *Extensional*
 \rightsquigarrow Vor- und Nachbedingungen sind durch Prädikate beschreiben.
- *Intensional*
 \rightsquigarrow Vor- und Nachbedingungen sind durch *Formeln einer Zusicherungssprache* beschrieben.

Kap. 4.3 Korrektheit und Vollständigkeit

178

Zur Vollständigkeit von HK_{PK} & HK_{TK}

Für den extensionalen Ansatz gilt:

Theorem [Vollständigkeit von HK_{PK} und HK_{TK}]

1. HK_{PK} ist vollständig, d.h. jede im Sinne partieller Korrektheit gültige Korrektheitsformel ist mit HK_{PK} ableitbar:
$$\models_{HK} \{p\} \pi \{q\} \Rightarrow \vdash_{HK} \{p\} \pi \{q\}$$
2. HK_{TK} ist vollständig, d.h. jede im Sinne totaler Korrektheit gültige Korrektheitsformel ist mit HK_{TK} ableitbar:
$$\models_{HK} [p] \pi [q] \Rightarrow \vdash_{HK} [p] \pi [q]$$

Beweis ... durch strukturelle Induktion über den Aufbau von π .

Kap. 4.3 Korrektheit und Vollständigkeit

179

Zur Vollständigkeit von HK_{PK} & HK_{TK}

Für intensionale Ansätze (durch unterschiedliche Wahlen der Zusicherungssprache) gilt Vollständigkeit i.a. nur relativ zur *Entscheidbarkeit* und *Ausdruckskraft* der Zusicherungssprache.

Intuition

- *Entscheidbarkeit*
...ist die Gültigkeit von Formeln der Zusicherungssprache algorithmisch verifizierbar bzw. falsifizierbar?
- *Ausdruckskraft*
...lassen sich alle Prädikate, insbesondere schwächste und schwächste liberale Vorbedingungen und Terminationsfunktionen, durch Formeln der Zusicherungssprache beschreiben?
 \rightsquigarrow *tieferlegende Frage*: ...lassen sich schwächste Vorbedingungen etc. syntaktisch ausdrücken?

Stichwort: Relative Vollständigkeit im Sinne von Cook.

Kap. 4.3 Korrektheit und Vollständigkeit

180

Kapitel 4.4 Beweis partieller Korrektheit: Zwei Beispiele

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

181

Die beiden Beispiele im Überblick 2(2)

Im Detail:

Beweise, dass die beiden Hoareschen Zusicherungen

$x := a; y := 1; \text{ while } x > 1 \text{ do } y := y * x; x := x - 1 \text{ od}$
 $\{a > 0\}$
 $\{y = a!\}$

und

$\{x \geq 0 \wedge y > 0\}$
 $q := 0; r := x; \text{ while } r \geq y \text{ do } q := q + 1; r := r - y \text{ od}$
 $\{x = q * y + r \wedge 0 \leq r < y\}$

gültig sind im Sinne partieller Korrektheit.

In der Folge geben wir die Beweise dafür in baumartiger Notation an...

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

183

Die beiden Beispiele im Überblick 1(2)

...Beweis partieller Korrektheit von Hoareschen Zusicherungen anhand zweier Programme zur Berechnung

- der Fakultät und
- der ganzzahligen Division mit Rest

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

182

Wegen Rückwärtszuweisungsregel wird der Rumpf der while-Schleife von hinten nach vorne bearbeitet:

```

{y * x! = a! ∧ x > 0 ∧ x > 1}
  y := y * x;
  {y * (x - 1)! = a! ∧ x - 1 > 0}
  x := x - 1; [ass]
  {y * x! = a! ∧ x > 0}
  
```

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

192

Lin. Beweisskizze f. Fakultätsbsp. (7)

Schluss der "Beweislücke" in der zugrundeliegenden Theorie:

```

{y * x! = a! ∧ x > 0 ∧ x > 1}
  ↓ [cons]
{(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
  y := y * x; [ass]
  {y * (x - 1)! = a! ∧ x - 1 > 0}
  x := x - 1; [ass]
  {y * x! = a! ∧ x > 0}
  
```

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

194

Lin. Beweisskizze f. Fakultätsbsp. (9)

Schritt 3

Zur gewünschten Nachbedingung verbleibt offenbar ebenfalls eine Beweislücke:

```

{y * x! = a! ∧ x > 0}
  while x > 1 do
  {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
      y := y * x; [ass]
      {y * (x - 1)! = a! ∧ x - 1 > 0}
      x := x - 1; [ass]
      {y * x! = a! ∧ x > 0}
      od [while]
  {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  {y = a!}
  
```

Lin. Beweisskizze f. Fakultätsbsp. (11)

Aus Platzgründen etwas verkürzt dargestellt:

```

{y * x! = a! ∧ x > 0}
  while x > 1 do
  {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
      y := y * x; [ass]
      {y * (x - 1)! = a! ∧ x - 1 > 0}
      x := x - 1; [ass]
      {y * x! = a! ∧ x > 0}
      od [while]
  {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y = a!}
  
```

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

196

Nach abermaliger Anwendung der [ass]-Regel erhalten wir...

```

{y * x! = a! ∧ x > 0 ∧ x > 1}
  {y * x! = a! ∧ x - 1 > 0}
  y := y * x; [ass]
  {y * (x - 1)! = a! ∧ x - 1 > 0}
  x := x - 1; [ass]
  {y * x! = a! ∧ x > 0}
  
```

...wobei noch eine "Beweislücke" verbleibt!

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

193

Lin. Beweisskizze f. Fakultätsbsp. (8)

Anwendung der [while]-Regel liefert nun wie gewünscht:

```

{y * x! = a! ∧ x > 0}
  while x > 1 do
  {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
      y := y * x; [ass]
      {y * (x - 1)! = a! ∧ x - 1 > 0}
      x := x - 1; [ass]
      {y * x! = a! ∧ x > 0}
      od [while]
  {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  
```

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

195

Lin. Beweisskizze f. Fakultätsbsp. (10)

Schluss der Beweislücke in der zugrundeliegenden Theorie:

```

{y * x! = a! ∧ x > 0}
  while x > 1 do
  {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
      y := y * x; [ass]
      {y * (x - 1)! = a! ∧ x - 1 > 0}
      x := x - 1; [ass]
      {y * x! = a! ∧ x > 0}
      od [while]
  {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y * x! = a! ∧ x > 0 ∧ x ≤ 1}
  ↓ [cons]
  {y * x! = a! ∧ x = 1}
  ↓ [cons]
  {y = a!}
  
```

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

197

Lin. Beweisskizze f. Fakultätsbsp. (12)

Schritt 4

Es verbleibt, die Beweislücke zur gewünschten Vorbedingung zu schließen:

```

{a > 0}
  x := a;
  y := 1;
  {y * x! = a! ∧ x > 0}
  while x > 1 do
  {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
      y := y * x; [ass]
      {y * (x - 1)! = a! ∧ x - 1 > 0}
      x := x - 1; [ass]
      {y * x! = a! ∧ x > 0}
      od [while]
  {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y = a!}
  
```

Einmalige Anwendung der [assl]-Regel liefert:

```

{a > 0}
x := a;
{1 * x! = a! ∧ x > 0}
  y := 1; [assl]
  {y * x! = a! ∧ x > 0}
  while x > 1 do
    {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {{(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
     y := y * x; [assl]
     {y * (x - 1)! = a! ∧ x - 1 > 0}
     x := x - 1; [assl]
     {y * x! = a! ∧ x > 0}
     od [while]
    {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y = a!}

```

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Lin. Beweisskizze f. Fakultätsbsp. (15)

Schluss der letzten Beweislücke in der zugrundeliegenden Theorie:

```

{a > 0}
↓ [cons]
{1 * a! = a! ∧ a > 0}
x := a; [assl]
{1 * x! = a! ∧ x > 0}
  y := 1; [assl]
  {y * x! = a! ∧ x > 0}
  while x > 1 do
    {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {{(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
     y := y * x; [assl]
     {y * (x - 1)! = a! ∧ x - 1 > 0}
     x := x - 1; [assl]
     {y * x! = a! ∧ x > 0}
     od [while]
    {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y = a!}

```

Kap. 4.4 Beweis partieller Korrektheit: Zwei Beispiele

Lin. Beweisskizze f. Fakultätsbsp. (17)

Damit haben wir insgesamt wie gewünscht gezeigt:

Das Hoaresche Tripel

```

{a > 0}
x := a; y := 1; while x > 1 do y := y * x; x := x - 1 od
{y = a!}

```

ist gültig im Sinne partieller Korrektheit.

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Das Beispiel im Überblick

Beweise, dass das Hoare-Tripel

```

{a > 0}
x := a; y := 1; while x > 1 do y := y * x; x := x - 1 od
{y = a!}

```

gültig ist im Sinne totaler Korrektheit.

Wir entwickeln den Beweis in der Folge Schritt für Schritt!

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Abermalige Anwendung der [assl]-Regel liefert:

```

{a > 0}
{1 * a! = a! ∧ a > 0}
  x := a; [assl]
  {1 * x! = a! ∧ x > 0}
  y := 1; [assl]
  {y * x! = a! ∧ x > 0}
  while x > 1 do
    {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {{(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
     y := y * x; [assl]
     {y * (x - 1)! = a! ∧ x - 1 > 0}
     x := x - 1; [assl]
     {y * x! = a! ∧ x > 0}
     od [while]
    {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y = a!}

```

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Überblick (16)

```

{a > 0}
↓ [cons]
{1 * a! = a! ∧ a > 0}
x := a; [assl]
{1 * x! = a! ∧ x > 0}
  y := 1; [assl]
  {y * x! = a! ∧ x > 0}
  while x > 1 do
    {y * x! = a! ∧ x > 0 ∧ x > 1}
    ↓ [cons]
    {{(y * x) * (x - 1)! = a! ∧ x - 1 > 0}
     y := y * x; [assl]
     {y * (x - 1)! = a! ∧ x - 1 > 0}
     x := x - 1; [assl]
     {y * x! = a! ∧ x > 0}
     od [while]
    {y * x! = a! ∧ x > 0 ∧ ¬(x > 1)}
  ↓ [cons]
  {y = a!}

```

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Kapitel 4.7 Beweis totaler Korrektheit: Ein Beispiel

Wahl von Invariante und Terminierungsterm

Schritt 1

„Träumen“...

- der Invariante: $y * x! = a! \wedge x > 0$
- des Terminierungsterms: $t \equiv x$
- von u : $u \equiv v \geq 0$

...um die [while]-Regel anwenden zu können.

Beachte:

- Aus der Wahl von $u \equiv v \geq 0$ und von $b \equiv x > 1$ folgt:
 - $M \equiv \{0, 1, 2, 3, 4, \dots\}$
 - $(v \geq 0)[b/a] \equiv x \geq 0$
- und somit insgesamt: $\vdash b \Rightarrow x \in M$ mit $(M, <)$ Noethersch geordnet.

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

Hinweis zur Notation: \equiv steht für syntaktisch gleich

Mit der vorherigen Wahl von I , t und u gilt:

$$\begin{aligned} M &=_{df} \{ \sigma(v) \mid \sigma \in \Sigma \wedge \llbracket u \rrbracket_B(\sigma) = tt \} \\ &= \{ \sigma(v) \mid \sigma \in \Sigma \wedge \llbracket v \geq 0 \rrbracket_B(\sigma) = tt \} \\ &= \{ \sigma(v) \mid \sigma \in \Sigma \wedge \text{grossenlych}(\llbracket v \rrbracket_A(\sigma), \llbracket 0 \rrbracket_A(\sigma)) \} \\ &= \{ \sigma(v) \mid \sigma \in \Sigma \wedge \text{grossenlych}(\sigma(v), \mathbf{0}) = tt \} \\ &= \{ \sigma(v) \mid \sigma \in \Sigma \wedge \sigma(v) \geq \mathbf{0} \} \\ &= \text{N} \cup \{ \mathbf{0} \} \end{aligned}$$

Damit haben wir insbesondere:

- $(M, <) = (\text{N} \cup \{ \mathbf{0} \}, <)$ ist noethersch geordnet.
- $u[t/x] = (v \geq 0)[x/v] = x \geq 0$

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

208

Bemerkung

Der Beweis wird wieder in Form einer linearen Beweisskizze präsentiert...

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

209

Bew. totaler Korrektheit: Fakultät (1)

Schritt 2

Behandlung des Rumpfs der while-Schleife...

Der Nachweis der Gültigkeit von

$$\begin{aligned} [y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0 \\ [y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w] \end{aligned}$$

$$y := y * x;$$

$$x := x - 1;$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

erlaube mithilfe der [while]-Regel den Übergang zu:

$$[y * x! = a! \wedge x > 0]$$

$$\text{while } x > 1 \text{ do}$$

$$y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0$$

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w]$$

$$y := y * x;$$

$$x := x - 1;$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

$$\text{od [while]}$$

$$[y * x! = a! \wedge x > 0 \wedge \neg(x > 1)]$$

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

211

Bew. totaler Korrektheit: Fakultät (2)

Behandlung des Rumpfs der while-Schleife im Detail:

$$y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0$$

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w]$$

$$y := y * x;$$

$$x := x - 1;$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

211

Bew. totaler Korrektheit: Fakultät (3)

Wegen Rückwärtszuweisungsregel wird der Rumpf der while-Schleife von hinten nach vorne bearbeitet:

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0]$$

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w]$$

$$y := y * x;$$

$$[y * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$x := x - 1; \text{ [ass]}$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

212

Bew. totaler Korrektheit: Fakultät (4)

Nach abermaliger Anwendung der [ass]-Regel erhalten wir...

$$y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0$$

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w]$$

$$[y * x) * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$y := y * x; \text{ [ass]}$$

$$[y * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$x := x - 1; \text{ [ass]}$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

...wobei noch eine "Beweislücke" verbleibt!

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

213

Bew. totaler Korrektheit: Fakultät (5)

Schluss der "Beweislücke" in der zugrundeliegenden Theorie:

$$y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0$$

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w]$$

↓ [cons]

$$[(y * x) * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$y := y * x; \text{ [ass]}$$

$$[y * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$x := x - 1; \text{ [ass]}$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

214

Bew. totaler Korrektheit: Fakultät (6)

Anwendung der [while]-Regel liefert nun wie gewünscht:

$$[y * x! = a! \wedge x > 0]$$

while $x > 1$ do

$$y * x! = a! \wedge x > 0 \wedge x > 1 \Rightarrow x \geq 0$$

$$[y * x! = a! \wedge x > 0 \wedge x > 1 \wedge x = w]$$

↓ [cons]

$$[(y * x) * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$y := y * x; \text{ [ass]}$$

$$[y * (x - 1)! = a! \wedge x - 1 > 0 \wedge x - 1 < w]$$

$$x := x - 1; \text{ [ass]}$$

$$[y * x! = a! \wedge x > 0 \wedge x < w]$$

od [while]

$$[y * x! = a! \wedge x > 0 \wedge \neg(x > 1)]$$

Kap. 4.7 Beweis totaler Korrektheit: Ein Beispiel

215

Bew. totaler Korrektheit: Fakultät (15)

Damit haben wir wie gewünscht insgesamt gezeigt:

Die Hoaresche Zusicherung

$$\{a > 0\}$$

$x := a; y := 1; \text{ while } x > 1 \text{ do } y := y * x; x := x - 1 \text{ od}$

$$\{y = a!\}$$

ist gültig im Sinne totaler Korrektheit.

Kap. 4.7 Beweiss totaler Korrektheit: Ein Beispiel

224

Kapitel 4.8 Ausblick

Kap. 4.8 Ausblick

225

Automatische Ansätze zur Programmverifikation (1)

... *Theorema*-Projekt am RISC, Linz: <http://www.theorema.org>

"The Theorema project aims at extending current computer algebra systems by facilities for supporting mathematical proving. The present early-prototype version of the Theorema software system is implemented in Mathematica. The system consists of a general higher-order predicate logic prover and a collection of special provers that call each other depending on the particular proof situations. The individual provers imitate the proof style of human mathematicians and produce human-readable proofs in natural language presented in nested cells. The special provers are intimately connected with the functors that build up the various mathematical domains.

The long-term goal of the project is to produce a complete system which supports the mathematician in creating interactive textbooks, i.e. books containing, besides the ordinary passive text, active text representing algorithms in executable format, as well as proofs which can be studied at various levels of detail, and whose routine parts can be automatically generated. This system will provide a uniform (logic and software) framework in which a working mathematician, without leaving the system, can get computer-support while looping through all phases of the mathematical problem solving cycle."

[...]

(Zitat von <http://www.theorema.org>)

Automatische Ansätze zur Programmverifikation (2)

Einige Artikel zu Programmverifikation mit *Theorema*:

- Laura Idico Kovacs and Tudor Jelebelean. *Practical Aspects of Imperative Program Verification using Theorema*. In Proceedings of the 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2003), Timisoara, Romania, October 1-4, 2003. apache.risc.uni-linz.ac.at/internals/ActivityDB/publications/download/risc_464/synasc03.pdf

- Laura Idico Kovacs and Tudor Jelebelean. *Generation of Invariants in Theorema*. In Proceedings of the 10th International Symposium of Mathematics and its Applications, Timisoara, Romania, November 6-9, 2003.

www.theorema.org/publication/2003/Laura/Pol1Timisoara.nov.pdf

Kap. 4.8 Ausblick

227

Kapitel 5 Worst-Case Execution Time Analyse

Kap. 5 Worst-Case Execution Time Analyse

228

Worst-Case Execution Time (WCET)-Analyse

Motivation:

- In vielen Anwendungsbereichen sind Aussagen über die Ausführungszeit erforderlich.

- Der Nachweis totaler Korrektheit garantiert zwar Terminierung, sagt aber nichts über den Ressourcen-, speziell den Zeitbedarf aus.

In der Folge:

- Erweiterung und Adaptierung des Beweissystems für totale Korrektheit, um solche Aussagen zu ermöglichen.

Kap. 5 Worst-Case Execution Time Analyse

230

In der Folge

Von Verifikation zu Analyse...

- *Worst-Case Execution Time-Analyse* als erstes Beispiel
- ...nach
- Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications – A Formal Introduction*. Wiley, 1992.

Kap. 5 Worst-Case Execution Time Analyse

229

Die grundlegende Idee (1)

...zur Zuordnung von Ausführungszeiten:

- *Leere Anweisung*
...Ausführungszeit in $\mathcal{O}(1)$, d.h. Ausführungszeit ist beschränkt durch eine Konstante.

- *Zuweisung*
...Ausführungszeit in $\mathcal{O}(1)$.

- *(Sequentielle) Komposition*

...Ausführungszeit entspricht, bis auf einen konstanten Faktor, der Summe der Ausführungszeiten der Komponenten.

Kap. 5 Worst-Case Execution Time Analyse

231

- **Fallunterscheidung**
...Ausführungszeit entspricht, bis auf einen konstanten Faktor, der größeren der Ausführungszeiten der beiden Zweige.
- (*while*)-**Schleife**
...Ausführungszeit der Schleife entspricht, bis auf einen konstanten Faktor, der Summe der wiederholten Ausführungszeiten des Rumpfes der Schleife.

Bemerkung: Verfeinerungen sind offenbar möglich.

- ...dieser grundlegenden Idee in 3 Schritten:
1. Angabe einer Semantik, die die Ausführungszeit arithmetischer und Boolescher Ausdrücke beschreibt.
 2. Erweiterung und Adaption der natürlichen Semantik von WHILE zur Bestimmung der Ausführungszeit eines Programms.
 3. Erweiterung und Adaption des Beweissystems für totale Korrektheit zum Nachweis über die Größenordnung der Ausführungszeit von Programmen.

Erster Schritt

Festlegung von Semantikfunktionen

- $\llbracket \cdot \rrbracket_{TA} : \mathbf{Aexpr} \rightarrow \mathbb{Z}$ und
- $\llbracket \cdot \rrbracket_{TB} : \mathbf{Bexpr} \rightarrow \mathbb{Z}$

zur Beschreibung der Auswertungszeit arithmetischer und Boolescher Ausdrücke (in Zeiteinheiten einer abstrakten Maschine).

Semantik zur Ausführungszeit der Auswertung arithmetischer Ausdrücke

$\llbracket \cdot \rrbracket_{TA} : \mathbf{Aexpr} \rightarrow \mathbb{Z}$ induktiv definiert durch

- $\llbracket n \rrbracket_{TA} =_df \mathbf{1}$
- $\llbracket x \rrbracket_{TA} =_df \mathbf{1}$
- $\llbracket a_1 + a_2 \rrbracket_{TA} =_df \llbracket a_1 \rrbracket_{TA} + \llbracket a_2 \rrbracket_{TA} + \mathbf{1}$
- $\llbracket a_1 * a_2 \rrbracket_{TA} =_df \llbracket a_1 \rrbracket_{TA} + \llbracket a_2 \rrbracket_{TA} + \mathbf{1}$
- $\llbracket a_1 - a_2 \rrbracket_{TA} =_df \llbracket a_1 \rrbracket_{TA} + \llbracket a_2 \rrbracket_{TA} + \mathbf{1}$
- $\llbracket a_1 / a_2 \rrbracket_{TA} =_df \llbracket a_1 \rrbracket_{TA} + \llbracket a_2 \rrbracket_{TA} + \mathbf{1}$
- ... (andere Operatoren analog, ggf. auch mit operationsspezifischen Kosten)

Anmerkungen zu $\llbracket \cdot \rrbracket_{TA}$ und $\llbracket \cdot \rrbracket_{TB}$

Die Semantikfunktionen

- $\llbracket \cdot \rrbracket_{TA}$ und $\llbracket \cdot \rrbracket_{TB}$
...beschreiben intuitiv die Anzahl der Zeiteinheiten, die eine (hier nicht spezifizierte) abstrakte Maschine zur Auswertung arithmetischer und Boolescher Ausdrücke benötigt.

Semantik zur Ausführungszeit der Auswertung Boolescher Ausdrücke

$\llbracket \cdot \rrbracket_{TB} : \mathbf{Bexpr} \rightarrow \mathbb{Z}$ induktiv definiert durch

- $\llbracket true \rrbracket_{TB} =_df \mathbf{1}$
- $\llbracket false \rrbracket_{TB} =_df \mathbf{1}$
- $\llbracket a_1 = a_2 \rrbracket_{TB} =_df \llbracket a_1 \rrbracket_{TA} + \llbracket a_2 \rrbracket_{TA} + \mathbf{1}$
- $\llbracket a_1 < a_2 \rrbracket_{TB} =_df \llbracket a_1 \rrbracket_{TA} + \llbracket a_2 \rrbracket_{TA} + \mathbf{1}$
- ... (andere Relatoren (z.B. \leq, \dots) analog)
- $\llbracket \neg b \rrbracket_{TB} =_df \llbracket b \rrbracket_{TB} + \mathbf{1}$
- $\llbracket b_1 \wedge b_2 \rrbracket_{TB} =_df \llbracket b_1 \rrbracket_{TB} + \llbracket b_2 \rrbracket_{TB} + \mathbf{1}$
- $\llbracket b_1 \vee b_2 \rrbracket_{TB} =_df \llbracket b_1 \rrbracket_{TB} + \llbracket b_2 \rrbracket_{TB} + \mathbf{1}$

Zweiter Schritt

Erweiterung und Adaption der

- natürlichen Semantik von WHILE
- zur Bestimmung der Ausführungszeit von Programmen.

Idee

Übergang zu Transitionen der Form

$$\langle \pi, \sigma \rangle \xrightarrow{t} \sigma'$$

mit der Bedeutung, dass π angesetzt auf σ nach t Zeiteinheiten in σ' terminiert.

...für das Beispiel von WHILE:

$$\begin{aligned}
& \text{[skip]}_{ms} \quad \frac{}{\langle \text{skip}, \sigma \rangle \xrightarrow{-1} \sigma} \\
& \text{[abort]}_{ms} \quad \frac{}{\langle \text{abort}, \sigma \rangle \xrightarrow{-1} \text{error}} \\
& \text{[ass]}_{ms} \quad \frac{}{\langle x := t, \sigma \rangle \xrightarrow{-1} \langle [t]_A(\sigma) / x \rangle} \\
& \text{[comp]}_{ms} \quad \frac{\langle \pi_1, \sigma \rangle \xrightarrow{-1} \sigma', \langle \pi_2, \sigma' \rangle \xrightarrow{-2} \sigma''}{\langle \pi_1; \pi_2, \sigma \rangle \xrightarrow{-1+2} \sigma''}
\end{aligned}$$

Kap. 5 Worst-Case Execution Time Analyse

240

Beispiel zur nat. "Zeit"-Semantik (1)

Sei $\sigma \in \Sigma$ mit $\sigma(x) = 3$.

Dann gilt:

$$\langle y := 1; \text{while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ od}, \sigma \rangle \longrightarrow \sigma[6/y][3/x]$$

$$\begin{aligned}
& \frac{}{\langle x := t, \sigma \rangle \xrightarrow{-1} \langle [t]_A(\sigma) / x \rangle} \quad \frac{}{\langle \pi_1; \pi_2, \sigma \rangle \xrightarrow{-1+2} \sigma''} \quad \frac{}{\langle \text{skip}, \sigma \rangle \xrightarrow{-1} \sigma} \quad \frac{}{\langle \text{abort}, \sigma \rangle \xrightarrow{-1} \text{error}} \\
& \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle}
\end{aligned}$$

Kap. 5 Worst-Case Execution Time Analyse

242

Dritter Schritt

Erweiterung und Adaption der

- des Beweiskalküls für totale Korrektheit um den Ausführungzeitaspekt von Programmen.

