Formale Methoden der Informatik

Prof. Dr. Karsten Wolf

Teil I
Semantik von Programmiersprachen

Prof. Dr. Karsten Wolf

Schwerpunkt: *Programmier*sprachen

- Syntax = Struktur von Sätzen der Sprache
  (Komplexität und Formale Sprachen: Grammatiken, Automaten)

- **Semantik = Bedeutung von Sätzen der Sprache**

- Pragmatik = Verwendung der Sprachen
  (in den Anwendungsgebieten)

Wozu Semantik?

- Programme abarbeiten
- Programme erstellen
- Programme ändern/austauschen
- Programme übersetzen
- Programme mit Spezifikation vergleichen

Wozu formale Semantik?

- Programme automatisch abarbeiten
  - Interpreter
- Programme automatisch erstellen
  - Konstruktion
- Programme automatisch ändern/austauschen
  - Transformation, Substitution, Optimierung
- Programme automatisch übersetzen
  - Compiler
- Programme automatisch mit Spezifikation vergleichen
  - Verifikation

Außerdem

- Informale Semantiken (Standards, Reports, ...) sind
  - unvollständig
  - missverständlich
  - fehlerhaft

- Folgen:
  - Portabilität als Problem
  - Schwierigkeiten beim Ablösen veralteter Programmiersprachen
  - Unmöglichkeit seriöser Verifikation
  - ...

- Formalisierung hat schon oft solche Probleme aufgedeckt

Nur mit formaler Semantik möglich:

- Beweisbar korrekte Interpreter/Compiler
- Beweisbar korrekte Programmoptimierung

- Konstruktion, Verifikation

Und schließlich...

Informatik = Syntax + Semantik + Pragmatik

Wir manipulieren Symbole, denken uns etwas dabei und verfolgen einen Zweck...

Fahrplan

- Kapitel 1: Eine einfache Programmiersprache:

ermöglicht Vergleich verschiedener Semantikarten

Fahrplan

- 2. Kapitel: Operationelle Semantik

Wie (in welchen Schritten) entsteht der Effekt eines Programms?

- Maschinensemantik
- Natürliche Semantik (Big-Step-Semantik)
- Strukturelle operationelle Semantik (Small-Step-Semantik)

Einsatz z.B. : Beweisbar korrekte Übersetzung

Fahrplan

- 3. Kapitel: Denotationelle Semantik

Was ist der Effekt eines Programms?

Funktion, die Eingaben eine Ausgabe zuordnet

Einsatz z.B. : Programmoptimierung

Abstract Interpretation = Informationsbeschaffung zur Compilezeit

Fahrplan

- 4. Kapitel: Axiomatische Semantik

Welche Aussagen kann ich über den Effekt eines Programms beweisen?

Logischer Kalkül

Einsatz z.B. : Programmverifikation

Kapitel 1

Eine einfache Programmiersprache

W

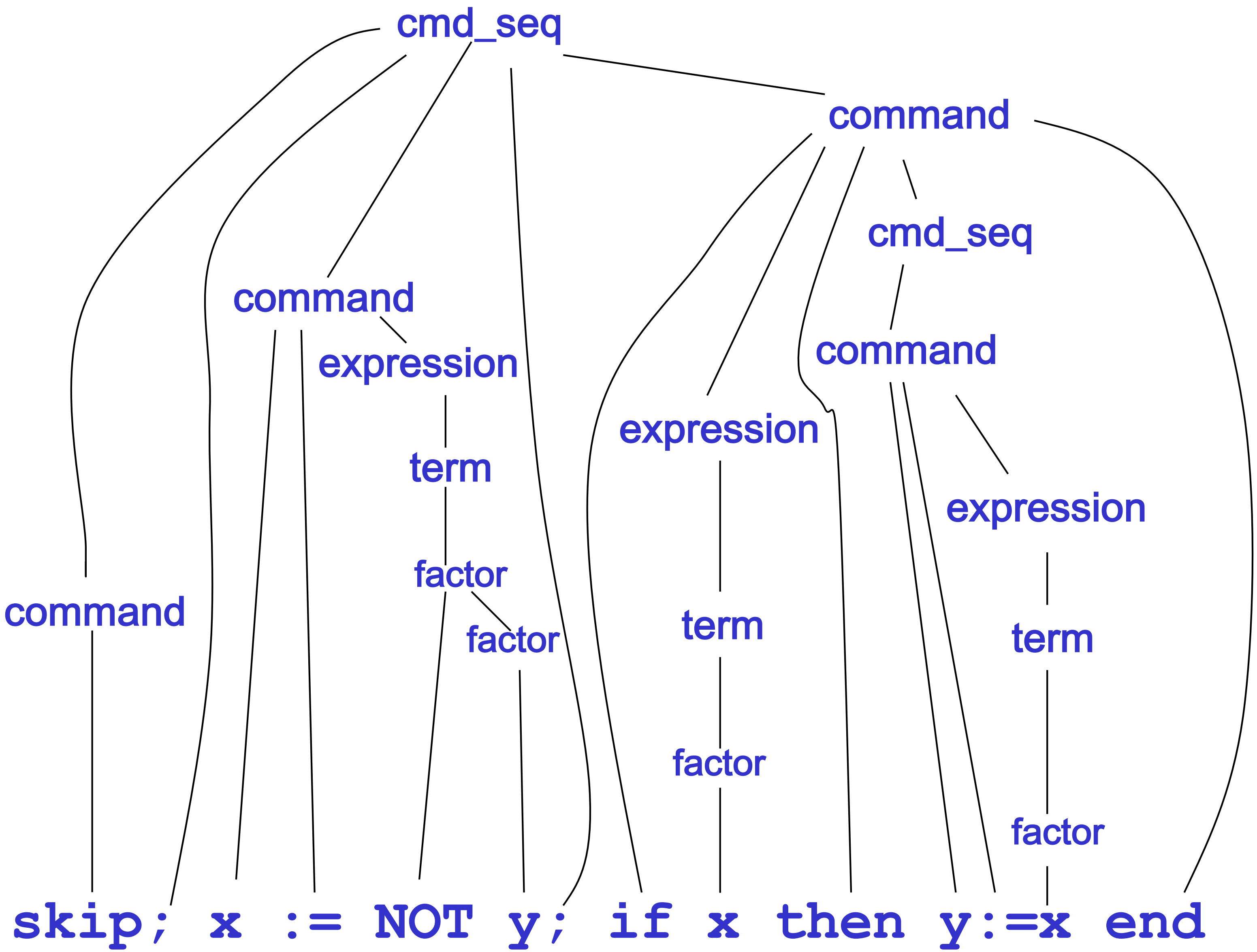
```
cmd_seq        = command | cmd_seq „;“ command.
command        = „identifier“ „:=“ expression | „skip“
               | „while“ expression „do“ cmd_seq „end“
               | „if“ expression „then“ cmd_seq  „end“
               | „if“ expression „then“ cmd_seq „else“
                 cmd_seq „end“.
expression   = term | expression („+“ | „-“ | „OR“) term.
term         = factor | term („*“ | „/“ |  „AND“) factor.
factor      = „NOT“ factor | „number“ | „identifier“
              | „true“ | „false“ | „(„ expression „)“
              | expression („<“|„>“|„<=“|„>=“|„<>“|„=“)
                expression
```

Abstrakte Syntax; Syntaxbaum

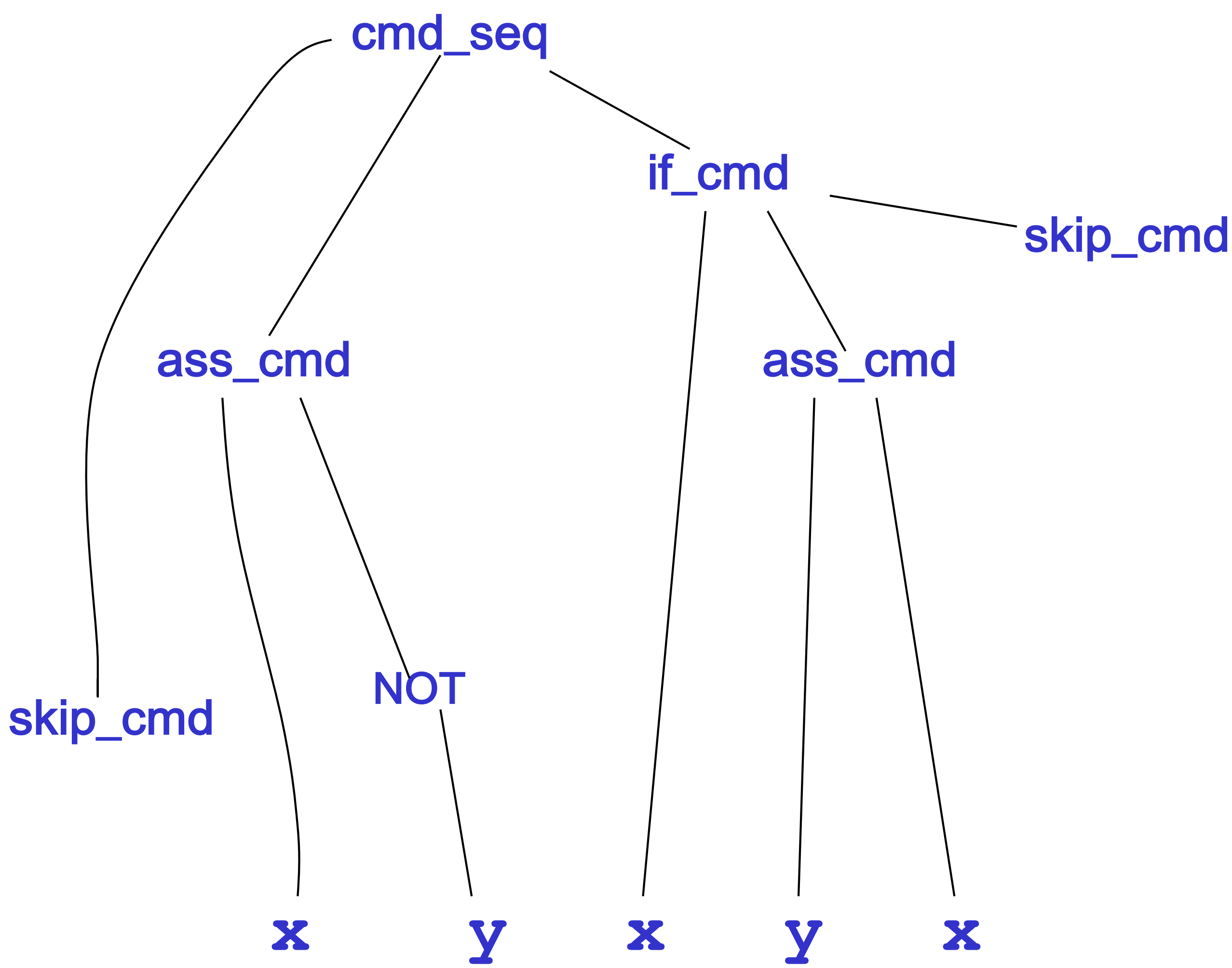
= Ableitungsbaum mit einigen Vereinfachungen, z.B.

- Verzicht auf struktursichernde Elemente: (,),;, end...
- Vereinheitlichung, z.B.
  `if B then S end` → `if B then S else skip end`
- Operanden in den Wurzelknoten von Expressions
- Listenstrukturen (z.B. cmd_seq) einebnen
- Umbenennungen (z.B. Expression→Term→Faktor) überspringen
- ...

Beispiel: Ableitungsbaum (schon etwas vereinfacht)



Beispiel: Syntaxbaum



Ausdruckstypen

Typ einer Expression:

- **`integer`**
  - Variable
  - Number
  - Expression mit Kopf +,-,*,/
- **`boolean`**
  - true, false
  - Expression mit Kopf OR, AND, NOT, <,>,<=,>=,<>,=

- Vereinbarung (für Einfachheit): Typen von Bedingungen
- (IF, WHILE) sei **`boolean`**; Typ von Anweisung sei **`integer`**

Kapitel 2

Operationelle Semantik

Idee

- Bedeutung eines Programms wird angegeben durch Transformation in ein Programm einer anderen, als bekannt vorausgesetzten Sprache/Maschine

Varianten

- Maschinensemantik: Übersetzung in Code einer konkreten Maschine

  Jeder Compiler; schwach wegen Maschinenbesonderheiten

- Abstrakte Maschinensemantik: Übersetzung in Code einer abstrakten Maschine

  WAM (Prolog), JVM (Java), P-Code (Pascal), Lilith/M-Code (Modula-2), .NET, ...; Pragmatisch, für Portabilität, einige nur durch informale Standards festgelegt

- Selbstdefinition: Ein Interpreter für Sprache X, geschrieben in X

  LISP, PROLOG; informal ok, formal unbrauchbar

- Structural Operational Semantics: Übersetzung in ein Automatenmodell (Small step)

  Zustand = Variablenbelegungen etc. Schritt = elementare Zustandstransformation durch Statement

- Natural Semantics: dasselbe (Big Step)

  Zustand = Variablenbelegungen etc. Schritt = Zustandstransformation durch Statement

Vorbereitung: Betrachten W

- Zustand: ein Wert für jede Variable

- Typ: Semantik eines Typs t ist ein Wertebereich (Menge) WB(t)
  - WB(integer) := {0,1,-1,2,-2,3,-3,...} WB(bool) := {tt,ff}

- Expression: wird interpretiert auf einem Zustand (liefert Wert), ändert Zustand selbst nicht

- Command: Ändert Zustand, liefert keinen Wert

Zustand in W

- Formal: Ein Zustand s ist eine Abbildung
  - Definitionsbereich ist $\mathcal{V}$ (Menge der Variablen des Programms)
  - Für $v \in \mathcal{V}$ ist $s(v)$ aus $\mathcal{WB}$(integer)

Abstrahiert von Implementationsdetails wie

- Speicheradresse
- Binärkodierung

Semantik einer Expression E

... ist eine Abbildung States → $\mathcal{WB}$(typ(E))

In W: Zu einer arithmetischen Expression sei $\mathcal{A}$ ⟦E⟧ ,

zu einer booleschen Expression sei $\mathcal{B}$ ⟦E⟧

ihre Semantik,

also: $\mathcal{A}$: AExp → (State → $\mathcal{WB}$(integer))

$\mathcal{B}$: BExp → (State → $\mathcal{WB}$(boolean))

Definition der Semantik: strukturelle Induktion über der Syntax einer Expression

Semantik einer Expression E

Sei s Zustand.

$\mathcal{A}\,[\![n]\!]\,(s)$ = value(n), falls n *number* (vertiefen wir nicht weiter)

$\mathcal{A}\,[\![x]\!]\,(s) = s(x)$,

$\mathcal{A}\,[\![E1+E2]\!]\,(s) = \mathcal{A}\,[\![E1]\!]\,(s) + \mathcal{A}\,[\![E2]\!]\,(s)$

$\mathcal{A}\,[\![E1-E2]\!]\,(s) = \mathcal{A}\,[\![E1]\!]\,(s) - \mathcal{A}\,[\![E2]\!]\,(s)$

$\mathcal{A}\,[\![E1*E2]\!]\,(s) = \mathcal{A}\,[\![E1]\!]\,(s) * \mathcal{A}\,[\![E2]\!]\,(s)$

$\mathcal{A}\,[\![E1/E2]\!]\,(s) = \lfloor \mathcal{A}\,[\![E1]\!]\,(s) : \mathcal{A}\,[\![E2]\!]\,(s) \rfloor$

$\mathcal{B}\,[\![true]\!]\,(s) = tt$

$\mathcal{B}\,[\![false]\!]\,(s) = ff$

$\mathcal{B}\,[\![E1\ OR\ E2]\!]\,(s) =$ ff, falls $\mathcal{B}\,[\![E1]\!]\,(s) = \mathcal{B}\,[\![E2]\!]\,(s) = ff$, sonst tt

$\mathcal{B}\,[\![E1\ AND\ E2]\!]\,(s) =$ tt, falls $\mathcal{B}\,[\![E1]\!]\,(s) = \mathcal{B}\,[\![E2]\!]\,(s) = tt$, sonst ff

$\mathcal{B}\,[\![NOT\ E]\!]\,(s) =$ tt, falls $\mathcal{B}\,[\![E]\!]\,(s) = ff$, sonst ff

$\mathcal{B}\,[\![E1<E2]\!]\,(s) =$ tt, falls $\mathcal{A}\,[\![E1]\!]\,(s) < \mathcal{A}\,[\![E2]\!]\,(s)$, sonst ff

analog: >, >=,<=, =, <>

2.1 Natural Semantics (NatS)

- Es geht nun um *commands*.

- Idee: Relation $\langle S,s\rangle \rightarrow s'$
  - „Gestartet in Zustand s, terminiert Command S und führt zu Zustand s' “

- Formulierung der Definitionen

  (a) Axiome, z.B. $\langle skip,s\rangle \rightarrow s$

  (b) Regeln , z.B.

Prämisse(n) *Bedingung*

$$\frac{\langle S2,s\rangle \rightarrow s'}{\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end},s\rangle \rightarrow s'} \quad \text{falls } \mathcal{B}[\![b]\!](s) = ff$$

Folgerung

Natural Semantics für W

$s[x \rightarrow k](x) = k$
$s[x \rightarrow k](y) = s(y) \quad (y \neq x)$

- $[\mathrm{skip}_{\mathrm{NatS}}]$ $\quad \langle \mathbf{skip}, s\rangle \rightarrow s$
- $[\mathrm{ass}_{\mathrm{NatS}}]$ $\quad \langle \mathbf{x := E}, s\rangle \rightarrow s[x \rightarrow \mathcal{A}\,[\![E]\!]\,(s)]$
- $[\mathrm{comp}_{\mathrm{NatS}}]$ $\quad \dfrac{\langle \mathbf{S1}, s\rangle \rightarrow s' \qquad \langle \mathbf{S2}, s'\rangle \rightarrow s''}{\langle \mathbf{S1;S2}, s\rangle \rightarrow s''}$
- $[\mathrm{if1}_{\mathrm{NatS}}]$ $\quad \dfrac{\langle \mathbf{S1}, s\rangle \rightarrow s'}{\langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, s\rangle \rightarrow s'}$ falls $\mathcal{B}\,[\![b]\!]\,(s) = tt$
- $[\mathrm{if2}_{\mathrm{NatS}}]$ $\quad \dfrac{\langle \mathbf{S2}, s\rangle \rightarrow s'}{\langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, s\rangle \rightarrow s'}$ falls $\mathcal{B}\,[\![b]\!]\,(s) = ff$
- $[\mathrm{while1}_{\mathrm{NatS}}]$ $\quad \dfrac{\langle \mathbf{S1}, s\rangle \rightarrow s' \quad \langle \mathbf{while\ b\ do\ S1\ end}, s'\rangle \rightarrow s''}{\langle \mathbf{while\ b\ do\ S1\ end}, s\rangle \rightarrow s''}$ falls $\mathcal{B}\,[\![b]\!]\,(s) = tt$
- $[\mathrm{while2}_{\mathrm{NatS}}]$ $\quad \langle \mathbf{while\ b\ do\ S\ end}, s\rangle \rightarrow s$ $\quad$ falls $B\,[\![b]\!]\,(s) = ff$

Zielzustand ermitteln

- **x:=x-y;y:=x+y;x:=y-x**
- Startzustand: s(x) = 3, s(y) = 7
- ges: Zielzustand
- Weg: Folgerungsbaum aus Regel/Axiom-*Instanzen*

<**x:=y-x**,{(x,-4),(y,3)>→{(x,7),(y,3)}

<**y:=x+y**,{(x,-4),(y,7)}>→{(x,-4),(y,3)}

<**y:=x+y;x:=y-x**,{(x,-4),(y,7)}>→{(x,7),(y,3)}

<**x:=x-y**,{(x,3),(y,7)}>→{(x,-4),(y,7)}

<**x:=x-y;y:=x+y;x:=y-x**,{(x,3),(y,7)}> → {(x,7),(y,3)}

Semantische Äquivalenz nachweisen

- wollen zeigen:
  - **`while b do S end`** ist semantisch äquivalent zu
  - **`if b then S; while b do S end else skip end`**

- Formal: Für alle s,s‘:
  - < **`while b do S end`**,s>→s‘ genau dann, wenn
  - < **`if b then S;while b do S end else skip end`**, s>→s‘

Beweis: Implikation

- Es gelte < **while b do S end**,s>→s‘.

- Erster Fall: Es gelte $\mathcal{B}[\![b]\!](s)=tt$.

  → [while1$_{NatS}$]: es gibt s“ mit <**S**,s>→s“ und <**while b do S end**,s“>→s‘

  → [comp$_{NatS}$]: <**S;while b do S end**,s>→s‘

  → [if1$_{NatS}$]:<**if b then S; while b do S end else skip end**,s>→s‘

- Zweiter Fall: Es gelte $\mathcal{B}[\![b]\!](s)=ff$.

  →[while2$_{NatS}$]: s=s‘

  Gleiches Resultat liefert [skip$_{NatS}$]: <**skip**,s>→s und folglich

  [if2$_{NatS}$]: <**if b then S; while b do S end else skip end**,s>→s‘.

Beweis: Replikation

- Es gelte <**if b then S; while b do S end else skip end**,s>→s‘.

- Erster Fall: Es gelte $\mathcal{B}[\![b]\!](s) = tt$.

  → $[\text{if1}_{NatS}]$: <**S; while b do S end**,s>→s‘

  → $[\text{comp}_{NatS}]$: es gibt s“ mit <**S**,s>→s“ und <**while b do S end**,s“>→s‘

  → $[\text{while1}_{NatS}]$: <**while b do S end**,s>→s‘

- Zweiter Fall: Es gelte $\mathcal{B}[\![b]\!](s) = ff$.

  → $[\text{if2}_{NatS}]$: <**skip**,s>→s‘, also s=s‘

  → $[\text{while2}_{NatS}]$: < **while b do S end**,s>→s‘.