Kap. 5 Worst-Case Execution Time Analyse

244

Idee (2)

Die Korrektheitsformel

$$\{p\} \pi \{e \Downarrow q\}$$

ist gültig gdw. für jeden Anfangszustand σ gilt: ist die Vorbereitung p in σ erfüllt, dann terminiert die zugehörige Berechnung von π angesetzt auf σ regulär mit einem Endzustand σ' und die Nachbedingung q ist in σ' erfüllt, und die benötigte Ausführungszeit ist in $\mathcal{O}(e)$.

Kap. 5 Worst-Case Execution Time Analyse

246

Ausführungszerspekt (2)

$$\begin{aligned}
& \text{[ff]}_{ms} \quad \frac{}{\langle \pi_1, \sigma \rangle \xrightarrow{-t} \sigma'} \quad \frac{}{\langle \pi_1 \text{ else } \pi_2, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \pi_1 \text{ else } \pi_2, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \\
& \text{[if]}_{ms} \quad \frac{}{\langle \pi_1 \text{ then } \pi_2, \sigma \rangle \xrightarrow{-t} \sigma'} \quad \frac{}{\langle \pi_1 \text{ then } \pi_2, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \pi_1 \text{ then } \pi_2, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \\
& \text{[while]}_{ms} \quad \frac{}{\langle \pi, \sigma \rangle \xrightarrow{-t} \sigma', \text{while } b \text{ do } \pi \text{ od}, \sigma' \rangle \xrightarrow{-t'} \sigma''} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \\
& \text{[while]}_{ff} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle}
\end{aligned}$$

Kap. 5 Worst-Case Execution Time Analyse

241

Beispiel zur nat. "Zeit"-Semantik (2)

Das gleiche Beispiel in etwas gefälliger Darstellung:

$$\begin{aligned}
& \frac{}{\langle x := t, \sigma \rangle \xrightarrow{-1} \langle [t]_A(\sigma) / x \rangle} \quad \frac{}{\langle \pi_1; \pi_2, \sigma \rangle \xrightarrow{-1+2} \sigma''} \quad \frac{}{\langle \text{skip}, \sigma \rangle \xrightarrow{-1} \sigma} \quad \frac{}{\langle \text{abort}, \sigma \rangle \xrightarrow{-1} \text{error}} \\
& \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle} \quad \frac{}{\langle \text{while } b \text{ do } \pi \text{ od}, \sigma \rangle \xrightarrow{-1} \langle [b]_B(\sigma) \rangle}
\end{aligned}$$

Kap. 5 Worst-Case Execution Time Analyse

243

Idee (1)

Übergang zu Korrektheitsformeln der Form

$$\{p\} \pi \{e \Downarrow q\}$$

wobei

- p und q Prädikate (wie bisher) und
- $e \in \text{Aexp}$ ein arithmetischer Ausdruck ist.

Kap. 5 Worst-Case Execution Time Analyse

245

Axiomatische Semantik zum Ausführungszerspekt (1)

$$\begin{aligned}
& \text{[skip]}_e \quad \frac{}{\{p\} \text{skip} \{1 \Downarrow p\}} \\
& \text{[ass]}_e \quad \frac{}{\{p \wedge e' = u\} \pi_1 \{e_1 \Downarrow u\} \wedge \{e_2 \leq u\}, \{r\} \pi_2 \{e_2 \Downarrow q\}} \\
& \text{[comp]}_e \quad \frac{}{\{p \wedge e' = u\} \pi_1 \{e_1 \Downarrow u\} \wedge \{e_2 \leq u\}, \{r\} \pi_2 \{e_2 \Downarrow q\}} \\
& \text{[ite]}_e \quad \frac{}{\{p \wedge b\} \pi_1 \{e_1 \Downarrow q\}, \{p \wedge \neg b\} \pi_2 \{e_2 \Downarrow q\}} \\
& \text{[cons]}_e \quad \frac{}{\{p\} \pi \{e' \Downarrow q\}} \\
& \text{wobei } u \text{ frische logische Variable ist} \\
& \text{wobei } (f \text{ für eine natürliche Zahl } k) p \Rightarrow p' \wedge e' \leq k * e \\
& \text{und } q' \Rightarrow q
\end{aligned}$$

Kap. 5 Worst-Case Execution Time Analyse

247

Axiomatische Semantik zum Ausführungszeitaspekt (2)

[while_z]
 $\{ \exists z. p(z) \} \text{ while } b \text{ do } \pi \{ \exists z. p(z) \wedge e \leq n \}$
 wobei $p(z + 1) \Rightarrow b \wedge e \geq e_1 + e'$, $p(0) \Rightarrow \neg b \wedge 1 \leq e$
 n eine frische logische Variable ist und
 z Werte aus den natürlichen Zahlen annimmt (d.h. $z \geq 0$)

Kap. 5 Worst-Case Execution Time Analyse

248

Beispiele (2)

Die Korrektheitsformel

$\{x > 0\} y := 1; \text{ while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ od } \{x \vee \text{True}\}$ ||

beschreibt, dass die Ausführungszeit des Fakultätsprogramms angesetzt auf einen Zustand, in dem x einen Wert größer als 0 hat, von der Größenordnung von x ist, also linear beschränkt ist.

Kap. 5 Worst-Case Execution Time Analyse

250

Programmmanalyse

...speziell *Datenflussanalyse*

Typische Fragen sind ...

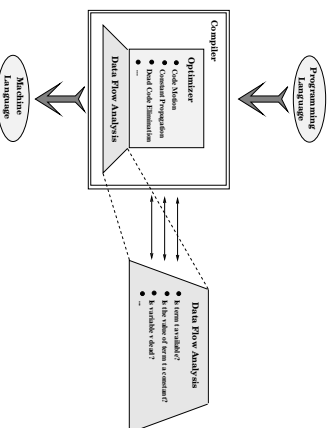
- Welchen **Wert** hat eine Variable an einer Programmstelle?
 \rightsquigarrow Konstantenausbreitung und Faltung
- Steht der Wert eines Ausdrucks an einer Programmstelle *verfügbar*?
 \rightsquigarrow (Partielle) Redundanzeliminatio
- Ist eine Variable *tot* an einer Programmstelle?
 \rightsquigarrow Elimination (partiell) toten Codes

Kap. 6: Programmmanalyse

252

Hintergrund und Motivation

...(Programm-) Analyse zur (Programm-) Optimierung



Kap. 6.1 Hintergrund und Motivation

254

Beispiele (1)

Die Korrektheitsformel

$\{x = 3\} y := 1; \text{ while } x \neq 1 \text{ do } y := y * x; x := x - 1 \text{ od } \{1 \vee \text{True}\}$ ||

beschreibt, dass die Ausführungszeit des Fakultätsprogramms angesetzt auf einen Zustand, in dem x den Wert 3 hat, von der Größenordnung von 1 ist, also durch eine Konstante beschränkt ist.

Kap. 5 Worst-Case Execution Time Analyse

249

Kapitel 6 Programmmanalyse

Kap. 6: Programmmanalyse

251

Kapitel 6.1 Hintergrund und Motivation

Kap. 6.1 Hintergrund und Motivation

253

In der Folge

Zentrale Fragen...

Grundlegendes ebenso ...

- Was heißt *Optimalität*
 \dots in Analyse und in Optimierung?
- Wie (scheinbar) Nebensächliches:
 \dots wie (scheinbar) Nebensächliches:
- Was ist eine *angemessene* Programmrepräsentation?

Kap. 6.1 Hintergrund und Motivation

255

Genauer werden wir unterscheiden...

- Intraprozedurale,
- interprozedurale,
- parallele,
- ...

Datenflussanalyse.

Kap. 6.1 Hintergrund und Motivation

256

Ingredientien Intraprozeduraler Datenflussanalyse:

- (Lokale) abstrakte Semantik
 1. Ein Datenflussanalyseverband $\mathcal{C} = (C, \cap, \cup, E, \perp, \top)$
 2. Ein Datenflussanalysefunktional $\llbracket \cdot \rrbracket : E \rightarrow (C \rightarrow C)$
 3. Anfangsinformation/-zusicherung $c_s \in C$
- Globalisierungsstrategien
 1. "Meet over all Paths"-Ansatz (MOP)
 2. Maximaler Fixpunktansatz (MaxFP)
- Generischer Fixpunktalgorithmus

Kap. 6.1 Hintergrund und Motivation

257

Ausblick

Hauptresultate:

- Sicherheits- (Korrektheits-) Theorem
- Kohärenz- (Vollständigkeits-) Theorem

Sowie:

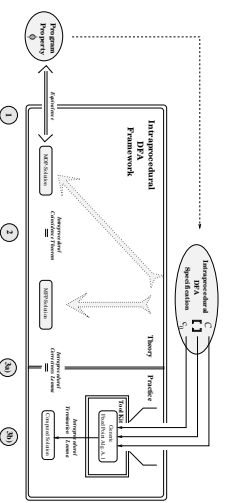
- Effektivitäts- (Terminierungs-) Theorem

Kap. 6.1 Hintergrund und Motivation

258

Ausblick: Intraprozedurale Datenflussanalyse (1)

...die (detaillierte) Werkzeugkistensticht:

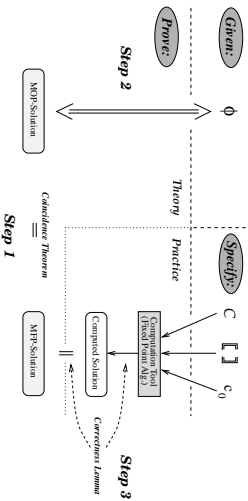


Kap. 6.1 Hintergrund und Motivation

259

Ausblick: Intraprozedurale Datenflussanalyse (2)

...bei genauerem Hinsehen:

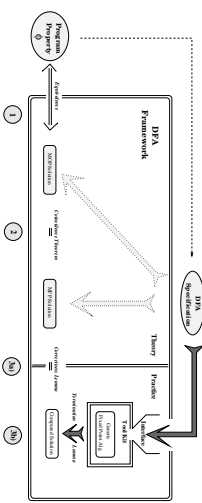


Kap. 6.1 Hintergrund und Motivation

260

Ausblick: DFA-Frameworks / DFA-Tools (1)

...aus größerer Ferne und Konzentration auf das Wesentliche:

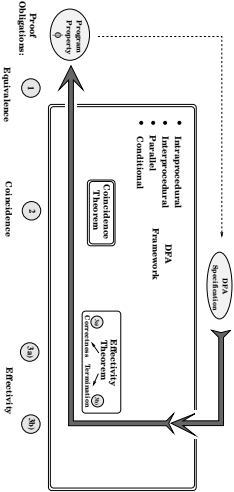


Kap. 6.1 Hintergrund und Motivation

261

Ausblick: DFA-Frameworks / DFA-Tools (2)

...das generelle Muster, die Werkzeugkistensticht:



Kap. 6.1 Hintergrund und Motivation

262

Ziel

Optimale Programminimierung...

...weiße Schimmel in der Informatik?

Kap. 6.1 Hintergrund und Motivation

263

Ohne Fleiß kein Preis!

In der Sprechweise der optimierenden Übersetzung...

...ohne *Analyse* keine Optimierung!

Kap. 6.1 Hintergrund und Motivation

264

Kapitel 6.2 Datenflussanalyse

Kap. 6.2 Datenflussanalyse

265

Zurück zum Anfang: Zur Programm-analyse

...speziell *Datenflussanalyse*

üblich ist...

- die Repräsentation von Programmen durch (nichtdeterministische) *Flussgraphen*

Kap. 6.2 Datenflussanalyse

266

Flussgraph

Ein (nichtdeterministischer) *Flussgraph* ist ein Quadrupel $G = (N, E, s, e)$ mit

- Knotenmenge (engl. *Nodes*) N
- Kantenmenge (engl. *Edges*) $E \subseteq N \times N$
- ausgezeichnetem Startknoten s ohne Vorgänger und
- ausgezeichnetem Endknoten e ohne Nachfolger

Knoten repräsentieren Programmpunkte, Kanten repräsentieren die Verzweigungsstruktur. Elementare Programmmanipulationen (Zuweisungen, Tests) können wahlweise durch Knoten oder Kanten repräsentiert werden.

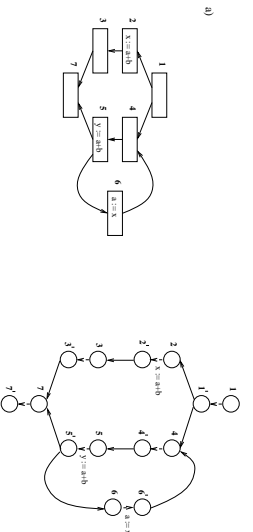
~ Darstellungsvarianten: Knoten- vs. kantenbenannte Flussgraphen

Kap. 6.2 Datenflussanalyse

267

Veranschaulichung

Knoten- vs. kantenbenannte Flussgraphen
(hier mit Einzelanweisungsbenennung)



Kap. 6.2 Datenflussanalyse

268

Flussgraphen

Darstellungsvarianten...

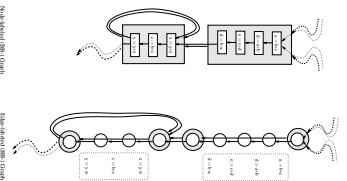
- Knotenbenannte Graphen
 - Einzelanweisungsgraphen (SI-Graphen)
 - Basisblockgraphen (BB-Graphen)
- Kantenbenannte Graphen
 - Einzelanweisungsgraphen (SI-Graphen)
 - Basisblockgraphen (BB-Graphen)

In der Folge werden wir bevorzugt kantenbenannte SI-Graphen betrachten.

Kap. 6.2 Datenflussanalyse

269

Knoten- vs. kantenbenannte Flussgraphen



Kap. 6.2 Datenflussanalyse

270

Bezeichnungen

Sei $G = (N, E, s, e)$ ein Flussgraph, seien m, n zwei Knoten aus N . Dann bezeichne:

- $P_G[m, n]$: ...die Menge aller Pfade von m nach n
- $P_G[m, n]$: ...die Menge aller Pfade von m zu einem Vorgänger von n
- $P_G[m, n]$: ...die Menge aller Pfade von einem Nachfolger von m nach n
- $P_G[m, n]$: ...die Menge aller Pfade von einem Nachfolger von m zu einem Vorgänger von n

Bem.: Wenn G aus dem Kontext eindeutig hervorgeht, schreiben wir einfacher auch P statt P_G .

Kap. 6.2 Datenflussanalyse

271

Datenflussanalysesemantik

- (Lokale) abstrakte Semantik
 - 1. Ein Datenflussanalyseverband $\mathcal{C} = (\mathcal{C}, \sqcap, \sqcup, \perp, \top)$
 - 2. Ein Datenflussanalysefunktional $\llbracket \cdot \rrbracket : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$
- Eine Anfangsinformation/-zusicherung: $c_s \in \mathcal{C}$

Kap. 6.2 Datenflussanalyse

272

Globalisierung einer lokalen abstrakten Semantik

Zwei Strategien:

- "Meet over all Paths"-Ansatz (MOP)
~> spezifizierende Lösung
- Maximaler Fixpunktansatz (MaxFP)
~> berechenbare Lösung

Kap. 6.2 Datenflussanalyse

273

Kapitel 6.3 MOP -Ansatz

Kap. 6.3 MOP -Ansatz

274

Der MOP -Ansatz

Zentral: Ausdehnung der lokalen abstrakten Semantik auf Pfad-
de

$$\llbracket p \rrbracket = d_f \begin{cases} Id_{\mathcal{C}} & \text{falls } q < 1 \\ \llbracket (e_2, \dots, e_q) \rrbracket \circ \llbracket e_1 \rrbracket & \text{sonst} \end{cases}$$

Kap. 6.3 MOP -Ansatz

275

Die MOP -Lösung

$$\forall c_s \in \mathcal{C} \forall n \in \mathbb{N}. \text{MOP}_{c_s}(n) = \sqcap \{ \llbracket p \rrbracket (c_s) \mid p \in \mathbf{P}[s, n] \}$$

Kap. 6.3 MOP -Ansatz

276

Kapitel 6.4 MaxFP -Ansatz

Kap. 6.4 MaxFP -Ansatz

277

Der MaxFP -Ansatz

Zentral: Das MaxFP -Gleichungssystem:

$$\text{inf}_{c_s}(n) = \begin{cases} \sqcap \{ \llbracket (m, n) \rrbracket (\text{inf}_{c_s}(m)) \} & \text{falls } n = s \\ \text{sonst} & \text{sonst} \end{cases}$$

Kap. 6.4 MaxFP -Ansatz

278

Die MaxFP -Lösung

$$\forall c_s \in \mathcal{C} \forall n \in \mathbb{N}. \text{MaxFP}_{\llbracket \cdot \rrbracket, c_s}(n) = d_f \text{inf}_{c_s}^*(n)$$

wobei $\text{inf}_{c_s}^*$ die größte Lösung des MaxFP -Gleichungssystems
bezeichnet.

Kap. 6.4 MaxFP -Ansatz

279

Eingabe: (1) Ein Flussgraph $G = (N, E, s, e)$, (2) eine (lokale) abstrakte Semantik bestehend aus einem Datenflussanalyseverband C , einem Datenflussanalysefunktional $[[\]]$: $E \rightarrow (C \rightarrow C)$, und (3) einer Anfangsinformation $c_s \in C$.

Ausgabe: Unter den Voraussetzungen des Effektivitätstheorems (später!) die *MaxFP*-solution. Abhängig von den Eigenschaften des Datenflussanalysefunktionals gilt dann:

(1) $[[\]]$ ist *distributiv*: Variable *inf* enthält für jeden Knoten die stärkste Nachbedingung bezüglich der Anfangsinformation c_s .

(2) $[[\]]$ ist *monoton*: Variable *inf* enthält für jeden Knoten eine sichere (d.h. untere) Approximation der stärksten Nachbedingung bezüglich der Anfangsinformation c_s .

Bemerkung: Die Variable *workset* steuert den iterativen Prozess. Ihre Elemente sind Knoten aus G , deren Annotation jüngst aktualisiert worden ist.

(Prolog-Implementierung von *inf* ohne *workset*)

FORALL $n \in N \setminus \{s\}$ DO *inf*[n] := \top OD;

inf[s] := c_s ;

workset := $\{s\}$;

(Hauptprozess: Iterative Fixpunktberechnung)

WHILE *workset* $\neq \emptyset$ DO

 CHOOSE $m \in \text{workset}$;

workset := *workset* $\setminus \{m\}$;

 (Aktualisiere die Nachfolgerumgebung von Knoten m)

 FORALL $n \in \text{succ}(m)$ DO

meet := $[[(m, n)]]$ \sqcap (*inf*[m]) \sqcap *inf*[n];

 IF *inf*[n] \supset *meet*

 THEN

inf[n] := *meet*;

workset := *workset* $\cup \{n\}$

 FI

 OD

 ESOOHC

OD.

Kapitel 6.5 Koinzidenz- und Sicherheitstheorem

Kap. 6.5 Koinzidenz- und Sicherheitstheorem

282

Korrektheit: Sicherheitstheorem

Theorem [Sicherheit (Safety)]

Die *MaxFP*-Lösung ist eine sichere (konservative), d.h. untere Approximation der *MOP*-Lösung, d.h.,

$$\forall c_s \in C \forall n \in N: \text{MaxFP}_{c_s}(n) \sqsubseteq \text{MOP}_{c_s}(n)$$

falls das Datenflussanalysefunktional $[[\]]$ monoton ist.

Kap. 6.5 Koinzidenz- und Sicherheitstheorem

284

Terminierung: Effektivitätstheorem

Theorem [Effektivität]

Der generische Fixpunktalgorithmus terminiert mit der *MaxFP*-Lösung, falls das Datenflussanalysefunktional monoton ist und der Verband die absteigende Kettenbedingung erfüllt.

Kap. 6.5 Koinzidenz- und Sicherheitstheorem

286

Hauptresultate

Zusammenhang von...

- *MOP* - und *MaxFP* -Lösung
 - Korrektheit
 - Vollständigkeit
- *MaxFP* -Lösung und generischem Algorithmus
 - Terminierung mit *MaxFP* -Lösung

Kap. 6.5 Koinzidenz- und Sicherheitstheorem

283

Vollständigkeit (und Korrektheit): Koinzidenztheorem

Theorem [Koinzidenz (Coincidence)]

Die *MaxFP*-solution stimmt mit der *MOP*-Lösung überein, d.h.,

$$\forall c_s \in C \forall n \in N: \text{MaxFP}_{c_s}(n) = \text{MOP}_{c_s}(n)$$

falls das Datenflussanalysefunktional $[[\]]$ distributiv ist.

Kap. 6.5 Koinzidenz- und Sicherheitstheorem

285

Nachzutragende Definitionen

...sind:

- Absteigende (aufsteigende) Kettenbedingung
- Monotonie und Distributivität von Datenflussanalysefunktionalen

Kap. 6.5 Koinzidenz- und Sicherheitstheorem

287

Definition [Ab-/aufsteigende Kettenbedingung]

Ein Verband $\tilde{C} = (C; \cap, \cup, \subseteq, \perp, \top)$ erfüllt

1. die *absteigende Kettenbedingung*, falls jede absteigende Kette stationär wird, d.h. für jede Kette $p_1 \supseteq p_2 \supseteq \dots \supseteq p_n \supseteq \dots$ gibt es einen Index $m \geq 1$ so dass $x_m = x_{m+j}$ für alle $j \in \mathbb{N}$ gilt
2. die *aufsteigende Kettenbedingung*, falls jede aufsteigende Kette stationär wird, d.h. für jede Kette $p_1 \subseteq p_2 \subseteq \dots \subseteq p_n \subseteq \dots$ gibt es einen Index $m \geq 1$ so dass $x_m = x_{m+j}$ für alle $j \in \mathbb{N}$ gilt

Kap. 6.5 Konduzenz- und Sicherheitstheorem

288

Zur Erinnerung: Oft nützlich

...ist folgende äquivalente Charakterisierung der Monotonie:

Lemma

Sei $\tilde{C} = (C; \cap, \cup, \subseteq, \perp, \top)$ ein vollständiger Verband und $f : C \rightarrow C$ eine Funktion auf C . Dann gilt:

$$f \text{ ist monoton} \iff \forall C' \subseteq C. f(\bigcap C') \subseteq \bigcap \{f(c) \mid c \in C'\}$$

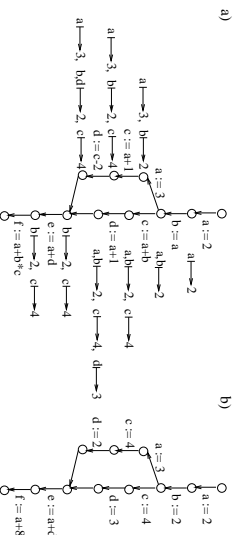
Kap. 6.5 Konduzenz- und Sicherheitstheorem

290

Kapitel 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

292



Beispiel: Einfache Konstanten

Ein typisches monotonen (nicht distributives) DFA-Problem...

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

294

...von Funktionen auf (Datenflussanalyse-) Verbänden.

Definition [Monotonie, Distributivität, Additivität]

Sei $\tilde{C} = (C; \cap, \cup, \subseteq, \perp, \top)$ ein vollständiger Verband und $f : C \rightarrow C$ eine Funktion auf C . Dann heißt f

1. *monoton* gdw $\forall a, d \in C. a \subseteq d \Rightarrow f(a) \subseteq f(d)$
(Erhalt der Ordnung der Elemente)
2. *distributiv* gdw $\forall C' \subseteq C. f(\bigcap C') = \bigcap \{f(c) \mid c \in C'\}$
(Erhalt der größten unteren Schranken)
3. *additiv* gdw $\forall C' \subseteq C. f(\bigcup C') = \bigcup \{f(c) \mid c \in C'\}$
(Erhalt der kleinsten oberen Schranken)

Kap. 6.5 Konduzenz- und Sicherheitstheorem

289

Monotonie und Distributivität

...von Datenflussanalysefunktionalen.

Definition

Ein Datenflussanalysefunktional $\llbracket \cdot \rrbracket : E \rightarrow (C \rightarrow C)$ heißt *monoton* (distributiv) gdw $\forall e \in E. \llbracket e \rrbracket$ ist monoton (distributiv).

Kap. 6.5 Konduzenz- und Sicherheitstheorem

291

Beispiel: Verfügbare Ausdrücke

...ein typisches distributives DFA-Problem.

• **Abstrakte Semantik für verfügbare Ausdrücke:**

1. *Datenflussanalyseverband:*
 $(C; \cap, \cup, \subseteq, \perp, \top) =_{df} (B; \wedge, \vee, \leq, ff, tt)$
2. *Datenflussanalysefunktional:* $\llbracket \cdot \rrbracket_{av} : E \rightarrow (B \rightarrow B)$ definiert durch
$$\forall e \in E. \llbracket e \rrbracket_{av} =_{df} \begin{cases} \text{Cs!ft} & \text{falls } \text{Comp } e \wedge \text{Transp } e \\ \text{Id}_B & \text{falls } \neg \text{Comp } e \wedge \text{Transp } e \\ \text{Cs!fff} & \text{sonst} \end{cases}$$

wobei

$$\tilde{B} =_{df} (B; \wedge, \vee, \leq, ff, tt)$$

den Verband der Wahrheitswerte bezeichnet mit $ff \leq tt$ und dem logischen "und" und "oder" als Schnitt- bzw. Vereinigungsoperation \cap and \cup .

Abstrakte Semantik für einfache Konstanten

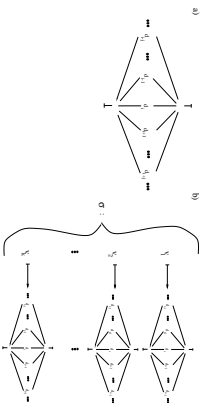
• **Abstrakte Semantik für einfache Konstanten:**

1. *Datenflussanalyseverband:*
 $(C; \cap, \cup, \subseteq, \perp, \top) =_{df} (\Sigma; \cap, \cup, \subseteq, \sigma_{\perp}, \sigma_{\top})$
2. *Datenflussanalysefunktional:*
 $\llbracket \cdot \rrbracket_{sc} : E \rightarrow (\Sigma \rightarrow \Sigma)$ definiert durch
$$\forall e \in E. \llbracket e \rrbracket_{sc} =_{df} \theta_e$$

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

295

Der "kanonische" Verband für Konstantenausbreitung/-faltung:



Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

296

Die Semantik von Termen $t \in \mathbf{T}$ ist gegeben durch die *Evaluationsfunktion*

$$\mathcal{E} : \mathbf{T} \rightarrow (\Sigma \rightarrow D)$$

die induktiv definiert ist durch:

$$\forall t \in \mathbf{T} \forall \sigma \in \Sigma. \mathcal{E}(t)(\sigma) =_{df} \begin{cases} \sigma(x) & \text{falls } t = x \in V \\ I_0(c) & \text{falls } t = c \in C \\ I_0(op)(\mathcal{E}(t_1)(\sigma), \dots, \mathcal{E}(t_n)(\sigma)) & \text{falls } t = op(t_1, \dots, t_n) \end{cases}$$

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

297

Nachzutragende Begriffe und Definitionen

... um die Definition der Termsemantik abzuschließen:

- Termsyntax
- Interpretation
- Zustand

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

298

Die Syntax von Termen (1)

Sei

- V eine Menge von Variablen und
- Op eine Menge von n -stelligem Operatoren, $n \geq 0$, sowie $C \subseteq Op$ die Menge der 0-stelligen Operatoren, der sog. *Konstanten* in Op .

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

299

Die Syntax von Termen (2)

Dann legen wir fest:

1. Jede Variable $v \in V$ und jede Konstante $c \in C$ ist ein Term.
2. Ist $op \in Op$ ein n -stelliger Operator, $n \geq 1$, und sind t_1, \dots, t_n Terme, dann ist auch $op(t_1, \dots, t_n)$ ein Term.
3. Es gibt keine weiteren Terme außer den nach den obigen beiden Regeln konstruierbaren.

Die Menge aller Terme bezeichnen wir mit \mathbf{T} .

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

300

Interpretation

Sei D' ein geeigneter Datenbereich (z.B. die Menge der ganzen Zahlen), seien \perp und \top zwei ausgezeichnete Elemente mit $\perp, \top \notin D'$ und sei $D =_{df} D' \cup \{\perp, \top\}$.

Eine *Interpretation* über \mathbf{T} und D ist ein Paar $I \equiv (D, I_0)$, wobei

- I_0 eine Funktion ist, die mit jedem 0-stelligen Operator $c \in Op$ ein Datum $I_0(c) \in D'$ und mit jedem n -stelligen Operator $op \in Op$, $n \geq 1$, eine totale Funktion $I_0(op) : D^n \rightarrow D$ assoziiert, die als *strikt* angenommen wird (d.h. $I_0(op)(d_1, \dots, d_n) = \perp$, wann immer es ein $j \in \{1, \dots, n\}$ gibt mit $d_j = \perp$)

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

301

Menge der Zustände

$$\Sigma =_{df} \{ \sigma \mid \sigma : V \rightarrow D \}$$

... bezeichnet die Menge der *Zustände*, d.h. die Menge der Abbildungen σ von der Menge der Programmvariablen V auf einen geeigneten (hier nicht näher spezifizierten) Datenbereich D .

Insbesondere

- $\sigma_{\perp} : \dots$ bezeichnet den wie folgt definierten *total undefinierten* Zustand aus Σ : $\forall v \in V. \sigma_{\perp}(v) = \perp$

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

302

Zustandstransformationsfunktion

Die *Zustandstransformationsfunktion*

$$\theta_l : \Sigma \rightarrow \Sigma, \quad l \equiv x := t$$

ist definiert durch:

$$\forall \sigma \in \Sigma \forall y \in V. \theta_l(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{falls } y = x \\ \sigma(y) & \text{sonst} \end{cases}$$

Kap. 6.6 Beispiele: Verfügbare Ausdrücke und Einfache Konstanten

303

Kapitel 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

304

Basisblöcke: Vermeyitliche Vorteile

...und die Ihnen gemeinhin aus ihrer Verwendung zugeschriebenen Vorteile ("Folk Knowledge"):

- *Performanz*: "...weil weniger Knoten in die teure iterative Fixpunktberechnung involviert sind".
- *Kompaktheit*: "...weil größere Programme in den Hauptspeicher passen".

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

306

MOP - Ansatz (Einzelanweisungen)

... für kantenbenannte Einzelanweisungsgraphen:

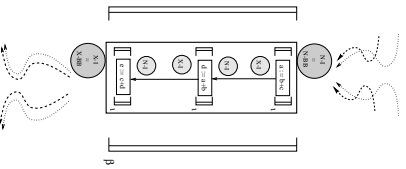
Die *MOP-Lösung*:

$$\forall c_s \in C \forall n \in N: MOP(\mathbb{I}, c_s)(n) = d_f \sqcap \{ \llbracket p \rrbracket, c_s \} | p \in P_G[s, n] \}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

308

Hierarchisierung durch Basisblöcke



In der Folge

...zum Einfluss der Repräsentation:

- Basisblöcke vs. Einzelanweisungen: Vor- und Nachteile
- Weitere Beispiele konkreter Datenflussanalysen/Datenflussanalyseprobleme

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

305

Basisblöcke: Sichere Nachteile

...und die in der Folge aus ihrer Verwendung behaupteten Nachteile:

- *Höhere konzeptuelle Komplexität*: ... Basisblöcke führen zu einer unerwünschten *Hierarchisierung*, die sowohl theoretische Überlegungen wie Implementierungen erschwert.
- *Notwendigkeit von Prä- und Postprozessen*: ... i.a. erforderlich, um die hierarchienutzten Zusatzprobleme zu behandeln (z.B. bei *dead code elimination*, *constant propagation*, ...); oder "trickbehäftete" Formulierungen nötig macht, um sie zu umgehen (z.B. bei *partial/redundancy elimination*).
- *Eingeschränkte Allgemeinheit*: ... bestimmte praktisch relevante Analysen und Optimierungen sind nur schwer oder gar nicht auf Basisblöckebene auszudrücken (z.B. *float variable elimination*).

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

307

MaxFP - Ansatz (Einzelanweisungen)

... für kantenbenannte Einzelanweisungsgraphen:

Die *MaxFP-Lösung*:

$$\forall c_s \in C \forall n \in N: MaxFP(\mathbb{I}, c_s)(n) = d_f \inf_{c_s}^*(n)$$

wobei $\inf_{c_s}^*$ die größte Lösung des *MaxFP* - Gleichungssystems bezeichnet:

$$\inf(n) = \begin{cases} c_s & \text{falls } n = s \\ \sqcap \{ \llbracket (m, n) \rrbracket, \inf(m) \} & \text{sonst} \end{cases}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

309

Vereinbarung

In der Folge werden

- Basisblockknoten mit fett gesetzten Buchstaben (**m**, **n**, ...)
- Einzelanweisungsknoten mit normal gesetzten Buchstaben (m, n, ...)

bezeichnet.

Weiters bezeichnen

- $\llbracket \rrbracket_\beta$ und
- $\llbracket \rrbracket_\beta$

(lokale) abstrakte Datenflussanalysefunktionale auf Basisblock- bzw. Einzelanweisungs- (Instruktions-) Ebene.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

311

... für knotenbenannte Basisblockgraphen:

Die MOP-Lösung: (Basisblockebene)

$$\forall c_s \in C \ \forall n \in N. \text{MOP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \\ (N\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)), X\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)))$$

mit

$$N\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \prod \{ \llbracket p \rrbracket_{\beta}(c_s) \mid p \in P_G[s, n] \} \\ X\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \prod \{ \llbracket p \rrbracket_{\beta}(c_s) \mid p \in P_G[s, n] \}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

312

MOP -Ansatz (Basisblöcke) (2)

Die MOP-Lösung: (Anweisungsebene)

$$\forall c_s \in C \ \forall n \in N. \text{MOP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \\ (N\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)), X\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)))$$

mit...

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

313

MOP -Ansatz (Basisblöcke) (3)

...mit

$$N\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \begin{cases} N\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(\text{block}(n))) \\ \text{falls } n = \text{start}(\text{block}(n)) \\ \llbracket p \rrbracket_{\beta}(N\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(\text{block}(n)))) \\ \text{sonst (p Prafixford} \\ \text{von start(block}(n)) \\ \text{bis (ausschlielich) } n) \end{cases}$$

$$X\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \llbracket p \rrbracket_{\beta}(X\text{-MOP}(\llbracket_{\beta, c_s} \rrbracket(\text{block}(n)))) \\ (\text{p Prafix von start(block}(n)) \\ \text{bis (einschlielich) } n)$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

314

MaxFP -Ansatz (Basisblöcke) (1)

...für knotenbenannte Basisblockgraphen:

Die MaxFP-Lösung: (Basisblockebene)

$$\forall c_s \in C \ \forall n \in N. \text{MaxFP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \\ (N\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)), X\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)))$$

mit

$$N\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \text{pre}_{c_s}^{\beta}(n) \quad \text{und}$$

$$X\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \text{post}_{c_s}^{\beta}(n)$$

wobei...

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

315

MaxFP -Ansatz (Basisblöcke) (2)

...wobei $\text{pre}_{c_s}^{\beta}$ und $\text{post}_{c_s}^{\beta}$ die größten Lösungen des folgenden Gleichungssystems bezeichnen:

$$\text{pre}(n) = \begin{cases} c_s \text{ falls } n = s \\ \prod \{ \text{post}(m) \mid m \in \text{pred}_G(n) \} \text{ sonst} \end{cases}$$

$$\text{post}(n) = \llbracket n \rrbracket_{\beta}(\text{pre}(n))$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

316

MaxFP -Ansatz (Basisblöcke) (3)

Die MaxFP-Lösung: (Anweisungsebene)

$$\forall c_s \in C \ \forall n \in N. \text{MaxFP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \\ (N\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)), X\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)))$$

mit

$$N\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \text{pre}_{c_s}^{\beta}(n) \quad \text{und}$$

$$X\text{-MFP}(\llbracket_{\beta, c_s} \rrbracket(n)) =_{df} \text{post}_{c_s}^{\beta}(n)$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

317

MaxFP -Ansatz (Basisblöcke) (4)

...wobei $\text{pre}_{c_s}^{\beta}$ und $\text{post}_{c_s}^{\beta}$ die größten Lösungen des folgenden Gleichungssystems bezeichnen:

$$\text{pre}(n) = \begin{cases} \text{pre}_{c_s}^{\beta}(\text{block}(n)) \\ \text{falls } n = \text{start}(\text{block}(n)) \\ \text{post}(m) \\ \text{sonst (m ist hier der eindeutig} \\ \text{bestimmte Vorganger von } n \\ \text{in block}(n)) \end{cases}$$

$$\text{post}(n) = \llbracket n \rrbracket_{\beta}(\text{pre}(n))$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

318

Verfügbarkeit von Ausdrücken (1)

...für knotenbenannte BB-Graphen:

Phase I: Die Basisblockebene

Lokale Prädikate: (assoziiert mit BB-Knoten)

- $\text{BB-XCOMP}_{\beta}(t)$: β enthält eine Anweisung t , die t berechnet, und weder ι , noch eine andere Anweisung von β nach ι modifiziert einen Operanden von t .
- $\text{BB-TRANSP}_{\beta}(t)$: β enthält keine Anweisung, die einen Operanden von t modifiziert.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

319

Verfügbarkeit von Ausdrücken (2)

Das Gleichungssystem von Phase I:

$$\text{BB-N-AVAIL}_\beta = \begin{cases} \text{ff} & \text{falls } \beta = s \\ \prod_{\beta \in \text{pred}(\beta)} \text{BB-X-AVAIL}_\beta & \text{sonst} \end{cases}$$

$$\text{BB-X-AVAIL}_\beta = \text{BB-N-AVAIL}_\beta \cdot \text{BB-TRANSP}_\beta + \text{BB-XCOMP}_\beta$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

320

Phase II: Die Anweisungsebene

Lokale Prädikate: (assoziiert mit EA-Knoten)

- **COMP** _{$\epsilon(t)$} : ι berechnet t .
- **TRANSP** _{$\epsilon(t)$} : ι modifiziert keinen Operanden von t .
- **BB-N-AVAIL**^{*}, **BB-X-AVAIL**^{*}: größte Lösung des Gleichungssystem von Phase I.

Das Gleichungssystem von Phase II:

$$\text{N-AVAIL}_\iota = \begin{cases} \text{BB-N-AVAIL}_{\text{b}^*\text{lock}(\iota)} & \text{falls } \iota = \text{start}(\text{b}^*\text{lock}(\iota)) \\ \text{X-AVAIL}_{\text{pred}(\iota)} & \text{sonst} \end{cases} \quad (\text{beachte: } |\text{pred}(\iota)| = 1)$$

$$\text{X-AVAIL}_\iota = \begin{cases} \text{BB-X-AVAIL}_{\text{b}^*\text{lock}(\iota)} & \text{falls } \iota = \text{end}(\text{b}^*\text{lock}(\iota)) \\ (\text{N-AVAIL}_\iota + \text{COMP}_\epsilon) \cdot \text{TRANSP}_\epsilon & \text{sonst} \end{cases}$$

Verfügbarkeit von Ausdrücken (4)

...für knotenbenannte EA-Graphen:

Lokale Prädikate: (assoziiert mit Knoten)

- **COMP** _{$\epsilon(t)$} : ι berechnet t .
- **TRANSP** _{$\epsilon(t)$} : ι modifiziert keinen Operanden von t .

Das Gleichungssystem:

$$\text{N-AVAIL}_\iota = \begin{cases} \text{ff} & \text{falls } \iota = s \\ \prod_{t \in \text{pred}(\iota)} \text{X-AVAIL}_t & \text{sonst} \end{cases}$$

$$\text{X-AVAIL}_\iota = (\text{N-AVAIL}_\iota + \text{COMP}_\epsilon) \cdot \text{TRANSP}_\epsilon$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

322

Verfügbarkeit von Ausdrücken (5)

...für kantenbenannte EA-Graphen:

Lokale Prädikate: (assoziiert mit EA-Kanten)

- **COMP** _{$\epsilon(t)$} : Anweisung ι von Kante ϵ berechnet t .
- **TRANSP** _{$\epsilon(t)$} : Anweisung ι von Kante ϵ ändert keinen Operanden von t .

Das Gleichungssystem:

$$\text{Avail}_n = \begin{cases} \text{ff} & \text{falls } n = s \\ \prod_{m \in \text{pred}(n)} (\text{Avail}_m + \text{COMP}_{(m,n)}) \cdot \text{TRANSP}_{(m,n)} & \text{sonst} \end{cases}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

323

Weitere Beispiele

- Constant folding (Konstantenfaltung)
- Faint Variable Elimination (Geistervariablenelimination)

...für die Varianten *knotenbenannte Basisblockgraphen* und *kantenbenannte Einzelanweisungsgraphen*.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

324

Konstantenfaltung: Einfache Konstanten

Zunächst zwei Hilfsfunktionen:

- Die Rückwärtssubstitution
- Die Zustandstransformation(sfunktion)

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

325

Rückwärtssubstitution & Zustands- transformation (1)

Sei $\iota \equiv (x := t)$ eine Anweisung. Dann definieren wir:

- Rückwärtssubstitution

$\delta_\iota : \mathbb{T} \rightarrow \mathbb{T}$ durch $\delta_\iota(s) =_{df} s[t/x]$ für alle $s \in \mathbb{T}$, wobei $s[t/x]$ die simultane Ersetzung aller Vorkommen von x in s durch t bezeichnet.

- Zustandstransformation (Erinnerung)

$$\theta_\iota(\sigma)(y) =_{df} \begin{cases} \mathcal{E}(t)(\sigma) & \text{falls } y = x \\ \sigma(y) & \text{sonst} \end{cases}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

326

Zusammenhang von δ und θ

Bezeichne \mathcal{I} die Menge aller Anweisungen.

Substitutionslemma

$$\forall t \in \mathbb{T} \forall \sigma \in \Sigma \forall \iota \in \mathcal{I}. \mathcal{E}(\delta_\iota(\sigma))(\sigma) = \mathcal{E}(t)(\theta_\iota(\sigma))$$

Beweis: ...induktiv über den Aufbau von t .

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

327

...für kantenbenannte Einzelanweisungsgraphen:

Bemerkung: Falscher CP-Macro!!!!

• $CalledProc_n \in \Sigma$

• $\sigma_0 \in \Sigma$ Anfangszusicherung

Das Gleichungssystem:

$\forall v \in V. CalledProc_n =$

$$\begin{cases} \sigma_0^{(v)} & \text{falls } n = s \\ \prod_1^t \mathcal{E}(\delta_{(m,n)}^{(v)})(CalledProc_m) & \text{sonst} \end{cases} \quad \text{falls } n = s$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

328

Rückwartssubstitution & Zustands- transformation (2)

Ausdehnung von δ und θ auf Prade (und somit insbesondere auch auf Basisblöcke):

- $\Delta_p : \mathbf{T} \rightarrow \mathbf{T}$ definiert durch $\Delta_p = df \delta_{n_q}$ für $q=1$ und durch $\Delta_{(n_1, \dots, n_{q-1})} \circ \delta_{n_q}$ für $q > 1$
- $\Theta_p : \Sigma \rightarrow \Sigma$ definiert durch $\Theta_p = df \theta_{n_1}$ für $q=1$ und durch $\Theta_{(n_2, \dots, n_q)} \circ \theta_{n_1}$ für $q > 1$.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

329

Zusammenhang von Δ und Θ

Bezeichne \mathcal{B} die Menge aller Basisblöcke.

Verallgemeinertes Substitutionslemma

$$\forall t \in \mathbf{T} \forall \sigma \in \Sigma \forall \beta \in \mathcal{B}. \mathcal{E}(\Delta_\beta(t))(\sigma) = \mathcal{E}(t)(\Theta_\beta(\sigma))$$

Beweis: ...induktiv über die Länge von p .

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

330

Einfache Konstanten (Basisblöcke) (1)

...für knotenbenannte Basisblockgraphen:

Phase I: Basisblockebene

Bemerkung:

- $\Delta_\beta(v) = df \delta_{l_1} \circ \dots \circ \delta_{l_q}(v)$, wobei $\beta \equiv l_1; \dots; l_q$.
- BB-N-CP $_{\beta}$, BB-X-CP $_{\beta}$, N-CP $_{\beta}$, X-CP $_{\beta} \in \Sigma$
- $\sigma_0 \in \Sigma$ Anfangszusicherung

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

331

Einfache Konstanten (Basisblöcke) (2)

Das Gleichungssystem von Phase I:

$$BB-N-CP_\beta = \begin{cases} \sigma_0 & \text{falls } \beta = s \\ \prod_1^m \{BB-X-CP_\beta \mid \beta \in pred(\beta)\} & \text{sonst} \end{cases}$$

$$\forall v \in V. BB-X-CP_\beta(v) = \mathcal{E}(\Delta_\beta(v))(BB-N-CP_\beta)$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

332

Einfache Konstanten (Basisblöcke) (3)

Phase II: Anweisungsebene

Vorberechnete Resultate (aus Phase I):

- BB-N-CP*, BB-X-CP*: die größte Lösung des Gleichungssystems von Phase I.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

333

Einfache Konstanten (Basisblöcke) (4)

Das Gleichungssystem von Phase II:

$$N-CP_\ell = \begin{cases} BB-N-CP_\ell^* \text{block}(\ell) & \text{falls } \ell = start(\text{block}(\ell)) \\ X-CP_{pred(\ell)} & \text{sonst (beachte: } |pred(\ell)| = 1) \end{cases}$$

$$\forall v \in V. X-CP_\ell(v) = \begin{cases} BB-X-CP_\ell^* \text{block}(\ell)(v) & \text{falls } \ell = start(\text{block}(\ell)) \\ \mathcal{E}(\delta_\ell(v))(N-CP_\ell) & \text{sonst} \end{cases}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

334

Geistervariablenelimination (1)

...für kantenbenannte Einzelanweisungsgraphen:

Lokale Prädikate: (assoziiert mit Einzelanweisungskanten)

- $USED_\epsilon(v)$: Anweisung ℓ von Kante ϵ benutzt v .
- $MOD_\epsilon(v)$: Anweisung ℓ von Kante ϵ modifiziert v .
- $Rel-Used_\epsilon(v)$: v ist eine Variable, die in der Anweisung ℓ von Kante ϵ vorkommt und von dieser Anweisung "zu leben gezwungen" wird (z.B. für ℓ eine Ausgabeanweisung).
- $Ass-Used_\epsilon(v)$: v ist eine in der Zuweisung ℓ von Kante ϵ rechtssseitig vorkommende Variable.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

335

Geistervariablenelimination (2)

Das Gleichungssystem:

$$\begin{aligned} \text{FAINT}_n(v) &= \prod_{m \in \text{Used}(n)} \overline{\text{Rel-Used}}_{(n,m)}(v) \cdot \\ & (\text{FAINT}_m(v) + \text{MOD}_{(n,m)}(v)) \cdot \\ & (\text{FAINT}_n(\text{LhsVar}_{(n,m)}) + \overline{\text{Ass-Used}}_{(n,m)}(v)) \end{aligned}$$

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

336

Geistervariablenelimination (3)

... ein typisches Beispiel für ein DFA-Problem, für das eine Formulierung

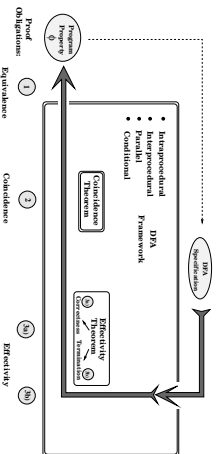
- auf (Knoten- und kantenbenannten) Einzelanweisungsgraphen offensichtlich ist,
- auf (knoten- und kantenbenannten) Basisblockgraphen alles andere als ersichtlich ist.

Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

337

Für uns reicht...

... die allgemeine Werkzeugkiste und das Wissen, dass je nach Repräsentationsvariante unterschiedlich aufwändige Spezifikations-, Implementierungs- und Beweisverpflichtungen entstehen.



Kap. 6.7 Pragmatik: Basisblöcke vs. Einzelanweisungen

338

Kapitel 7 Optimierung

Kap. 7 Optimierung

339

Kapitel 7.1 Motivation

Kap. 7.1 Motivation

340

Optimale Programmoptimierung

Zum Aufwärmen...

- Einige einfache Beispiele (siehe Tafel)
- Anschließend zum echten Einstieg...
- *Partielle Redundanzelimination (PRE)* alias *Code Motion (CM)*

Kap. 7.1 Motivation

341

Kap. 7.2 Einfache Konstanten

Kap. 7.2 Einfache Konstanten

342

Kap. 7.3 Geistervariablenelimination

Kap. 7.3 Geistervariablenelimination

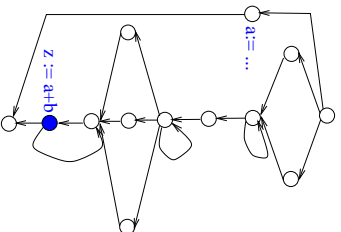
343

Kap. 7.4 Partielle Redundanzeliminati- on

Kap. 7.4 Partielle Redundanzeliminaton

344

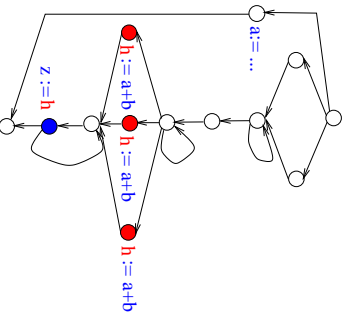
PRE – besonders wirksam im Zshg. mit Schleifen



Kap. 7.4 Partielle Redundanzeliminaton

346

Aber welches soll es sein? Dieses? Das vorige?



Kap. 7.4 Partielle Redundanzeliminaton

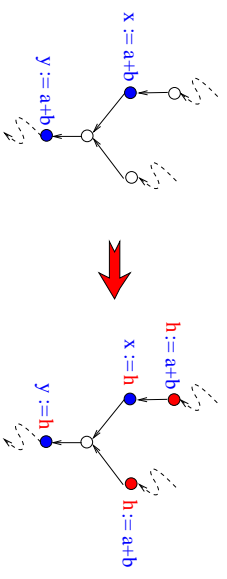
348

Auf die (Optimierungs-) Ziele kommt es an!

Kap. 7.4 Partielle Redundanzeliminaton

350

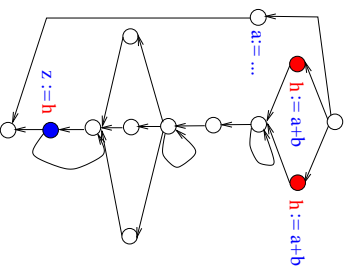
...Im Kern um die Vermeidung von Mehrfachberechnungen von
Werten



Kap. 7.4 Partielle Redundanzeliminaton

345

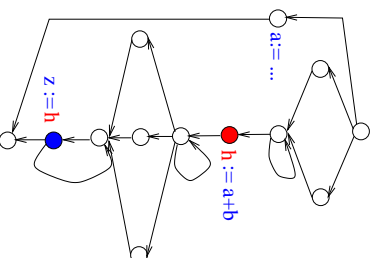
Ein redundanzfreies Programm



Kap. 7.4 Partielle Redundanzeliminaton

347

Oder vielleicht dieses?