Terminierung

- Ausführung von S in s
  - terminiert, falls es ein s‘ gibt mit <S,s>→s‘
  - kreist, falls es kein s‘ gibt mit <S,s>→s‘ (dann: kein endlicher Folgerungsbaum)

- Ausführung von S
  - terminiert immer, falls S in allen Zuständen s terminiert
  - kreist immer, falls S in allen Zuständen s kreist

- Beispiele:
  - Jedes while-freie Statement in W terminiert immer
  - **`while true do skip end`** kreist immer
  - **`while x > 1 do y:=0; while y < x do y:=y+2 end; if x = y then x := x / 2 else x := 3 * x + 1 end end`** immer-Terminierung ungeklärt

Beweis: Jedes while-freie S terminiert immer

- (Induktion über Struktur desFolgerungsbaums)
- [$skip_{NatS}$], [ass_{NatS}]: Zu jedem s ex. s‘; Baum hat Tiefe 0
- [$if1_{NatS}$], [$if2_{NatS}$]: Ind.-Voraussetzung liefert: es gibt ein ein s‘ mit $\langle S1,s\rangle \rightarrow s'$ bzw. $\langle S1,s\rangle \rightarrow s'$
- [$comp_{NatS}$]:Ind.-Voraussetzung liefert: Es gibt ein s‘ mit $\langle S1,s\rangle \rightarrow s'$ und es gibt ein s“ mit $\langle S2,s'\rangle \rightarrow s''$

Beweis: while true do skip end kreist immer

- Einzige anwendbare Regel: [while1$_{NatS}$]

$$\frac{\langle \text{skip}, s\rangle \to s \qquad \langle \text{while true do skip end}, s\rangle \to s'}{\langle \text{while true do skip end}, s\rangle \to s'}$$

- 2. Prämisse und Folgerung identisch

→ kein endlicher Ableitungsbaum generierbar

Determinismus

- Zu jedem Command S in W und jedem s gibt es max. ein s‘ mit <S,s>→s‘

- Beweis: (Induktion über Struktur des Folgerungsbaums)
- [skip$_{NatS}$], [ass$_{NatS}$], [while2$_{NatS}$]: s‘ von s jeweils funktional abhängig
- [if1$_{NatS}$], [if2$_{NatS}$]: Ind.-Voraussetzung liefert: es gibt max. ein s‘ mit <S1,s>→s‘ bzw. <S1,s>→s‘
- [comp$_{NatS}$]: Ind.-Voraussetzung liefert: Es gibt max. ein s‘ mit <S1,s>→s‘ und es gibt max. ein s“ mit <S2,s‘>→s“
- [while1$_{NatS}$]:Ind.-Voraussetzung liefert: Es gibt max. ein s‘ mit <S,s>→s‘ und max. ein s“ mit <`while b do S end`,s‘>→s“

Zum Vergleich mit anderen Semantiken

- definieren semantische Funktion $\mathcal{S}_{NatS}$:Cmd→(State↪State) wie folgt:
- $\mathcal{S}_{NatS}$ ⟦S⟧ (s) := s‘, falls <S,s>→s‘, und *n.def.*, sonst
- Rechtfertigung: Determiniertheit von W
- Bemerkung: Für Sprachen mit Nichtdeterminismus wäre $\mathcal{S}$:Cmd→(State→℘(State))

Zusammenfassung: Natural Semantics

- Zu jedem Command: Zustandsüberführung
  - Ermittlung per Folgerungsbaum

- Können mittels Semantik
  - Werte von Zielzuständen ermitteln
  - semantische Äquivalenz von Konstrukten nachweisen
  - über Terminierung und Determiniertheit argumentieren

2.2 Structural Operational Semantics

- Idee: Semantik eines Commands ist *Folge elementarer* Zustandstransformationen

- Zum Vergleich: Natural Semantics liefert eine Zustandsüberführung für das *gesamte* Command, gerechtfertigt durch Überführungen (meist) einfacherer Commands

Notation

- <S,s> ⇒ <S‘,s‘> : Um S in s auszuführen, muss man erst nach s‘ übergehen und dort S‘ ausführen

- <S,s> ⇒ s‘ : Ausführung von S in s geht in s‘ über und terminiert dort

Structural Operational Semantics für W

- $[\text{skip}_{\text{sos}}]$ $\langle \mathbf{skip}, s\rangle \Rightarrow s$

- $[\text{ass}_{\text{sos}}]$ $\langle \mathbf{x:=E}, s\rangle \Rightarrow s[x \to \mathcal{A} [\![E]\!] (s)]$
- $[\text{comp1}_{\text{sos}}]$ $$\frac{\langle \mathbf{S1}, s\rangle \Rightarrow s'}{\langle \mathbf{S1;S2}, s\rangle \Rightarrow \langle \mathbf{S2}, s'\rangle}$$
- $[\text{comp2}_{\text{sos}}]$ $$\frac{\langle \mathbf{S1}, s\rangle \Rightarrow \langle \mathbf{S1'}, s'\rangle}{\langle \mathbf{S1;S2}, s\rangle \Rightarrow \langle \mathbf{S1';S2}, s'\rangle}$$
- $[\text{if1}_{\text{sos}}]$ $\langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, s\rangle \Rightarrow \langle \mathbf{S1}, s\rangle$
  falls $\mathcal{B} [\![b]\!] (s) = tt$
- $[\text{if2}_{\text{sos}}]$ $\langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, s\rangle \Rightarrow \langle \mathbf{S2}, s\rangle$
  falls $\mathcal{B} [\![b]\!] (s) = ff$
- $[\text{while}_{\text{sos}}]$ $\langle \mathbf{while\ b\ do\ S\ end}, s\rangle \Rightarrow \langle \mathbf{if\ b\ then\ S;\ while\ b\ do\ S\ end\ else\ skip\ end}, s\rangle$

Abarbeitungsssequenzen

- endlich, terminierend, z.B. (mit s(x) = a, s(y) = b)
  - <**x:=x–y;y:=x+y;x:=y–x**,s> ⇒
    <**y:=x+y;x:=y–x**,s[x→a-b]>⇒
    <**x:=y–x**,s[x→a-b,y→a]>⇒
    s[x→b,y→a]

- unendlich, nicht terminierend, z.B.
  - <**while true do skip end**,s> ⇒
    <**if true then skip;while true do skip end else skip end**,s> ⇒
    < **skip;while true do skip end**,s> ⇒
    <**while true do skip end**,s> ⇒...

- endlich, verklemmend (endet in einem <S‘,s‘> - kommt in W nicht vor, aber in Erweiterungen)

Semantische Äquivalenz

- S1 und S2 sind semantisch äquivalent, falls folgende Bedingungen für alle s, s', S' gelten:

  – $\langle S1,s\rangle \Rightarrow^* s'$ gdw. $\langle S2,s\rangle \Rightarrow^* s'$

  – Wenn $\langle S1,s\rangle \Rightarrow^* \langle S',s'\rangle$ (ohne Nachfolger) so gibt es ein $S2'$ mit $\langle S2,s\rangle \Rightarrow^* \langle S2',s''\rangle$ (ohne Nachfolger)

  – Wenn $\langle S2,s\rangle \Rightarrow^* \langle S',s'\rangle$ (ohne Nachfolger) so gibt es ein $S1'$ mit $\langle S1,s\rangle \Rightarrow^* \langle S1',s''\rangle$ (ohne Nachfolger)

  – Wenn $\langle S1,s\rangle \Rightarrow^*$ (unendlich), so $\langle S2,s\rangle \Rightarrow^*$ (unendlich).

Argumentieren über Terminierung, Determiniertheit analog zu NatS

Unterschiede:

- SOS unterscheidet Terminierung und Verklemmung
- Argumentation durch Induktion über Abarbeitungssequenzen statt über Folgerungsbaum

Zum Vergleich mit anderen Semantiken

- definieren semantische Funktion $\mathcal{S}_{SOS}$:Cmd→(State↪State) wie folgt:

- $\mathcal{S}_{SOS}$ ⟦S⟧ (s) := s‘, falls <S,s>⇒* s‘, und *n.def.*, sonst

- Rechtfertigung: Determiniertheit von W

Vergleich: SOS versus NatS

- Für jedes Command aus W gilt: $\mathcal{S}_{SOS} [\![S]\!] = \mathcal{S}_{NatS} [\![S]\!]$
- Zum Beweis: Zeigen
  - Wenn $\langle S,s \rangle \rightarrow s'$ so $\langle S,s \rangle \Rightarrow^* s'$
  - Wenn $\langle S,s \rangle \Rightarrow^* s'$ so $\langle S,s \rangle \rightarrow s'$

Beweis Teil 1

- per Induktion über dem Folgerungsbaum
- Es gelte <S,s>→s‘ (also ex. endlicher Folgerungsbaum)
- Fall [$skip_{NatS}$]: Also s‘=s; SOS liefert <**skip**,s>⇒s
- Fall [ass_{NatS}]: Also s‘ = s[x→$\mathcal{A}$ ⟦E⟧ (s)]; SOS dito.
- Fall [$comp_{NatS}$]: Es gibt also s“ mit <S1,s>→s“ und <S2,s“>→s‘.
  Nach Ind.Vor: <S1,s>⇒*s“ und <S2,s“>⇒*s‘. Aus <S1,s>⇒*s“
  kann man folgern: <S1;S2,s>⇒*<S2,s“>. Also <S1;S2,s>⇒*s‘
- Fall [$if1_{NatS}$]: Nach Ind.Vor: <S1,s>⇒*s‘. SOS liefert also
  <**if b then S1 else S2 end**,s> ⇒ <S1,s> ⇒*s‘.
- Fall [$if2_{NatS}$]: Analog.
- Fall [$while1_{NatS}$]: Nach Ind.Vor: <S,s>⇒*s“ und <while b do S end,s“>⇒*s‘.
  Also <**while b do S end**,s> ⇒ <**if b then S; while b do S end else skip end**,s> ⇒ < **S; while b do S end**,s> ⇒* <**while b do S end**,s“> ⇒*s‘
- Fall [$while2_{NatS}$]: Also s=s‘; SOS liefert <**while b do S end**,s> ⇒ <**if b then S; while b do S end else skip end**,s> ⇒ <**skip**,s> ⇒s.

Beweis Teil 2

- per Induktion über Abarbeitungssequenzen
- Es gelte <S,s>⇒*s‘ (also ex. endlicher Folgerungsbaum)
- Fall [skip$_{SOS}$], [ass$_{SOS}$]: leicht zu sehen
- Fall [comp1/2$_{SOS}$]: Man kann zeigen: Es gibt ein s“ mit <S1;S2,s>⇒*s“ und <S2,s“>⇒*s‘. <S1;S2,s>⇒*s“ kann man folgern: <S1,s>⇒*s“. Nach Ind.Vor: <S1,s>s →s“ und <S2,s“> → s‘. Also: <S1;S2,s>→s‘.
- Fall [if1$_{SOS}$]: <**`if b then S1 else S2 end`**,s>⇒<S1,s>⇒*s‘. Nach Ind.Vor: <S1,s>⇒*s‘. NatS liefert also <**`if b then S1 else S2 end`**,s> → s‘.
- Fall [if2$_{SOS}$]: Analog.
- Fall [while$_{SOS}$]: <**`while b do S end`**,s> ⇒ <**`if b then S; while b do S end else skip end`**,s> ⇒* s‘. Nach Ind.Vor: <**`if b then S; while b do S end else skip end`**,s> →s‘. Wegen sem. Äquivalenz auch <while b do S end,s>→s‘.

2.3 Erweiterungen von W: 1. Abort

- Neues command: **`abort`**.
- Semantik informal: Programm abbrechen, also *nicht fortsetzen*
- SOS/NatS von W mit **`abort`** = SOS/NatS von W ohne **`abort`**!
  (sprich: auf abort keine Regeln/Axiome anwendbar)

- Dadurch in beiden Semantiken: **`skip`** und **`abort`** nicht äquivalent

- SOS → Ausführung von **`abort`** führt zu Verklemmung
- NatS → Ausführung von abort = Es gibt keinen Folgerungsbaum

- In SOS sind **`abort`** und **`while true do skip end`** nicht äquivalent
- In NatS sind **`abort`** und **`while true do skip end`** äquivalent!

Erweiterungen von W: 2. Nichtdeterminismus

- Neues command: **`S1 or S2`**.
- Semantik informal: nichtdeterministische Wahl zw. S1 und S2
- $[\mathrm{or1}_{\mathrm{NatS}}]$: $$\frac{\langle S1,s\rangle \rightarrow s'}{\langle \mathtt{S1\ or\ S2},s\rangle \rightarrow s'}$$ $[\mathrm{or2}_{\mathrm{NatS}}]$: $$\frac{\langle S2,s\rangle \rightarrow s'}{\langle \mathtt{S1\ or\ S2},s\rangle \rightarrow s'}$$

- $[\mathrm{or1}_{\mathrm{SOS}}]$: $\langle \mathtt{S1\ or\ S2},s\rangle \Rightarrow \langle S1,s\rangle$

  $[\mathrm{or2}_{\mathrm{SOS}}]$: $\langle \mathtt{S1\ or\ S2},s\rangle \Rightarrow \langle S2,s\rangle$

- Betrachten (while true do skip end) or x:x-y;y:=x+y;x:=x-y;
- NatS → Es gibt einen endlichen Folgerungsbaum
- SOS → Es gibt sowohl endliche als auch unendliche Abarbeitungssequenzen
- → NatS versteckt *mögliches* kreisendes Verhalten

Erweiterungen von W: 3. Nebenläufigkeit

- Neues command: **`S1 par S2`**.
- Semantik informal: nebenläufige Ausführung von S1 und S2
- $[\text{par1}_{\text{sos}}]$: $\dfrac{\langle \mathtt{S1}, s\rangle \Rightarrow s'}{\langle \mathtt{S1\ par\ S2}, s\rangle \Rightarrow \langle \mathtt{S2}, s'\rangle}$ $[\text{par2}_{\text{sos}}]$: $\dfrac{\langle \mathtt{S2}, s\rangle \Rightarrow s'}{\langle \mathtt{S1\ par\ S2}, s\rangle \Rightarrow \langle \mathtt{S1}, s'\rangle}$

$[\text{par3}_{\text{sos}}]$: $\langle \mathtt{S1}, s\rangle \Rightarrow \langle \mathtt{S1}', s'\rangle$

$$\langle \mathtt{S1\ par\ S2}, s\rangle \Rightarrow \langle \mathtt{S1}'\ \mathtt{par\ S2}, s'\rangle$$

$[\text{par4}_{\text{sos}}]$: $\langle \mathtt{S2}, s\rangle \Rightarrow \langle \mathtt{S2}', s'\rangle$

$$\langle \mathtt{S1\ par\ S2}, s\rangle \Rightarrow \langle \mathtt{S1\ par\ S2}', s'\rangle$$

- In NatS nicht ausdrückbar, weil zu jedem Command ein *atomarer* Zustandsübergang definiert ist!

Erweiterungen von W: 4. Blöcke

- mit Variablendeklarationen (dürfen weiter außen bereits deklariert sein)
- Syntax:

```
command = ... | „begin“ declaration cmd_seq „end“
```

- Verwendung eines Namens bezieht sich immer auf die innerste Deklaration.

```
begin   var x,y : integer;
         y := 1; x := 2;
           begin     var x : integer;
                        x := 3; y := x + 1
           end ;
           x := x + y
end
```

Natural Semantics für W + Blöcke

- setzen voraus: Für **`declarationlist`** L sei dv(L) die Menge der in L deklarierten Variablen
- Deklaration selbst hat keine Wirkung auf den Zustand
- Nach Verlassen von **`block`** müssen Variablenwerte auf den Inhalt vor Betreten des Blocks zurückgesetzt werden

$$[\text{block}_{\text{NatS}}]: \quad \frac{\langle S,s\rangle \to s'}{\langle \textbf{begin}\ DL\ S\ \textbf{end}, s\rangle \to s'[\{x \to s(x) \mid x \in dv(DL)\}]}$$

Erweiterungen von W: Prozeduren

- Syntax:

```
declaration = „var“ ...
            | „proc“ „identifier“ „is“ block
cmd         = „call“ „identifier“
```

- In Blockstrukturen können Bezeichner mehrfach deklariert sein. Es wird immer die innerste Deklaration/Definition verwendet. Unterscheiden:
- Statische/dynamische Scopes für
  - Prozeduren
  - Variablen

  - Statisch: Auswahl nach Einbettung von Blöcken im Quelltext
  - Dynamisch: Auswahl nach Einbettung in der Aufrufstruktur

Beispiel statisch/dynamisch

```
begin  var x; x:= 0;
  proc p is begin x := x * 2 end
  proc q is begin call p end
  begin var x; x := 5;
     proc p is x := x + 1;
     call q;
           y := x;
  end
end
```

Proc statisch/Var statisch: y = 5
Proc statisch/Var dynamisch: y = 10
Proc dynamisch/Var dynamisch: y = 6

Prozedurumgebungen

- Brauchen: Zuordnung von Namen zu Blöcken:
  - env: Names → Commands
- Betreten von Blöcken = Aktualisierung von env:
  - upd(proc p is S; DL, env) = upd(DL,env[p$\mapsto$S])
  - upd(ε,env) = env
- Alle Semantikregeln stehen im Kontext einer Umgebung env:
  - env $\vdash$ <skip,s> → s

Natural Semantics für W mit Proz. (dyn. Scopes)

- $[skip_{NatS}]$ $env \vdash \langle \mathbf{skip}, s\rangle \to s$

- $[ass_{NatS}]$ $env \vdash \langle \mathbf{x:=E}, s\rangle \to s[x \to \mathcal{A}\llbracket E\rrbracket(s)]$
- $[comp_{NatS}]$ $$\frac{env \vdash \langle \mathbf{S1}, s\rangle \to s' \quad env \vdash \langle \mathbf{S2}, s'\rangle \to s''}{env \vdash \langle \mathbf{S1;S2}, s\rangle \to s''}$$
- $[if1_{NatS}]$ $$\frac{env \vdash \langle \mathbf{S1}, s\rangle \to s'}{env \vdash \langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, s\rangle \to s'} \quad \text{falls } \mathcal{B}\llbracket b\rrbracket(s) = tt$$
- $[if2_{NatS}]$ $$\frac{env \vdash \langle \mathbf{S2}, s\rangle \to s'}{env \vdash \langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, s\rangle \to s'} \quad \text{falls } \mathcal{B}\llbracket b\rrbracket(s) = ff$$
- $[while1_{NatS}]$ $$\frac{env \vdash \langle \mathbf{S}, s\rangle \to s' \quad env \vdash \langle \mathbf{while\ b\ do\ S\ end}, s'\rangle \to s''}{env \vdash \langle \mathbf{while\ b\ do\ S1\ end}, s\rangle \to s''} \quad \text{falls } \mathcal{B}\llbracket b\rrbracket(s) = tt$$
- $[while2_{NatS}]$ $env \vdash \langle \mathbf{while\ b\ do\ S\ end}, s\rangle \to s$ falls $\mathcal{B}\llbracket b\rrbracket(s) = ff$
- [blockNatS] $$\frac{upd(DL, env) \vdash \langle S, s\rangle \to s'}{env \vdash \langle \mathbf{begin\ DL\ S\ end}, s\rangle \to s'[\{x \to s(x) \mid x \in dv(DL)\}]}$$
- $[call_{NatS}]$ $$\frac{env \vdash \langle S, s\rangle \to s'}{env \vdash \langle \text{call } p, s\rangle \to s'} \quad \text{falls } env(p) = S$$

Natural Sem. für W mit Proz. (gemischte Scopes)

- müssen env verfeinern: brauchen Zuordnung von Namen zu Blöcken *zum Zeitpunkt der Definition* einer Prozedur:
  - env: Names → Commands x Environments
  - upd(proc p is S; DL, env) = upd(DL,env[p↦(S,env)])
  - upd(ε,env) = env

- $[\text{call}_{\text{NatS}}]$ $$\frac{\text{env}'[p\mapsto(S,\text{env}')] \vdash \langle S,s\rangle \to s'}{\text{env} \vdash \langle \text{call } p,s\rangle \to s'}$$ falls env(p) = (S,env')

Nur relevant für rekursiv gerufene Prozeduren

Natural Sem. für W mit Proz. (static Scopes)

- müssen Konzept des Zustands verfeinern:
  - bisher: Variable verweist auf Wert. Hier:
    - Variable verweist auf *Location* *(Env_V:Names →Loc)*
    - Location verweist auf Wert

      (Sto: Loc →WB(integer/boolean))
  - new: → Loc   liefert eine bislang nicht verwendete Location

  - Assignments ändern Sto
  - Blöcke ändern Env
  - Ausdrucksauswertung bezieht Werte, indem von Namen über Env auf eine Location und von dieser über sto auf einen Wert verweisen wird:

    $s = sto \circ env_V$

Variablendeklarationen

- Variablendeklarationen definieren Variablenumgebungen:

$$[\mathrm{var}_{\mathrm{NatS}}]: \quad \frac{\langle DL, env_V[x \rightarrow new()] \rangle \rightarrow env_V'}{\langle var\ x : type;\ DL \rangle \rightarrow env_V'}$$

$$[\mathrm{none}_{\mathrm{NatS}}]: \quad \langle \varepsilon, env_V \rangle \rightarrow env_V$$

- Prozedurdeklarationen aktualisieren Prozedurumgebungen:

$$upd(proc\ p\ is\ S;DL\ , env_V, env_P) = upd(DL, env_V, env_P[p \rightarrow (S, env_V, env_P)])$$

$$upd(\varepsilon, env_V, env_P) = env_P$$

Natural Semantics für W mit Proz. (st. Scopes)

- [skip$_{NatS}$] $env_V, env_P \vdash \langle \mathbf{skip}, sto\rangle \rightarrow sto$

- [ass2$_{NatS}$] $env_V, env_P \vdash \langle \mathbf{x:=E}, sto\rangle \rightarrow sto[env_V(x) \rightarrow \mathcal{A}[\![E]\!](sto \circ env_V)]$
- [comp$_{NatS}$] $$\frac{env_V, env_P \vdash \langle \mathbf{S1}, sto\rangle \rightarrow sto' \quad env_V, env_P \vdash \langle \mathbf{S2}, sto'\rangle \rightarrow sto''}{env_V, env_P \vdash \langle \mathbf{S1;S2}, sto\rangle \rightarrow sto''}$$
- [if1$_{NatS}$] $$\frac{env_V, env_P \vdash \langle sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \mathbf{if\ b\ then\ S1\ else\ S2\ end}, sto\rangle \rightarrow sto'} \quad \text{falls } \mathcal{B}[\![b]\!](s) = tt$$
- [if2$_{NatS}$] analog
- [while1$_{NatS}$] analog
- [while2$_{NatS}$] analog
- [blockNatS] $$\frac{\begin{array}{c}\langle DL_{Var}, env_V\rangle \rightarrow env_V' \\ upd(DL_{Proc}, env_V', env_P) \vdash \langle S, sto\rangle \rightarrow sto'\end{array}}{env_V, env_P \vdash \langle \mathbf{begin\ DL\ S\ end}, sto\rangle \rightarrow sto'}$$
- [call$_{NatS}$] $$\frac{env_V', env_P' \vdash \langle S, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle call\ p, sto\rangle \rightarrow sto'} \quad \text{mit } env_P(p) = (S, env_V', env_P')$$

Lektionen aus Erweiterungen

- SOS besser für Nichtdeterminismus und Parallelität
- NatS besser für Blockstrukturen
- Komplexe Sprachen erfordern angepasste Zustandsdefinitionen (env, Loc, Sto,...)