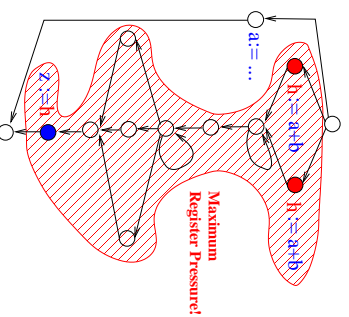


Kap. 7.4 Partielle Redundanzeliminaton

349

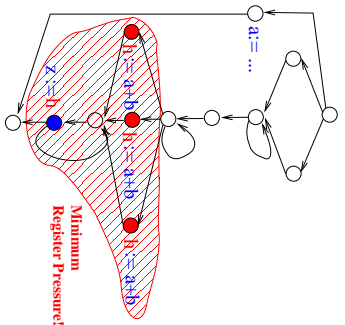
Die erste Transformation

...redundanzfrei, aber maximaler Registerdruck



Kap. 7.4 Partielle Redundanzeliminaton

351



Auf die (Optimierungs-) Ziele kommt es an!

In unseren Beispielen...

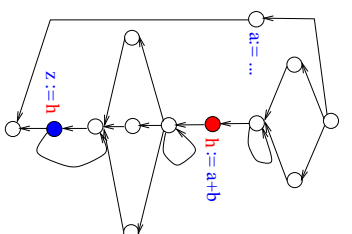
- **Performanz:** Vermeidung von Mehrfachberechnungen
 ~ Berechnungsqualität/-optimalität
- **Registerdruck:** Vermeidung unnötigen Schiebens
 ~ Lebenszeitqualität/-optimalität
- **Platz:** Vermeidung von Codereplikation
 ~ Platzqualität/-optimalität

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

Bezeichnungen (1)

Sei $G = (N, E, s, e)$ ein Flussgraph. Dann bezeichnen:

- $pred(n) =_{df} \{m \mid (m, n) \in E\}$: Menge aller *Vorgänger*
- $succ(n) =_{df} \{m \mid (n, m) \in E\}$: Menge aller *Nachfolger*
- $source(e)$, $dest(e)$: *Anfangs-* und *Endknoten* einer Kante
- **Endlicher Pfad:** Kantenfolge (e_1, \dots, e_k) mit $dest(e_i) = source(e_{i+1})$ für alle $1 \leq i < k$
- Statt Kantenfolgen betrachten wir entsprechend auch Knotenfolgen als Pfade, so zweckmäßig.



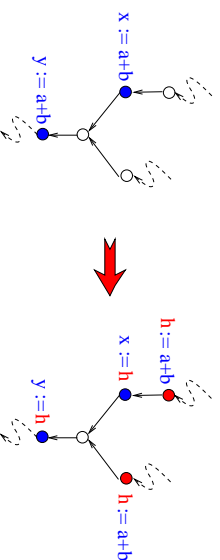
Zum Nachdenken

Sei P ein Programm mit Vorkommen partiell redundanter Berechnungen.

Ist es immer möglich, P so in ein Programm P' zu transformieren, dass P und P' bedeutungsgleich sind, P' aber frei von partiell redundanten Berechnungen ist?

In Medias Res – PRE

Ziel: ...die Vermeidung von Mehrfachberechnungen von Werten



Bezeichnungen (2)

- $p = (e_1, \dots, e_k)$ *Pfad* von m nach n , falls $source(e_1) = m$ und $dest(e_k) = n$
- $P[m, n]$: Menge aller Pfade von m nach n
- λ_p : *Länge* von p , d.h. die Anzahl der Kanten von p
- ϵ : Pfad der Länge 0
- $N_I \subseteq N$: Menge der *Join-Knoten*, d.h. Menge der Knoten mit mehr als einem Vorgänger
- $N_B \subseteq N$: Menge der *Branch-Knoten*, d.h. Menge der Knoten mit mehr als einem Nachfolger

Vereinbarung

Ohne Beschränkung der Allgemeinheit...

- Jeder Knoten in einem Flussgraphen liegt auf einem Pfad von s nach e

Intuition: Es gibt keine unerreichen Teile in einem Flussgraphen.

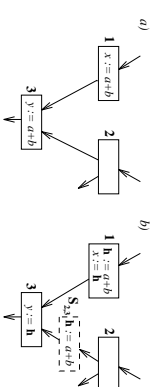
...eine generell übliche Vereinbarung für Analyse und Optimierung!

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

360

Eine Kante heißt *kritisch*, wenn sie von einem Branch- zu einem Join-Knoten führt.

Zur Illustration: ...mit Knoten **S**2,3 als *künstlichem (synthetic) Knoten*, der die kritische Kante von Knoten **2** nach **3** spaltet.



Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

361

Verfahrensspezifische Vereinbarung

Ohne Beschränkung der Allgemeinheit...

In der Folge betrachten wir Flussgraphen...

- in Form knotenbenannter EA-Graphen,
- bei denen alle Kanten, die in einem Join-Knoten enden, durch Einfügen eines sog. *künstlichen* Knotens aufgespalten sind,

...eine PRE-spezifische Vereinbarung.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

362

..dieser Vereinbarung:

- Der PRE-Prozess vereinfacht sich dadurch.

~> *Berechnungsoptimale* Ergebnisse können bereits erzielt werden, wenn erforderliche Hilfsvariableninitialisierungen einheitlich an Knotenanfängen erfolgen.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

363

Bemerkung

Berechnungsoptimale Ergebnisse sind auch möglich, wenn ausschließlich kritische Kanten gespaltet werden.

Dann aber muss ein PRE-Algorithmus in der Lage sein, sowohl N- als auch X-Initialisierungen (an Knoten) durchführen zu können.

Prinzipiell ist das kein Problem; mit vorstehender Vereinbarung ist die Präsentation des PRE-Algorithmus aber noch einfacher.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

364

Arbeitsplan

In der Folge werden wir definieren...

- Die Menge der PRE-Transformationen
- Die Menge der *zulässigen* PRE-Transformationen
- Die Menge der *berechnungsoptimalen* PRE-Transformationen
- Die BCM-Transformation als spezielle berechnungsoptimale PRE-Transformation

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

365

Die Menge der PRE-Transformationen

Generelles (Transformations-) Muster für einen Term t ...

- Deklariere eine neue Hilfsvariable h für t in G
- Füge an einigen Knoten von G die Anweisung $h := t$ ein
- Ersetze einige der originalen Vorkommen von t in G durch h

Bem.: t wird oft auch als *Kandidatenausdruck* bezeichnet.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

366

Beobachtung

Zwei (auf Knoten definierte) Prädikate

- $Insert_{CM}$
- $Repl_{CM}$

sind ausreichend, eine PRE- (bzw. CM-) Transformation vollständig zu beschreiben (beachte: die Deklaration der Hilfsvariablen h ist für jede CM-Transformation identisch und braucht deshalb nicht gesondert betrachtet zu werden).

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

367

CM-Transformationen

...bezeichne CM die Menge aller CM-Transformationen (für den Kandidatenausdruck t).

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

368

Beobachtung

Offenbar ist nicht jede Transformation in CM bedeutungserhaltend und damit akzeptabel.

Das führt uns auf den Begriff der *zulässigen* CM-Transformationen...

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

369

Zulässige CM-Transformationen

Sei $CM \in CM$.

CM heißt *zulässig*, wenn CM *sicher* und *korrekt* ist.

Intuition:

- *Sicher*: ...es gibt keinen Pfad, auf dem durch Einfügen einer Initialisierung ein neuer Wert berechnet wird.
- *Korrekt*: ...die Hilfsvariable ist an jeder Benutzungsstelle "richtig" initialisiert, d.h. sie enthält denselben Wert, den eine Neuberechnung von t an dieser Stelle liefert.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

370

Zur Formalisierung

...sind folgende (lokale) Prädikate erforderlich.

- $Comp(n)$: n enthält ein Vorkommen des Kandidatenausdrucks t .
- $Transp(n)$: n ist transparent für t , d.h., n weist keinem Operanden von t einen (neuen) Wert zu.

Ebenfalls nützlich:

- $Comp_{CM}(n) \stackrel{\text{def}}{=} Insert_{CM}(n) \vee Comp(n) \wedge \neg Repl_{CM}(n)$: Programmpunkte, an denen nach Anwendung von CM der Ausdruck t berechnet wird.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

371

Globalisierung von Prädikaten auf Pfade

Sei p ein Pfad und bezeichne p_i den i -ten Knoten von p .

Mit diesen Bezeichnungen treffen wir die folgende Vereinbarung:

- $Predicate^V(p) \iff \forall 1 \leq i \leq \lambda_p. Predicate(p_i)$
- $Predicate^E(p) \iff \exists 1 \leq i \leq \lambda_p. Predicate(p_i)$

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

372

Sicherheit und Korrektheit

Definition [Sicherheit und Korrektheit]

Sei $n \in N$. Wir definieren:

1. $Safe(n) \iff \stackrel{\text{def}}{=} \forall \langle n_1, \dots, n_k \rangle \in \mathbf{P}[s, e] \forall i. (n_i = n) \Rightarrow$
 $j) \exists j < i. Comp(n_j) \wedge Transp^V(n_j, \dots, n_{i-1}) \vee$
 $ii) \exists j \geq i. Comp(n_j) \wedge Transp^V(n_i, \dots, n_{j-1})$
2. Sei $CM \in CM$. Dann:
 $Correct_{CM}(n) \iff \stackrel{\text{def}}{=} \forall \langle n_1, \dots, n_k \rangle \in \mathbf{P}[s, n]$
 $\exists i. Insert_{CM}(n_i) \wedge Transp^V(n_i, \dots, n_{k-1})$

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

373

Aufwärts- und Abwärtssicherheit

Die Einschränkung der Definition für *Sicherheit* auf (I) bzw. (II) führt auf die Begriffe

- *Aufwärtssicherheit*
- *Abwärtssicherheit*

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

374

Intuition

Eine Berechnung t ist an einer Programmstelle n

- *aufwärtssicher*, wenn t auf allen Pfaden von s nach n berechnet wird und auf die jeweils letzte Berechnung von t keine Modifikation eines Operanden von t mehr erfolgt.
- *abwärtssicher*, wenn t auf allen Pfaden von n nach e berechnet wird und der jeweils ersten Berechnung von t keine Modifikation eines Operanden von t vorausgeht.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

375

Aufwärts- und Abwärtssicherheit

Definition [Aufwärts- und Abwärtssicherheit]

- $\forall n \in \mathbb{N}. U\text{-Safe}(n) \iff \#f$
 $\forall p \in \mathbf{P}[s, n] \exists i < \lambda_p. \text{Comp}(p_i) \wedge \text{Transp}^V(p[i, \lambda_p])$
- $\forall n \in \mathbb{N}. D\text{-Safe}(n) \iff \#f$
 $\forall p \in \mathbf{P}[n, e] \exists i \leq \lambda_p. \text{Comp}(p_i) \wedge \text{Transp}^V(p[1, i])$

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

376

Damit können wir jetzt genauer definieren...

Definition [Zulässige CM-Transformation]

Eine CM-Transformation $CM \in \mathcal{CM}$ heißt *zulässig* gdw für jeden Knoten $n \in \mathbb{N}$ gelten folgende beide Eigenschaften:

- $\text{Insert}_{CM}(n) \Rightarrow \text{Safe}(n)$
- $\text{Repl}_{CM}(n) \Rightarrow \text{Correct}_{CM}(n)$

Die Menge aller zulässigen CM-Transformationen bezeichnen wir mit \mathcal{CM}_{Adm} .

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

377

Erste Aussagen... (1)

Korrektheitslemma

$$\forall CM \in \mathcal{CM}_{Adm} \forall n \in \mathbb{N}. \text{Correct}_{CM}(n) \Rightarrow \text{Safe}(n)$$

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

378

Erste Aussagen... (2)

Sicherheitslemma

$$\forall n \in \mathbb{N}. \text{Safe}(n) \iff D\text{-Safe}(n) \vee U\text{-Safe}(n)$$

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

379

Berechnungsbesser, berechnungsoptimal

Eine CM-Transformation $CM \in \mathcal{CM}_{Adm}$ heißt *berechnungsbesser* als eine CM-Transformation CM' $\in \mathcal{CM}_{Adm}$ gdw

$$\forall p \in \mathbf{P}[s, e]. |\{i \mid \text{Comp}_{CM}(p_i)\}| \leq |\{i \mid \text{Comp}_{CM'}(p_i)\}|$$

Bemerkung: Die Relation "berechnungsbesser" ist eine Quasiordnung, d. h. eine reflexive und transitive Relation.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

380

Berechnungsoptimalität

Definition [Berechnungsoptimale CM-Transformation]

Eine zulässige CM-Transformation $CM \in \mathcal{CM}_{Adm}$ heißt *berechnungsoptimal* gdw CM ist berechnungsbesser als jede andere zulässige CM-Transformation.

Wir bezeichnen die Menge der berechnungsoptimalen CM-Transformationen mit $\mathcal{CM}_{CompOpt}$.

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

381

Konzeptuell

...PRE kann als zweistufiger Prozess gesehen werden

- Vorziehen von Ausdrücken (Expression hoisting)
...Vorziehen von Ausdrücken an "frühere" sichere Berechnungspunkte
- Beseitigung total redundanter Ausdrücke (Total redundancy elimination)
...beseitigen von Berechnungen, die durch das Vorziehen total redundant geworden sind

Kap. 7.4.1 Nützliche Begriffe und Bezeichnungen

382

Kap. 7.4.2 Busy Code Motion

Kap. 7.4.2 Busy Code Motion

383

Extreme Strategie – Frühhestitprinzip

Platziere Berechnungen so früh wie möglich...

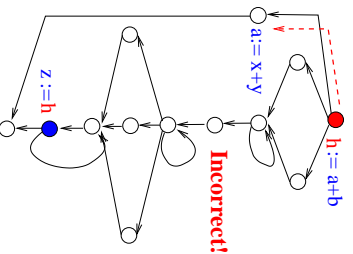
- Theorem [Berechnungsoptimalität]
 - ... vorziehen von Ausdrücken zu ihren frühesten sicheren Berechnungspunkten liefert berechnungsoptimale Programme
- ~> ... bekannt als Busy Code Motion

Kap. 7.4.2 Busy Code Motion

384

Beachte: Frühhest heißt in der Tat...

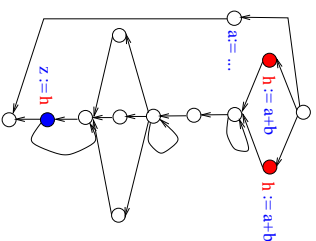
...so früh wie möglich, aber nicht früher!



Kap. 7.4.2 Busy Code Motion

386

Platzieren von Berechnungen so früh wie möglich...
...liefert berechnungsoptimale Programme.



Kap. 7.4.2 Busy Code Motion

385

Busy Code Motion

Intuition:

Platziere Berechnungen so *früh* wie möglich im Programm, ohne Sicherheit und Korrektheit zu verletzen!

Beachte: Berechnungen werden dadurch so weit wie möglich entgegen des Kontrollflusses verschoben

~> ...liefert die Motivation für die Wahl der Bezeichnung *busy*.

Kap. 7.4.2 Busy Code Motion

387

Frühhestit

Definition [Frühhestit]

$\forall n \in \mathbb{N}. \text{Earliest}(n) =_{df} \text{Safe}(n) \wedge$

$$\left\{ \begin{array}{l} \text{tt} \\ \vee \\ m \in \text{pred}(n) \end{array} \right. \text{falls } n = s \quad \neg \text{Transp}(m) \vee \neg \text{Safe}(m) \text{ sonst}$$

Kap. 7.4.2 Busy Code Motion

388

Die BCM-Transformation

- $\text{Insert}_{BCM}(n) =_{df} \text{Earliest}(n)$
- $\text{Repl}_{BCM}(n) =_{df} \text{Comp}(n)$

Kap. 7.4.2 Busy Code Motion

389

Das BCM-Theorem

BCM-Theorem

Die *BCM*-Transformation ist berechnungsoptimal, d.h., $BCM \in \mathcal{CM}_{Comp}^{Opt}$.

Der Beweis für das BCM-Theorem stützt sich ab auf das *Frühhestit*- und das *BCM-Lemma*...

Kap. 7.4.2 Busy Code Motion

390

Das Frühhestitlemma

Frühhestitlemma

Sei $n \in \mathbb{N}$. Dann gilt:

1. $\text{Safe}(n) \Rightarrow \forall p \in \mathbf{P}[s, n] \exists i \leq \lambda_p. \text{Earliest}(p) \wedge \text{Transp} \forall [i, \lambda_p]$
2. $\text{Earliest}(n) \iff D\text{-Safe}(n) \wedge \bigwedge_{m \in \text{pred}(n)} (\neg \text{Transp}(m) \vee \neg \text{Safe}(m))$
3. $\text{Earliest}(n) \iff \text{Safe}(n) \wedge \forall CM \in \mathcal{CM}_{Adm}. \text{Correct}_{CM}(n) \Rightarrow \text{Insert}_{CM}(n)$

Kap. 7.4.2 Busy Code Motion

391

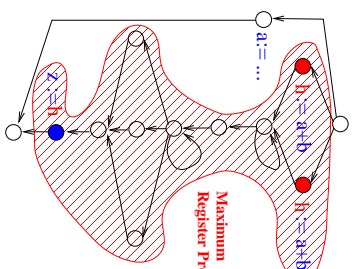
BCM-Lemma

Sei $p \in \mathbf{P}[s, e]$. Dann gilt:

1. $\forall i \leq \lambda_p. \text{Insert}_{BCM}(p_i) \iff \exists j \geq i. p[i, j] \in \text{FU-LTRg}(BCM)$
2. $\forall CM \in \mathcal{CM}_{\lambda_{min}} \forall i, j \leq \lambda_p. p[i, j] \in \text{LTRg}(BCM) \Rightarrow \text{Comp}_{CM}^{\exists}(p[i, j])$
3. $\forall CM \in \mathcal{CM}_{\text{CompOpt}} \forall i \leq \lambda_p. \text{Comp}_{CM}(p_i) \Rightarrow \exists j \leq i \leq l. p[i, j] \in \text{FU-LTRg}(BCM)$

Kap. 7.4.2 Busy Code Motion

392



Kap. 7.4.2 Busy Code Motion

393

Duale extreme Strategie – Spättestzeitprinzip

Platziere Berechnungen so spät wie möglich...

- Theorem [Optimalität]
 - ... vorziehen von Ausdrücken so wenig wie möglich, aber so weit wie nötig (um berechnungsoptimal zu werden), liefert berechnungsoptimale Programme mit minimalem Registerdruck
 - ~ ... bekannt als Lazy Code Motion

Kap. 7.4.3 Lazy Code Motion

395

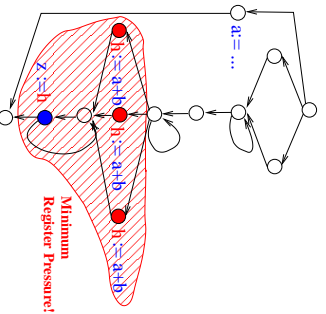
Kap. 7.4.3 Lazy Code Motion

Kap. 7.4.3 Lazy Code Motion

394

Die LCM-Transformation

... ebenfalls berechnungsoptimal, aber mit minimalem Registerdruck!



Kap. 7.4.3 Lazy Code Motion

396

Lazy Code Motion

Intuition:

Platziere Berechnungen so spät wie möglich im Programm, ohne Sicherheit, Korrektheit und Berechnungsoptimalität zu verletzen!

Beachte: Berechnungen werden dadurch so wenig wie möglich entgegen des Kontrollflusses verschoben

~ ... liefert die Motivation für die Wahl der Bezeichnung lazy.

Kap. 7.4.3 Lazy Code Motion

397

Arbeitsplan

In der Folge werden wir definieren...

- Die Menge der **lebenszeitoptimalen PRE-Transformationen**
- Die LCM-Transformation als eindeutig bestimmte einzige lebenszeitoptimale PRE-Transformation

Kap. 7.4.3 Lazy Code Motion

398

Zur Formalisierung

... ist der Begriff des Lebenszeitbereichs zentral.

Sei $CM \in \mathcal{CM}$.

- Lebenszeitbereich
 - $\text{LTRg}(CM) =_{df} \{p \mid \text{Insert}_{CM}(p_1) \wedge \text{Rep}_{CM}(p_{\lambda_p}) \wedge \neg \text{Insert}_{CM}^{\exists}(p[1, \lambda_p])\}$
- **Erstbenutzungslebenszeitbereich**
 - $\text{FU-LTRg}(CM) =_{df} \{p \in \text{LTRg}(CM) \mid \forall q \in \text{LTRg}(CM). (q \sqsubseteq p) \Rightarrow (q=p)\}$

Kap. 7.4.3 Lazy Code Motion

399

Erstbenutzungslebenszeitbereichslemma

Sei $CM \in \mathcal{CM}$, $p \in \mathbf{P}[s, e]$ und seien i_1, i_2, j_1, j_2 Indizes so dass $p[i_1, j_1] \in FU-LTRg(CM)$ und $p[i_2, j_2] \in FU-LTRg(CM)$. Dann gilt:

- entweder stimmen $p[i_1, j_1]$ und $p[i_2, j_2]$ überein, d.h. $i_1 = i_2$ und $j_1 = j_2$, oder
- $p[i_1, j_1]$ und $p[i_2, j_2]$ sind disjunkt, d.h., $j_1 < i_2$ oder $j_2 < i_1$.

Kap. 7.4.3 Lazy Code Motion

400

Lebenszeitbesser, lebenszeitoptimal

Eine CM-Transformation $CM' \in \mathcal{CM}$ heißt lebenszeitbesser als eine CM-Transformation $CM \in \mathcal{CM}$ gdw

$$\forall p \in LTRg(CM) \exists q \in LTRg(CM'), p \sqsubseteq q$$

Bemerkung. Die Relation "lebenszeitbesser" ist eine partielle Ordnung, d.h. eine reflexive, transitive und antisymmetrische Relation.

Kap. 7.4.3 Lazy Code Motion

401

Lebenszeitoptimalität

Definition [Lebenszeitoptimale CM-Transformation]

Eine berechnungsoptimale CM-Transformation $CM \in \mathcal{CM}_{CompOpt}$ heißt lebenszeitoptimal gdw CM ist lebenszeitbesser als jede andere berechnungsoptimale CM-Transformation.

Wir bezeichnen die Menge der lebenszeitoptimalen CM-Transformationen mit \mathcal{CM}_{LTOpt} .

Kap. 7.4.3 Lazy Code Motion

402

Wdhg: Mengen und Relationen 1(2)

Sei M eine Menge und R eine Relation auf M , d.h. $R \subseteq M \times M$.

Dann heißt R ...

- reflexiv gdw. $\forall m \in M. m R m$
- transitiv gdw. $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow m R p$
- antisymmetrisch gdw. $\forall m, n \in M. m R n \wedge n R m \Rightarrow m = n$

Wo wir dabei sind...

- symmetrisch gdw. $\forall m, n \in M. m R n \Leftrightarrow n R m$
- total gdw. $\forall m, n \in M. m R n \vee n R m$

Kap. 7.4.3 Lazy Code Motion

403

Wdhg: Mengen und Relationen 2(2)

Eine Relation R auf M heißt

- Quasiordnung gdw. R ist reflexiv und transitiv
- partielle Ordnung gdw. R ist reflexiv, transitiv und antisymmetrisch

Zur Vollständigkeit sei noch ergänzt...

- Äquivalenzrelation gdw. R ist reflexiv, transitiv und symmetrisch

...eine partielle Ordnung ist also eine antisymmetrische Quasiordnung, eine Äquivalenzrelation eine symmetrische Quasiordnung.

Kap. 7.4.3 Lazy Code Motion

404

Eindeutigkeit lebenszeitoptimaler PRE

Offensichtlich gilt:

$$\mathcal{CM}_{LTOpt} \subseteq \mathcal{CM}_{CompOpt} \subseteq \mathcal{CM}_{Adm} \subset \mathcal{CM}$$

Es gilt sogar weitergehend:

Theorem [Eindeutigkeit lebenszeitoptimaler CM-Transformationen]

$$|\mathcal{CM}_{LTOpt}| \leq 1$$

Kap. 7.4.3 Lazy Code Motion

405

Zur Entwicklung der LCM-Transformation

Zunächst folgende Beobachtung:

Lemma

$\forall CM \in \mathcal{CM}_{CompOpt} \forall p \in LTRg(CM) \exists q \in LTRg(BCM). p \sqsubseteq q.$

Intuitiv:

- Keine berechnungsoptimale CM-Transformation platziert die Berechnungen früher als die BCM-Transformation
- Die BCM-Transformation ist diejenige berechnungsoptimale CM-Transformation mit maximalem Registerdruck

Kap. 7.4.3 Lazy Code Motion

406

Verzögerbarkeit

Definition [Verzögerbarkeit]

$\forall n \in N. Delayed(n) \xrightarrow{\lambda f}$

$\forall p \in \mathbf{P}[s, n] \exists i \leq \lambda_p. \text{Earliest}(p) \wedge \neg \text{Comp}^3(p[i, \lambda_p])$

Kap. 7.4.3 Lazy Code Motion

407

Das Verzögerbarkeitslemma

Verzögerbarkeitslemma

1. $\forall n \in \mathbb{N}$. $Delayed(n) \Rightarrow D\text{-Safe}(n)$
2. $\forall p \in \mathbf{P}[s, e] \forall i \leq \lambda_p$. $Delayed(p_i) \Rightarrow \exists j \leq i \leq l$. $p[j, l] \in FU\text{-LFRg}(BCM)$
3. $\forall CM \in \mathcal{CM}_{CompOpt} \forall n \in \mathbb{N}$. $Comp_{CM}(n) \Rightarrow Delayed(n)$

Kap. 7.4.3 Lazy Code Motion

408

Spätetheit

Definition [Spätetheit]

$\forall n \in \mathbb{N}$. $Latest(n) =_{df} Delayed(n) \wedge (Comp(n) \vee \bigvee_{m \in succ(n)} \neg Delayed(m))$

Kap. 7.4.3 Lazy Code Motion

409

Das Spätetheitlemma

Spätetheitlemma

1. $\forall p \in LFRg(BCM) \exists i \leq \lambda_p$. $Latest(p_i)$
2. $\forall p \in LFRg(BCM) \forall i \leq \lambda_p$. $Latest(p_i) \Rightarrow \neg Delayed^{\exists}(p[i, \lambda_i])$

Kap. 7.4.3 Lazy Code Motion

410

Die ALCM-Transformation

Die "Almost Lazy Code Motion" Transformation...