2.4 Eine beweisbar korrekte Implementation

- Brauchen:
  - Maschine
  - Formale Semantik dieser Maschine
  - Übersetzung von W in die Sprache der Maschine
  - Beweis der Übereinstimmung der Semantik eines W-Programms mit der seiner Übersetzung

Die Maschine AM

Besitzt:

- einen Programmspeicher für eine Anweisungssequenz

- einen Kellerspeicher für die Ausdrucksauswertung

  Eine Kellerkonfiguration ist Element von
  $(\mathcal{WB}(\text{integer}) \cup \mathcal{WB}(\text{boolean}))^*$

- einen allgemeinen Speicher für Variablenwerte

  $s \in \text{State}$

Anweisungen von AM

- PUSH *n*: Konstante n auf Keller schreiben
- TRUE, FALSE: analog
- ADD: oberste Kellersymbole entnehmen, addieren, Ergebnis auf Keller schreiben
- MULT, SUB, DIV, OR, NEG, EQ, NEQ, LE, GE, LT, GT: analog
- FETCH *x*: Variablenwert aus allgemeinem Speicher lesen und auf Keller schreiben
- STORE *x*: oberstes Kellersymbol entfernen und als Wert in Variable x speichern
- NOOP: nix tun
- BRANCH (c1,c2): oberstes Kellersymbol (bool) entfernen und abhängig vom Wahrheitswert c1 oder c2 ausführen
- LOOP(c1,c2): c1 ausführen, oberstes Kellersymbol (bool) entfernen, bei ff beenden, sonst c2 ausführen und von vorn
- c1 : c2: Hintereinanderausführung

Operationelle Semantik für AM

- Zustand (Konfiguration) besteht aus
  - verbleibender Anweisungssequenz
  - Stackkonfiguration
  - Zustand (Variablenwerte)

Operationelle Semantik für AM

- <PUSH *n*:c,e,s> ▷ <c, n e, s>
- analog: TRUE, FALSE
- <ADD:c,x y e,s> ▷ <c, (x+y) e, s>
- analog: SUB MULT DIV AND OR NEG EQ NEQ LT GT LE GE
- <FETCH *x*:c,e,s> ▷ <c, s(*x*) e, s>
- <STORE *x*:c,y e,s> ▷ <c, e, s[x →y]>
- <NOOP:c,e,s> ▷ <c,e,s>
- <BRANCH(c1,c2):c,x e,s> ▷ <c1:c,e,s>, falls x = tt
- <BRANCH(c1,c2):c,x e,s> ▷ <c2:c,e,s>, falls x = ff
- <LOOP(c1,c2):c,e,s> ▷ <c1:BRANCH(c2:LOOP(c1,c2),NOOP):c,e,s>

Beispiel

- Ausführung von PUSH 1:FETCH x:ADD:STORE x bei leerem Stack und Initialzustand x = 3:

< PUSH 1:FETCH x:ADD:STORE x,ε,s> ▷

< FETCH x:ADD:STORE x,1,s> ▷

< ADD:STORE x,3 1,s> ▷

< STORE x,4,s> ▷

< ε, ε,s[x→4]>

(Terminierung)

Die Übersetzung: Expressions

- Ziel: expression E → Anweisungssequenz $\mathcal{C}$(E) mit:
  - $<\mathcal{C}(E),e,s> \triangleright^* <\varepsilon, \mathcal{A}/\mathcal{B}\ [\![E]\!]\ (s)\ e,s>$

- $\mathcal{C}$(x) = FETCH *x*
- $\mathcal{C}$(n) = PUSH *n*
- $\mathcal{C}$(true) = TRUE, $\mathcal{C}$(false) = FALSE
- $\mathcal{C}$(E1 + E2) = $\mathcal{C}$(E2):$\mathcal{C}$(E1):ADD ; andere Operationen analog

Die Übersetzung: Statements

- $\mathcal{C}$(x := E) = $\mathcal{C}$(E): STORE *x*
- $\mathcal{C}$(skip) = NOOP
- $\mathcal{C}$(S1;S2) = $\mathcal{C}$(S1):$\mathcal{C}$(S2)
- $\mathcal{C}$(if b then S1 else S2 end) = $\mathcal{C}$(b):BRANCH($\mathcal{C}$(S1), $\mathcal{C}$(S2))
- $\mathcal{C}$(while b do S end) = LOOP($\mathcal{C}$(b), $\mathcal{C}$(S))

Beispiel

$\mathcal{C}$(y:=1;while NOT (x=1) do y:=y*x;x:=x-1 END)

=

PUSH 1: STORE y:

LOOP(PUSH 1:FETCH x:EQ:NEG,

FETCH x:FETCH y:MULT:STORE y:

PUSH 1:FETCH x:SUB:STORE x)

Korrektheitsbeweis: Plan

1. Zeigen für beliebige Expressions von W:

   $<\mathcal{C}(E),e,s> \triangleright^* <\varepsilon, \mathcal{A}/\mathcal{B} [\![E]\!] (s)\ e,s>$

   (und Zwischenkonfigurationen haben Stack der Form w e)

2. Zeigen für beliebige Commands:

   $\mathcal{S}_{NatS} [\![S]\!] = \mathcal{S}_{AM} [\![\mathcal{C}(S)]\!]$ für alle S aus W.

   2.1 Wenn $<S,s> \rightarrow s'$, so $<\mathcal{C}(S),\varepsilon,s> \triangleright^* <\varepsilon,\varepsilon,s'>$

   2.2 Wenn $<\mathcal{C}(S),\varepsilon,s> \triangleright^* <\varepsilon,e,s'>$, so $<S,s> \rightarrow s'$ und $e=\varepsilon$

Korrektheit der Expressions

- z.Z.: $<\mathcal{C}(E),e,s> \triangleright^* <\varepsilon,\mathcal{A}/\mathcal{B}\ [\![E]\!]\ (s)\ e,s>$
- Induktion über Struktur von E:
  - $<\mathcal{C}(n),e,s>=<PUSH\ n,e,s> \triangleright$ (nach Semantik AM)
    $<\varepsilon,n=\mathcal{A}\ [\![n]\!]\ (s)\ e,s>$
  - $<\mathcal{C}(x),e,s>=<FETCH\ x,e,s> \triangleright$ (nach Semantik AM)
    $<\varepsilon,s(x)=\mathcal{A}\ [\![x]\!]\ (s)\ e,s>$
  - $<\mathcal{C}(E1+E2),e,s>=< \mathcal{C}(E2): \mathcal{C}(E1):ADD,e,s> \triangleright^*$ (nach IVor)
    $<\mathcal{C}(E1):ADD,\ \mathcal{A}\ [\![E2]\!]\ (s)\ e,s> \triangleright^*$ (nach IVor)
    $<ADD,\ \mathcal{A}\ [\![E1]\!]\ (s)\ \mathcal{A}\ [\![E2]\!]\ (s)\ e,s> \triangleright$ (nach Sem. AM)
    $<\varepsilon,\ \mathcal{A}\ [\![E1]\!]\ (s) + \mathcal{A}\ [\![E2]\!]\ (s) = \mathcal{A}\ [\![E1+E2]\!]\ (s)\ e,s>$
  - andere Operationen analog
  - Außerdem: Zwischenkonfigurationen haben Stack w e mit $w \neq \varepsilon$

Korrektheit der Statements I

- z.Z.: Wenn $\langle S,s\rangle \rightarrow s'$, so $\langle \mathcal{C}(S),\varepsilon,s\rangle \triangleright^* \langle\varepsilon,\varepsilon,s'\rangle$
- Induktion über Folgerungsbaum:
  - Sei $\langle x{:=}E,s\rangle \rightarrow s'$. $\langle \mathcal{C}(x{:=}E),\varepsilon,s\rangle = \langle \mathcal{C}(E):\mathrm{STORE}\ x,\varepsilon,s\rangle$
    $\triangleright^* \langle \mathrm{STORE}\ x, \mathcal{A}[\![E]\!](s),s\rangle$ (wg. Korrektheit Expressions)
    $\triangleright \langle\varepsilon, \varepsilon,s[x\rightarrow \mathcal{A}[\![E]\!](s)\rangle$ (nach Semantik STORE)
  - Sei $\langle \mathrm{skip},s\rangle \rightarrow s'$, also $s=s'$. $\langle \mathcal{C}(\mathrm{skip}),\varepsilon,s\rangle = \langle \mathrm{NOOP},\varepsilon,s\rangle = \langle\varepsilon,\varepsilon,s\rangle$
  - Sei $\langle S1;S2,s\rangle \rightarrow s'$. Also ex. s'' mit $\langle S1,s\rangle \rightarrow s''$ und $\langle S2,s''\rangle \rightarrow s'$.
    $\langle \mathcal{C}(S1;S2),\varepsilon,s\rangle = \langle \mathcal{C}(S1):\mathcal{C}(S2),\varepsilon,s\rangle \triangleright^* \langle \mathcal{C}(S2),\varepsilon,s''\rangle$ (nach IVor)
    $\triangleright^* \langle\varepsilon,\varepsilon,s'\rangle$ (nach IVor)
  - Sei $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end},s\rangle \rightarrow s'$.
- 1. Fall: $\mathbf{B}[\![b]\!](s)(b) = tt$, also $\langle S1,s\rangle \rightarrow s'$.
    $\langle \mathcal{C}(\text{if } b \text{ then } S1 \text{ else } S2 \text{ end}),\varepsilon,s\rangle = \langle \mathcal{C}(b):\mathrm{BRANCH}(\mathcal{C}(S1),\mathcal{C}(S2)),\varepsilon,s\rangle$
    $\triangleright^* \langle \mathrm{BRANCH}(\mathcal{C}(S1),\mathcal{C}(S2)), \mathbf{B}[\![b]\!](s)=tt,s\rangle$ (wg. Korrektheit Exp.)
    $\triangleright \langle \mathcal{C}(S1), \varepsilon,s\rangle$ (wg. Semantik BRANCH)
    $\triangleright \langle\varepsilon, \varepsilon,s'\rangle$ (IVor); 2. Fall analog.
  - Sei $\langle \text{while } b \text{ do } S \text{ end},s\rangle \rightarrow s'$.
    - 1. Fall: $\mathbf{B}[\![b]\!](s)(b) = tt$, also $\langle S,s\rangle \rightarrow s''$ und $\langle \text{while } b \text{ do } S \text{ end},s''\rangle \rightarrow s'$.
      $\langle \mathcal{C}(\text{while } b \text{ do } S \text{ end}),\varepsilon,s\rangle = \langle \mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)),\varepsilon,s\rangle$
    $\triangleright \langle \mathcal{C}(b):\mathrm{BRANCH}(\mathcal{C}(S):\mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)),\mathrm{NOOP}), \varepsilon,s\rangle$ (Semantik LOOP)
    $\triangleright^* \langle \mathrm{BRANCH}(\mathcal{C}(S):\mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)),\mathrm{NOOP}), \mathbf{B}[\![b]\!](s)=tt,s\rangle$ (Korrektheit Exp.)
    $\triangleright \langle \mathcal{C}(S):\mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)), \varepsilon,s\rangle$ (Semantik BRANCH)
    $\triangleright^* \langle \mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)), \varepsilon,s''\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$ (IVor) ; 2. Fall einfach.

Korrektheit der Statements II

- z.Z.: Wenn $\langle \mathcal{C}(S),\varepsilon,s\rangle \triangleright^* \langle\varepsilon,e,s'\rangle$, so $\langle S,s\rangle \rightarrow s'$ und $e=\varepsilon$
- Induktion über Länge der Maschinenbefehlssequenz:
  - Sei $\langle \mathcal{C}(x:=E),\varepsilon,s\rangle = \langle \mathcal{C}(E):\mathrm{STORE}\ x,\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$
    $\langle \mathcal{C}(E),\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \mathcal{A}[\![E]\!](s),s\rangle$ (Nach IVor, Korrektheit Exp.)
    $\rightarrow s' = s[x \rightarrow \mathcal{A}[\![E]\!](s)]$ (Semantik STORE), also $\langle x:=E,s\rangle \rightarrow s'$.
  - Skip: einfach
  - Sei $\langle \mathcal{C}(S1;S2), \varepsilon,s\rangle = \langle \mathcal{C}(S1):\mathcal{C}(S2),\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$. Zerlegen Abarbeitungsfolge:
    $\langle \mathcal{C}(S1):\mathcal{C}(S2),\varepsilon,s\rangle \triangleright^* \langle \mathcal{C}(S2),e,s''\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$. Nach IVor: $e=\varepsilon$, $\langle S1,s\rangle \rightarrow s''$ und (wg. IVor) $\langle S2,s''\rangle \rightarrow s'$. Also $\langle S1;S2,s\rangle \rightarrow s'$.
  - Sei $\langle \mathcal{C}(\text{if } b \text{ then } S1 \text{ else } S2 \text{ end}),\varepsilon,s\rangle = \langle \mathcal{C}(b):\mathrm{BRANCH}(\mathcal{C}(S1),\mathcal{C}(S2)),\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$.
    Nach IVor, Korrektheit Exp.: $\langle \mathcal{C}(b),\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \mathcal{B}[\![b]\!](s),s\rangle$
    - 1. Fall: $\mathcal{B}[\![b]\!](s)(b) = tt$.
      Dann $\langle \mathrm{BRANCH}(\mathcal{C}(S1),\mathcal{C}(S2)), \mathcal{B}[\![b]\!](s),s\rangle \triangleright \langle \mathcal{C}(S1), \varepsilon,s\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$.
      Nach IVor $\langle S1,s\rangle \rightarrow s'$, also $\langle \text{if } b \text{ then } S1 \text{ else } S2 \text{ end},s\rangle \rightarrow s'$. 2. Fall analog.
  - Sei $\langle \mathcal{C}(\text{while } b \text{ do } S \text{ end}),\varepsilon,s\rangle \triangleright \langle \mathcal{C}(b):\mathrm{BRANCH}(\mathcal{C}(S):\mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)),\mathrm{NOOP}),\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$. Nach IVor, Korrektheit Exp.: $\langle \mathcal{C}(b),\varepsilon,s\rangle \triangleright^* \langle\varepsilon, \mathcal{B}[\![b]\!](s),s\rangle$.
    - 1. Fall: $\mathcal{B}[\![E]\!](s)(b) = tt$, also $\langle \mathcal{C}(S):\mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)),\varepsilon,s\rangle \triangleright^*$
      . $\langle \mathrm{LOOP}(\mathcal{C}(b),\mathcal{C}(S)),e,s''\rangle \triangleright^* \langle\varepsilon, \varepsilon,s'\rangle$
      Nach IVor: $e=\varepsilon$ und $\langle S,s\rangle \rightarrow s''$ und $\langle \text{while } b \text{ do } S \text{ end},s''\rangle \rightarrow s'$.
      Also $\langle \text{while } b \text{ do } S \text{ end},s\rangle \rightarrow s'$. 2. Fall einfach.

Zusammenfassung Kapitel 2

- Formale operationelle Semantik abstrahiert so weit wie möglich von technischen Rahmenbedingungen (Wertkodierung, Speicheradressen, ...) → verbleibende Flexibilität bei Implementierung
- Formale operationelle Semantik ermöglicht Argumentation über
  - Terminierung
  - semantische Äquivalenz
  - Determiniertheit
  - Korrektheit einer Übersetzung

Kapitel 3

Denotationelle Semantik

Idee

- Hatten aus operationeller Semantik gewonnen:
  - $\mathcal{S}$: Command → (State↪State)
  - Zur Bestimmung von S: Folgerungsbaum bzw. Abarbeitungssequenzen

- Ziel nun: Definition von $\mathcal{S}$ ohne Umwege, induktiv über Struktur eines Commands.

- Insbesondere: $\mathcal{S}$(C) soll definierbar sein allein aus den Werten von $\mathcal{S}$ für die unmittelbaren Teilstrukturen von C

„*Kompositionalität*“

3.1 Direct-Style Semantik für W

- $\mathcal{S}_{ds}$ ⟦x:=E⟧ (s) = s[x → $\mathcal{A}$ ⟦E⟧ (s)]

- $\mathcal{S}_{ds}$ ⟦skip⟧ = id

- $\mathcal{S}_{ds}$ ⟦S1;S2⟧ = $\mathcal{S}_{ds}$ ⟦S2⟧ o $\mathcal{S}_{ds}$ ⟦S1⟧

- $\mathcal{S}_{ds}$ ⟦if b then S1 else S2 end ⟧ =
  cond($\mathcal{B}$ ⟦b⟧ ,$\mathcal{S}_{ds}$ ⟦S1⟧ ,$\mathcal{S}_{ds}$ ⟦S2⟧)

- $\mathcal{S}_{ds}$ ⟦while b do S end ⟧ = FIX(F)
  mit F(g) = cond($\mathcal{B}$ ⟦b⟧ ,g o $\mathcal{S}_{ds}$ ⟦S⟧ , id)

Bedingte Anweisungen

- cond: (State $\rightarrow$ {tt,ff}) x (State $\hookrightarrow$ State) x (State $\hookrightarrow$ State)
  $\rightarrow$ (State $\hookrightarrow$ State)

- $$\text{cond}(p,g1,g2)\ (s) = \begin{cases} g1(s), \text{ falls } p(s) = tt \\ g2(s), \text{ falls } p(s) = ff \end{cases}$$

Bleibt: Semantik der While-Anweisung

- $\mathcal{S}_{ds}$ $[\![$while b do S end $]\!]$ = FIX(F)

  mit F(g) = cond($\mathcal{B}$ $[\![b]\!]$,g o $\mathcal{S}_{ds}$ $[\![S]\!]$, id)

- FIX(F) („Fixpunkt von F“) ist Lösung der Gleichung X = F(X).
  FIX: ((State↪State)→(State↪State)) →(State↪State)

- Wieso?
  - Sinnvoll: $\mathcal{S}_{ds}$ $[\![$while b do S end $]\!]$
    = $\mathcal{S}_{ds}$ $[\![$if b then S; while b do S end else skip end$]\!]$
    = cond($\mathcal{B}$ $[\![b]\!]$, $\mathcal{S}_{ds}$ $[\![$while b do S end $]\!]$ o $\mathcal{S}_{ds}$ $[\![S]\!]$, id)

  also: $\mathcal{S}_{ds}$ $[\![$while b do S end $]\!]$
  ist (eine!) Lösung von X = F(X)

Beispiel für Fixpunkt

- Betrachten S = while x<>0 do skip end
- $F_S(g)(s) = \begin{cases} g(s), \text{ falls } s(x) \neq 0 \\ s, \text{ falls } s(x) = 0 \end{cases}$
- g1 mit $g1(s) = \begin{cases} \text{undef}, s(x) \neq 0 \\ s, s(x)=0 \end{cases}$ ist Fixpunkt:

Falls s(x) = 0, so F(g1)(s) = s = g1(s)

Falls s(x) ≠ 0, so F(g1)(s) = g1(s) (= undef)

- g2 mit g2(s) = undef (für alle s) ist kein Fixpunkt.

Für s mit s(x) = 0: F(g2)(s) = s ≠ g2(s) = undef

Probleme:

1. Es gibt Funktionale, die keinen Fixpunkt haben, z.B.

$$F(g) = \begin{cases} g1, \text{ falls } g = g2 \\ g2, \text{ falls } g \neq g2 \end{cases}$$

Lösung: Werden zeigen, dass die in der Semantik von W verwendeten Funktionale Fixpunkte besitzen

2. Es gibt Funktionale, die mehr als einen Fixpunkt besitzen, z.B. ist jede Funktion g^* mit $g^*(s) = s$, falls $s(x) = 0$, ein Fixpunkt des Funktionals der vorigen Folie

Lösung: Werden Bedingungen erarbeiten, die von genau einem Fixpunkt eines verwendeten Funktionals erfüllt sind.

Welcher Fixpunkt

- Betrachten while b do S end, gestartet in s0.

- Fall A: terminiert
- Fall B: terminiert nicht, weil ein Untercommand nicht terminiert
- Fall C: terminiert nicht, weil Schleife selbst nicht abbricht

- Fall A: Terminiert → ex. s1,...,sn mit
  - $\mathcal{B}$ ⟦b⟧ (s1)= ...= $\mathcal{B}$ ⟦b⟧ (sn-1)= tt, $\mathcal{B}$ ⟦b⟧ (sn)= ff.
  - $\mathcal{S}_{ds}$ ⟦S⟧ (si) = si+1 für i<n
  - Sei g0 Fixpunkt. →(für i<n) g0(si) = F(g0)(si)

    = cond($\mathcal{B}$ ⟦b⟧ (si), g0 o $\mathcal{S}_{ds}$ ⟦S⟧ (si),id)

    = g0 o $\mathcal{S}_{ds}$ ⟦S⟧ (si) = g0(si+1).
  - Für i = n: g0(sn) = F(g0)(sn)

    = cond($\mathcal{B}$ ⟦b⟧ (sn), g0 o $\mathcal{S}_{ds}$ ⟦S⟧ (sn),id)

    = id(sn) = sn Also: g0(s0) = sn. Das ist, was wir wollen.

Welcher Fixpunkt

- Fall B: terminiert nicht, weil ein Untercommand nicht terminiert
- ex. s1,...,sn mit
  - $\mathcal{B}$ ⟦b⟧ (s1)= ...= $\mathcal{B}$ ⟦b⟧ (sn)= tt,
  - $\mathcal{S}_{ds}$ ⟦S⟧ (si) = si+1 für i<n
  - $\mathcal{S}_{ds}$ ⟦S⟧ (sn) = undef

  - Sei g0 Fixpunkt. →(für i<n) g0(si) = g0(si+1) (wie vorher).
  - Für i = n: g0(sn) = F(g0)(sn)

    = cond($\mathcal{B}$ ⟦b⟧ (sn), g0 o $\mathcal{S}_{ds}$ ⟦S⟧ (sn), id)

    = undef Also: g0(s0) = undef. Das ist, was wir wollen.

Welcher Fixpunkt

- Fall C: terminiert nicht, weil Schleife selbst nicht abbricht.
- ex. s1,...,sn,... mit
  - $\mathcal{B}$ ⟦b⟧ (s1)= ...= $\mathcal{B}$ ⟦b⟧ (sn) = ... = tt,
  - $\mathcal{S}_{ds}$ ⟦S⟧ (si) = si+1 für i<n

  - Sei g0 Fixpunkt. →(für alle i) g0(si) = g0(si+1).
  - Aus diesen Gleichungen lässt sich kein Wert für g0(s0) ermitteln!

$$\text{Für F mit F(g)(s)} = \begin{cases} \text{g(s), falls } x \neq 0 \\ \text{s, falls } x = 0 \end{cases}$$

(gebildet für while x<>0 do skip end)

ist jedes g Fixpunkt, das g(s) = s für alle s mit s(x) = 0 hat.

Welcher Fixpunkt: Zusammenfassung

- Fall A und B: Jeder Fixpunkt liefert das gewünschte Ergebnis
- Fall C: Betrachten while x<>0 do skip end
  - vorige Folie: jedes g Fixpunkt, das g(s) = s für alle s mit s(x) = 0 hat.
  - intuitiv: $\mathcal{S}_{ds}$ $[\![$ while x<>0 do skip end $]\!]$ (s0) $= \begin{cases} \text{undef}, s0(x) \neq 0 \\ s0,\ s0(x)=0 \end{cases}$

Dies ist der kleinste Fixpunkt g0 im folgenden Sinn:

Wenn g0(s) = s‘ (also definiert), so für alle anderen Fixpunkte g: g(s) = s‘.

Fixpunkttheorie

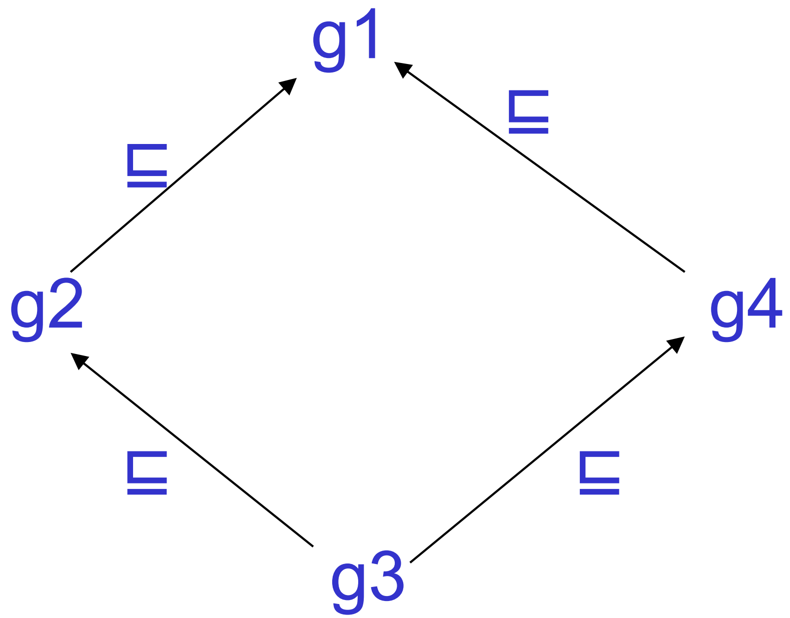
- Mathematik, die sich mit Fixpunkten beschäftigt
- Ziel: Handwerkszeug, um Existenz und Eindeutigkeit der in der Semantik von W vorkommenden Fixpunkte zu zeigen

- Reden über Halbordnungen
- Unsere Halbordnung: $\sqsubseteq$ mit $g1 \sqsubseteq g2$, falls für alle s: $g1(s) = s' \rightarrow g2(s)=s'$
  - ist reflexiv: für alle g: $g \sqsubseteq g$
  - ist transitiv: für alle g1,g2,g3:Wenn $g1 \sqsubseteq g2$ und $g2 \sqsubseteq g3$, so $g1 \sqsubseteq g3$
  - ist antisymmetrisch: Wenn $g1 \sqsubseteq g2$ und $g2 \sqsubseteq g1$, so $g1=g2$

$\sqsubseteq$ entspricht der Relation Inklusion auf der Menge der zu g gehörenden geordneten Paare.

Beispiel

- $g1(s) = s$ für alle s
- $g2(s) = s$, falls $s(x) \geq 0$, undef sonst
- $g3(s) = s$, falls $s(x) = 0$, undef sonst
- $g4(s) = s$, falls $s(x) \leq 0$, undef sonst



Hasse-Diagramm

Begriffe für Halbordnungen

- x heißt Minimum einer Halbordnung, falls für alle y: $x \sqsubseteq y$
- Wenn eine Halbordnung ein Minimum besitzt, ist es eindeutig.
- In der Halbordnung $\sqsubseteq$ auf der Menge (State $\hookrightarrow$ State) ist $\perp$ mit $\perp(s)$ = undef (für alle s) das Minimum
- M heißt Kette, falls für alle $x,y \in M$: $x \sqsubseteq y$ oder $y \sqsubseteq x$
- z.B.: Wenn $gi(s) = \begin{cases} s, \text{ falls } s(x) < i \\ \text{undef, sonst,} \end{cases}$

  so $\{gi \mid i>0\}$ unendliche Kette

Begriffe für Halbordnungen II

- x heißt obere Schranke einer Menge M, falls für alle $y \in M: y \sqsubseteq x$.

- x heißt kleinste obere Schranke von M, falls für alle Schranken x' von M gilt: $x \sqsubseteq x'$.

- Falls M eine kleinste obere Schranke hat, so ist sie eindeutig bestimmt. Schreiben: $\sqcup M$

- $\sqsubseteq$ ist kettenvollständig, d.h., jede Kette M besitzt eine kleinste obere Schranke, nämlich

$$\sqcup M(s) = \begin{cases} s', & \text{ex.} g \in M \text{ mit } g(s) = s' \\ \text{undef}, & \text{sonst} \end{cases}$$

($\leq$ auf N ist nicht kettenvollständig – N hat keine obere Schranke)

Begriffe für Halbordnungen III

- Jede kettenvollständige Halbordnung hat ein kleinstes Element $\perp$, nämlich $\perp = \sqcup\emptyset$ (Jedes Element ist ob. Schranke von $\emptyset$)
- Sei F: (State $\hookrightarrow$ State) → (State $\hookrightarrow$ State)
- F heißt monoton, falls: Wenn $g1 \sqsubseteq g2$, so $F(g1) \sqsubseteq F(g2)$
- Wenn F1, F2 monoton, so auch F1 o F2
- Wenn M Kette und F monoton, so $\{F(x) \mid x \in M\}$ Kette und $\sqcup\{F(x) \mid x \in M\} \sqsubseteq F(\sqcup M)$
- F (monoton) heißt stetig, falls für alle nichtleeren Ketten M: $\sqcup\{F(x) \mid x \in M\} = F(\sqcup M)$ und strikt, falls $\sqcup\emptyset = F(\sqcup\emptyset)$

Ein Fixpunkttheorem

- Sei F eine stetige (also auch monotone) Funktion in einer kettenvollständigen Halbordnung. Dann ist

  FIX F = $\sqcup\{\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots\}$ der kleinste Fixpunkt von F.

Beweis:

- $\{\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots\}$ ist Kette, weil $\perp \sqsubseteq F(\perp)$, also $F^i(\perp) \sqsubseteq F^{i+1}(\perp)$ (Monotonie), also $F^i(\perp) \sqsubseteq F^{i+j}(\perp)$ (Transitivität). Also ist $\sqcup\{\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots\}$ wohldefiniert.
- FIX F ist Fixpunkt, weil $F(\text{FIX } F) = F(\sqcup\{\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots\})$

  $= \sqcup (\{F(\perp), F(F(\perp)), \ldots, F(F^i(\perp)), \ldots\})$ (F stetig)

  $= \sqcup (\{\perp, F(\perp), F(F(\perp)), \ldots, F(F^i(\perp)), \ldots\})$ ($\perp$ ist Minimum)

  = FIX F.
- FIX F ist kleinster Fixpunkt, denn für anderen Fixpunkt d gilt: $\perp \sqsubseteq d$ (Minimum), also $F^n(\perp) \sqsubseteq F^n(d)$ (Monotonie), also $F^n(\perp) \sqsubseteq d$ (d Fixp.), also d obere Schranke von $\{\perp, F(\perp), F(F(\perp)), \ldots, F^i(\perp), \ldots\}$, also FIX $F \sqsubseteq d$.

Schlussfolgerung

- Zur Korrektheit der Semantikdefinition für W reicht es zu zeigen, dass die der Fixpunktbildung zugrundeliegenden Funktionale monoton und stetig sind.

Stetigkeit der Funktionale in der Direct-Style-Semantik

- F in der Def. der Direct-Style-Semantik:
  - $F(g) = \text{cond}(\mathcal{B}\,[\![b]\!], g \circ \mathcal{S}_{ds}\,[\![S]\!], id)$
    $= F1(F2(g))$ mit
    $F1(g) = \text{cond}(\mathcal{B}\,[\![b]\!], g, id)$
    $F2(g) = g \circ \mathcal{S}_{ds}\,[\![S]\!]$

  Zeigen
  1. F1 stetig
  2. F2 stetig
  3. Wenn F1, F2 stetig, so auch F1 o F2

Stetigkeit von F1

Für beliebige g0: State$\hookrightarrow$State und p:State $\rightarrow$\{tt,ff\} ist F1(g) = cond(p,g,g0) stetig.

Beweis:

(1) Zeigen: F1 monoton. Sei $g1 \sqsubseteq g2$, z.Z. $F1(g1) \sqsubseteq F1(g2)$

Sei F1(g1)(s) = s‘.

- Fall: p(s) = tt. Dann F1(g1)(s) = g1(s) = s‘ = g2(s) =F1(g2)(s).
- Fall: p(s) = ff. Dann F1(g1)(s) = g0(s) = F1(g2)(s).

(2) Zeigen: F1 stetig. Sei Y nichtleere Kette.

z.Z. $F1(\sqcup Y) \sqsubseteq \sqcup\{F1(y)|y\in Y\}$ ($\sqsupseteq$ gilt immer)

Sei $F1(\sqcup Y)$ (s) = s‘. Zeigen: es gibt ein $g\in Y$ mit F1(g)(s)=s‘:

- Fall: p(s) = tt, dann $F1(\sqcup Y)$ (s) = $(\sqcup Y)$ (s), also muss es ein $g\in Y$ geben mit g(s) = s‘.
- Fall: p(s) = ff, dann $F1(\sqcup Y)$ (s) = g0(s) = F1(g)(s) für bel.$g\in Y$

Wenn es ein $g\in Y$ mit F1(g)(s)=s‘ gibt, ist aber $\sqcup\{F1(y)|y\in Y\}$(s) = s‘.

Stetigkeit von F2

- Sei g0: State $\hookrightarrow$State und F2(g) = g o g0. Dann ist F2 stetig.

Beweis:

(1) Zeigen: F2 monoton.

Sei g1$\sqsubseteq$g2 und F2(g1)(s) = s‘. Also ex. s“ mit g0(s) = s“, g1(s“)=s‘.

Wegen g1$\sqsubseteq$g2 ist g2(s“)=s‘, also F2(g2)(s) = g2(g0(s)) = s‘.

(2) Zeigen: F2 stetig.

Sei Y nichtleere Kette und F2($\sqcup$Y)(s) = s‘. Also

$\sqcup$Y(g0(s)) = s‘. Also ex. g$\in$Y mit g(g0(s)) = s‘.

Also $\sqcup${F2(y)|y$\in$Y}(s) = s‘.

Stetigkeit der Verkettung

Seien F1, F2 stetig. Dann F1oF2 stetig.

Beweis:

(1) Da F1,F2 monoton, so auch F1oF2.

(2) Zeigen: F1oF2 stetig.

Sei Y nichtleere Kette.

Weil F2 stetig: $F2(\sqcup Y) = \sqcup \{F2(y) | y \in Y\}$

Weil F1 stetig: $F1(\sqcup \{F2(y) | y \in Y\}) = \sqcup \{F1(F2(y)) | y \in Y\}$.

Also $F1oF2(\sqcup Y) = \sqcup \{F1oF2(y) | y \in Y\}$.

Zusammenfassung Direct-Style-Semantik

- kompositional: Semantik eines Konstrukts allein auf der Basis seiner Teilkonstrukte definiert
- Kleinste Fixpunkte werden zur Definition der Semantik von Schleifen verwendet
- Fixpunkttheorie liefert Existenz solcher Fixpunkte

- ohne Beweis: Für alle Commands S von W ist
  $\mathcal{S}_{ds}\ [\![S]\!] = \mathcal{S}_{SOS}\ [\![S]\!]$

3.2 Erweiterung: Prozeduren

- Am Beispiel statischer Scopes

- Verwenden (z.T. erneut) die Konzepte
  - Loc: Menge von Locations (= Menge der ganzen Zahlen)
  - new: Loc→Loc (new(x) := x+1) /* Ordnet einer Location die nächste zu */
  - Store:Loc ∪{next} →Z /* Speicher; next speichert Adresse der 1. bislang unbenutzen Location */
  - Env_V :Var→Loc
  - Env_P: Pnames → (Store ↪ Store)
  - Funktion lookup: Env_V x Store → State; lookup(env,sto)(x) = sto(env(x)) /* Transformiert env + sto in state */

Direct-Style Semantik für W mit Env,Sto

- $\mathcal{S}'_{ds}$ $[\![x:=E]\!]$ $(env_V)(env_P)(sto)$ = $sto[env_V(x) \rightarrow \mathcal{A}$ $[\![E]\!]$ $(lookup\ (env_V,sto))]$

- $\mathcal{S}'_{ds}$ $[\![skip]\!]$ $(env_V)(env_P)$= id

- $\mathcal{S}'_{ds}$ $[\![S1;S2]\!]$ $(env_V)(env_P)$ =
  $(\mathcal{S}'_{ds}$ $[\![S2]\!]$ $(env_V)(env_P))$ o $(\mathcal{S}'_{ds}$ $[\![S1]\!]$ $(env_V)(env_P))$

- $\mathcal{S}'_{ds}$ $[\![$if b then S1 else S2 end $]\!]$ $(env_V)(env_P)$=
  cond($\mathcal{B}$ $[\![b]\!]$ o lookup(env_V,.), $\mathcal{S}'_{ds}$ $[\![S1]\!]$ $(env_V)(env_P)$, $\mathcal{S}'_{ds}$ $[\![S2]\!]$ $(env_V)(env_P)$)

- $\mathcal{S}'_{ds}$ $[\![$while b do S end $]\!]$ $(env_V)(env_P)$= FIX(F) mit
  F(g) = cond($\mathcal{B}$ $[\![b]\!]$ o lookup(env_V,.), g o $\mathcal{S}'_{ds}$ $[\![S]\!]$ $(env_V)(env_P)$, id)

Deklarationen

Effekt einer Variablendeklaration = Änderung von env_V ... $\mathcal{DV}$

Effekt einer Prozedurdeklaration = Änderung von env_P ... $\mathcal{DP}$

$\mathcal{DV}\ [\![e]\!] = id$

$\mathcal{DV}\ [\![var\ x\ ;\ decl]\!]\ (env_V,sto) = \mathcal{DV}\ [\![decl]\!]$
$(env_V[x \rightarrow l],sto(l \rightarrow 0,next \rightarrow new\ l])$ mit $l = sto(next)$

(0 = Initialwert für x)

$\mathcal{DP}\ [\![e]\!]\ (env_V) = id$

$\mathcal{DP}\ [\![proc\ p\ is\ S\ ;\ decl]\!]\ (env_V,env_P) =$

$\mathcal{DP}\ [\![decl]\!]\ (env_V,env_p[p \rightarrow \mathcal{S}'_{ds}\ [\![S]\!]\ (env_V)(env_P)])$

Blöcke, Prozedurrufe

- $\mathcal{S}'_{ds}$ $[\![$begin Dec_V Dec_P S end$]\!]$ $(env_V)(env_P)(sto)$ =

  $\mathcal{S}'_{ds}$ $[\![S]\!]$ $(env'_V)(env'_P)(sto')$ mit

  $\mathcal{DV}$ $[\![Dec_V]\!]$ $(env_V, sto) = (env'_V, sto')$ und

  $\mathcal{DP}$ $[\![Dec_P]\!]$ $(env'_V, env_P) = env'_P$

- $\mathcal{S}'_{ds}$ $[\![$call p$]\!]$ $(env_V)(env_P)$ = $env_P(p)$

3.3 Continuation-Style Semantik

- Am Beispiel einer Erweiterung von W: Exceptions

- command = ...
  | begin S1 catch e: S2 end
  | throw e

Continuation

- beschreibt den Effekt der Ausführung des restlichen Programms

- c: State $\hookrightarrow$ State

- Continuation-style semantics:
- S_{cs}: Commands → (Cont → Cont)
- .... ; S ; ...   (c)   →   .... ; S ; ... (c')

CS-Semantik für W (ohne Exceptions)

- $\mathcal{S}_{cs}$ $[\![x:=E]\!]$ (c)(s) = c (s[x$\rightarrow\mathcal{A}$ $[\![E]\!]$ (s)])
- $\mathcal{S}_{cs}$ $[\![skip]\!]$ = id
- $\mathcal{S}_{cs}$ $[\![S1;S2]\!]$ = $\mathcal{S}_{cs}$ $[\![S1]\!]$ o$\mathcal{S}_{cs}$ $[\![S2]\!]$
- $\mathcal{S}_{cs}$ $[\![\text{if b then S1 else S2 end}]\!]$ (c) =
  cond($\mathcal{B}$ $[\![b]\!]$, $\mathcal{S}_{cs}$ $[\![S1]\!]$ (c), $\mathcal{S}_{cs}$ $[\![S2]\!]$ (c))
- $\mathcal{S}_{cs}$ $[\![\text{while b do S end}]\!]$ = FIX G
  wobei G(g)(c) = cond($\mathcal{B}$ $[\![b]\!]$, $\mathcal{S}_{cs}$ $[\![S]\!]$ (g(c)), c)

Mit den bekannten Techniken kann man Existenz der Fixpunkte und Äquivalenz zu anderen Semantiken nachweisen:
Für alle Commands S und Continuations c gilt:

- $\mathcal{S}_{cs}$ $[\![S]\!]$ (c) = c o $\mathcal{S}_{ds}$ $[\![S]\!]$

Exceptions

- Brauchen Information, welcher Handler (catch-Block) zur welcher Exception gehört:

- Env_E : Exception $\rightarrow$ Cont

- S_{cs}‘ : (Command x Env_E) $\rightarrow$ (Cont$\rightarrow$Cont)

CS-Semantik für W (mit Exceptions)

- $\mathcal{S}_{cs}' \llbracket x{:=}E \rrbracket (env)(c)(s) = c\,(s[x \rightarrow \mathcal{A}/\mathcal{B} \llbracket E \rrbracket (s)])$
- $\mathcal{S}_{cs}' \llbracket skip \rrbracket (env) = id$
- $\mathcal{S}_{cs}' \llbracket S1;S2 \rrbracket (env) = \mathcal{S}_{cs}' \llbracket S1 \rrbracket (env) \circ \mathcal{S}_{cs}' \llbracket S2 \rrbracket (env)$
- $\mathcal{S}_{cs}' \llbracket \text{if } b \text{ then } S1 \text{ else } S2 \text{ end} \rrbracket (env)(c) =$
  $cond(\mathcal{B} \llbracket b \rrbracket, \mathcal{S}_{cs}' \llbracket S1 \rrbracket (env)(c), \mathcal{S}_{cs}' \llbracket S1 \rrbracket (env)(c))$
- $\mathcal{S}_{cs}' \llbracket \text{while } b \text{ do } S \text{ end} \rrbracket (env) = FIX\ G$
  wobei $G(g)(c) = cond(\mathcal{B} \llbracket b \rrbracket, \mathcal{S}_{cs}' \llbracket S \rrbracket (env)(g(c)), c)$
- $\mathcal{S}_{cs}' \llbracket \text{begin } S1 \text{ catch } e\text{: } S2 \text{ end} \rrbracket (env)(c) =$
  $\mathcal{S}_{cs}' \llbracket S1 \rrbracket (env[e \rightarrow \mathcal{S}_{cs}' \llbracket S2 \rrbracket (env)(c)])$
- $\mathcal{S}_{cs}' \llbracket \text{throw } E \rrbracket (env)(c) = env(e)$

Mit den bekannten Techniken kann man Existenz der Fixpunkte und Äquivalenz zu anderen Semantiken nachweisen

3.3 Statische Programmanalyse

- Ziel: Information über die Semantik eines Programms ohne Ausführung
- Anwendung
  - Programmcodeoptimierung
  - Programmverifikation
- Rahmenbedingungen
  - muss immer (und relativ schnell) terminieren (auch bei nicht terminierenden Programmen)
  - darf unscharfe („weiss nicht“), aber nie falsche Ergebnisse liefern

Beispiele

- Expressions
  - Konstantenpropagation (Konstante Teilausdrücke gleich im Compiler ausrechnen)
  - Vorzeichenanalyse (vereinfacht z.B. Typkonvertierung, Ausnahmebehandlung bei Division durch 0)
  - Feldgrenzenüberwachung
- Daten
  - Pointeranalyse (may point to/must point to; Vereinfachung von Zugriffen)
  - Referenzanalyse (Wieviele Pointer zeigen auf mich?; Garbage collection
- Kontrollfluss
  - Very busy expressions (Werte, die in jedem Kontrollzweig noch einmal verwendet werden)
  - Tote Zweige

.... und viele andere mehr

Anwendungsbeispiel

- Übersetzung eines ALGOL (Pascal)-Arrays in ein C-Array

- Algol (Pascal):
  - A : array [0:n,0:m] of integer
  - Zugriff: A[i,j]

- C:
  - int * A
  - Zugriff: * (A + i * (m+1) + j)

Brute-Force-Übersetzung ALGOL-C

ALGOL:

```
i := 0;
while i <= n do
    j := 0;
    while j <= m do
            A[i,j] := B[i,j] + C[i,j];
            j := j + 1;
    end
    i := i + 1;
end
```

C:

```
i = 0;
while(i <=n) {
    j = 0;
    while(j <=m) {
            tmp = A + i * (m+1) + j;
            *tmp = *(B + i * (m+1)+j)
                  + *(C + i*(m+1)+j);
            j = j + 1;
    }
    i = i + 1;
}
```

Analyse: Available Expressions
Ziel: Common Subexpression elimination

C alt:

```
i = 0;
while(i <=n) {
    j = 0;
    while(j <=m) {
        tmp = A + i * (m+1) + j;
        *tmp = *(B + i * (m+1)+j)
            + *(C + i*(m+1)+j);
        j = j + 1;
    }
    i = i + 1;
}
```

erste Berechnung

Folgeberechnung

C neu:

```
i = 0;
while(i <=n) {
    j = 0;
    while(j <=m) {
        t1 = i * (m+1)+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
}
```

Analyse: Schleifeninvarianten
Ziel: Verschiebung von Code aus Schleife

C alt:

```
i = 0;
while(i <=n) {
    j = 0;
    while(j <=m) {
          t1 = i * (m+1)+j;
          tmp = A + t1;
          *tmp = *(B + t1)
                + *(C + t1);
          j = j + 1;
    }
    i = i + 1;
}
```

Invariant!

C neu:

```
i = 0;
while(i <=n) {
    j = 0;
    t2 = i * (m+1);
    while(j <=m) {
          t1 = t2+j;
          tmp = A + t1;
          *tmp = *(B + t1)
                + *(C + t1);
          j = j + 1;
    }
    i = i + 1;
}
```

Analyse: Induktive Variablen
Ziel: Komplexitätsreduktion

C alt:

Induktion

```
i = 0;
while(i <=n) {
    j = 0;
    t2 = i * (m+1);
    while(j <=m) {
        t1 = t2+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
}
```

C neu:

```
i = 0;
t3 = 0;
while(i <=n) {
    j = 0;
    t2 = t3;
    while(j <=m) {
        t1 = t2+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
    t3 = t3 + m + 1;
}
```

Analyse: Equivalent expressions
Ziel: Copy propagation

C alt:

```
i = 0;
t3 = 0;
while(i <=n) {
    j = 0;
    t2 = t3;
    while(j <=m) {
        t1 = t2+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
    t3 = t3 + m + 1;
}
```

äquivalent

C neu:

```
i = 0;
t3 = 0;
while(i <=n) {
    j = 0;
    t2 = t3;
    while(j <=m) {
        t1 = t3+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
    t3 = t3 + m + 1;
}
```

Analyse: Live variables
Ziel: dead code elimination

C alt:

```
i = 0;
t3 = 0;
while(i <=n) {
    j = 0;
    t2 = t3;
    while(j <=m) {
        t1 = t3+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
    t3 = t3 + m + 1;
}
```

not live

C neu:

```
i = 0;
t3 = 0;
while(i <=n) {
    j = 0;
    while(j <=m) {
        t1 = t3+j;
        tmp = A + t1;
        *tmp = *(B + t1)
            + *(C + t1);
        j = j + 1;
    }
    i = i + 1;
    t3 = t3 + m + 1;
}
```

Fazit des Beispiels

- Optimierungspotential kann auch ohne „Verschulden“ eines Programmierers entstehen

  weitere Beispiele:
  - Feldgrenzenüberwachung
  - Garbage collection
  - Virtuelle/Abstrakte Methoden
  - ...

- Optimierung setzt Analyse voraus
  - korrekt (was behauptet wird, muss stimmen)
  - nicht notwendig exakt (manches stimmt, wird aber nicht behauptet) – Preis: weniger Optimierung
  - effizient

Eine Analyse im Detail: Abhängigkeitsanalyse

- Einige Variablen werden als „in“, andere als „out“ deklariert.
- Frage: Hängt der Wert der „out“-Variablen nach Ausführung des Programms funktional von den Werten den „in“-Variablen vor Ausführung ab?

- y = x*z + 25, $y \in out$
  - ok, falls $x,z \in in$
  - D? sonst (dubios = möglicherweise nicht ok)

- y := 1; while x<>1 do y:= y*x;x:=x-1, $x \in in$, $y \in out$
  - ok
  - ohne Initialisierung von y: D?, weil $y \notin in$ und Endwert von y vom Anfangswert von y abhängt

Lösung: Nichtstandardsemantik

- Zustand s → Abstrakter Zustand p; pro Variable (statt Wert)Eigenschaft mit Werten ...