- $Insert_{ALCM}(n) =_{df} Latest(n)$
- $Repl_{ALCM}(n) =_{df} Comp(n)$

Kap. 7.4.3 Lazy Code Motion

411

Fast lebenszeitoptimal

Definition [Fast lebenszeitoptimale CM-Transformation]

Eine lebenszeitoptimale CM-Transformation $CM \in \mathcal{CM}_{CompOpt}$ heißt *fast lebenszeitoptimal* gdw

$$\forall p \in LFRg(CM). \lambda_p \geq 2 \Rightarrow \forall CM' \in \mathcal{CM}_{CompOpt} \exists q \in LFRg(CM'). p \sqsubseteq q$$

Wir bezeichnen die Menge der fast lebenszeitoptimalen CM-Transformationen mit \mathcal{CM}_{ALLOpt} .

Kap. 7.4.3 Lazy Code Motion

412

Das ALCM-Theorem

ALCM-Theorem

Die $ALCM$ -Transformation ist fast lebenszeitoptimal, d.h., $ALCM \in \mathcal{CM}_{ALLOpt}$.

Kap. 7.4.3 Lazy Code Motion

413

Isolierte Berechnungen

Definition [CM-Isolation]

$\forall CM \in \mathcal{CM} \forall n \in \mathbb{N}$. $Isolated_{CM}(n) \iff_{df} \forall p \in \mathbf{P}[n, e] \forall 1 < i \leq \lambda_p$. $Repl_{CM}(p_i) \Rightarrow Insert_{CM}^{\exists}(p[1, i])$

Kap. 7.4.3 Lazy Code Motion

414

Das Isolationslemma

Isolationslemma

1. $\forall CM \in \mathcal{CM} \forall n \in \mathbb{N}$. $Isolated_{CM}(n) \iff \forall p \in LFRg(CM). (n) \sqsubseteq p \Rightarrow \lambda_p = 1$
2. $\forall CM \in \mathcal{CM}_{CompOpt} \forall n \in \mathbb{N}$. $Latest(n) \Rightarrow (Isolated_{CM}(n) \iff Isolated_{BCM}(n))$

Kap. 7.4.3 Lazy Code Motion

415

Die LCM-Transformation

- $Insert_{LCM}(n) =_{df} Latest(n) \wedge \neg Isolated_{BCM}(n)$
- $Repl_{LCM}(n) =_{df} Comp(n) \wedge \neg(Latest(n) \wedge Isolated_{BCM}(n))$

Kap. 7.4.3 Lazy Code Motion

416

Das LCM-Theorem

LCM-Theorem

Die LCM -Transformation ist lebenszeitoptimal, d.h., $LCM \in CM_{LifeTime}$.

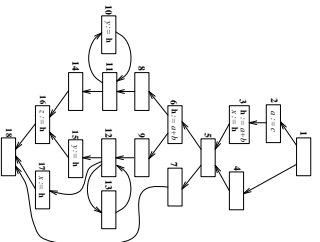
Kap. 7.4.3 Lazy Code Motion

417

Kap. 7.4.4 Ein größeres Beispiel

Kap. 7.4.4 Ein größeres Beispiel

418

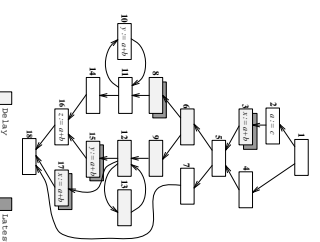


Ein größeres Beispiel zur Illustration (2)

Das Resultat der BCM-Transformation...

Kap. 7.4.4 Ein größeres Beispiel

420



Ein größeres Beispiel zur Illustration (3)

Verzögerte und späteste Berechnungspunkte...

Kap. 7.4.4 Ein größeres Beispiel

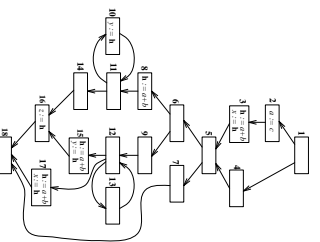
421

late

isolated

Ein größeres Beispiel zur Illustration (4)

Das Resultat der ALCM-Transformation...

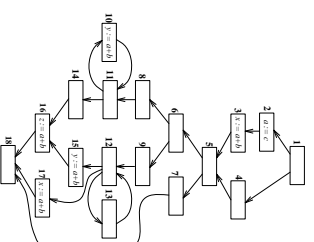


Kap. 7.4.4 Ein größeres Beispiel

422

Ein größeres Beispiel zur Illustration (1)

Das Ausgangsprogramm...

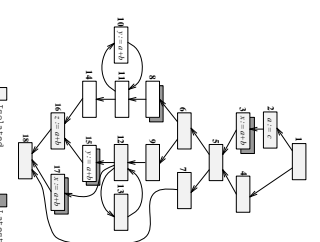


Kap. 7.4.4 Ein größeres Beispiel

419

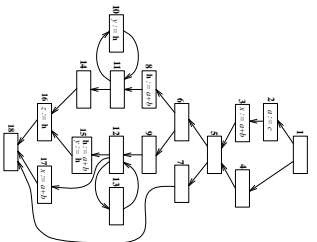
Ein größeres Beispiel zur Illustration (5)

Späteste und isolierte Berechnungspunkte...



Kap. 7.4.4 Ein größeres Beispiel

423



Zur Implementierung der BCM-Transformation auf EA-Graphen

...auf Einzelanweisungs niveau, hier für knotenbenannte EA-Graphen.

Beachte: ...wir nehmen für das folgende an, dass nur kritische Kanten gespalten sind (deshalb N- und X-Einsetzungen).

Kap. 7.4.5 Implementierungspragmatik

Busy Code Motion (EA-1)

1. Die Analysen für Aufwärts- und Abwärtsicherheit

Lokale Prädikate:

- $COMP_i(t)$: t berechnet
- $TRANSP_i(t)$: t modifiziert keinen Operanden von i .

Busy Code Motion (EA-2)

Das Gleichungssystem für Aufwärtsicherheit:

$$N\text{-USAFE}_i = \begin{cases} \text{ff} & \text{falls } i = s \\ \prod_{t \in \text{pred}(i)} X\text{-USAFE}_t & \text{sonst} \end{cases}$$

$$X\text{-USAFE}_i = (N\text{-USAFE}_i + \text{COMP}_i) \cdot \text{TRANSP}_i$$

Busy Code Motion (EA-4)

2. Die Transformation: Einsetzungs- und Ersetzungspunkte

Lokale Prädikate:

- $N\text{-USAFE}_i^*$, $X\text{-USAFE}_i^*$, $N\text{-DSAFE}_i^*$, $X\text{-DSAFE}_i^*$: größte Lösungen der Gleichungssysteme für Aufwärts- und Abwärtsicherheit aus Schritt 1.

Busy Code Motion (EA-3)

Das Gleichungssystem für Abwärtsicherheit:

$$N\text{-DSAFE}_i = \text{COMP}_i + X\text{-DSAFE}_i \cdot \text{TRANSP}_i$$

$$X\text{-DSAFE}_i = \begin{cases} \text{ff} & \text{falls } i = e \\ \prod_{t \in \text{succ}(i)} N\text{-DSAFE}_t & \text{sonst} \end{cases}$$

Busy Code Motion (EA-5)

$$N\text{-INSERT}_i^{\text{BCM}} =_{df} N\text{-DSAFE}_i^* \cdot \prod_{t \in \text{pred}(i)} (X\text{-USAFE}_t^* + X\text{-DSAFE}_t^*)$$

$$X\text{-INSERT}_i^{\text{BCM}} =_{df} X\text{-DSAFE}_i^* \cdot \text{TRANSP}_i$$

$$\text{REPLACE}_i^{\text{BCM}} =_{df} \text{COMP}_i$$

Zur Implementierung der BCM-Transformation auf BB-Graphen (1)

...auf Basisblockniveau, hier für knotenbenannte BB-Graphen.

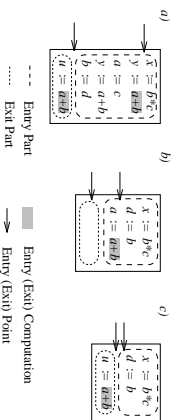
Beachte: ... wir nehmen für das folgende an, dass (1) nur kritische Kanten gespalten sind (deshalb N- und X-Einsetzungen), und (2) dass alle Redundanzen innerhalb eines Basisblocks schon durch einen Präprozess beseitigt sind.

Kap. 7.4.5 Implementierungspragmatik

432

Zur Implementierung der BCM-Transformation auf BB-Graphen (3)

Zur Illustration von Eingangs- und Ausgangsteil eines Basisblocks...



Kap. 7.4.5 Implementierungspragmatik

434

Busy Code Motion (BB-2)

Das Gleichungssystem für Aufwärtssicherheit:

$$BB\text{-}N\text{-}USAFE_{\beta} = \begin{cases} \text{ff} & \text{falls } \beta = s \\ \prod_{\beta_{\text{Prädik}}(t)} (BB\text{-}X\text{COMP}_{\beta} + BB\text{-}X\text{-}USAFE_{\beta}) & \text{sonst} \end{cases}$$

$$BB\text{-}X\text{-}USAFE_{\beta} = (BB\text{-}N\text{-}USAFE_{\beta} + BB\text{-}N\text{COMP}_{\beta}) \cdot BB\text{-}TRANSP_{\beta}$$

Kap. 7.4.5 Implementierungspragmatik

436

Busy Code Motion (BB-4)

2. Die Transformation: Einsetzungs- und Ersetzungspunkte

Lokale Prädikate:

- $BB\text{-}N\text{-}USAFE_{\beta}^*$, $BB\text{-}X\text{-}USAFE_{\beta}^*$, $BB\text{-}N\text{-}DSAFE_{\beta}^*$, $BB\text{-}X\text{-}DSAFE_{\beta}^*$: größte Lösungen der Gleichungssysteme für Aufwärts- und Abwärtssicherheit aus Schritt 1.

Kap. 7.4.5 Implementierungspragmatik

438

t-verfeinerte Flussgraphen...

Bezüglich einer Berechnung *t* lässt sich ein Basisblock *n* in zwei disjunkte Teile unterteilen:

- ein *Eingangsteil* (*entry part*), der aus allen Anweisungen bis zu und einschließlich der letzten Modifikation von *t* besteht
- ein *Ausgangsteil* (*exit part*), der aus den verbleibenden Anweisungen von *n* besteht.

Beachte: ein nichtleerer Basisblock hat stets einen nichtleeren Eingangsteil; im Unterschied dazu kann der Ausgangsteil leer sein (Zur Illustration siehe folgende Abbildung).

Kap. 7.4.5 Implementierungspragmatik

433

Busy Code Motion (BB-1)

1. Die Analysen für Aufwärts- und Abwärtssicherheit

Lokale Prädikate:

- $BB\text{-}N\text{COMP}_{\beta}(t)$: β enthält eine Anweisung *l*, die *t* berechnet, und der keine Anweisung vorausgeht, die einen Operanden von *t* modifiziert.
- $BB\text{-}X\text{COMP}_{\beta}(t)$: β enthält eine Anweisung *l*, die *t* berechnet, und weder *l* noch irgendeine andere Anweisung von β nach *l* modifiziert einen Operanden von *t*.
- $BB\text{-}TRANSP_{\beta}(t)$: β enthält keine Anweisung, die einen Operanden von *t* modifiziert.

Kap. 7.4.5 Implementierungspragmatik

435

Busy Code Motion (BB-3)

Das Gleichungssystem für Abwärtssicherheit:

$$BB\text{-}N\text{-}DSAFE_{\beta} = BB\text{-}N\text{COMP}_{\beta} + BB\text{-}X\text{-}DSAFE_{\beta} \cdot BB\text{-}TRANSP_{\beta}$$

$$BB\text{-}X\text{-}DSAFE_{\beta} = BB\text{-}X\text{COMP}_{\beta} + \begin{cases} \text{ff} & \text{falls } \beta = e \\ \prod_{\beta_{\text{Prädik}}(t)} BB\text{-}N\text{-}DSAFE_{\beta} & \text{sonst} \end{cases}$$

Kap. 7.4.5 Implementierungspragmatik

437

Busy Code Motion (BB-5)

$$N\text{-}INSERT_{\beta}^{\text{BCM}} =_{df} BB\text{-}N\text{-}DSAFE_{\beta} \cdot \prod_{\beta_{\text{Prädik}}(t)} (BB\text{-}X\text{-}USAFE_{\beta}^* + BB\text{-}X\text{-}DSAFE_{\beta}^*)$$

$$X\text{-}INSERT_{\beta}^{\text{BCM}} =_{df} BB\text{-}X\text{-}DSAFE_{\beta} \cdot BB\text{-}TRANSP_{\beta}$$

$$N\text{-}REPLACE_{\beta}^{\text{BCM}} =_{df} BB\text{-}N\text{COMP}_{\beta}$$

$$X\text{-}REPLACE_{\beta}^{\text{BCM}} =_{df} BB\text{-}X\text{COMP}_{\beta}$$

Kap. 7.4.5 Implementierungspragmatik

439

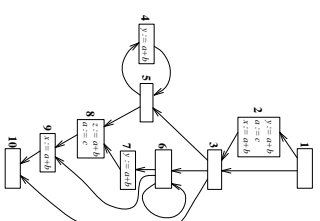
Die Gleichungssysteme für LCM

Ähnlich!

Kap. 7.4.5 Implementierungspragmatik

440

Das Ausgangsprogramm...

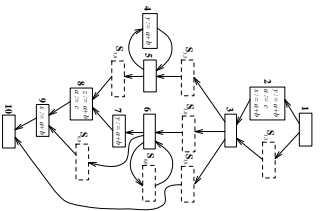


Kap. 7.4.5 Implementierungspragmatik

441

Ein größeres BB-Beispiel zur Illustration (2)

Das Ausgangsprogramm mit Kritischen Kanten gesparten...



Kap. 7.4.5 Implementierungspragmatik

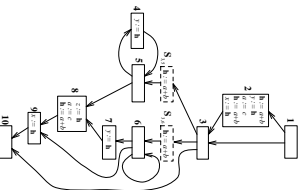
442

Kap. 7.4.5 Implementierungspragmatik

443

Ein größeres BB-Beispiel zur Illustration (4)

Das Ergebnis der BCM-Transformation...



Kap. 7.4.5 Implementierungspragmatik

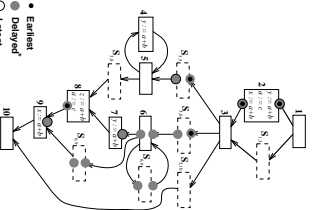
444

Kap. 7.4.5 Implementierungspragmatik

445

Ein größeres BB-Beispiel zur Illustration (6)

Das Ergebnis der ALCM-Transformation...



Kap. 7.4.5 Implementierungspragmatik

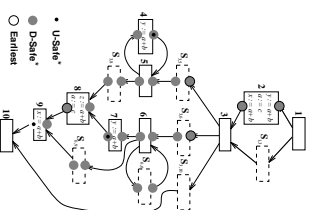
446

Kap. 7.4.5 Implementierungspragmatik

447

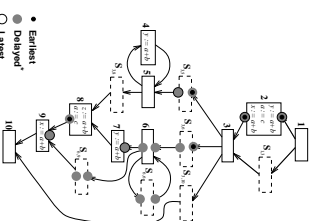
Ein größeres BB-Beispiel zur Illustration (3)

Die Berechnung der frühesten Berechnungspunkte...



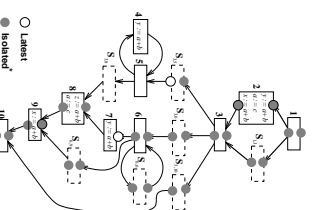
Ein größeres BB-Beispiel zur Illustration (5)

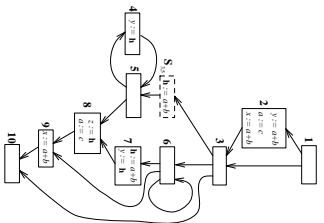
Die Berechnung der spätesten Berechnungspunkte...



Ein größeres BB-Beispiel zur Illustration (7)

Die Berechnung isolierter Berechnungspunkte...





Kap. 7.4.5 Implementierungsparamatik

Die bahnbrechende PRE-Formulierung von Morel/Renvoise

PRE ist untrennbar mit den Namen von E. Morel und C. Renvoise verknüpft. Ihr 1979 vorgestelltes Verfahren kann als "Urvater" aller CM-Verfahren angesehen werden und war bis in die 90er-Jahre hinein das "state of the art"-Verfahren für PRE.

Kennzeichnend für dieses Verfahren sind:

- 3 unidirektionale Bitvektoranalysen (AV, ANT, PAV)
- 1 bidirektionales Bitvektoranalyse (PP)

Kap. 7.4.5 PRE gemäß Morel/Renvoise

PRE gemäß Morel/Renvoise (2)

- Vorziehbarkeit (Anticipability):

$$\text{ANTIN}(n) = \text{COMP}(n) + \text{TRANSP}(n) * \text{ANTOUT}(n)$$

$$\text{ANTOUT}(n) = \begin{cases} \text{ff} & \text{falls } n = e \\ \prod_{m \in \text{stück}(n)} \text{ANTIN}(m) & \text{sonst} \end{cases}$$

Kap. 7.4.5 PRE gemäß Morel/Renvoise

PRE gemäß Morel/Renvoise (4)

- Platzierung möglich (Placement Possible):

$$\text{PPIN}(n) = \begin{cases} \text{CONST}(n)^* & \text{ff falls } n = s \\ \prod_{m \in \text{prk}(n)} (\text{PPOUT}(m) + \text{AVOUT}(m))^* & \text{sonst} \\ (\text{COMP}(n) + \text{TRANSP}(n) * \text{PPOUT}(n)) & \text{sonst} \end{cases}$$

$$\text{PPOUT}(n) = \begin{cases} \text{ff} & \text{falls } n = e \\ \prod_{m \in \text{stück}(n)} \text{PPIN}(m) & \text{sonst} \end{cases}$$

wobei $\text{CONST}(n) =_{df} \text{ANTIN}(n) * (\text{PAVIN}(n) + \text{COMP}(n) * \text{TRANSP}(n))$

Kap. 7.4.5 PRE gemäß Morel/Renvoise

Kapitel 7.4.5 PRE gemäß Mo-rel/Renvoise

Kap. 7.4.5 PRE gemäß Morel/Renvoise

PRE gemäß Morel/Renvoise (1)

- Verfügbarkeit (Availability):

$$\text{AVIN}(n) = \begin{cases} \text{ff} & \text{falls } n = s \\ \prod_{m \in \text{pred}(n)} \text{AVOUT}(m) & \text{sonst} \end{cases}$$

$$\text{AVOUT}(n) = \text{TRANSP}(n) * (\text{COMP}(n) + \text{AVIN}(n))$$

Kap. 7.4.5 PRE gemäß Morel/Renvoise

PRE gemäß Morel/Renvoise (3)

- Partielle Verfügbarkeit (Partial Availability):

$$\text{PAVIN}(n) = \begin{cases} \text{ff} & \text{falls } n = s \\ \sum_{m \in \text{pred}(n)} \text{PAVOUT}(m) & \text{sonst} \end{cases}$$

$$\text{PAVOUT}(n) = \text{TRANSP}(n) * (\text{COMP}(n) + \text{PAVIN}(n))$$

Kap. 7.4.5 PRE gemäß Morel/Renvoise

PRE gemäß Morel/Renvoise (5)

- Initialisierung:

$$\text{INSIN}(n) =_{df} \text{ff}$$

$$\text{INSOUT}(n) =_{df} \text{PPOUT}(n) * \text{AVOUT}(n) * (-\text{PPIN}(n) + \text{TRANSP}(n))$$

- Ersetzung: $\text{REPLACE}(n) =_{df} \text{COMP}(n) * \text{PPIN}(n)$

Kap. 7.4.5 PRE gemäß Morel/Renvoise

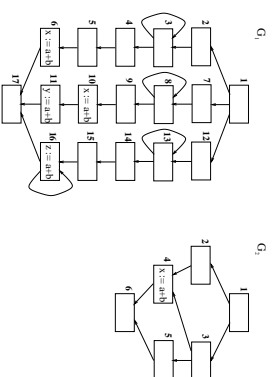
...gemäß Morel/Renvoise:

- Konzeptuell
 - Fehlende Berechnungsoptimalität
 - ~ nur aufgrund nicht gespaltener kritischer Kanten
 - Fehlende Lebzeitoptimalität
 - ~ Heuristische Behandlung
 - Technisch
 - Bidirektionalität
 - ~ konzeptuell und berechnungsmäßig komplexer
- ...das Transformationsergebnis liegt (nicht vorhersagbar) zwischen denen von BCM- und LCM-Transformation.

Kap. 7.4.5 PRE gemäß Morel/Renvoise

456

...folgende zwei Beispiele mithilfe des PRE-Verfahrens von Morel/Renvoise zu optimieren:



Kap. 7.4.5 PRE gemäß Morel/Renvoise

457

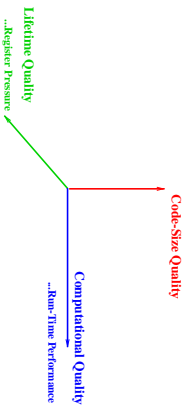
Kap. 7.4.6 Sparse Code Motion

Kap. 7.4.6 Sparse Code Motion

458

In der Folge...

(Modulare) Erweiterung von LCM, um Anwenderprioritäten zu berücksichtigen!



Kap. 7.4.6 Sparse Code Motion

460

There is more than speed!

Kap. 7.4.6 Sparse Code Motion

462

Heutzutage...

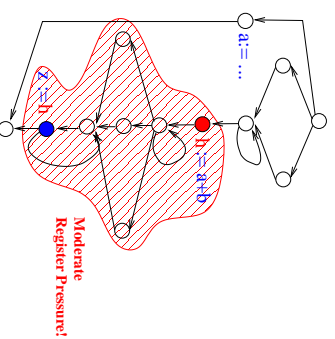
Lazy Code Motion ist...

- ...der de-facto Standardalgorithmus für PRE, der in aktuellen state-of-the-art Übersetzern zum Einsatz kommt
 - Gnu compiler family
 - Sun Sparc compiler family
 - ...

Kap. 7.4.6 Sparse Code Motion

459

...um auch diese Transformation zu ermöglichen:



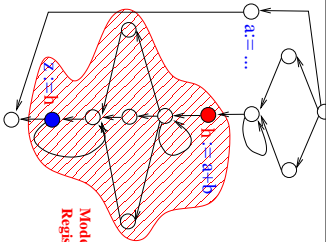
Kap. 7.4.6 Sparse Code Motion

461

There is more than speed!

Kap. 7.4.6 Sparse Code Motion

463



Der Weltmarkt für Mikroprozessoren 1999

Chip-Kategorie	Verkaufte Anzahl
Eingebettet 4-bit	2000 Millionen
Eingebettet 8-bit	4700 Millionen
Eingebettet 16-bit	700 Millionen
Eingebettet 32-bit DSP	400 Millionen
Desktop 32/64-bit	600 Millionen
	150 Millionen

~ 2%

... David Tenenhouse (Intel Director of Research). Hauptvortrag auf dem 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, Dezember 1999.

Code für eingebettete Systeme

Anforderungen...

- Performance (oft Echtzeitanforderungen)
- Codegröße (system-on-chip, on-chip RAM/ROM)
- ...

Für eingebettete Systeme...

...Codegröße ist oft eine kritischere Größe als Geschwindigkeit!

Angesichts dieses Trends...

...wie unterstützen traditionelle Übersetzer- und Optimierungstechnologien das spezielle Anforderungsprofil von Code für eingebettete Systeme?



Leider nur wenig.

1999

Chip-Kategorie	Verkaufte Anzahl
Eingebettet 4-bit	2000 Millionen
Eingebettet 8-bit	4700 Millionen
Eingebettet 16-bit	700 Millionen
Eingebettet 32-bit DSP	400 Millionen
Desktop 32/64-bit	600 Millionen
	150 Millionen

... David Tenenhouse (Intel Director of Research). Hauptvortrag auf dem 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, Dezember 1999.

Man denke an...

... domänenspezifische Prozessoren eingesetzt in eingebetteten Systemen

- **Telekommunikation**
 - Mobiltelefone, Pager, ...
- **Heimelektronik**
 - MP3-Spieler, Kameras, Spielkonsolen, ...
- **Autobilbereich**
 - GPS-Navigation, Airbags, ...
- ...

Code für eingebettete Systeme (Fortsetzung)

Anforderungen (und wie sie häufig adressiert werden...):

- Assem bierprogrammierung
- Händische Postoptimierung
- Schwächen...

Schwächen...

- Verzögerte Marktreife/-eintritt
- ...die Probleme werden zunehmend größer mit wachsender Komplexität.

Generell beobachtet man...