... z.B. 0,1,2,3, viele, irgendein (Referenzanalyse)

... z.B. <0,<=0,=0,>=0,>0,beliebig (Vorzeichenanalyse)

... z.B. 0,1,2,3,...,variabel (Konstantenpropagation)

... z.B. OK,D? (Abhängigkeitsanalyse)

Allen Wertebereichen gemeinsam: bilden kettenvollständige Halbordnungen

Grund: Wollen/Müssen Fixpunkte ausrechnen

Rechnen mit OK und D?

- $OK \sqsubseteq D?$

- $(x \sqcup y$ meint $\sqcup\{x,y\})$
- $OK \sqcup OK = OK$
- $OK \sqcup D? = D? \sqcup OK = D? \sqcup D? = D?$

Abstrakter Zustand

- p : (V + {control})→ {OK,D?}
- control Bestandteil eines abstrakten Zustandes z.B. wegen
- if x = 1 then y := 1 else y := 2
- p(x) = D?, alle anderen ok
- Kontrollfluss hängt möglicherweise nicht von „in“ ab, also auch Wert von y möglicherweise nicht.

Halbordnung auf abstrakten Zuständen

- Haben: Halbordnung $\sqsubseteq$ auf {OK,D?}

- Def.: Halbordnung auf abstrakten Zuständen:
  - $p \sqsubseteq p'$ falls für alle $x \in V+\{control\}$: $p(x) \sqsubseteq p'(x)$

  - Ist Halbordnung
  - ist kettenvollständig und $\sqcup M$ ist derjenige abstrakte Zustand mit $\sqcup M\ (x) = \sqcup\{p(x) \mid p \in M\}$

Beweis

- z.Z.: $\sqsubseteq$ ist kettenvollständig und $\sqcup M$ ist derjenige abstrakte Zustand mit $\sqcup M\ (x) = \sqcup\{p(x) \mid p \in M\}$

(Wohldefiniertheit) Sei M Kette.

Nach Def. $\sqsubseteq$ ist für jedes x $\{p(x) \mid p \in M\}$ Kette.

Da $\sqsubseteq$ auf {OK,D?} kettenvollständig, ex. $\sqcup\{p(x) \mid p \in M\}$.

(Obere Schranke) Sei $p \in M$.

Weil $\sqcup$ kleinste obere Schranke von $\{p(x) \mid p \in M\}$,

ist $p(x) \sqsubseteq \sqcup\{p(x) \mid p \in M\}$ für alle x.

(Kleinste obere Schranke). Sei p^* obere Schranke von M.

Zeigen: $\sqcup M \sqsubseteq p^*$. Weil p^* obere Schranke von M, ist für alle $p \in M$: $p \sqsubseteq p^*$. Also für alle x: $p(x) \sqsubseteq p^*(x)$. Also $\sqcup\{p(x) \mid p \in M\} \sqsubseteq p^*(x)$.

Also: $\sqcup M \sqsubseteq p^*$.

Die Analyse: Ausdrücke

- $\mathcal{PA}$: Arithm.Expressions →(Abstr.State →{OK,D?})

- $\mathcal{PA}$ ⟦n⟧ (p) = OK, falls p(control) = OK, sonst D?
- $\mathcal{PA}$ ⟦x⟧ (p) = p(x), falls p(control) = OK, sonst D?
- $\mathcal{PA}$ ⟦E1+E2⟧ (p) = $\mathcal{PA}$ ⟦E1⟧ (p) ⊔ $\mathcal{PA}$ ⟦E2⟧ (p) (-*/ analog)

- $\mathcal{PB}$: Bool.Expressions →(Abstr.State →{OK,D?})

- analog

Die Analyse: Commands

- $\mathcal{PS}$: Commands →(Abstr.State →Abstr.State)

- $\mathcal{PS}$ ⟦x:=E⟧ (p) = p[x→$\mathcal{PA}$/$\mathcal{PB}$(E)(p)]
- $\mathcal{PS}$ ⟦skip⟧ = id
- $\mathcal{PS}$ ⟦S1;S2⟧ = $\mathcal{PS}$ ⟦S2⟧ o $\mathcal{PS}$ ⟦S1⟧
- $\mathcal{PS}$ ⟦if b then S1 else S2 end⟧ = cnd($\mathcal{PB}$(b), $\mathcal{PS}$ ⟦S1⟧ ,$\mathcal{PS}$ ⟦S2⟧)
  - cnd(f,h1,h2) (p) = h1(p)⊔h2(p), falls f(p) = OK, sonst LOST.
  - LOST(p)(x) = D? für alle x (einschließlich control)
- $\mathcal{PS}$ ⟦while b do S end⟧ = FIX H mit

  H(g) = cnd($\mathcal{PB}$(b), go$\mathcal{PS}$ ⟦S⟧ ,id)

Beispiele

- y := x
  – p(x) = OK, p(y) = D?, p(control) = OK
  → $\mathcal{PS}$ ⟦y:=x⟧ (p)(x) = $\mathcal{PS}$ ⟦y:=x⟧ (p)(y) =
  $\mathcal{PS}$ ⟦y:=x⟧ (p)(control) = OK
  - p(x) = D?, p(y) = OK, p(control) = OK
  → $\mathcal{PS}$ ⟦y:=x⟧ (p)(x) = $\mathcal{PS}$ ⟦y:=x⟧ (p)(y) = D?,
  $\mathcal{PS}$ ⟦y:=x⟧ (p)(control) = OK

- if x=x then z:= y else y := z
  - p(x) = p(y) = OK, p(z) = D?
  → $\mathcal{PS}$ ⟦ if x=x then z:= y else y := z ⟧ (p)(z) = D?
  - p(x) = D?, p(y) = p(z) = OK
  → $\mathcal{PS}$ ⟦ if x=x then z:= y else y := z ⟧ (p)(x,y,z,control) = D?

Beispiele

- y := 1; while x <> 1 do y:= y*x;x:=x-1 end
  - p(x) = OK, p(y) = D?, p(control) = OK

    ... müssen Fixpunkt berechnen:

    FIX H mit

    H(g) = cnd($\mathcal{PB}$ ⟦x<>1⟧ , go$\mathcal{PS}$ ⟦ y:= y*x;x:=x-1 ⟧ ,id),

    also H(g)(p) = LOST, falls p(control) = D? oder p(x) = D?, sonst g(p) ⊔ p

    1. ⊥(p) = INIT für alle p (INIT ordnet allen Variablen OK zu)
    2. H(⊥)(p) = LOST, falls p(control) = D? oder p(x) = D?, sonst p
    3. H(H(⊥))(p) = LOST, falls p(control) = D? oder p(x) = D?, sonst p

    → FIX H = H(⊥)

  → $\mathcal{PS}$ ⟦y := 1; while x <> 1 do y:= y*x;x:=x-1 end ⟧ (p)(x)

  = $\mathcal{PS}$ ⟦y:=x⟧ (p)(x,y,control) = OK

Aussagen

1. Diese Analyse ist wohldefiniert
   - Technik: Fixpunkttheorie

2. Diese Analyse ist sicher: Wenn die Analyse für eine Variable OK ergibt, ist diese funktional abhängig von den Inputs.
   - Def.: $s \equiv_p s'$, falls: Wenn p(control) und p(x) = ok, so s(x) = s'(x)
   - Satz: Für alle S: Wenn $s \equiv_p s'$, so kreist S sowohl in s als auch s', oder $S_{ds}[\![s]\!] \equiv_{PS[\![S]\!]} S_{ds}[\![s']\!]$

Terminierung der Analyse

- 1. Aussage (leicht zu sehen, aber schlecht geschätzt):

Fixpunkt für while b do S end ist

$\sqcup\{\perp, F(\perp), F(F(\perp)), \ldots, F^{(2^{m+1})^{2^{(m+1)}}}(\perp)\} = F^{(2^{m+1})^{2^{(m+1)}}}(\perp)$

wobei m die Zahl der in b und S vorkommenden Variablen ist

Gründe:
- nicht vorkommende Variablen irrelevant
- für vorkommende Variablen + control: 2^{m+1} Zustände
- für vorkommende Variablen + control: $(2^{m+1})^{2^{(m+1)}}$ Funktionen

2. Aussage (besser geschätzt, aber mit detaillierter Analyse):
$(m+1)^2$ Iterationen reichen.

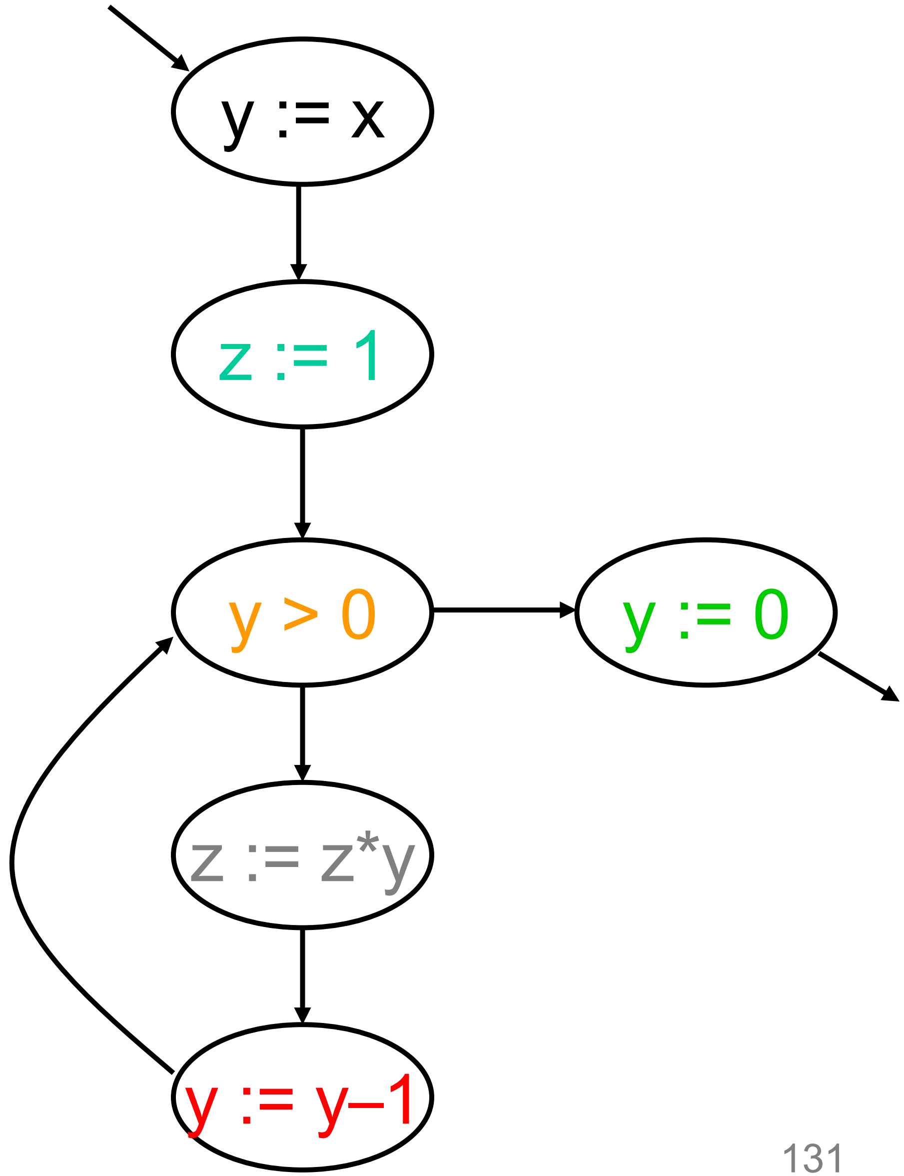
Andere Analysen

- Nutzen: Flussgraph
  - Knoten für Zuweisungen und Tests
  - Kanten x→y für „nach x kann möglicherweise y ausgeführt werden

- ohne semantikfreie (d.h. rein struktursichernde Elemente

Flussgraph: Beispiel

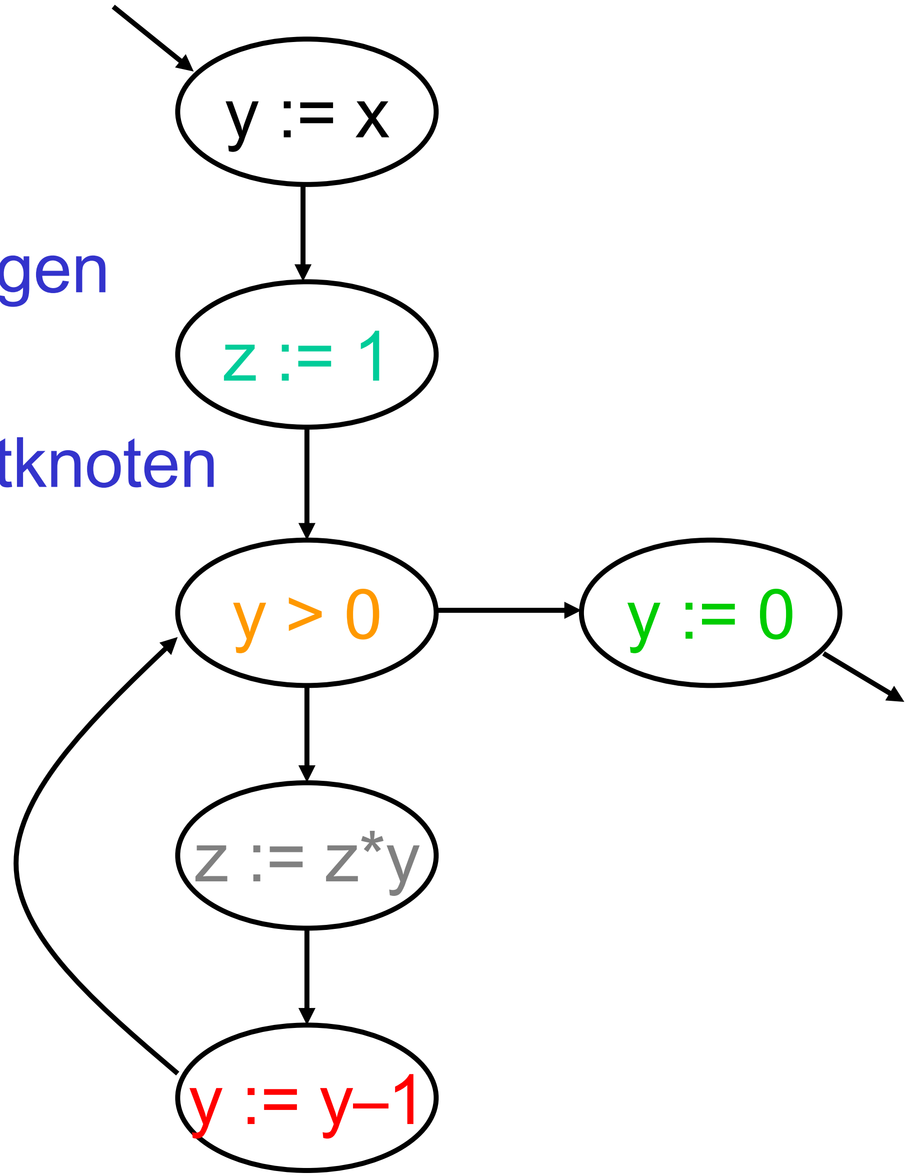
```
y := x;
z := 1;
while y > 0 do
    z := z * y;
    y := y – 1;
end
y := 0;
```



Allgemein für Sprache W

- Für Statement S Knotenmengen
  - nodes(S) – alle Knoten
  - init(S) – ein Element, Startknoten
  - final(S) – Endknoten

- Kantenmenge
  - flow(S) – alle Kanten



Induktive Definition

- für S = x:= E
  - k := new node(„x := E“)
  - nodes(S)=init(S)=final(S) = {k}
  - flow(S) = Ø
- für S = skip analog
- für S = S1;S2
  - nodes(S) = nodes(S1) ∪ nodes(S2)
  - init(S) = init(S1)
  - final(S) = final(S2)
  - flow(S) = flow(S1) ∪ flow(S2) ∪ final(S1) x init(S2)
- für S = if b then S1 else S2 end
  - k := new node(„b“)
  - nodes(S) = nodes(S1) ∪ nodes(S2) ∪ {k}
  - init(S) = {k}
  - final(S) = final(S1) ∪ final(S2)
  - flow(S) = flow(S1) ∪ flow(S2) ∪ {k} x (init(S1) ∪ init(S2))
- für S = while b do S1 end
  - k = new node(„b“)
  - nodes(S) = nodes(S1) ∪ {k}
  - init(S) = final(S) = {k}
  - flow(S) = flow(S1) ∪ {k} x init(S1) ∪ final(S1) x {k}

Analyse 1: Available Expressions

- Zu einem Knoten des Flussgraphen, bestimme alle Expressions, die auf allen Pfaden
  - bereits berechnet sind
  - seitdem nicht modifiziert wurden

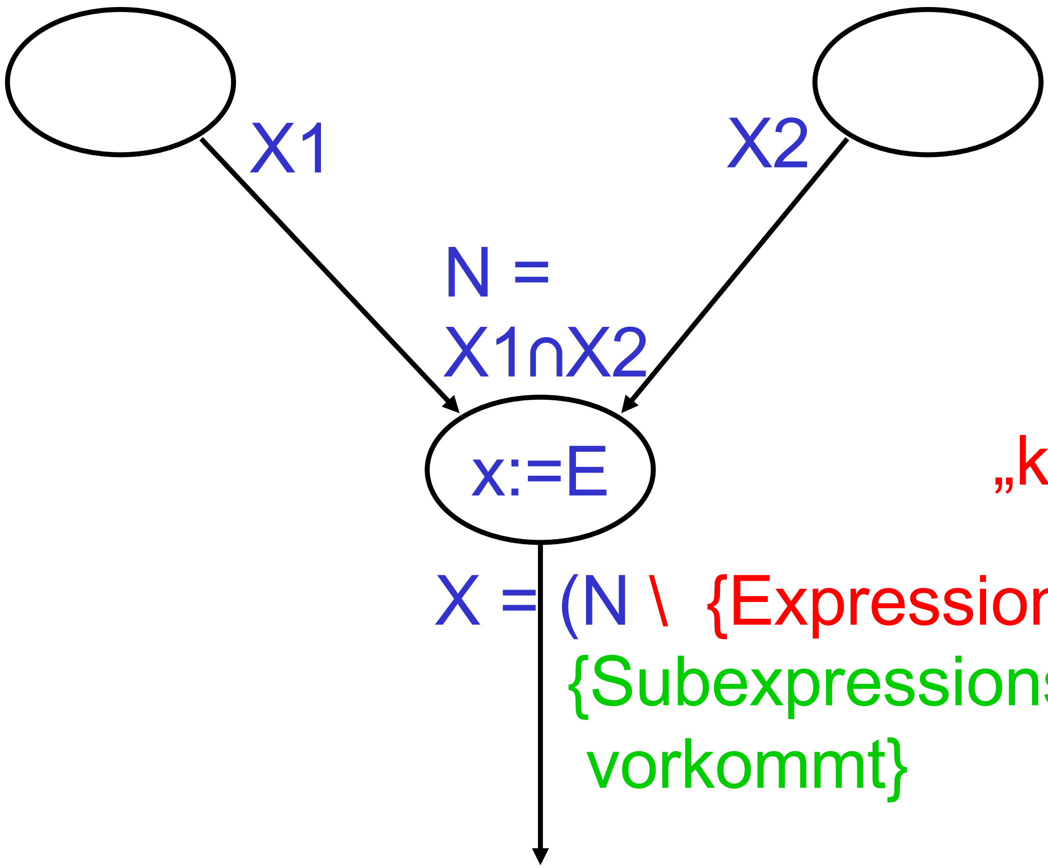
- Beispiel

x := a+b; y := a*b; while y > a+b do a := a +1;x := a+b; end

kann ggf umgeformt werden zu

x := a+b; y := a*b; while y > x do a := a +1;x := a+b; end

Available Expressions: Idee



„kill“

$X = (N \setminus$ {Expressions in denen x vorkommt}$) \cup$ {Subexpressions von E, in denen kein x vorkommt}

„Gen“

Available Expressions: Ausführung

$\mathrm{kill}(x := E) = \{ E' \in EXP \mid x \sqsubseteq E' \}$
$\mathrm{kill}(\mathrm{skip}) = \emptyset$
$\mathrm{kill}(b) = \emptyset$

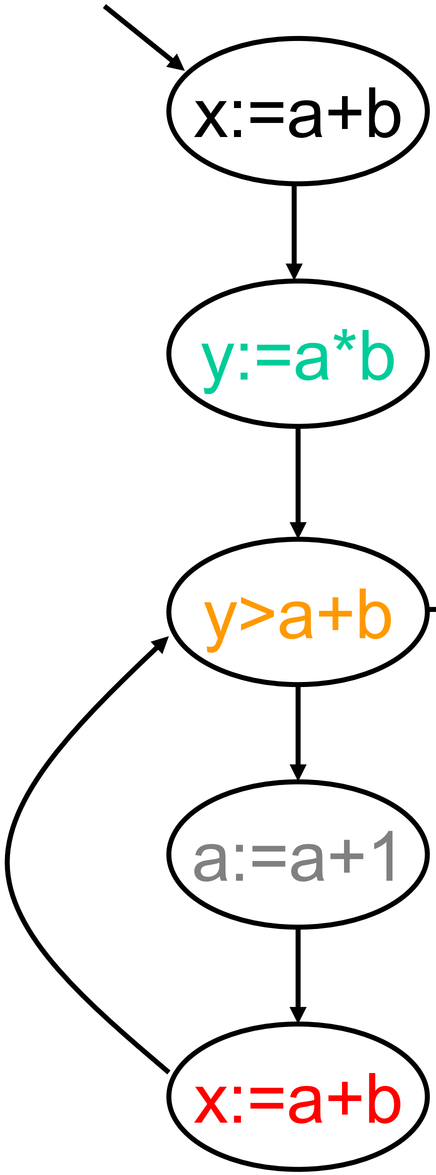
$\mathrm{gen}(x := E) = \{E' \mid E' \sqsubseteq E, !x \sqsubseteq E'\}$
$\mathrm{gen}(\mathrm{skip}) = \emptyset$
$\mathrm{gen}(b) = \{E' \mid E' \sqsubseteq b\}$

Bestimmung der Available Expressions mittels Lösung von Datenflussgleichungen

- Pro Knoten k Variable k_{entry}, k_{exit}
- Gleichungen
  - $k_{entry} = \emptyset$ für $k \in \mathrm{init}(S)$
  - $k_{entry} = \bigcap_{(k',k) \in \mathrm{flow}(S)} k'_{exit}$ für $k \notin \mathrm{init}(S)$
  - $k_{exit} = (k_{entry} \setminus \mathrm{kill}(k)) \cup \mathrm{gen}(k)$