...einen **Trend hin zu Hochsprachenprogrammierung (C/C++)**

Unbestritten...

Traditionelle Optimierungen...

- ...sind nahezu ausschließlich auf Performanceoptimierung getrimmt
- ...sind nicht codegrößen sensitiv und bieten i.a. keinerlei Kontrolle über ihren Einfluss auf die Codegröße

... gilt dies für Optimierungen, die auf Codeverschiebung beruhen

Dazu gehören insbesondere

- Partial redundancy elimination
- Partial dead-code elimination
- Partial redundant-assignment elimination
- Strength reduction
- ...

Kap. 7.4.6 Sparse Code Motion

472

Erinnerung am Beispiel von LCM

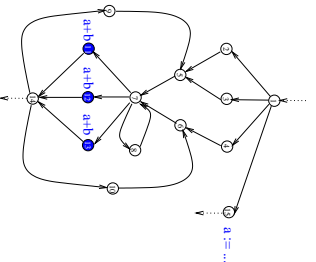
LCM kann konzeptuell als Ergebnis eines zweistufigen Prozesses gesehen werden...

1. Ausdrucksvorziehen (hoisting expressions)
 - ...zu ihren "frühesten" sicheren Berechnungspunkten
2. Ausdrucksverzögern (sinking expressions)
 - ...zu ihren "spätesten" sicheren und berechnungsoptimalen Berechnungspunkten

Kap. 7.4.6 Sparse Code Motion

474

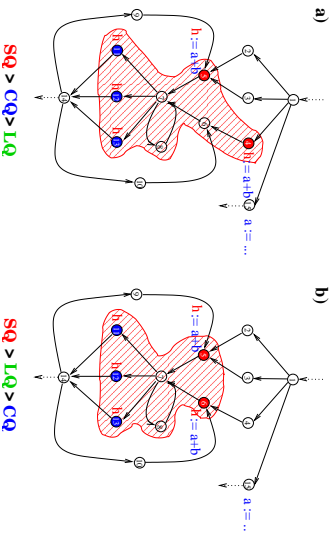
Laufendes Beispiel



Kap. 7.4.6 Sparse Code Motion

476

Laufendes Beispiel (Fortsetzung)



Kap. 7.4.6 Sparse Code Motion

478

PRE kann konzeptuell als zweistufiger Prozess gesehen werden...

1. Ausdrucksvorziehen
 - ...vorziehen von Ausdrücken an "frühere" sichere Berechnungspunkte
2. Totale Redundanzeliminierung
 - ...eliminieren von Berechnungen, die durch das Vorziehen total redundant geworden sind

Kap. 7.4.6 Sparse Code Motion

473

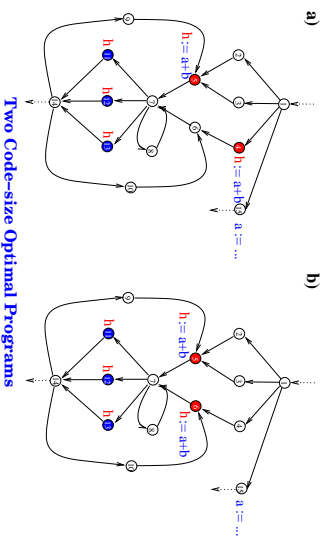
Auf dem Weg zu codegrößensensitiver PRE...

- Hintergrund: Klassische PRE
 - ~ Busy CM (BCM) / Lazy CM (LCM) (Knoop et al., PLDI'92)
 - Ausgezeichnet mit dem ACM SIGPLAN Most Influential PLDI Paper Award 2002 (for 1992)
 - Ausgewählt für "20 Years of the ACM SIGPLAN PLDI: A Selection" (60 Artikel aus ca. 600 Artikeln)
- Codegrößensensitive PRE (Knoop et al., POPL'00)
 - ~ ...: modulare Erweiterung von BCM/LCM
 - * Problemmodellierung und -lösung
 - ...basiert auf graphtheoretischen Hilfsmitteln
 - * Hauptergebnisse
 - ...: **Korrektheit, Optimalität**

Kap. 7.4.6 Sparse Code Motion

475

Laufendes Beispiel (Fortsetzung)

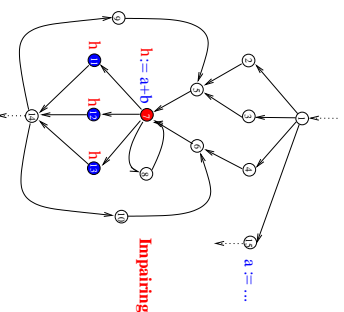


Kap. 7.4.6 Sparse Code Motion

477

Laufendes Beispiel (Fortsetzung)

Beachte: Folgende Transformation ist unerwünscht!



Kap. 7.4.6 Sparse Code Motion

479

Das Problem

- ...wir erhalten wir eine codegrößeminimale Platzierung der Berechnungen, d.h. eine Platzierung, die
 - zulässig (semantik- & performanzerhaltend)
 - codegrößeminimal ist?

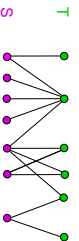
Lösung: Eine neue Sicht auf PRE

...betrachte PRE als ein Austauschproblem: Austauschen der ursprünglichen Berechnungen gegen neu eingesetzt

Der Clou: Benutze Graphtheorie!

...finde das Austauschproblem auf die Berechnung sog. tight sets in bipartiten Graphen zurück basierend auf maximalen matchings!

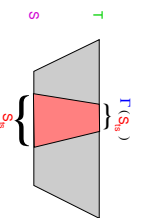
Bipartite Graphen



Tight Set

...eines bipartiten Graphen $(S \cup T, E)$: Teilmenge $S_{ts} \subseteq S$ mit

$$\forall S' \subseteq S: |S_{ts}| - |E(S_{ts})| \geq |S'| - |E(S')$$



Zwei Varianten: (1) **Größte** Tight Sets (2) **Kleinste** Tight Sets

Offensichtlich

...können wir auf vorgefertigte Standardalgorithmen aus der Graphtheorie zurückgreifen, um

- **Maximale Matchings** und
- **Tight sets** zu berechnen.

Damit reduziert sich unser PRE-Problem auf...

...die Konstruktion des bipartiten Graphen, der das Problem modelliert!

Algorithmus LTS (Largest Tight Sets)

Eingabe: Bipartiter Graph $(S \cup T, E)$, maximales Matching M .

Ausgabe: Größte tight set $T_{LTS}(S) \subseteq S$.

```

SM := S; D := {t ∈ T | t is unmatched};
WHILE D ≠ ∅ DO
  choose some x ∈ D; D := D \ {x};
  IF x ∈ S
    THEN SM := SM ∪ {x};
    D := D ∪ {y | {x,y} ∈ M}
  ELSE D := D ∪ (T(x) ∩ SM)
  FI
OD;
T_{LTS}(S) := SM
  
```

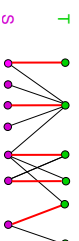
Wir müssen beantworten...

- Wo sind Initialisierungen vorzunehmen und wo sind ursprüngliche Berechnungen zu ersetzen?
- ...und Beweisen

- Warum dies korrekt (semantikerhaltend) ist
- Wie sich dies auf die Codegröße auswirkt
- Warum dies "optimal" bezüglich einer vorgegebenen Priorisierung von Zielen ist?

Für jede dieser Fragen werden wir ein spezielles Theorem angeben, das uns die entsprechende Antwort liefert!

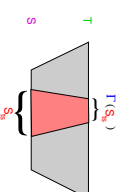
Bipartite Graphen



Tight Set

...eines bipartiten Graphen $(S \cup T, E)$: Teilmenge $S_{ts} \subseteq S$ mit

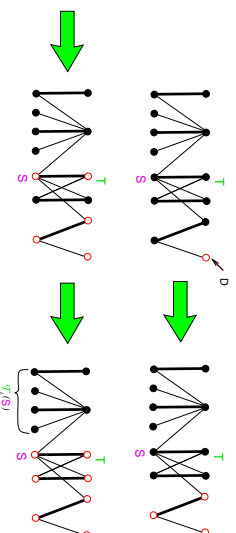
$$\forall S' \subseteq S: |S_{ts}| - |E(S_{ts})| \geq |S'| - |E(S')$$



Zwei Varianten: (1) **Größte** Tight Sets (2) **Kleinste** Tight Sets

Zur Berechnung größter/kleinsten Tight Sets

...auf Grundlage maximaler Matchings



Algorithmus STS (Smallest Tight Sets)

Eingabe: Bipartiter Graph $(S \cup T, E)$, maximales Matching M .

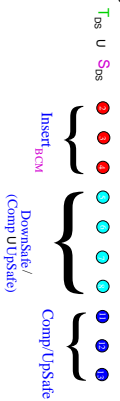
Ausgabe: Kleinste tight set $T_{STS}(S) \subseteq S$.

```

SM := ∅; A := {s ∈ S | s is unmatched};
WHILE A ≠ ∅ DO
  choose some x ∈ A; A := A \ {x};
  IF x ∈ S
    THEN SM := SM ∪ {x};
    A := A ∪ (T(x) \ SM)
  ELSE A := A ∪ {y | {x,y} ∈ M}
  FI
OD;
T_{STS}(S) := SM
  
```

Modellierung des Austauschproblems

Die Menge der Knoten



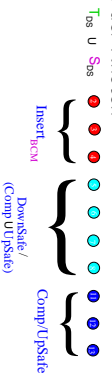
Die Menge der Kanten...

Kap. 7.4.6 Sparse Code Motion

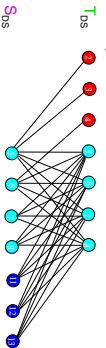
488

Modellierung des Austauschproblems

Die Menge der Knoten



Der bipartite Graph



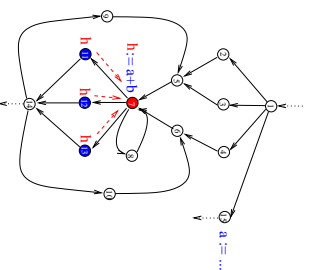
Die Menge der Kanten ... $\forall n \in S_{Dps} \forall m \in T_{Dps}$

$$\{n, m\} \in E_{Dps} \iff \#f \ m \in \text{Closure}(\text{pred}(n))$$

Kap. 7.4.6 Sparse Code Motion

490

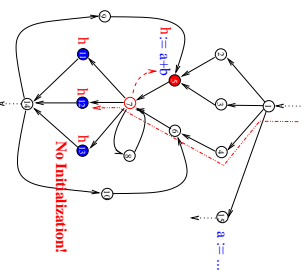
DownSafety-Hüllen – Die zentrale Idee 1(4)



Kap. 7.4.6 Sparse Code Motion

492

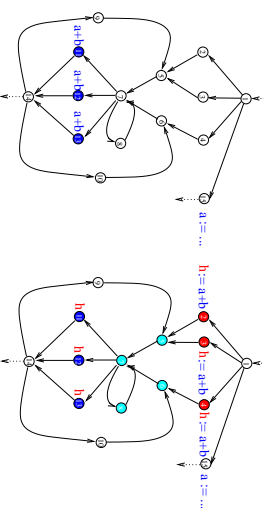
DownSafety-Hüllen – Die zentrale Idee 3(4)



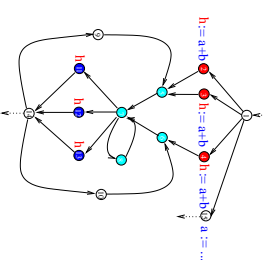
Kap. 7.4.6 Sparse Code Motion

494

a)



b)



Kap. 7.4.6 Sparse Code Motion

489

DownSafety-Hüllen

DownSafety-Hüllen

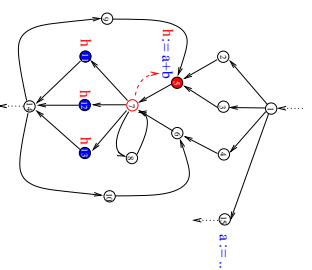
Für $n \in \text{DownSafe/Upsafe}$ ist die DownSafety-Hülle $\text{Closure}(n)$ die kleinste Menge von Knoten, so dass

1. $n \in \text{Closure}(n)$
2. $\forall m \in \text{Closure}(n) \setminus \text{Comp_succ}(m) \subseteq \text{Closure}(n)$
3. $\forall m \in \text{Closure}(n) \cdot \text{pred}(m) \cap \text{Closure}(n) \neq \emptyset \Rightarrow \text{pred}(m) \setminus \text{Upsafe} \subseteq \text{Closure}(n)$

Kap. 7.4.6 Sparse Code Motion

491

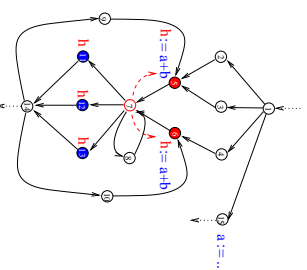
DownSafety-Hüllen – Die zentrale Idee 2(4)



Kap. 7.4.6 Sparse Code Motion

493

DownSafety-Hüllen – Die zentrale Idee 4(4)



Kap. 7.4.6 Sparse Code Motion

495

DownSafety-Hüllen

DownSafety-Hüllen

Für $n \in \text{DownSafe/Unsafe}$ ist die DownSafety-Hülle $\text{Closure}(n)$ die kleinste Menge von Knoten, so dass

- $n \in \text{Closure}(n)$
- $\forall m \in \text{Closure}(n) \setminus \text{Comp}, \text{succ}(m) \subseteq \text{Closure}(n)$
- $\forall m \in \text{Closure}(n), \text{pred}(m) \cap \text{Closure}(n) \neq \emptyset \Rightarrow \text{pred}(m) \setminus \text{Unsafe} \subseteq \text{Closure}(n)$

Kap. 7.4.6 Sparse Code Motion

496

DownSafety-Regionen

Einige Teilmengen von Knoten sind in besonderer Weise ausgezeichnet. Wir nennen diese Mengen DownSafety-Regionen...

- Eine Menge $R \subseteq N$ von Knoten heißt DownSafety-Region gdw
 - $\text{Comp} \setminus \text{Unsafe} \subseteq R \subseteq \text{DownSafe} \setminus \text{Unsafe}$
 - $\text{Closure}(R) = R$

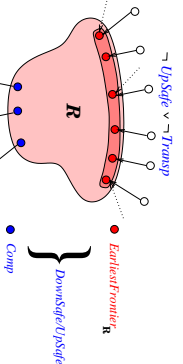
Kap. 7.4.6 Sparse Code Motion

497

Fundamental...

Initialisierungstheorem

Initialisierungen zulässiger PRE-Transformationen erfolgen stets am "ruhigsten Rand" von DownSafety-Regionen.



...charakterisiert erstmals alle semantikerhaltenden PRE-Transformationen.

Kap. 7.4.6 Sparse Code Motion

498

Hauptergebnisse / Erste Frage

1. Wo Initialisierungen vornehmen, warum ist es korrekt?

Intuitiv: am frühesten Rand der von der tight set induzierten DS-region...

Theorem 1 [Tight Sets: Initialisierungspunkte]

Sei $TS \subseteq SD_S$ eine tight set.

Dann ist $R_{TS} =_{df} \Gamma(TS) \cup (\text{Comp} \setminus \text{Unsafe})$

eine DownSafety-Region mit $\text{Body}_{R_{TS}} = TS$

Korrektheit

...unmittelbares Korollar aus Theorem 1 und dem Initialisierungstheorem

Kap. 7.4.6 Sparse Code Motion

500

Hauptergebnisse / Dritte Frage

3. Warum ist das Resultat optimal, d.h., codegrößenminimal?

Aufgrund einer inhärenten Eigenschaft von tight sets (non-negative deficiency)...

Optimalitätstheorem [Die Transformation]

Sei $TS \subseteq SD_S$ eine tight set.

- **Initialisierungspunkte:**
 $\text{Insert}_{TS, \text{Comp}} =_{df} \text{EarliestFrontier}_{R_{TS}} = R_{TS} \setminus TS$
- **Platzgewinn:**
 $\text{defic}(TS) =_{df} |TS| - |\Gamma(TS)| \geq 0$ max.

Kap. 7.4.6 Sparse Code Motion

502

Die Schlüsselfragen

...bezüglich Korrektheit und Optimalität:

1. Wo Initialisierungen vornehmen, warum ist es korrekt?
2. Wie ist der Effekt auf die Codegröße?
3. Warum ist das Resultat optimal, d.h., codegrößenminimal?

...drei Theoreme werden jeweils eine dieser Fragen beantworten.

Kap. 7.4.6 Sparse Code Motion

499

Hauptergebnisse / Zweite Frage

2. Wie ist der Effekt auf die Codegröße?

Intuitiv: Die Differenz aus eingesetzten und ersetzten Berechnungen...

Theorem 2 [DownSafety-Regionen: Platzgewinn]

Sei R eine DownSafety-Region

mit $\text{Body}_R =_{df} R \setminus \text{EarliestFrontier}_R$

Dann

- Platzgewinn aufgrund Einsetzens an

$\text{EarliestFrontier}_R:$

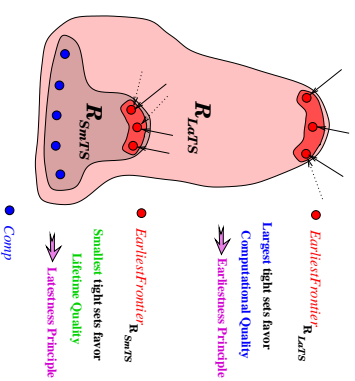
$|\text{Comp} \setminus \text{Unsafe}| - |\text{EarliestFrontier}_R| =$

$|\text{Body}_R| - |\Gamma(\text{Body}_R)| \quad_{df} = \text{defic}(\text{Body}_R)$

Kap. 7.4.6 Sparse Code Motion

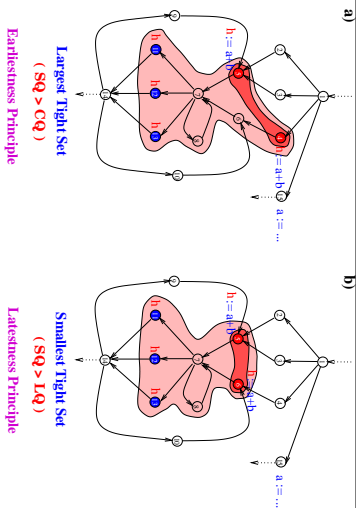
501

Größe vs. kleinste Tight Sets: Der Einfluss



Kap. 7.4.6 Sparse Code Motion

503

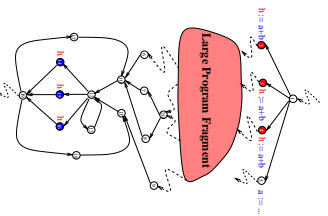


...auf einen Blick 2(2)

Choice of Priority	Apply	To	Using	Yields	Auxiliary Information Required
LQ					
SQ	Not meaningful: The identity, i.e., G itself is optimal!				
				Subsumed by $SQ > CQ$ and $SQ > LQ$!	
CQ	BCM	G			$\text{upset}(G), \text{downset}(G)$
$CQ > LQ$	LCM	G			$\text{upset}(G), \text{downset}(G), \text{delay}(G)$
$SQ \leq CQ$	SpCM	G	Largest tight set	$\text{SpCM}_{LTS}(G)$	$\text{upset}(G), \text{downset}(G)$
$SQ > CQ$	SpCM	G	Smallest tight set		$\text{upset}(G), \text{downset}(G)$
$CQ > SQ$	SpCM		Largest tight set		$\text{upset}(G), \text{downset}(G), \text{delay}(G)$
$CQ > SQ > LQ$	SpCM		Smallest tight set		$\text{upset}(LCM(G)), \text{downset}(LCM(G))$
$SQ > CQ > LQ$	SpCM	DL/SpCM _{LTS} (G)	Smallest tight set		$\text{upset}(G), \text{downset}(G), \text{delay}(\text{SpCM}_{LTS}(G)), \text{downset}(\text{DL/SpCM}_{LTS}(G))$

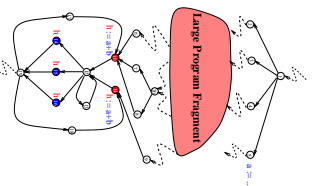
Flexibilität (2)

BCM ... Ein berechnungsoptimales Programm (CQ)

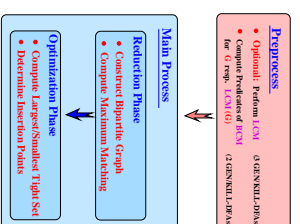


Flexibilität (4)

SpCM ... A Code-Size & Lifetime Optimal Program ($SQ > LQ$)

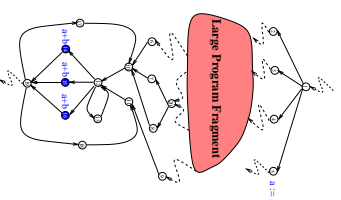


Blick 1(2)



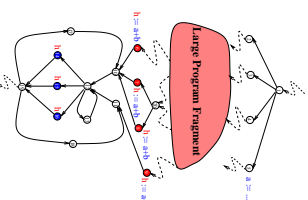
Flexibilität (1)

Das Ausgangsprogramm ...



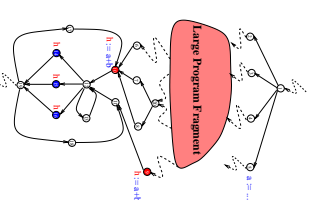
Flexibilität (3)

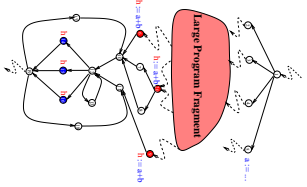
LCM ... A Computationally & Lifetime Opt. Program ($CQ > LQ$)



Flexibilität (5)

SpCM ... A Computationally and Lifetime Best Code-Size Optimal Program ($SQ > CQ > LQ$)





Ein Rückblick (fortgesetzt)

- **2000:** ... *origin of code-size sensitive PRE* [Knoop et al., POPL 2000]
 - ~ ... first to allow prioritization of goals
 - ~ ... rigorously be proven correct and optimal
 - ~ ... first to bridge the gap between traditional compilation and compilation for embedded systems
- ca. since 1997: ... *a new strand of research on PRE*
 - ~ Speculative PRE: Gupta, Horspool, Soffa, Xue, Scholz, Knoop, ...
- **2005:** ... *another fresh look at PRE (as maximum flow problem)*
 - ~ Unifying PRE and Speculative PRE [Jingling Xue and J. Knoop]

Warum lohnt es sich, PRE zu betrachten? (1)

Es ist...

- Relevant ... weit verbreitet in der Praxis
- Generell ... eine Familie von Optimierungen denn eine einzelne Optimierung
- Wohlverstanden ... bewiesen korrekt und *optimal*
- Herausfordernd ... konzeptuell einfach, aber weist eine Reihe kopfnussaufgebender Phänomene auf

Kap. 7.5 Optimalitätsphänomene

... auf die Entwurfstrategie von PRE:

- **1958:** ... *first glimpse of PRE*
 - ~ Ershov's work on *On Programming of Arithmetic Operations*
- < **1979** ... Special Techniques
 - ~ Total Redundancy Elimination, Loop Invariant Code Motion
- **1979:** ... *origin of contemporary PRE*
 - ~ Morel/Rennoise's seminal work on PRE
- **1992:** ... LCM [Knoop et al., PLDI'92]
 - ~ ... first to achieve comp. optimality with minimum register pressure
 - ~ ... first to rigorously be proven correct and optimal

Namen sind Nachrichten

Ein anderer Rückblick...

- < **1979** ... Special Techniques
 - ~ Total Redundancy Elimination, Loop Invariant Code Motion
- **1979** ... Partial Redundancy Elimination
 - ~ **Pioneering** ... Morel/Rennoise's bidirectional algorithm [1979]
 - ~ **Heuristic improvements** ... Dhamdhere [1988, 1991], Drechsler/Stadel [1988], Sorkin [1989], Dhamdhere/Rosen/Zadeck [1992], ...
- **1992** ... BCM & LCM [Knoop et al., PLDI'92]
 - ~ BCM ... first to achieve Computational Optimality: Earliestness Principle
 - ~ LCM ... first to achieve Comp. & Lifetime Optimality: Latestness Principle
 - ~ ... first to be purely unidirectional, however, not yet code-size sensitive.
- **2000/2004:** Code-Size Sensitive PRE [Knoop et al., POPL 2000, LCTES 2004]

Warum lohnt es sich, PRE zu betrachten? (2)

Zu guter letzt, PRE ist...

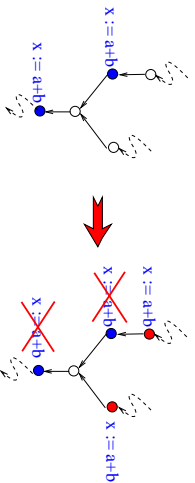
- Währlich klassisch ... blickt auf eine lange Geschichte zurück
 - Morel, E. and Rennoise, C. *Global Optimization by Suppression of Partial Redundancies*. CACM 22 (2), 96 - 103, 1979.
 - Ershov, A. P. *On Programming of Arithmetic Operations*. CACM 1 (8), 3 - 6, 1958.

Zurück zu CM im allgemeinen

Traditionell

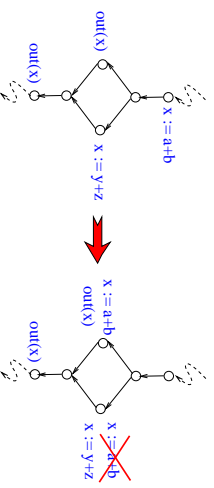
- Code (C) heißt Ausdrücke
 - Motion (M) heißt vorziehen
- Aber...
- CM ist mehr als vorziehen von Ausdrücken und PR(E)EI

...Anweisungen sind Code.