Beispiel: kill, gen

x := a+b; y := a*b; while y > a+b do a := a +1;x := a+b; end



kill: Ø gen: {a+b}

kill: Ø gen: {a*b}

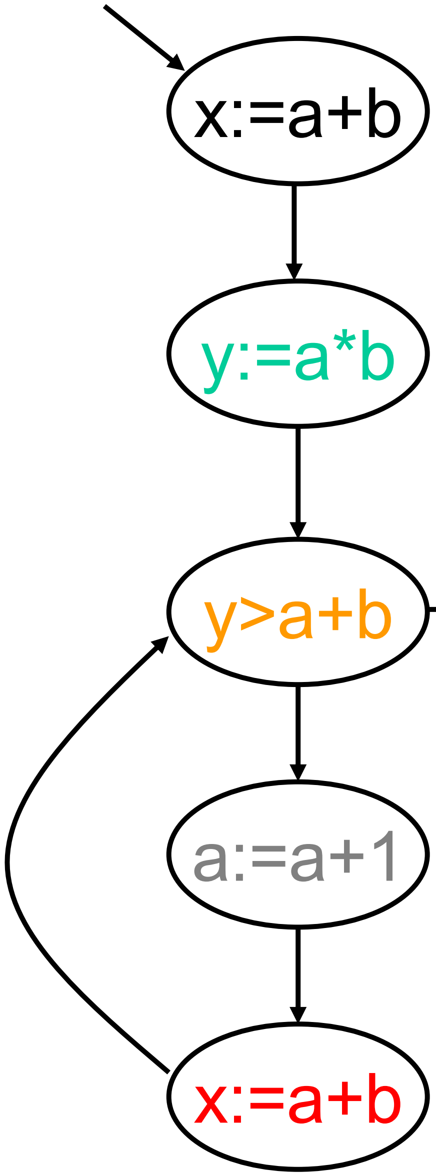
kill: Ø gen: {a+b}

kill: {a+b,a*b,a+1} gen: Ø

kill: Ø gen: {a+b}

Beispiel: Gleichungen

x := a+b; y := a*b; while y > a+b do a := a +1;x := a+b; end



entry = Ø — exit = entry∪{a+b}

entry = exit — exit = entry ∪{a*b}

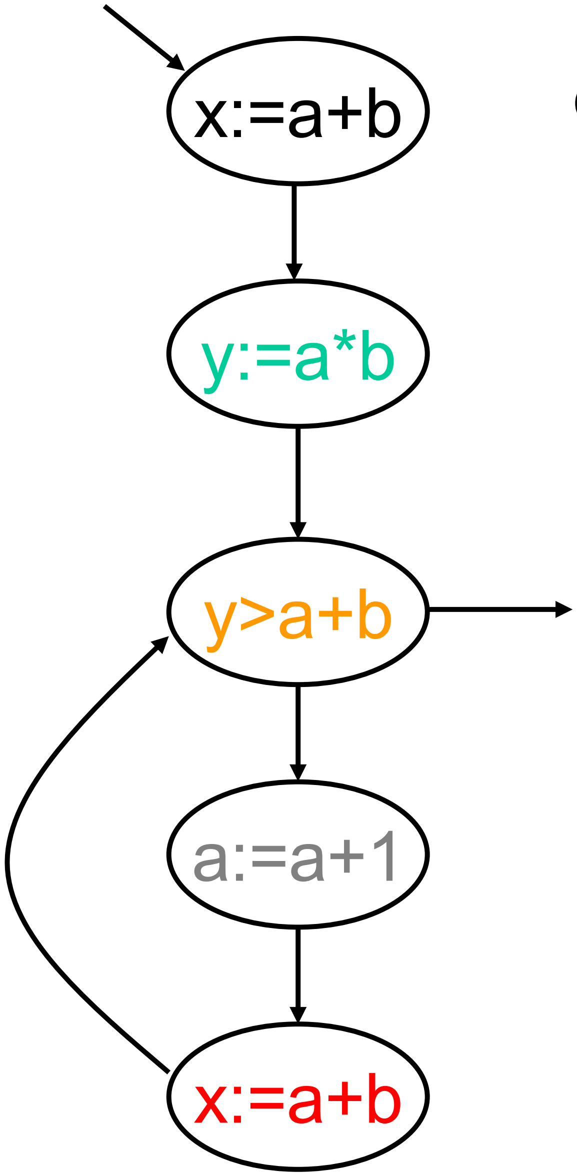
entry = exit ∩ exit — exit = entry ∪{a+b}

entry = exit — exit = entry \ {a+b,a*b,a+1}

entry = exit — exit = entry ∪ {a+b}

Beispiel:
Lösung = größte Lösung der Gleichungen

x := a+b; y := a*b; while y > a+b do a := a +1;x := a+b; end



entry = ∅ exit = {a+b}

entry = {a+b} exit = {a+b, a*b}

entry = {a+b} exit = {a+b}

entry = {a+b} exit = ∅

entry = ∅ exit = {a+b}

Bestimmung der größten Lösung

- Setze alle Variablen auf Menge aller Epressions
- REPEAT
  - für alle Gleichungen parallel
    - linke Seite := rechte Seite
  - UNTIL nothing changes

Beispiel: Start

entry = Ø
a+b,a*b,a+1

exit = entry∪{a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry ∪{a*b}
a+b,a*b,a+1

entry = exit ∩ exit
a+b,a*b,a+1

exit = entry ∪{a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry \ {a+b,a*b,a+1}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry ∪ {a+b}
a+b,a*b,a+1

Beispiel: 1. Iteration

entry = Ø
~~a+b,a*b,a+1~~

exit = entry∪{a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry ∪{a*b}
a+b,a*b,a+1

entry = exit ∩ exit
a+b,a*b,a+1

exit = entry ∪{a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry \ {a+b,a*b,a+1}
~~a+b,a*b,a+1~~

entry = exit
a+b,a*b,a+1

exit = entry ∪ {a+b}
a+b,a*b,a+1

Beispiel: 2. Iteration

entry = ∅

exit = entry ∪ {a+b}
a+b, ~~a*b,a+1~~

entry = exit
a+b,a*b,a+1

exit = entry ∪ {a*b}
a+b,a*b,a+1

entry = exit ∩ exit
a+b,a*b,a+1

exit = entry ∪ {a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry \ {a+b,a*b,a+1}

entry = exit
~~a+b,a*b,a+1~~

exit = entry ∪ {a+b}
a+b,a*b,a+1

Beispiel: 3. Iteration

entry = ∅

exit = entry∪{a+b}
a+b

entry = exit
a+b,~~a*b,a+1~~

exit = entry ∪{a*b}
a+b,a*b,a+1

entry = exit ∩ exit
a+b,a*b,a+1

exit = entry ∪{a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry \ {a+b,a*b,a+1}

entry = exit

exit = entry ∪ {a+b}
a+b,~~a*b,a+1~~

Beispiel: 4. Iteration

entry = ∅

exit = entry∪{a+b}
a+b

entry = exit
a+b

exit = entry ∪{a*b}
a+b,a*b,~~a+1~~

entry = exit ∩ exit
a+b,~~a*b,a+1~~

exit = entry ∪{a+b}
a+b,a*b,a+1

entry = exit
a+b,a*b,a+1

exit = entry \ {a+b,a*b,a+1}

entry = exit

exit = entry ∪ {a+b}
a+b

Beispiel: 5. Iteration

entry = ∅

exit = entry∪{a+b}
a+b

entry = exit
a+b

exit = entry ∪{a*b}
a+b,a*b

entry = exit ∩ exit
a+b

exit = entry ∪{a+b}
a+b,~~a*b,a+1~~

entry = exit
a+b,a*b,a+1

exit = entry \ {a+b,a*b,a+1}

entry = exit

exit = entry ∪ {a+b}
a+b

Beispiel: 6. Iteration

entry = ∅

exit = entry∪{a+b}
a+b

entry = exit
a+b

exit = entry ∪{a*b}
a+b,a*b

entry = exit ∩ exit
a+b

exit = entry ∪{a+b}
a+b

entry = exit
a+b,~~a*b,a+1~~

exit = entry \ {a+b,a*b,a+1}

entry = exit

exit = entry ∪ {a+b}
a+b

Beispiel: 7. Iteration = no change

entry = $\emptyset$

exit = entry $\cup$ {a+b}
a+b

entry = exit
a+b

exit = entry $\cup$ {a*b}
a+b,a*b

entry = exit $\cap$ exit
a+b

exit = entry $\cup$ {a+b}
a+b

entry = exit
a+b

exit = entry \ {a+b,a*b,a+1}

entry = exit

exit = entry $\cup$ {a+b}
a+b

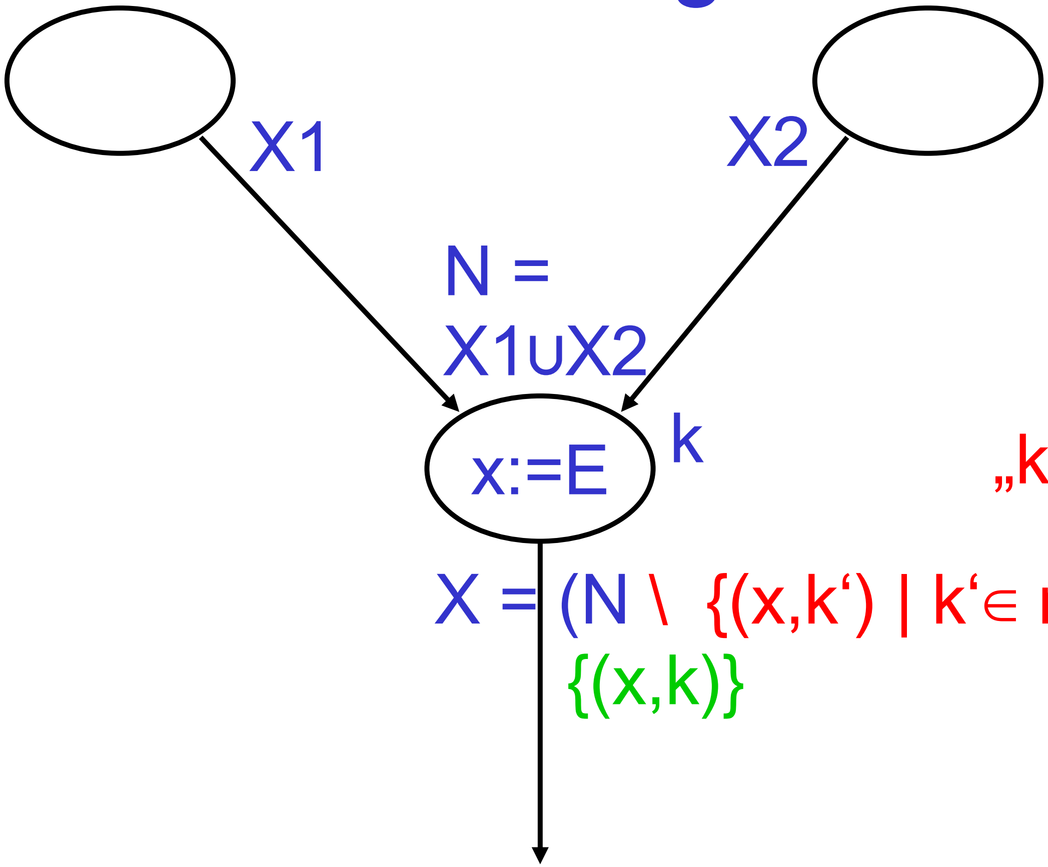
Analyse: Reaching definitions

- Für einen Knoten k: Welche Zuweisungen sind auf mind. einem Pfad zu k noch nicht überschrieben?

- Beispiel

x := 5; y := 1; while x > 1 do y := x*y; x := x -1 end

Reaching definitions: Idee



$$X = (N \setminus \{(x,k') \mid k' \in nodes(S) \cup \{?\}\}) \cup \{(x,k)\}$$

„kill“

„Gen“

Reaching definitions: Ausführung

kill(x := E) = { (x,k) | (k$\in$ nodes(S)$\cup${?}}
kill(skip) = Ø
kill(b) = Ø

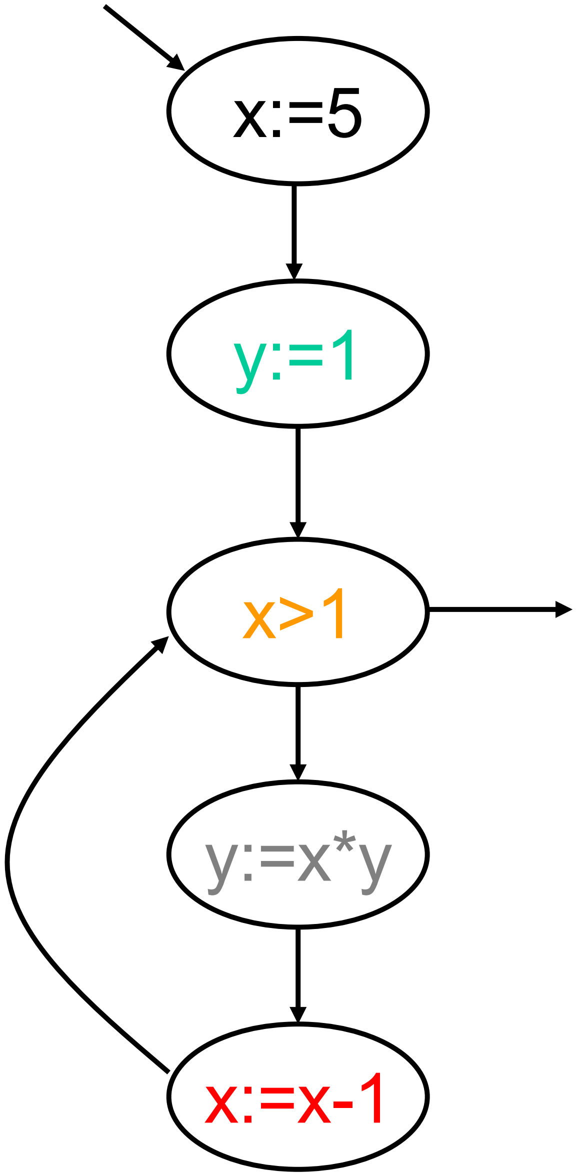
gen(x := E) = {(x,this)}
gen(skip) = Ø
gen(b) = Ø

Bestimmung der reaching definitions mittels Lösung von Datenflussgleichungen

- Pro Knoten k Variable k_{entry}, k_{exit}
- Gleichungen
  - $k_{entry} = \{(x,?) \mid x \in VAR\}$ für $k \in init(S)$
  - $k_{entry} = \bigcup_{(k',k) \in flow(S)} k'_{exit}$ für $k \notin init(S)$
  - $k_{exit} = (k_{entry} \setminus kill(k)) \cup gen(k)$

Beispiel: kill, gen

x := 5; y := 1; while x >1 do y := x*y;x := x-1; end



kill: {(x,?),(x,o),(x,o)} gen: {(x,o)}

kill: {(y,?),(y,o),(y,o)} gen: {(y,o)}

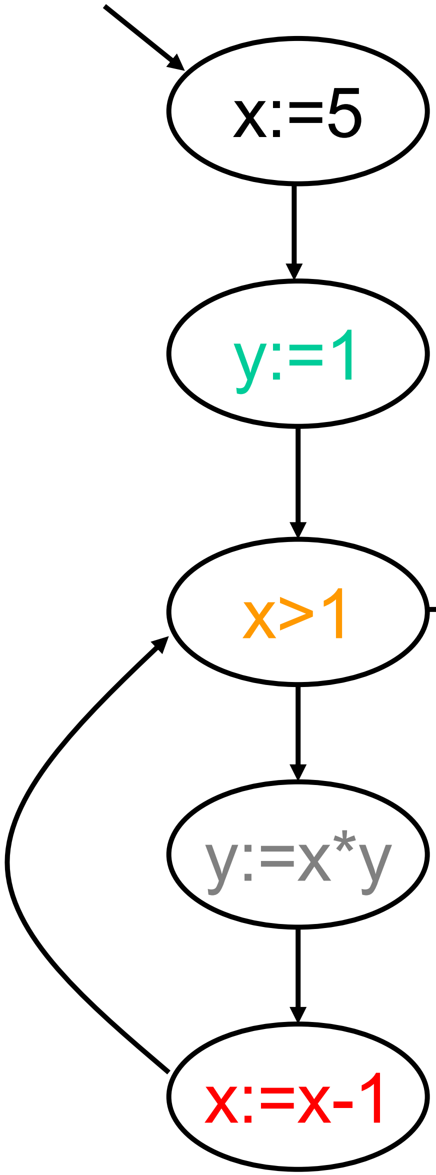
kill: Ø gen: Ø

kill: {(y,?),(y,o),(y,o)} gen: {(y,o)}

kill: {(x,?),(x,o),(x,o)} gen: {(x,o)}

Beispiel: Gleichungen

x := 5; y := 1; while x >1 do yx := x*y;x := x-1; end



entry = {(x,?),(y,?)}
exit = entry \ {(x,?),(x,o),(x,o)} ∪ {(x,o)}

entry = exit
exit = entry \ {(y,?),(y,o),(y,o)} ∪ {(y,o)}

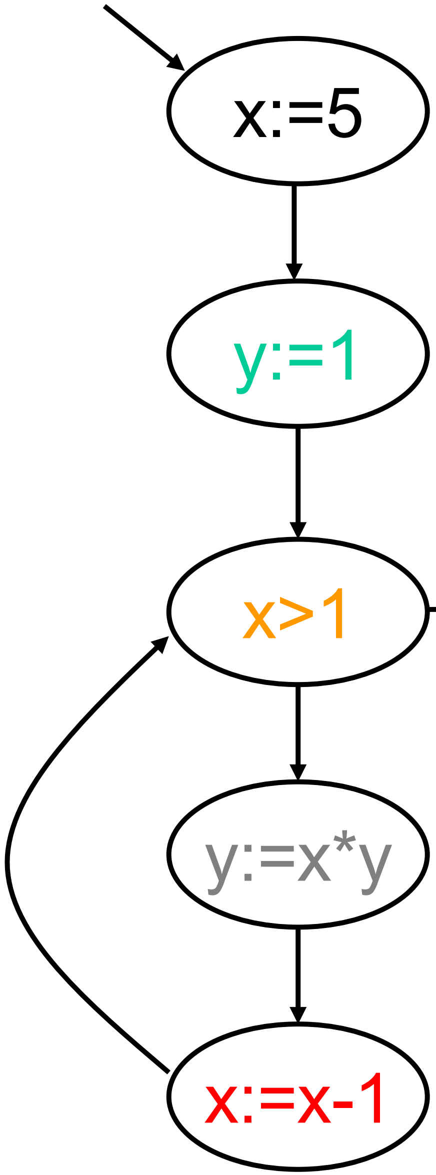
entry = exit ∪ exit
exit = entry

entry = exit
exit = entry \ {(y,?),(y,o),(y,o)} ∪ {(y,o)}

entry = exit
exit = entry \ {(x,?),(x,o),(x,o)} ∪ {(x,o)}

Beispiel: Lösung = kleinste Lösung

x := 5; y := 1; while x >1 do yx := x*y;x := x-1; end



entry = {(x,?),(y,?)
exit = {(y,?),(x,o)}

entry = {(y,?),(x,o)}
exit = { (x,o),(y,o)}

entry = {(x,o),(x,o),(y,o),(y,o)}
exit = {(x,o),(x,o),(y,o),(y,o)}

entry = {(x,o),(x,o),(y,o),(y,o)}
exit = {(x,o),(x,o),(y,o)}

entry = {(x,o),(x,o),(y,o)}
exit = {(x,o),(y,o)}

Analyse: Very busy expressions

- Expression ist very busy an einem Knoten, falls ihr Wert auf jedem Kontrollpfad noch einmal benutzt wird (ohne dass vorkommende Variablen ihren Wert ändern)

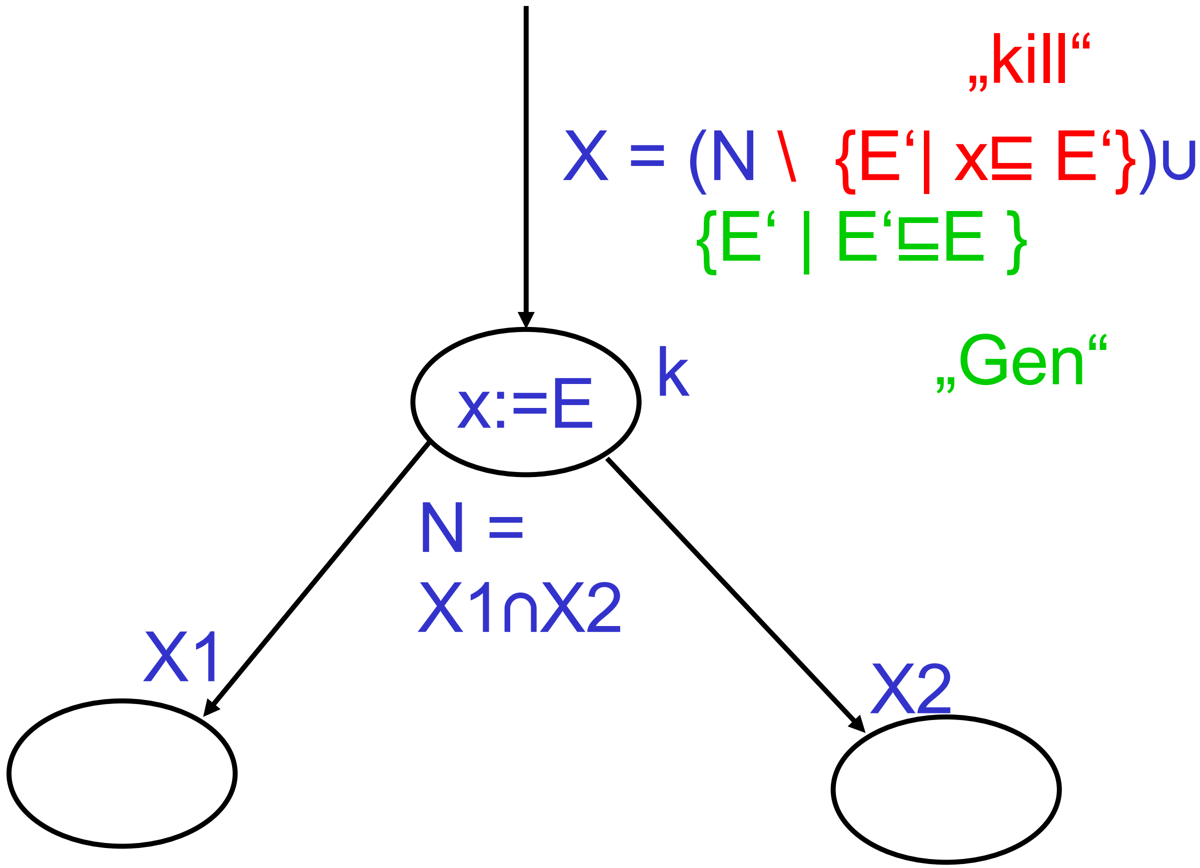
Ziel: Very busy expressions können gleich berechnet werden

Beispiel:

very busy: b-a, a-b

if a>b then x:= b-a ;y := a-b else y:= b-a ;x:= a-b end

Very busy expressions: Idee



Very busy expressions: Ausführung

$\text{kill}(x := E) = \{ E' \mid (x \sqsubseteq E'\}$
$\text{kill}(\text{skip}) = \emptyset$
$\text{kill}(b) = \emptyset$

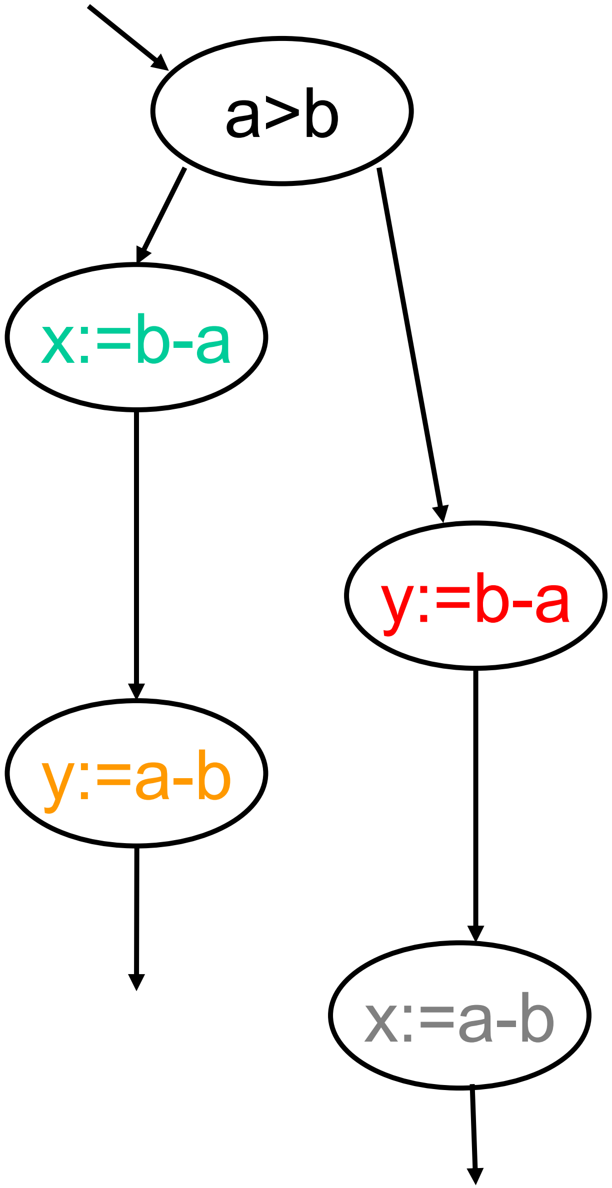
$\text{gen}(x := E) = \{ E' \mid E' \sqsubseteq E\}$
$\text{gen}(\text{skip}) = \emptyset$
$\text{gen}(b) = \{ E' \mid E' \sqsubseteq b\}$

Bestimmung der Very busy Expressions mittels Lösung von Datenflussgleichungen

- $k_{exit} = \emptyset$ für $k \in \text{final}(S)$
- $k_{exit} = \bigcap_{(k,k') \in \text{flow}(S)} k'_{entry}$ für $k \notin \text{final}(S)$
- $k_{entry} = (k_{exit} \setminus \text{kill}(k)) \cup \text{gen}(k)$

Beispiel: kill, gen

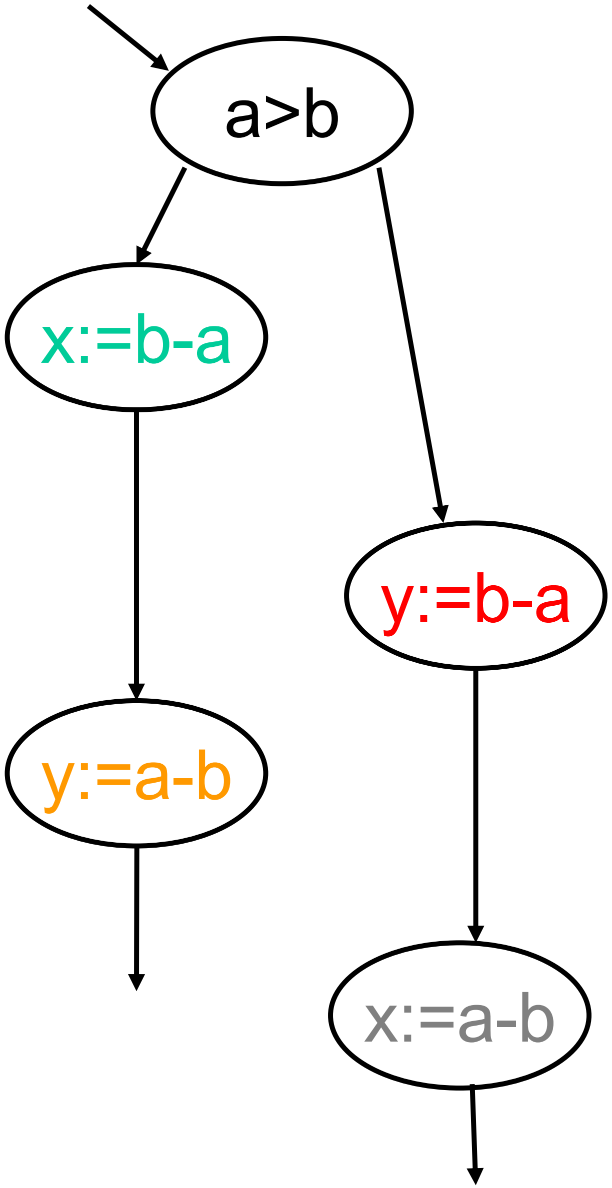
if a>b then x:= b-a ;y := a-b else y:= b-a ;x:= a-b end



kill: Ø	gen: Ø
kill: Ø	gen: {b-a}
kill: Ø	gen: {b-a}
kill: Ø	gen: {a-b}
kill: Ø	gen: {a-b}

Beispiel: Gleichungen

if a>b then x:= b-a ;y := a-b else y:= b-a ;x:= a-b end



entry = exit
exit = entry ∩ entry

entry = exit ∪ {b-a}
exit = entry

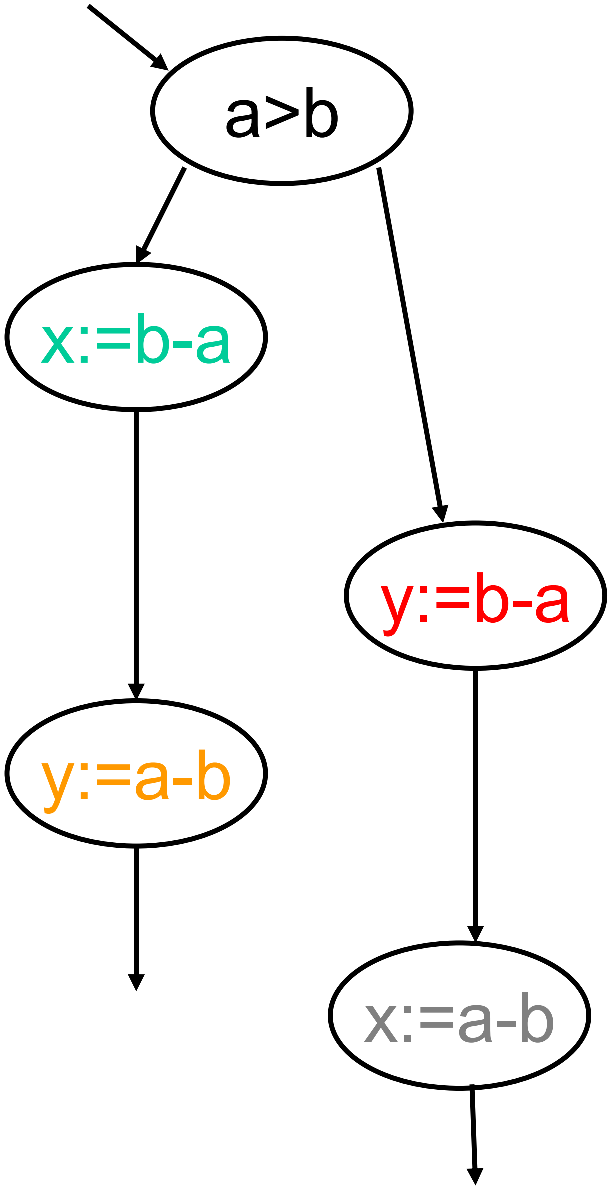
entry = exit ∪ {b-a}
exit = entry

entry = exit ∪ {a-b}
exit = Ø

entry = exit ∪ {a-b}
exit = Ø

Beispiel: Lösung = größter Fixpunkt

if a>b then x:= b-a ;y := a-b else y:= b-a ;x:= a-b end



entry = {a-b,b-a}
exit = {a-b,b-a}

entry = {a-b,b-a}
exit = {a-b}

entry = {a-b, b-a}
exit = {a-b}

entry = {a-b}
exit = Ø

entry = {a-b}
exit = Ø

Analyse: Live variables

- Variable ist live an einem Knoten, falls ihr Wert auf mindestens einem Kontrollpfad noch einmal benutzt wird (ohne dass ihr Wert überschrieben wurde)

Ziel: Zuweisungen an Variablen, die nicht live sind, können gestrichen werden

Beispiel:

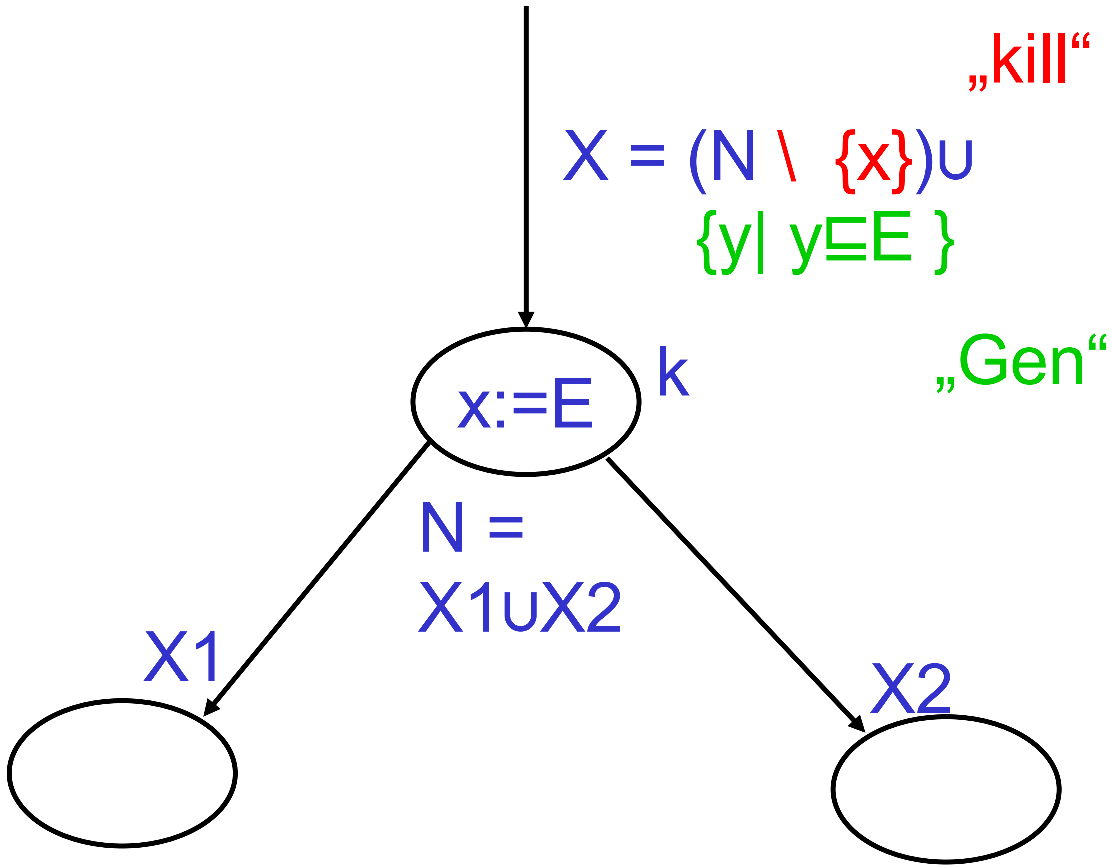
↓

x:=2;y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;

erlaubt Transformation zu:

y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;

Live variables: Idee



Live variables: Ausführung

$\text{kill}(x := E) = \{x\}$
$\text{kill}(\text{skip}) = \emptyset$
$\text{kill}(b) = \emptyset$

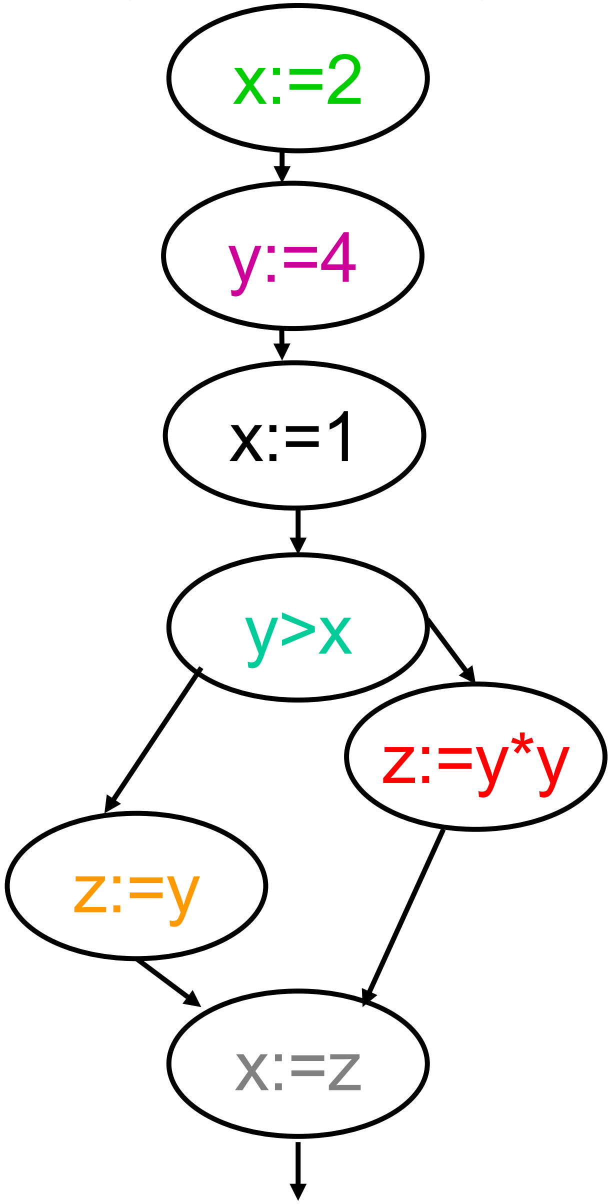
$\text{gen}(x := E) = \{ y \mid y \sqsubseteq E\}$
$\text{gen}(\text{skip}) = \emptyset$
$\text{gen}(b) = \{ y \mid y \sqsubseteq b\}$

Bestimmung der Live variables mittels Lösung von Datenflussgleichungen

- $k_{exit} = \emptyset$ für $k \in \text{final}(S)$
- $k_{exit} = \bigcup_{(k,k') \in \text{flow}(S)} k'_{entry}$ für $k \notin \text{final}(S)$
- $k_{entry} = (k_{exit} \setminus \text{kill}(k)) \cup \text{gen}(k)$

Beispiel: kill, gen

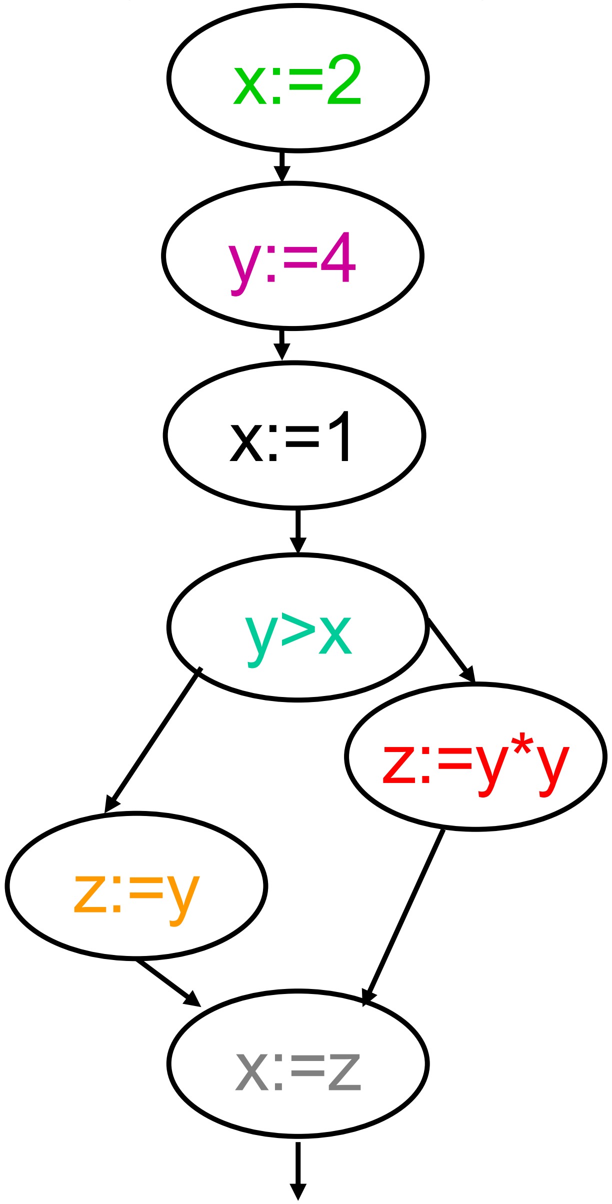
x:=2;y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;



kill: {x}	gen: Ø
kill: {y}	gen: Ø
kill: {x}	gen: Ø
kill: Ø	gen: {x,y}
kill: {z}	gen: {y}
kill: {z}	gen: {y}
kill: {x}	gen: {z}

Beispiel: Gleichungen

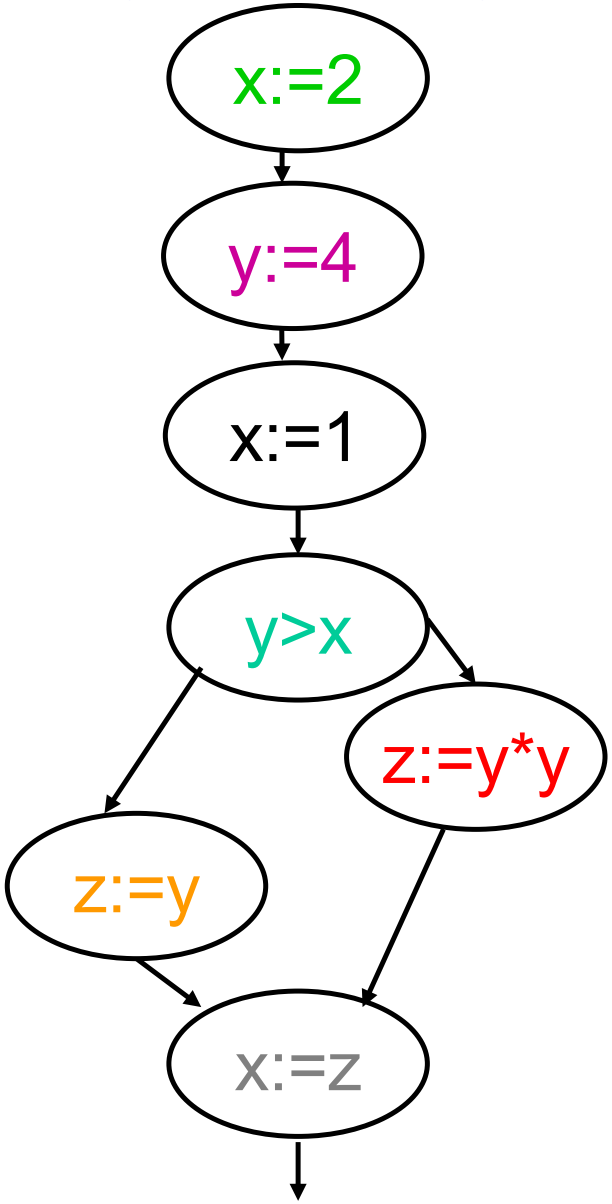
x:=2;y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;



entry = exit\{x}
exit = entry
entry = exit\{y}
exit = entry
entry = exit\{x}
exit = entry
entry = exit ∪{x,y}
exit = entry ∪ entry
entry = exit\{z}∪{y}
exit = entry
entry = exit\{z}∪{y}
exit = entry
entry = exit\{x}∪{z}
exit = Ø

Beispiel: Lösung (kleinste)

x:=2;y:=4;x:=1;if y>x then z:=y else z:=y*y end; x := z;



entry = Ø
exit = Ø
entry = Ø
exit = {y}
entry = {y}
exit = {x,y}
entry = {x,y}
exit = {y}
entry = {y}
exit = {z}
entry = {y}
exit = {z}
entry = {z}
exit = Ø

Fazit bis hier

May-Analyse (∪) (auf mind. einem Pfad...)	Must-Analyse (∩) (auf allen Pfaden ...)	
z.B reaching definitions	z.B. available expressions	Vorwärtsanalyse
z.B. Live variables	z.B. very busy expressions	Rückwärtsanalyse
kleinste Fixpunkte	größte Fixpunkte	

Konstantenpropagation

- Für einen Knoten k und eine Variable x soll festgestellt werden,

  ob x in k stets den gleichen Wert liefert (wenn ja welchen)

Ziel: Ersetzung der Variable durch Konstante

Beispiel:

x := 6; y := 3; while x > y do x := x-1;z := y*y end

erlaubt Transformation

x := 6; y := 3; while x > 3 do x := x-1;z := 9 end

Konstantenpropagation

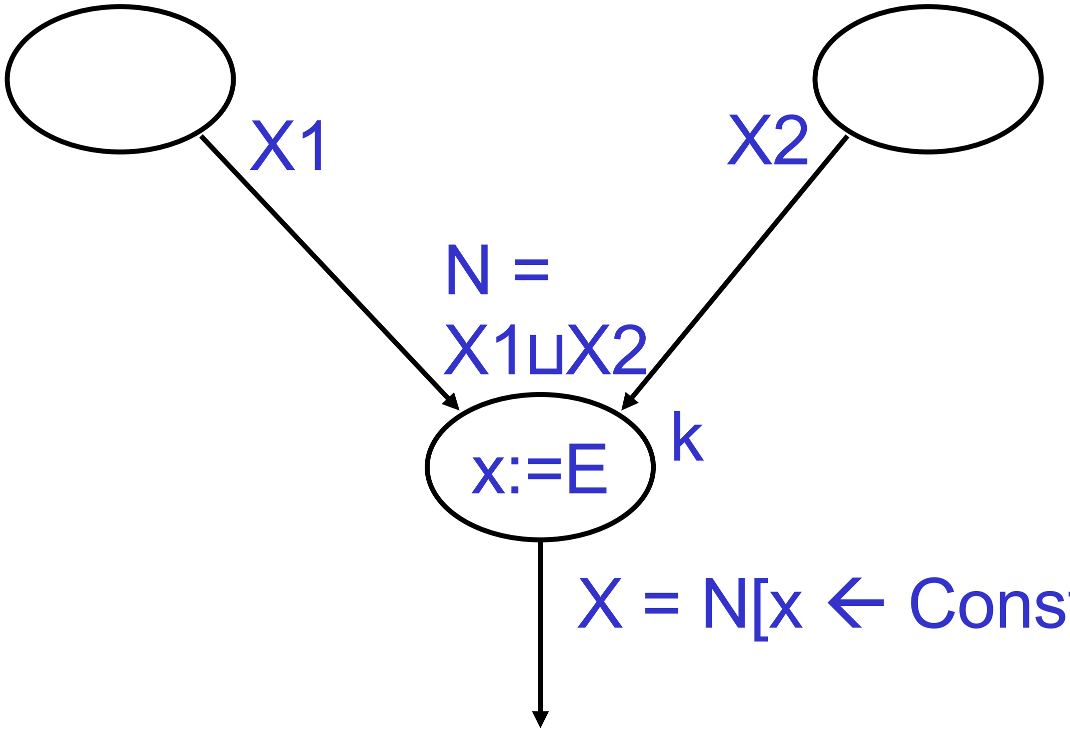
- Idee: Für jede Variable x, für jeden Punkt im Kontrollfluss
  - $\perp$ ... keine Information über x verfügbar
  - $n \in Z$ x hat hier immer Wert n
  - $\top$ ... x könnte hier verschiedene Werte haben

$\perp \sqsubseteq n$, $n \sqsubseteq \top$ $n \neq n' \Rightarrow n \not\sqsubseteq n'$ $x \sqcup y$ = kleinste obere Schranke von x,y

- Zustand: s: VAR $\rightarrow Z \cup \{\top, \perp\}$
- $s \sqcup s'$ definiert durch $s \sqcup s'(x) = s(x) \sqcup s'(x)$ für alle $x \in$ VAR

Konstantenpropagation

... ist Vorwärtsanalyse



X = N[x ← Const(E,N)], falls N≠⊥, sonst ⊥

Const(x,N) = N(x)
Const(n,N) = n
Const(E1 op E2,N) = Const(E1,N) op‘ Const(E2,N)
op‘ wie op, nur ⊤ op‘ x = x op‘ ⊤ = ⊤

Kapitel 4

Axiomatische Semantik

Idee

Statt:

- Wie kommt der Effekt eines Programms zustande (operationell)
- Was ist der Effekt eines Programms? (denotationell)

Nun:

- Was kann ich über den Effekt eines Programms aussagen?