- Hier heißt CM Elimination partiell redundanter Anweisungen (PRAE)

...Anweisungen auch verzögert werden.



- CM heißt jetzt Elimination partiell toten Codes (PDCE)

Über den Entwurfsraum von CM-Algorithmen...

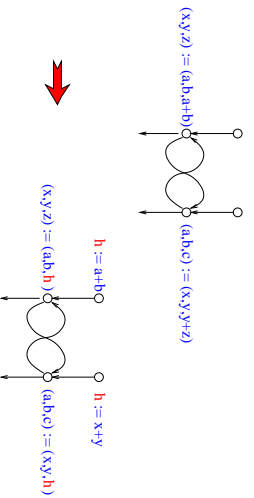
Allgemeiner...

- Code heißt Ausdrücke/Anweisungen
- Motion heißt vorziehen/verzögern

Code / Motion	Hoisting	Sinking
Expressions	EH	/.
Assignments	AH	AS

Semantisches Code Motion...

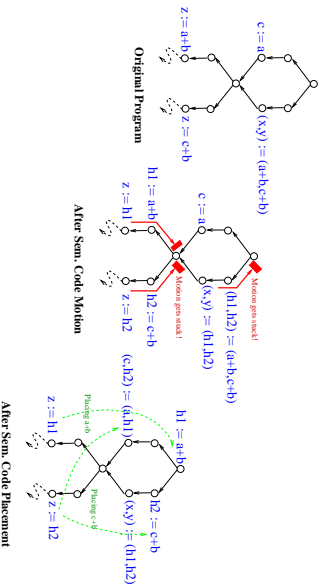
erlaubt mächtigere Optimierungen!



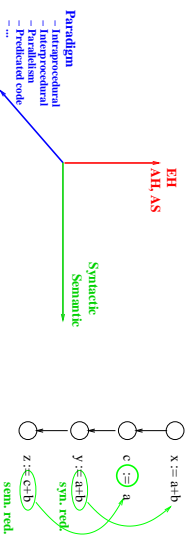
(Beispiel von B. Steffen, TAPSOFT-87)

(I) Code Motion vs. Code Placement

...sind keine Synonyme!



Weitere Verfeinerung des Entwurfsraums von CM-Algorithmen...



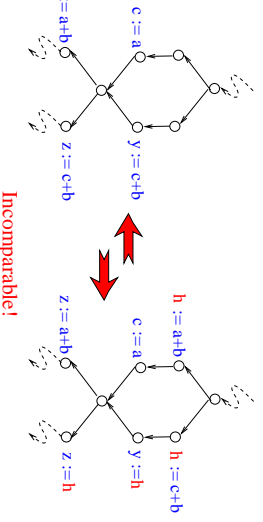
Optimalitätsergebnisse sind sehr empfindlich!

Drei Beispiele sollen dies belegen...

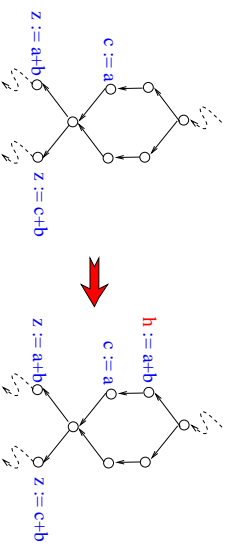
- (I) Code motion vs. code placement
- (II) Abhängigkeiten elementarer Transformationen
- (III) Paradigmenabhängigkeiten

Schlechter noch...

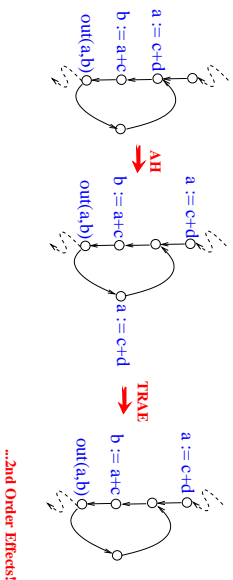
Optimalität ist verloren!



Performanz kann verloren gehen, wenn naiv angewandt!



Abhängigkeiten von Transformationen



~ ... Partially Redundant Assignment Elimination (PRAE)

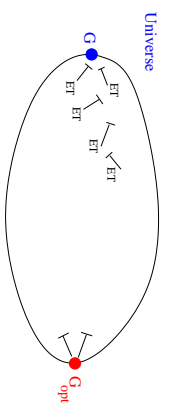
PRAE/PDCE – Optimalitätsergebnisse

Ableitungsrelation \vdash ...

- PRAE... $G \vdash_{AH,TRAE} G'$ (ET={AH, TRAE})
- PDCE... $G \vdash_{AS,TDCE} G'$ (ET={AS, TDCE})

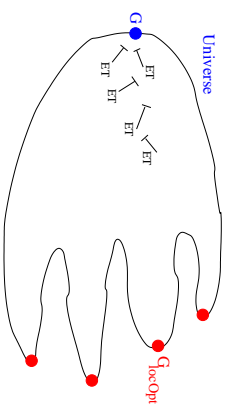
Wir können beweisen ...

Optimalitätstheorem
Für PRAE und PDCE ist \vdash_{ET} konfluent und terminierend

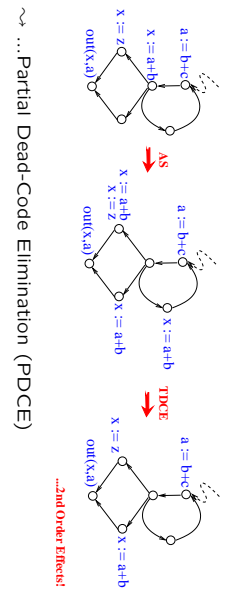


Konfluenz...

...und folglich (globale) Optimalität ist verloren!



(ii) Abhängigkeiten von Transformationen



~ ... Partial Dead-Code Elimination (PDCE)

Konzeptuell

...können wir PREE, PRAE und PDCE wie folgt verstehen:

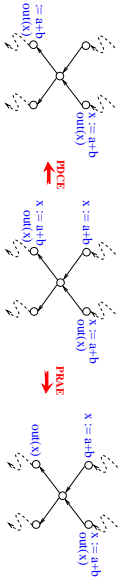
- PREE = AH ; TREE
- PRAE = (AH + TRAE)*
- PDCE = (AS + TDCE)*

Betrachte jetzt...

- Assignment Placement AP
AP = (AH + TRAE + AS + TDCE)*

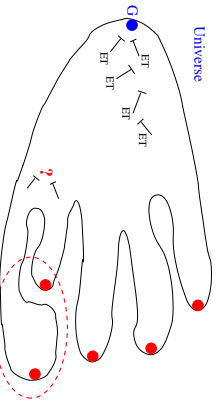
...sollte noch mächtiger sein!

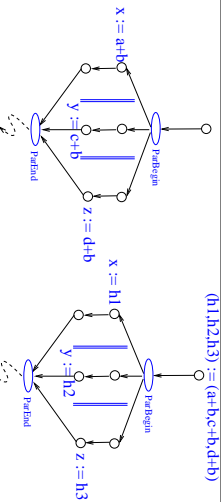
In der Tat, aber...



Noch schlechter...

...es gibt Szenarien, in denen wir mit Universen wie dem folgenden enden können:





... ein naiver Transfer der Transformationsstrategie führt hier zu einem im wesentlichen sequentiellen Programm!

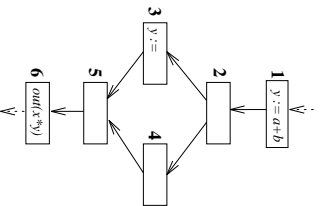
Kap. 7.5 Partial Dead-Code und Faint Code Elimination

Elimination partiell toten/geisterhaften Codes

Wir unterscheiden zwei Spielarten...

- Partial Dead-Code Elimination (PDCE)
- Partial Faint-Code Elimination (PFCE)

Ein Beispiel: Nach PDCE



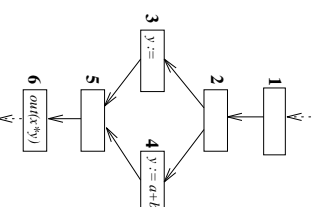
PDCE/PDFE: Anweisungsmuster

Bezeichne in der Folge...

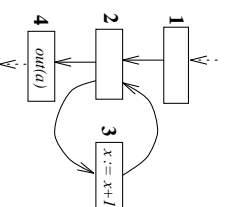
- α ein sog. *Anweisungsmuster*:
- \mathcal{AP} die Menge aller Anweisungsmuster

$$\alpha \equiv x := t$$

Ein Beispiel: Das Ausgangsprogramm



Geisterhaft, aber nicht tot: Ein Beispiel



PDCE/PDFE: Verzögerbarkeit von Anweisungen (1)

Definition [Assignment Sinking]

An *assignment sinking* for α is a program transformation that

- eliminates some occurrences of α ,
- inserts instances of α at the entry or the exit of some basic blocks being reachable from a basic block with an eliminated occurrence of α .

Weisungen (2)

Definition [Local Barriers]

The sinking of an assignment pattern α is *blocked* by an instruction that

- modifies an operand of t or
- uses the variable x or
- modifies the variable x .

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

544

Definition [Admissible Assignment Sinking]

An assignment sinking for α is *admissible*, iff it satisfies:

1. The removed assignments are *substituted*, i.e., on every program path leading from n to e , where an occurrence of α has been eliminated at n , an instance of α has been inserted at a node m on the path such that α is not blocked by any instruction between n and m .
2. The inserted assignments are *justified*, i.e., on each program path from s to n , where an occurrence of α has been inserted at n , an occurrence of α has been eliminated at a node m on the path such that α is not blocked by any instruction between m and n .

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

545

PDCE/PDFE: Elimination von Anweisungen (1)

Definition [Assignment Elimination]

An *assignment elimination* for α is a program transformation that eliminates some original occurrences of α in the argument program.

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

546

PDCE/PDFE: Zulässige Elimination von Anweisungen (2)

Definition [Dead (Faint) Code Elimination]

A *dead (faint) code elimination* for an assignment pattern α is an assignment elimination for α , where some dead (faint) occurrences of α are eliminated.

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

547

PDCE/PDFE: Ein neues Phänomen

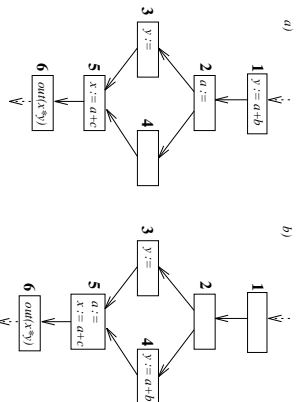
...sog. *second order* Effekte:

- *Sinking-Elimination* Effekte
- *Sinking-Sinking* Effekte
- *Elimination-Sinking* Effekte
- *Elimination-Elimination* Effekte

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

548

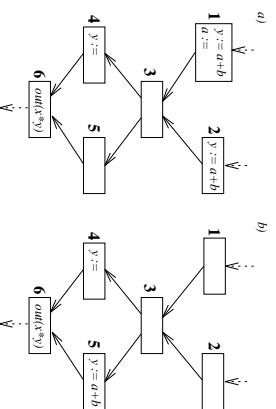
Ein Beispiel für "Sinking-Sinking"-Effekte



Kap. 7.5 Partial Dead-Code und Faint Code Elimination

549

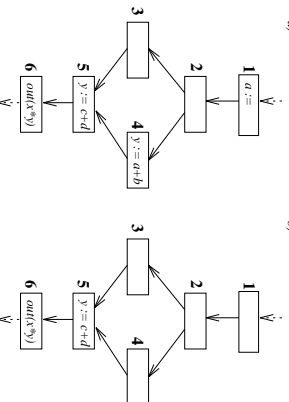
Ein Beispiel für "Elimination-Sinking"-Effekte



Kap. 7.5 Partial Dead-Code und Faint Code Elimination

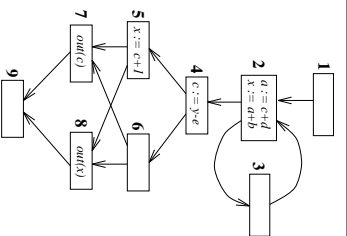
550

Ein Beispiel für "Elimination-Elimination"-Effekte



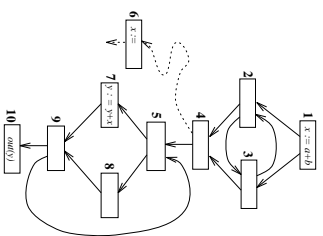
Kap. 7.5 Partial Dead-Code und Faint Code Elimination

551



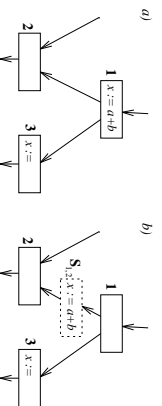
Kap. 7.5 Partial Dead-Code und Faint Code Elimination

Auch für PDCE/PFCE: Spalten kritischer Kanten erforderlich (1)



Kap. 7.5 Partial Dead-Code und Faint Code Elimination

Erinnerung: Kritische Kanten

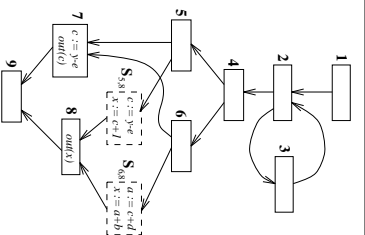


Kap. 7.5 Partial Dead-Code und Faint Code Elimination

Schreibweisen und Vereinbarungen

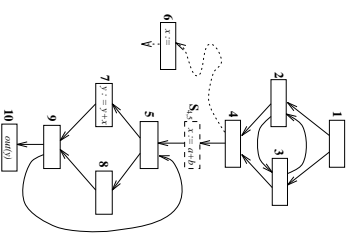
- $G \vdash_{PDCE} G'$ bzw. $G \vdash_{PFCE} G'$:
... G' resultiert aus G durch Anwendung einer zulässigen Verzögerungs- oder (dead/faint) Eliminationstransformation.
- $\tau \in \{PDCE, PFCE\}$:
...Bezeichner für PDCE bzw. PFCE.
- $G_{\tau} =_{df} \{G' \mid G \vdash_{\tau} G'\}$:
...das aus G durch sukzessive Anwendung von PDCE- bzw. PFCE-Transformationen entstehende *Universum*.

Kap. 7.5 Partial Dead-Code und Faint Code Elimination



Kap. 7.5 Partial Dead-Code und Faint Code Elimination

Auch für PDCE/PFCE: Spalten kritischer Kanten erforderlich (2)



Kap. 7.5 Partial Dead-Code und Faint Code Elimination

PDCE/PDFE: Die Definition

Definition [Partial Dead (Faint) Code Elimination]
Partial dead (faint) code elimination PDCE (PFCE) is an arbitrary sequence of admissible assignment sinkings and dead (faint) code eliminations.

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

PDCE/PDFE: Optimalität

Definition [Optimality of PDCE (PFCE)]

1. Let $G', G'' \in G_{\tau}$. Then G' is better than G'' , in signs $G'' \sqsupset G'$, if and only if
 $\forall p \in P[s, e] \forall \alpha \in AP. \alpha\#(pG') \leq \alpha\#(pG'')$
where $\alpha\#(pG')$ and $\alpha\#(pG'')$ denote the number of occurrences of the assignment pattern α on p in G' and G'' , respectively.
2. $G^* \in G_{\tau}$ is optimal if and only if G^* is better than any other program in G_{τ} .

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

PDCE/PDFE: Relation "besser"

Die Relation \preceq ist eine...

- Quasiordnung (d.h. reflexiv, transitiv, aber nicht antisymmetrisch)

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

560

Sei

- $\underline{G}_\tau = g_\tau (\bar{\Sigma} \cap \Gamma_\tau)^*$, und sei
- $\mathcal{F}_\tau \subseteq \{f \mid f : g_\tau \rightarrow g_{\tau'}\}$ eine endliche Familie von Funktionen mit
 1. *Monotonicity*:
 $\forall G', G'' \in g_\tau \ \forall f \in \mathcal{F}_\tau$
 $G' \underline{G}_\tau G'' \Rightarrow f(G') \underline{G}_\tau f(G'')$
 2. *Dominance*:
 $\forall G', G'' \in g_\tau, G' \Gamma_\tau G'' \Rightarrow \exists f \in \mathcal{F}_\tau, G'' \underline{G}_\tau f(G')$

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

561

PDCE/PDFE: Existenz optimaler Programme

Theorem [Existence of Optimal Programs]

g_τ has an optimal element (wrt \preceq) which can be computed by any sequence of function applications that contains all elements of \mathcal{F}_τ 'sufficiently' often.

Proof ...by means of a generalized version of the fixed point theorem of Knaster/Tarski (cf. Gese, Knoop, Lüttgen et al. (CC'96))

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

562

PDCE/PDFE: Existenz optimaler Programme

Bemerkungen:

- PDCE und PFCE erfüllen die Anforderungen des vorstehenden Optimalitätstheorems
- Das optimale Programm ist eindeutig bestimmt bis auf irrelevante Umsortierungen von Anweisungen in Basisblöcken.

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

563

Die PDCE/PDFE-Analysen (1)

Lokale Prädikate für *Dead Code Elimination* (DCE) und *Faint Code Elimination* (FCE)...

- $\text{USED}_l(x)$: x is a right-hand side variable of the instruction l ,*
- $\text{Rel-Used}_l(x)$: x is a right-hand side variable of the relevant instruction l .
- $\text{Ass-Used}_l(x)$: x is a right-hand side variable of the assignment statement l .
- $\text{MOD}_l(x)$: x is the left-hand side variable of the instruction l .

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

564

Die PDCE/PDFE-Analysen (2)

Die DCE-Analyse...

The Dead Variable Analysis:

$$\text{N-DEAD}_l = \neg \text{USED}_l * (\text{X-DEAD}_l + \text{MOD}_l)$$

$$\text{X-DEAD}_l = \prod_{l \in \text{succ}(l)} \text{N-DEAD}_l$$

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

565

Die PDCE/PDFE-Analysen (3)

Die FCE-Analyse...

The Faint Variable Analysis: (Slotwise simultaneously for all variables x)

$$\text{N-FAINT}_l(x) = \neg \text{Rel-Used}_l(x) *$$

$$(\text{X-FAINT}_l(x) + \text{MOD}_l(x)) *$$

$$(\text{X-FAINT}_l(\text{LhsVar}_l) + \neg \text{Ass-Used}_l(x))$$

$$\text{X-FAINT}_l(x) = \prod_{l \in \text{succ}(l)} \text{N-FAINT}_l(x)$$

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

566

Die PDCE/PDFE-Analysen (4)

Lokale Prädikate für *Assignment Sinking* (AS)...

- $\text{LOC-DELAYED}_n(\alpha)$: There is a sinking candidate of α in n .
- $\text{LOC-BLOCKED}_n(\alpha)$: The sinking of α is blocked by some instruction of n .

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

567

Die AS-Analyse, im Kern eine Verzögerbarkeitsanalyse...

Deliability Analysis:

$$\begin{aligned} \text{N-DELAYED}_n &= \begin{cases} \text{ff} & \text{if } n = s \\ \prod_{m \in \text{pred}(n)} \text{X-DELAYED}_m & \text{otherwise} \end{cases} \\ \text{X-DELAYED}_n &= \text{LOC-DELAYED}_n + \text{N-DELAYED}_n * \text{-LOC-BLOCKED}_n \end{aligned}$$

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

568

Die PDCE/PDFE-Analysen (6)

Die aus der AS-Analyse resultierenden Einsetzungspunkte...

Insertion Points:

$$\begin{aligned} \text{N-INSERT}_n &=_{df} \text{N-DELAYED}_n * \text{LOC-BLOCKED}_n \\ \text{X-INSERT}_n &=_{df} \text{X-DELAYED}_n * \sum_{m \in \text{succ}(n)} \text{-N-DELAYED}_m \end{aligned}$$

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

569

PDCE/PDFE: Hauptresultate (1)

Bezeichnen

- $pdce$ und
- $pfce$

die aus vorigen Analysen abgeleiteten Algorithmen für die

- *Elimination partiell toter Anweisungen* und
- *Elimination partiell geisterhafter Anweisungen*

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

570

PDCE/PDFE: Hauptresultate (2)

Bezeichnen

- G_{pdce} und
- G_{pfce}

die aus G durch $pdce$ und $pfce$ resultierenden Programme, sowie

- G_{PDCE} und
- G_{PDFE}

die von den Elementarttransformation von $pdce$ und $pfce$ aufgespannten Universen für G .

Dann gilt...

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

571

PDCE/PDFE: Hauptresultate (3)

Theorem [Correctness]

1. $G_{pdce} \in G_{PDCE}$
2. $G_{pfce} \in G_{PDFE}$

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

572

PDCE/PDFE: Hauptresultate (4)

Theorem [Optimality Theorem]

1. G_{pdce} is optimal in G_{PDCE}
2. G_{pfce} is optimal in G_{PDFE}

Kap. 7.5 Partial Dead-Code und Faint Code Elimination

573

Elimination partiell redundanter Anweisungen

Auch hier neue Phänomene...

Wir können unterscheiden:

- (Pure) Expression Motion (PUEM) (bzw. PUPREE)
- (Pure) Assignment Motion (PUAM) (bzw. PUPRAE)
- ...sowie
- Uniform Expression and Assignment Motion (AM)

Dazu einige Beispiele...

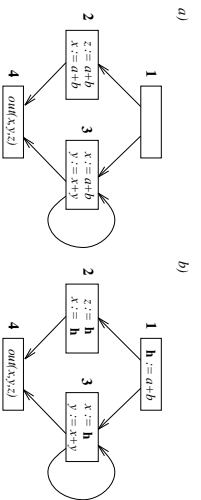
Kap. 7.7 Assignment Motion

574

Kap. 7.7 Assignment Motion

575

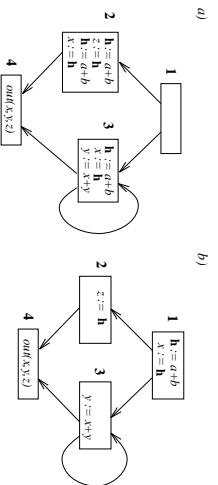
Expression Motion (PuEM)



Kap. 7.7 Assignment Motion

576

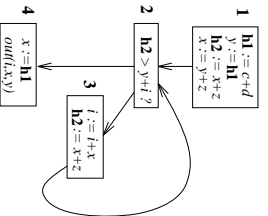
Uniform Expression and Assignment Motion (AM)



Kap. 7.7 Assignment Motion

578

Ein komplexeres Beispiel: Optimiertes Programm



Kap. 7.7 Assignment Motion

580

AM: Der einheitliche PuEM/PuAM-Algorithmus

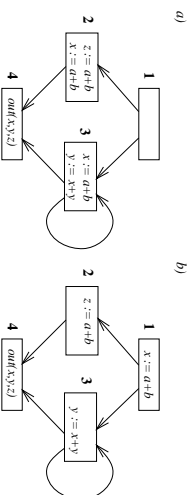
Ein dreistufiger Algorithmus...

- *Präprozess*
Esetze jedes Vorkommen einer Anweisung $x := t$ durch die Sequenz $h_x := t; x := h_x$.
- *Hauptprozess*
Wiederholte Anwendung von
 - Assignment Hoisting (AH) und
 - Totally Redundant Assignment Elimination (TRAЕ) bis Stabilität eintritt.
- *Postprozess*
Aufräumen isolierter Initialisierungen

Kap. 7.7 Assignment Motion

582

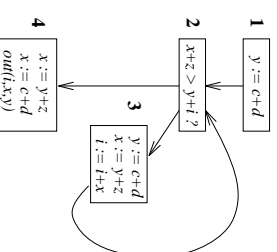
Assignment Motion (PuAM)



Kap. 7.7 Assignment Motion

577

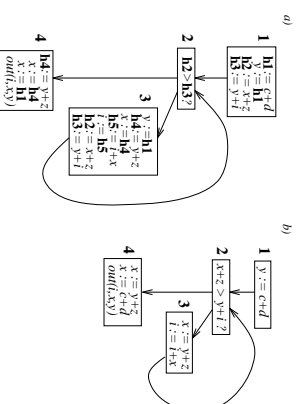
Ein komplexeres Beispiel: Ausgangsprogramm



Kap. 7.7 Assignment Motion

579

Zum Vergleich: Separate Effekte von PuEM & PuAM



Kap. 7.7 Assignment Motion

581

Bemerkung

...dank des Präprozesses wird PuEM durch PuAM abgedeckt und einheitlich erfasst!

Kap. 7.7 Assignment Motion

583

AM: Ein schon bekanntes Phänomen

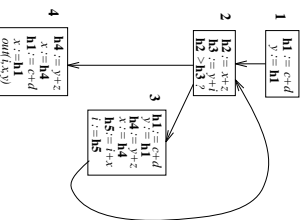
...sog. *second order* Effekte:

- *Hoisting-Hoisting* Effekte
- *Hoisting-Elimination* Effekte
- *Eliminator-Hoisting* Effekte
- *Elimination-Elimination* Effekte

Kap. 7.7 Assignment Motion

584

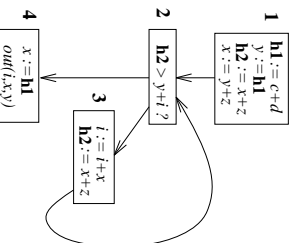
Nach dem Präprozess



Kap. 7.7 Assignment Motion

586

Das Endresultat: Nach dem Postprozess



Kap. 7.7 Assignment Motion

588

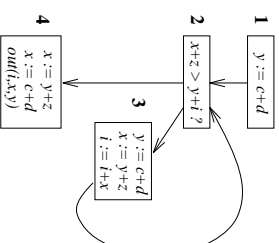
Hauptergebnisse (2)

Theorem [Expression-Optimality]

$G_{GlobAlg}$ is expression-optimal in G , i.e., it requires at most as many expression evaluations at run-time than any other program that can be obtained via EM and AM transformations.

Kap. 7.7 Assignment Motion

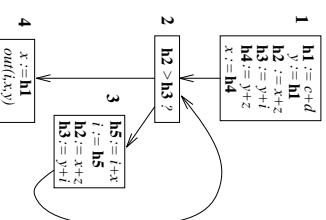
590



Kap. 7.7 Assignment Motion

585

Nach dem Hauptprozess



Kap. 7.7 Assignment Motion

587

Hauptergebnisse (1)

Analog zu PDCE/PFCE...

Theorem [Correctness] $G_{GlobAlg} \in G$

Kap. 7.7 Assignment Motion

589

Hauptergebnisse (3)

Theorem [Relative Assignment-Optimality]

$G_{GlobAlg}$ is relatively assignment-optimal in G , i.e., it is impossible to decrease the number of assignments required by $G_{GlobAlg}$ at run-time by means of EM and AM transformations.

Kap. 7.7 Assignment Motion

591

Hauptergebnisse (4)

Theorem [Relative Temporary-Optimality]

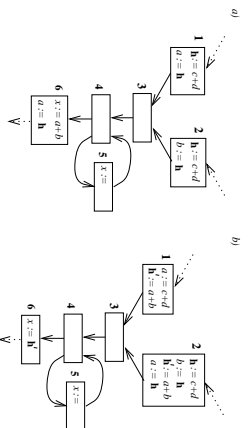
G_{Global} is relatively temporary-optimal in G , i.e. it is impossible to decrease the number of assignments to temporaries or the length of temporary lifetimes in G_{Global} by a corresponding assignment sinking.

Kap. 7.7 Assignment Motion

592

Warum nur "relative Optimalität" (2)

...und folgende zwei unvergleichbare Transformationsresultate:

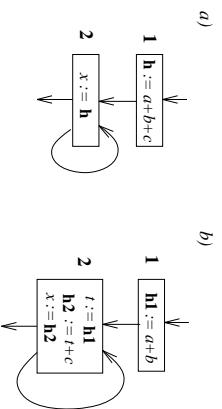


Kap. 7.7 Assignment Motion

594

Anderer Phänomene: Komplex- vs. 3-Adresscode (2)

Der Effekt von PUEM...



Kap. 7.7 Assignment Motion

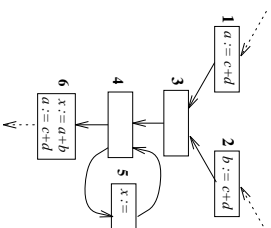
596

Kap. 7.8 Fazit und Literaturhinweise

Kap. 7.8 Fazit und Literaturhinweise

598

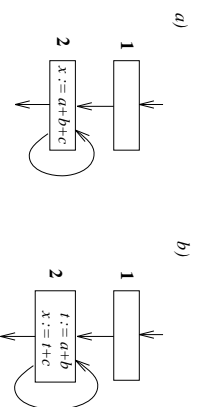
Betrachte dazu...



Kap. 7.7 Assignment Motion

593

Anderer Phänomene: Komplex- vs. 3-Adresscode (1)

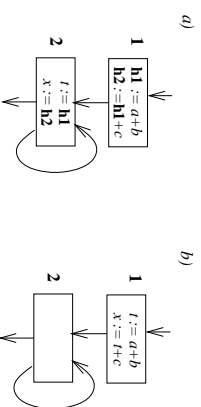


Kap. 7.7 Assignment Motion

595

Anderer Phänomene: Komplex- vs. 3-Adresscode (3)

Die Effekte von (PUEM + Copy Propagation) und von (Uniform PUEM&PUAM)...



Kap. 7.7 Assignment Motion

597

Ein Fazit bzw...

Die Frage nach dem Sinn des Lebens, was haben wir alles erreicht bzw...

- Was haben wir alles betrachtet?
- Das wenigste!

Oder umgekehrt...

- Was haben wir alles nicht betrachtet?
- Das meiste!

Kap. 7.8 Fazit und Literaturhinweise

599

- *Erweiterungen syntaktischer PRE neben PDCE/PRAE*
 - Lazy Strength Reduction
 - ...
- *Semantische Erweiterungen*
 - Semantic Code Motion/Code Placement
 - Semantic Strength Reduction
 - ...
- *Sprachausweitungen*
 - Interprozeduralität
 - Parallelität
 - ...

Kap. 7.8 Fazit und Literaturhinweise

600

Insbesondere nicht (oder nicht im Detail) (1)

- *Dynamische, profigestützte Erweiterungen*
 - Spekulative PRE
 - ...
- ...

Kap. 7.8 Fazit und Literaturhinweise

601

Literaturhinweise (1)

- *Syntaktische PRE*
 - Knoop, J., Rütting, O., and Steffen, B. Retrospective: Lazy Code Motion. In "20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979 - 1999): A Selection", ACM SIGPLAN Notices 39, 4 (2004), 460 - 461 & 462-472.
 - Knoop, J., Rütting, O., and Steffen, B. Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Systems 16, 4 (1994), 1117 - 1155.
 - Rütting, O., Knoop, J., and Steffen, B. Sparse code motion. In Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000) (Boston, MA, Jan. 19 - 21, 2000), ACM New York, (2000), 170 - 183.

Kap. 7.8 Fazit und Literaturhinweise

602

Literaturhinweise (2)

- *Elimination partiell toten Codes*
 - Knoop, J., Rütting, O., and Steffen, B. Partial dead code elimination. In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94) (Orlando, FL, USA, June 20 - 24, 1994), ACM SIGPLAN Notices 29, 6 (1994), 147 - 158.
- *Elimination partiell redundanter Anweisungen*
 - Knoop, J., Rütting, O., and Steffen, B. The power of assignment motion. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95) (La Jolla, CA, USA, June 18 - 21, 1995), ACM SIGPLAN Notices 30, 6 (1995), 233 - 245.

Kap. 7.8 Fazit und Literaturhinweise

603

Literaturhinweise (3)

- *BB- vs. EA-Graphen*
 - Knoop, J., Koschitzki, D., and Steffen, B. Basic-block graphs: Living dinosaur? In Proceedings of the 7th International Conference on Compiler Construction (CC'98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1383 (1998), 65 - 79.
- *Schieben vs. Platzieren*
 - Knoop, J., Rütting, O., and Steffen, B. Code motion and code placement: Just synonyms? In Proceedings of the 7th European Symposium On Programming (ESOP 98) (Lisbon, Portugal, March 30 - April 3, 1998), Springer-Verlag, Heidelberg, LNCS 1381 (1998), 154 - 169.

Kap. 7.8 Fazit und Literaturhinweise

604

Literaturhinweise (4)

- *Spekulative vs. klassische PRE*
 - Scholz, B., Horspool, N., and Knoop, J. Optimizing for space and time usage with speculative partial redundancy elimination. In Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2004) (Washington, DC, June 11 - 13, 2004), ACM SIGPLAN Notices 39, 7 (2004), 221 - 230.
 - Xue, J., Knoop, J. A fresh look at PRE as a maximum flow problem. In Proceedings of the 15th International Conference on Compiler Construction (CC 2006) (Vienna, Austria, March 25 - April 2, 2006), Springer-Verlag, Heidelberg, LNCS 3923 (2006), 139 - 154.

Kap. 7.8 Fazit und Literaturhinweise

605

Literaturhinweise (5)

- *Weitere Techniken und spezielle Verfahren*
 - Geiser, A., Knoop, J., Lütgen, G., Rütting, O., and Steffen, B. Non-monotone fixpoint iterations to resolve second order effects. In Proceedings of the 6th International Conference on Compiler Construction (CC'96) (Linköping, Sweden, April 24 - 26, 1996), Springer-Verlag, Heidelberg, LNCS 1060 (1996), 106 - 120.
 - Knoop, J., and Mehofer, E. Optimal distribution assignment placement. In Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97) (Passau, Germany, August 26 - 29, 1997), Springer-V., Heidelberg, LNCS 1300 (1997), 364 - 373.
 - Knoop, J., Rütting, O., and Steffen, B. Lazy strength reduction. Journal of Programming Languages 1, 1 (1993), 71 - 91.
 - ...: siehe auch www.comp.lang.tuwien.ac.at/knoop

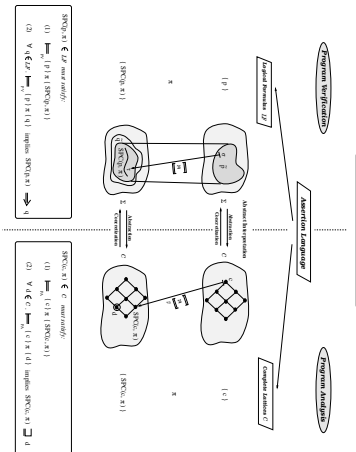
Kap. 7.8 Fazit und Literaturhinweise

606

Kapitel 8 Programmverifikation und -analyse im Vergleich

Kap. 8 Programmverifikation und -analyse im Vergleich

607



Kap. 8 Programmverifikation und -analyse im Vergleich 608

Kapitel 9 Reverse Datenflussanalyse

Kap. 9 Reverse Datenflussanalyse 610

Reverse abstrakte Semantik

Reverse abstrakte Semantik

1. *Datenflussanalyseverband* $\mathcal{C} = (C, \cap, \cup, \subseteq, \perp, \top)$
2. *Reverses Datenflussanalysefunktional*
 $\llbracket \cdot \rrbracket_R : E \rightarrow (C \rightarrow C)$ definiert durch

$$\forall e \in E \forall c \in C: \llbracket e \rrbracket_R(c) = \bigcap \{ c' \mid \llbracket e \rrbracket(c') \supseteq c \}$$

wobei $\llbracket \cdot \rrbracket : E \rightarrow (C \rightarrow C)$ eine abstrakte Semantik auf C ist.

Kap. 9.1 Grundlagen 612

Zusammenhang von $\llbracket \cdot \rrbracket$ und $\llbracket \cdot \rrbracket_R$ (2)

Lemma

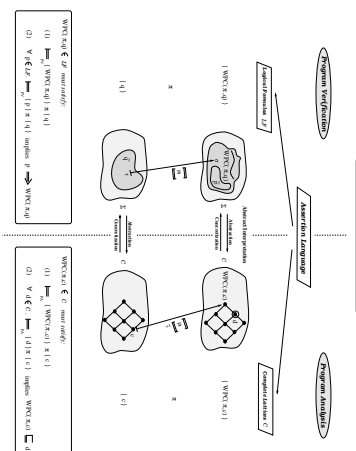
Sei $\llbracket \cdot \rrbracket$ ein Datenflussanalysefunktional. Dann gilt für jede Kannte $e \in E$:

1. $\llbracket e \rrbracket_R \circ \llbracket e \rrbracket \subseteq \text{Id}_C$, falls $\llbracket e \rrbracket$ monoton ist.
2. $\llbracket e \rrbracket \circ \llbracket e \rrbracket_R \supseteq \text{Id}_C$, falls $\llbracket e \rrbracket$ distributiv ist.

Sprechweise in der Theorie "Abstrakter Interpretation":

- $\llbracket e \rrbracket$ und $\llbracket e \rrbracket_R$ bilden eine Galois-Verbindung.

Kap. 9.1 Grundlagen 614



Kap. 8 Programmverifikation und -analyse im Vergleich 609

Kapitel 9.1 Grundlagen

Kap. 9.1 Grundlagen 611

Zusammenhang von $\llbracket \cdot \rrbracket$ und $\llbracket \cdot \rrbracket_R$ (1)

Lemma

Sei $\llbracket \cdot \rrbracket$ ein Datenflussanalysefunktional. Dann gilt für jede Kannte $e \in E$:

1. $\llbracket e \rrbracket_R$ ist wohldefiniert und monoton.
2. $\llbracket e \rrbracket_R$ ist additiv, falls $\llbracket e \rrbracket$ distributiv ist.

Kap. 9.1 Grundlagen 613

Zusammenhang von $\llbracket \cdot \rrbracket$ und $\llbracket \cdot \rrbracket_R$ (3)

Hilfssatz

1. $\forall n \in N' \cap N. P_C[s, n] = P_C[s, n]$
2. $\forall q \in N' \setminus \{s\}. P_C[s, q] = P_C[s, q]$
3. $\forall \alpha_S \in C \forall n \in N' \cap N. MOP_{(C', \alpha_S)}(n) = MOP_{(C, \alpha_S)}(n)$
4. $MOP_{(C, \alpha_S)}(q) = MOP_{(C, \alpha_S)}(q)$

Kap. 9.1 Grundlagen 615

Kapitel 9.1.1 R -JOP-Ansatz

Kap. 9.1.1 R -JOP-Ansatz

616

Der R -JOP-Ansatz

Die R -JOP-Lösung:

$$\forall c_q \in C \ \forall n \in N. R\text{-JOP}_{c_q}(n) =_{df} \sqcup \{ \llbracket p \rrbracket_R(c_q) \mid p \in P[n, q] \}$$

Kap. 9.1.1 R -JOP-Ansatz

617

Kapitel 9.1.2 R -MinFP-Ansatz

Kap. 9.1.2 R -MinFP-Ansatz

618

Der R -MinFP-Ansatz

Das R -MinFP-Gleichungssystem:

$$\mathit{rednf}(n) = \begin{cases} c_q & \text{falls } n = q \\ \sqcup \{ \llbracket (n, m) \rrbracket_R(\mathit{rednf}(m)) \mid m \in \mathit{succ}(n) \} & \text{sonst} \end{cases}$$

Bezeichne $\mathit{rednf}_{c_q}^*$ die kleinste Lösung dieses Gleichungssystems bzgl. $c_q \in C$.

Die R -MinFP-Lösung:

$$\forall c_q \in C \ \forall n \in N. R\text{-MinFP}_{c_q}(n) =_{df} \mathit{rednf}_{c_q}^*(n)$$

Kap. 9.1.2 R -MinFP-Ansatz

619

Der generische R -MinFP-Alg. (1)

Input: (1) A flow graph $G = (N, E, s, e)$, (2) a program point q , (3) a reverse abstract semantics (i.e., a data-flow lattice C , and a reverse data-flow functional $\llbracket _ \rrbracket_R : E \rightarrow (C \rightarrow C)$ induced by a functional $\llbracket _ \rrbracket : E \rightarrow (C \rightarrow C)$), and (4) a component information $c_q \in C$.

Output: Under the assumption of termination (cf. Theorem 77), the R -MinFP-solution. Depending on the properties of the underlying reverse data-flow functional, this has the following interpretation.

(1) $\llbracket _ \rrbracket_R$ is additive: Variable $\mathit{rednf}[s]$ stores the weakest context information on of c_q , i.e., the least data-flow fact which must be ensured at the program entry in order to guarantee c_q at q . If this is \perp , the requested component information cannot be satisfied at all.

(2) $\llbracket _ \rrbracket_R$ is monotonic: Variable $\mathit{rednf}[s]$ stores a lower bound of the weakest context candidate of c_q . Generally, this is not a sufficient context information itself. Hence, except for the special case $\mathit{rednf}[s] = \perp$, which implies that c_q cannot be satisfied by any consistent context information, nothing can be concluded from the value of $\mathit{rednf}[s]$.

Remark: The variable $\mathit{workset}$ controls the iterative process. Its elements are nodes of G , whose informations annotating them have recently been updated.

Kap. 9.1.2 R -MinFP-Ansatz

620

Der generische R -MinFP-Alg. (2)

(Prologue: Initialization of the annotation array rednf , and the variable $\mathit{workset}$)

FORALL $n \in N \setminus \{q\}$ DO $\mathit{rednf}[n] := \perp$ OD;

$\mathit{rednf}[q] := c_q$;

$\mathit{workset} := \{q\}$;

Kap. 9.1.2 R -MinFP-Ansatz

621

Der generische R -MinFP-Alg. (3)

(Main process: Iterative fixed point computation)

WHILE $\mathit{workset} \neq \emptyset$ DO

 CHOOSE $m \in \mathit{workset}$;

$\mathit{workset} := \mathit{workset} \setminus \{m\}$;

 (Update the predecessor-environment of node m)

 FORALL $n \in \mathit{pred}(m)$ DO

$\mathit{join} := \llbracket (n, m) \rrbracket_R(\mathit{rednf}[m]) \sqcup \mathit{rednf}[n]$;

 IF $\mathit{rednf}[n] \sqsubset \mathit{join}$

 THEN

$\mathit{rednf}[n] := \mathit{join}$;

$\mathit{workset} := \mathit{workset} \cup \{n\}$

 FI

 OD

 ESOOHC

OD.

Kap. 9.1.2 R -MinFP-Ansatz

622

Kapitel 9.1.3 Reverses Koinzidenz- und Sicherheitstheorem

Kap. 9.1.3 Reverses Koinzidenz- und Sicherheitstheorem

623

Reversees Sicherheitstheorem

Reversees Sicherheitstheorem

Die $R\text{-MinFP}$ -Lösung ist eine obere (d.h. sichere) Approximation der $R\text{-JOP}$ -Lösung, d.h.,

$$\forall c_q \in C \forall n \in N: R\text{-MinFP}_{c_q}(n) \supseteq R\text{-JOP}_{c_q}(n)$$

Kap. 9.1.3 Reversees Koinzidenz- und Sicherheitstheorem

624

Reversees Koinzidenztheorem

Reversees Koinzidenztheorem

Die $R\text{-MinFP}$ -Lösung stimmt mit der $R\text{-JOP}$ -Lösung überein, d.h.,

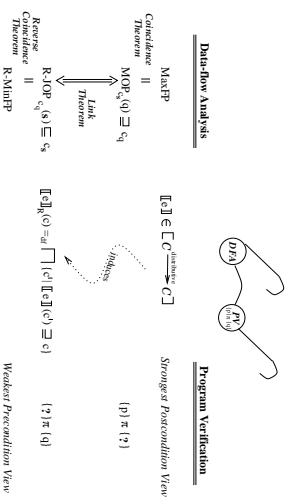
$$\forall c_q \in C \forall n \in N: R\text{-MinFP}_{c_q}(n) = R\text{-JOP}_{c_q}(n)$$

falls $\llbracket \cdot \rrbracket$ distributiv ist.

Kap. 9.1.3 Reversees Koinzidenz- und Sicherheitstheorem

625

DFA vs. Verifikation: Überblick



Kap. 9.2 DFA, RDFA und DD-DFA

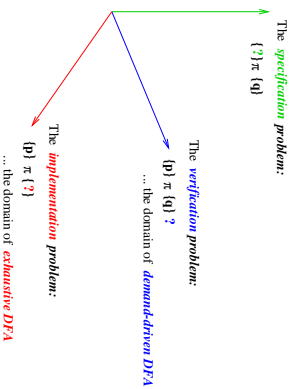
627

Kapitel 9.2 DFA, RDFA und DD-DFA

Kap. 9.2 DFA, RDFA und DD-DFA

626

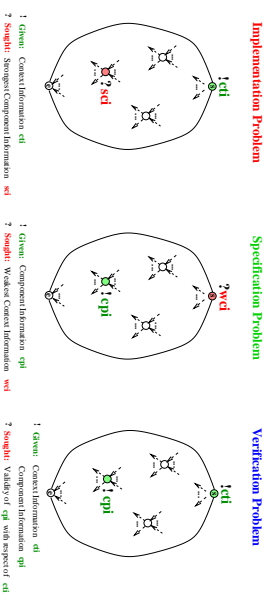
Drei unterschiedliche Problemperspektiven (1)



Kap. 9.2 DFA, RDFA und DD-DFA

628

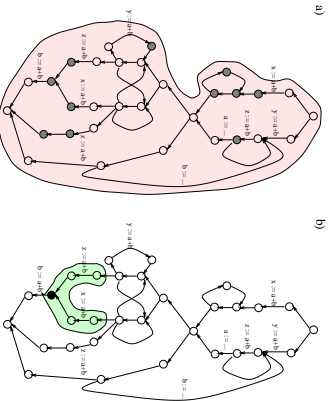
Drei unterschiedliche Problemperspektiven (2)



Kap. 9.2 DFA, RDFA und DD-DFA

629

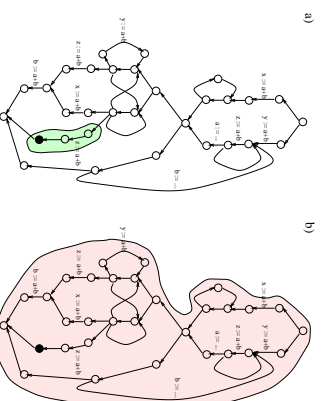
Bsp: Verfügbarkeit an einem Punkt (1)



Kap. 9.2 DFA, RDFA und DD-DFA

630

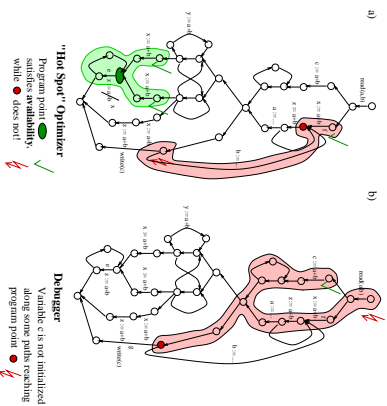
Bsp: Verfügbarkeit an einem Punkt (2)



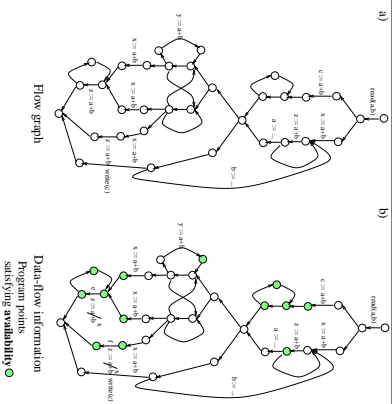
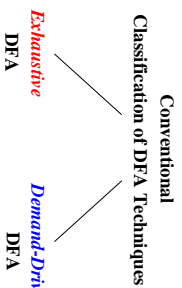
Kap. 9.2 DFA, RDFA und DD-DFA

631

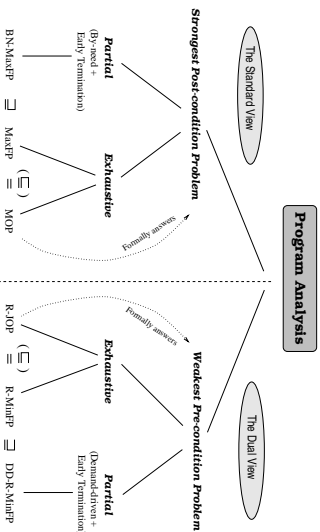
Anwendung: "Hot Spot" Optimierer und Debugger (2)



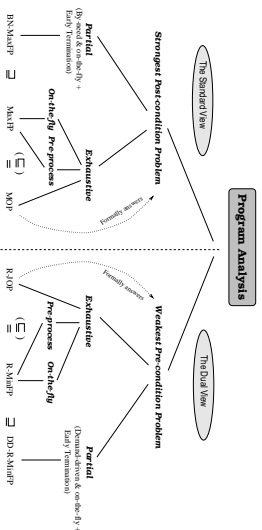
Erschöpfende vs. anforderungsgetriebene DFA (1)



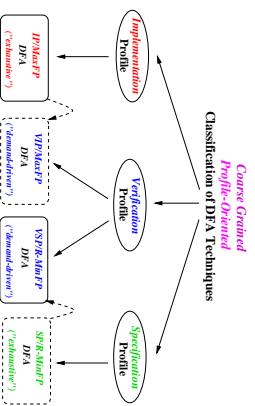
Erschöpfende vs. anforderungsgetriebene DFA (2)



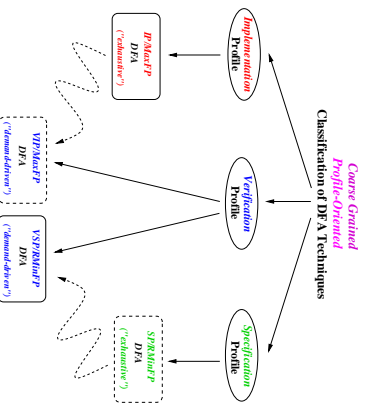
Erschöpfende vs. anforderungsgetriebene DFA (3)

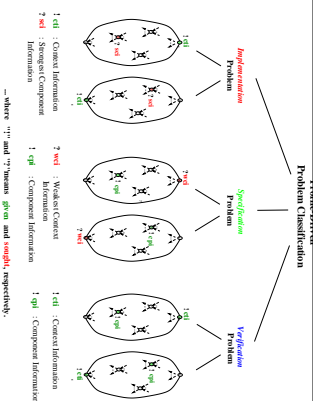


Eine andere Sicht (1)

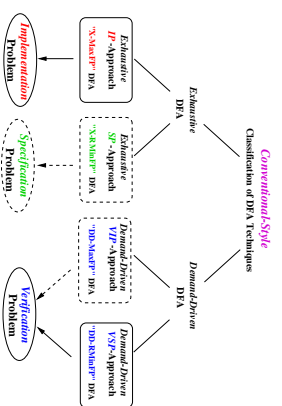
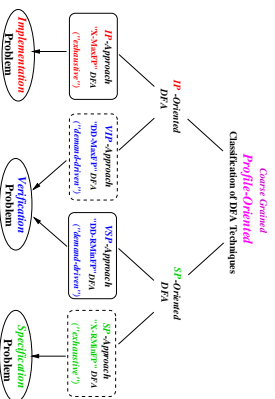


Eine andere Sicht (2)





Zum Abschluss: Problemorientiert (2)



Hot-Spot Program Optimization

...in größerem Detail: Siehe Zusatzfolien!