Mittel: Hoare-Tripel

$\{P\}\ S\ \{Q\}$

Precondition Postcondition

bedeutet:

Wenn S terminiert und vor Abarbeitung von S die Aussage P gilt, so gilt nach Abarbeitung von S die Aussage Q

partielle Korrektheit

$\{P\}\ S\ \{\Downarrow Q\}$

bedeutet:

Wenn vor Abarbeitung von S die Aussage P gilt, dann terminiert S und nach seiner Abarbeitung gilt die Aussage Q

totale Korrektheit

Pre- und Postconditions

- P,Q sind Prädikate: Ordnen Variablen(belegung) einen Wahrheitswert zu
- Beispiele: x < y, x > 0, usw.
- Variable: zwei Sorten
  - Programmvariablen: Wert ändert sich durch die Programmausführung
  - logische Variablen: Wert ändert sich nicht durch die Programmausführung („symbolische Konstanten“)
- Bsp: {x = n} y:=1;while x<>1 do y:=x*y;x:=x-1 end {y = n!}

4.1 Die axiomatische Semantik von W (partielle Korrektheit)

- $[\text{ass}]_{par}$: $\{P[x \rightarrow \mathcal{A}\,[\![E]\!]\,]\}\ x := E\ \{P\}$
- $[\text{skip}]_{par}$: $\{P\}\ \text{skip}\ \{P\}$
- $[\text{comp}]_{par}$: $\dfrac{\{P\}\ S1\ \{Q\}\ ,\ \{Q\}\ S2\ \{R\}}{\{P\}\ S1\ ;\ S2\ \{R\}}$
- $[\text{if}]_{par}$: $\dfrac{\{P \wedge \mathcal{B}\,[\![b]\!]\}\ S1\ \{Q\}\ ,\ \{P \wedge \neg\mathcal{B}\,[\![b]\!]\}\ S2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}\ \{Q\}}$
- $[\text{while}]_{par}$: $\dfrac{\{P \wedge \mathcal{B}\,[\![b]\!]\}\ S\ \{P\}}{\{P\}\ \text{while } b \text{ do } S \text{ end}\ \{P \wedge \neg\mathcal{B}\,[\![b]\!]\}}$
- $[\text{cons}]_{par}$: $\dfrac{\{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q'\}}$ falls $P' \Rightarrow P \wedge Q \Rightarrow Q'$

Weiteres Vorgehen

- Anwendung in der Programmverifikation, Erweiterungen
- Theorie: Widerspruchsfreiheit und Vollständigkeit

Beispiel: Fakultätsberechnung

Def Fakultät (induktiv): $0! = 1$; $(n+1)! = (n+1)\ n!$

Ziel: {x=n} y:=1; while x <> 1 do y:=y*x;x:=x-1 end $\{n>0 \wedge y=n!\}$

Beweis:

I := „ $x>0 \Rightarrow (y*x! = n! \wedge n \geq x)$ “ Es folgt:

wegen $[ass]_{par}$: $\{I[x \rightarrow x-1]\}$ x:=x-1 {I} und

$\{I[y \rightarrow y*x, x \rightarrow x-1]\}$ y:=y*x $\{I[x \rightarrow x-1]\}$;

wegen $[comp]_{par}$: $\{I[y \rightarrow y*x, x \rightarrow x-1]\}$ y:=y*x;x:=x-1 {I};

Es gilt: $I \wedge x \mathrel{<>} 1 \Rightarrow I[y \rightarrow y*x, x \rightarrow x-1]$, also

wegen $[cons]_{par}$: $\{I \wedge x \mathrel{<>} 1\}$ y:=y*x;x:=x-1 {I};

wegen $[while]_{par}$: {I} while x<>1 do y:=y*x;x:=x-1 end $\{I \wedge x=1\}$;

Es gilt: $I \wedge x=1 \Rightarrow (y = n! \wedge n > 0)$, also

wegen $[cons]_{par}$: {I} while x<>1 do y:=y*x;x:=x-1 end $\{y = n! \wedge n > 0\}$;

wegen $[ass]_{par}$: $\{I[y \rightarrow 1]\}$ y:=1 {I};

Es gilt: $x=n \Rightarrow I[y \rightarrow 1]$; also

wegen $[cons]_{par}$: {x=n} y:= 1 {I} und

wegen $[comp]_{par}$: {x=n} y:=1; while x <> 1 do y:=y*x;x:=x-1 end $\{n>0 \wedge y=n!\}$

Beispiel: ggt-Berechnung

Zeigen: $\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$

```
while x<>y do
   if x>y then
        x := x –y
   else
        y := y – x
   end
end
```

$\{ x = ggt(a,b) \}$

I = „ $x>0 \wedge y > 0 \wedge ggt(x,y) = ggt(a,b)$“

Beispiel: ggt-Berechnung

```
{x=a∧y=b∧a>0∧b>0}
{I}
while x<>y do
    if x>y then
          x := x –y
    else
          y := y – x
    end
end
```

I = „ $x>0 \land y > 0 \land \ ggt(x,y) = ggt(a,b)$“

Beispiel: ggt-Berechnung

```
{x=a∧y=b∧a>0∧b>0}
{I}
while x<>y do
   {I ∧ x <>y}
   if x>y then
         x := x –y
   else
         y := y – x
   end
end
```

I = „ x>0 ∧ y > 0 ∧ ggt(x,y) = ggt(a,b)“

Beispiel: ggt-Berechnung

$\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$

$\{I\}$

while x<>y do

    $\{I \wedge x <> y\}$

    if x>y then

        $\{I \wedge x<>y \wedge x>y\} \Rightarrow \{x-y>0 \wedge y>0 \wedge ggt(x-y,y)=ggt(a,b)\}$

        x := x –y

    else

        $\{I \wedge x<>y \wedge x \leq y\} \Rightarrow \{I \wedge x<y\} \Rightarrow \{y-x>0 \wedge x>0 \wedge ggt(x,y-x)=ggt(a,b)\}$

        y := y – x

    end

end

I = „ $x>0 \wedge y > 0 \wedge ggt(x,y) = ggt(a,b)$"

Beispiel: ggt-Berechnung

$\{x=a \wedge y=b \wedge a>0 \wedge b>0\}$
$\{I\}$
while x<>y do
    $\{I \wedge x <> y\}$
    if x>y then
        $\{I \wedge x<>y \wedge x>y\} \Rightarrow \{x-y>0 \wedge y>0 \wedge ggt(x-y,y)=ggt(a,b)\}$
        x := x –y
        $\{I\}$
    else
        $\{I \wedge x<>y \wedge x \leq y\} \Rightarrow \{I \wedge x<y\} \Rightarrow \{y-x>0 \wedge x>0 \wedge ggt(x,y-x)=ggt(a,b)\}$
        y := y – x
        $\{I\}$
    end
    $\{I\}$
end
$\{I \wedge x=y\} \Rightarrow \{x = ggt(a,b)\}$

I = „ $x>0 \wedge y > 0 \wedge ggt(x,y) = ggt(a,b)$“

Hilfssatz: $ggt(x,y) = ggt(x-y,y)$

- Vor: $x>0$, $y>0$, $x-y>0$

- $ggt(x,y)$ teilt x, $ggt(x,y)$ teilt y, also $ggt(x,y)$ teilt $x-y$.
- Also $ggt(x,y)$ ist gemeinsamer Teiler von $x-y$ und y
- Also $ggt(x,y) \leq ggt(x-y,y)$

- $ggt(x-y,y)$ teilt $x-y$, $ggt(x-y,y)$ teilt y, also $ggt(x-y,y)$ teilt $y+(x-y)=x$
- Also $ggt(x-y,y)$ ist gemeinsamer Teiler von x und y
- Also $ggt(x-y,y) \leq ggt(x,y)$

- Also: $ggt(x,y) = ggt(x-y,y)$

Arrays

- Hatten: $[\text{ass1}]_{par}$: $\{P[x \rightarrow \mathcal{A}[\![E]\!]]\}\ x := E\ \{P\}$

- Naiv: $\{a[3] = 1\}\ a[i] := 4\ \{a[3] = 1\}$
  - nur richtig für $i \neq 3$

- Problemanalyse: Verschiedene Zugriffe auf ein und dasselbe Element: a[3]   a[i] (i=3) ...
- Lösung: Array als ganzheitliche Variable auffassen
- a‘=write(a,i,n) ist ein Array mit a‘[j] = n, falls j=i, a‘[j]=a[j], sonst.
- Also:
- $[\text{arr1}]_{par}$: $\{P[a \rightarrow write(a, \mathcal{A}[\![i]\!], \mathcal{A}[\![E]\!])]\}\ a[i] := E\ \{P\}$

Beispiel: Maximum-Berechnung

Zeigen:

```
{n>0}
i := 0;
m := a[0];
while i<n do
        i := i+1;
        if a[i] > m then
                m := a[i]
        else
                skip
        end
end
{m = max {a[0],...a[n]}}
```

Beispiel: Maximum-Berechnung

Zeigen: $\{n \geq 0\}$

```
i := 0;
```

$\{n \geq i \wedge i=0\}$

```
m := a[0];
```

$\{m = a[0] \wedge i=0\} \Rightarrow \{i \leq n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y \leq i \Rightarrow a[y] = m\}$

```
while i<n do
        i := i+1;
    if a[i] > m then
        m := a[i]
    else
        skip
    end
end
```

$\{m = \max \{a[0],...a[n]\}$

Beispiel: Maximum-Berechnung

Zeigen: $\{n \geq 0\}$

```
        i := 0;
        {n≥i∧i=0}
        m := a[0];
    {m = a[0]∧i=0} ⇒ {i≤n∧für alle x: x≤i ⇒a[x] ≤m∧es gibt ein y: y≤i ⇒a[y] = m}
        while i<n do
        {i<n∧für alle x: x≤i ⇒a[x] ≤m∧es gibt ein y: y≤i ⇒a[y] = m}
                i := i+1;
             if a[i] > m then
                   m := a[i]
             else
                   skip
             end
         end
         {m = max {a[0],...a[n]}}
```

Beispiel: Maximum-Berechnung

Zeigen: $\{n \geq 0\}$

$i := 0;$

$\{n \geq i \wedge i = 0\}$

$m := a[0];$

$\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

while $i < n$ do

$\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

$i := i+1;$

$\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

if $a[i] > m$ then

$m := a[i]$

else

skip

end

end

$\{m = \max \{a[0],...a[n]\}\}$

Beispiel: Maximum-Berechnung

Zeigen: $\{n \geq 0\}$

$i := 0;$

$\{n \geq i \wedge i = 0\}$

$m := a[0];$

$\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

while $i < n$ do

$\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

$i := i+1;$

$\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

if $a[i] > m$ then

$\{a[i] > m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

$m := a[i]$

else

$\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

skip

end

end

$\{m = \max \{a[0], \ldots a[n]\}\}$

Beispiel: Maximum-Berechnung

```
Zeigen:   {n≥0}
          i := 0;
          {n≥i∧i=0}
          m := a[0];
     {m = a[0]∧i=0} ⇒ {i≤n∧für alle x: x≤i ⇒a[x] ≤m∧es gibt ein y: y≤i ⇒a[y] = m}
          while i<n do
          {i<n∧für alle x: x≤i ⇒a[x] ≤m∧es gibt ein y: y≤i ⇒a[y] = m}
                         i := i+1;
               {i≤n∧für alle x: x<i ⇒a[x] ≤m∧es gibt ein y: y<i ⇒a[y] = m}
               if a[i] > m then
              {a[i]>m∧i≤n∧für alle x: x<i ⇒a[x] ≤m∧es gibt ein y: y<i ⇒a[y] = m}
          ⇒  {i≤n∧für alle x: x≤i ⇒a[x] ≤a[i]∧es gibt ein y: y≤i ⇒a[y] = a[i]}
                        m := a[i]
            {i≤n∧für alle x: x≤i ⇒a[x] ≤m∧es gibt ein y: y≤i ⇒a[y] = m}
               else
            {a[i]≤m ∧i≤n∧für alle x: x<i ⇒a[x] ≤m∧es gibt ein y: y<i ⇒a[y] = m}
                        skip
            {a[i]≤m ∧i≤n∧für alle x: x<i ⇒a[x] ≤m∧es gibt ein y: y<i ⇒a[y] = m}
           ⇒{i≤n∧für alle x: x≤i ⇒a[x] ≤m∧es gibt ein y: y≤i ⇒a[y] = m}
               end
           end
           {m = max {a[0],...a[n]}
```

Beispiel: Maximum-Berechnung

Zeigen: $\{n \geq 0\}$

$i := 0;$

$\{n \geq i \wedge i = 0\}$

$m := a[0];$

$\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y \leq i \wedge a[y] = m\}$

while $i<n$ do

$\{i<n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y \leq i \Rightarrow a[y] = m\}$

$i := i+1;$

$\{i \leq n \wedge \text{für alle } x\text{: } x<i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y<i \Rightarrow a[y] = m\}$

if $a[i] > m$ then

$\{a[i]>m \wedge i \leq n \wedge \text{für alle } x\text{: } x<i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y<i \Rightarrow a[y] = m\}$

$\Rightarrow \{i \leq n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq a[i] \wedge \text{es gibt ein } y\text{: } y \leq i \Rightarrow a[y] = a[i]\}$

$m := a[i]$

$\{i \leq n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y \leq i \Rightarrow a[y] = m\}$

else

$\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x\text{: } x<i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y<i \Rightarrow a[y] = m\}$

skip

$\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x\text{: } x<i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y<i \Rightarrow a[y] = m\}$

$\Rightarrow \{i \leq n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y \leq i \Rightarrow a[y] = m\}$

end

$\{i \leq n \wedge \text{für alle } x\text{: } x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y\text{: } y \leq i \Rightarrow a[y] = m\}$

end

$\{m = \max \{a[0],...a[n]\}\}$

Beispiel: Maximum-Berechnung

Zeigen: $\{n \geq 0\}$

$i := 0;$

$\{n \geq i \wedge i = 0\}$

$m := a[0];$

$\{m = a[0] \wedge i = 0\} \Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

while $i < n$ do

$\{i < n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

$i := i+1;$

$\{i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

if $a[i] > m$ then

$\{a[i] > m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

$m := a[i]$

$\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

else

$\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

skip

$\{a[i] \leq m \wedge i \leq n \wedge \text{für alle } x: x < i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y < i \Rightarrow a[y] = m\}$

$\Rightarrow \{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

end

$\{i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

end

$\{i \geq n \wedge i \leq n \wedge \text{für alle } x: x \leq i \Rightarrow a[x] \leq m \wedge \text{es gibt ein } y: y \leq i \Rightarrow a[y] = m\}$

$\Rightarrow \{m = \max \{a[0], ... a[n]\}\}$

Beispiel: Array-Elemente tauschen

$\{a[i] = a \wedge a[j] = b\}$

$h := a[i];$

$\{h = a \wedge a[j] = b\}$

$a[i] := a[j];$

$\{a[i] = b \wedge h = a\}$

$a[j] := h;$

$\{a[i] = b \wedge a[j] = a\}$

funktioniert auch, wenn $i = j$!

Beispiel: Dutch National Flag

- Problem:
- geg: Array a[0 .. n] mit $a[i] \in \{blau, weiss, rot\}$

Ziel: Array umsortieren (allein durch Swap-Operationen) derart, dass alle blau-Einträge vor allen weiss-Einträgen, und diese vor allen rot-Einträgen liegen.

Algorithmus

```
b := 0; w:=0; r := n;
while w ≤ r do
        case a[w]
            blau:    swap(a[b],a[w]);
                     w := w+1; b:= b+1;
            weiss:   w:=w+1;
            rot:     swap(a[w],a[r]);
                     r := r-1;
        end
end
```

Abschweifung: Hoare-Logik im Programmieralltag

- C, JAVA, EIFFEL ... unterstützen Assertions
- Schleifeninvarianten nach eigener Erfahrung extrem hilfreich
- Datenstruktur-Invarianten essentiell, um komplexe Strukturen zu beherrschen
- „Design by contract“ „Rely-Guarantee-Paradigm“:

{P} S {Q} = „sicherst Du mir P zu, garantiere ich Dir Q“

Weitere Beispiele

- Zeigen: Frühes Überlegen von Schleifeninvarianten hilft bei der Problemlösung
- Schleifeninvarianten kondensieren algorithmische Ideen

Sattelsuche

- Gegeben Matrix von Zahlen
- Zeilen enthalten aufsteigende Werte
- Spalten enthalten aufsteigende Werte
- Wissen: k kommt irgendwo in der Matrix vor

- Frage: Wo?

- Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

Sattelsuche

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

Zeilen: 0 .. m-1

Spalten: 0 .. n-1

Gesuchter Wert k bei [i,j]

Start: $0 \leq i < m$ und $0 \leq j < n$

Idee: Suchraum eingrenzen

Start mit iu := 0, io := m, ju := 0, jo := n

Ziel: io = iu +1, jo = ju+1

Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	*10*	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?

- a) irgendwo in der Mitte   a[i,j] > k   a[i,j] < k

Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?

- a) irgendwo in der Mitte   a[i,j] > k   a[i,j] < k
- b) irgendwo am Rand

Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?

- a) irgendwo in der Mitte $a[i,j] > k$ $a[i,j] < k$
- b) irgendwo am Rand
- c) Ecke links oben oder rechts unten

Wie Suchraum einschränken?

Beispiel:

1	3	4	5	7	10
2	5	8	10	13	20
5	6	8	10	20	42
8	9	10	10	100	200

- Ein Element abfragen; welches?

- a) irgendwo in der Mitte   a[i,j] > k   a[i,j] < k
- b) irgendwo am Rand
- c) Ecke links oben oder rechts unten
- d) Ecke links unten oder rechts oben

Algorithmus+Beweis

```
{k kommt in a vor}
iu := 0; io := m; ju := 0; jo := n;
{es gibt i,j mit iu≤i<io und ju≤j<jo und a[i,j] = k}
while io – iu > 1 OR jo – ju > 1 do
   case a[io-1,ju]
      = k:   iu := io - 1; jo := ju+1;
      < k:   ju := ju + 1;
      > k:   io := io – 1;
   end
end
```

Größtes True-Quadrat

Geg: Matrix aus Booleans

Ges: Größe einer größten quadratischen Teilmatrix, die nur TRUE-Einträge hat

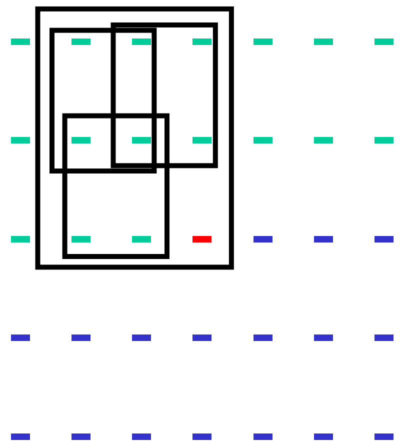
T	T	F	F	T	T	F
T	T	T	F	T	T	T
T	T	T	T	T	T	T
T	T	T	F	T	T	T
T	F	T	T	T	T	T

Größtes True-Quadrat

Idee: Zahlenmatrix, die für jeden Matrixeintrag i,j die Größe des größten True-Quadrates mit i,j als rechter unterer Ecke angibt

Warum?

T	T	F	F	T	T	F
T	T	T	F	T	T	T
T	T	T	T	T	T	T
T	T	T	F	T	T	T
T	F	T	T	T	T	T



Größtes True-Quadrat

Algorithmus und Beweis:
(Der Einfachheit halber Annahme: b[x,y] = 0 für x<0 oder y<0)

```
z := 0; s := 0;
{Für alle [i,j] mit i<z oder i=z und j<s ist b[i,j] die Größe des
  größten True-Quadrates mit [i,j] als rechter unterer Ecke }
while z < m do
       while s < n do
              case a[z,s]
                     F :     b[z,s] := 0;
                     T :     b[z,s] := MIN(b[z-1,s],b[z-1,s-1],b[z,s-1])+1
               end;
                s := s + 1;
        end
        z := z + 1; s := 0;
end
```

Erweiterung: Prozeduren

- nichtrekursiv:

- $[\text{call1}]_{\text{par}}$: $\dfrac{\{P\}\ S\ \{Q\}}{\{P\}\ \text{call}\ p\ \{Q\}}$ (proc p is S)

- rekursiv:
- $[\text{call2}]_{\text{par}}$: $\dfrac{\{P\}\ \text{call}\ p\ \{Q\}\ \vdash \{P\}\ S\ \{Q\}}{\{P\}\ \text{call}\ p\ \{Q\}}$ (proc p is S)

Beispiel: Quicksort

```
qs(0,n);

qs(von,bis) is
     b := von; w := von+1; r := bis;
     p := a[b];
    while w ≤ r do
           case
               a[w] < p:     swap(a[b],a[w]);
                             w := w+1; b:= b+1;
                a[w] = p:   w:=w+1;
               a[w] > p:    swap(a[w],a[r]);
                                 r := r-1;
           end
    end
    if von < b-1 then qs(von,b-1) end;
    if r+1 < bis then qs(r+1,bis) end;
end
```

Dutch National Flag
blau := <p
weiss := =p
rot := >p

Beweis: Quicksort

```
{0 < n}
qs(0,n);

{von < bis]
qs(von,bis) is

   b := von; w := von+1; r := bis;
     {b <w ≤ r+1}
      p := a[b];
    while w ≤ r do
          case
             a[w] < p:     swap(a[b],a[w]);
                          w := w+1; b:= b+1;
              a[w] = p:   w:=w+1;
             a[w] > p:    swap(a[w],a[r]);
                             r := r-1;
          end
    end
    {b<w≤r+1∧für alle i : (i<b⇒a[i]<p)∧(i≥b∧i<w⇒a[i]=p)∧(i>r⇒a[i]>p)}
    if von < b-1 then qs(von,b-1) end;
    {b<w≤r+1∧für alle i : (i<b⇒a[i]<p)∧(i≥b∧i<w⇒a[i]=p)∧(i>r⇒a[i]>p) ∧ sortiert(von,b-1) }
    if r+1 < bis then qs(r+1,bis) end;
    {b<w≤r+1∧für alle i : (i<b⇒a[i]<p)∧(i≥b∧i<w⇒a[i]=p)∧(i>r⇒a[i]>p) ∧ sortiert(von,b-1) ∧ sortiert(r+1,bis) }
   ⇒ {sortiert(von,bis)}
end
{sortiert(von,bis)}
```

4.2 Axiomatische Semantik von W (totale Korrektheit)

- $[\text{ass1}]_{\text{tot}}$: $\{P[x \to \mathcal{A}⟦E⟧]\}\ x := E\ \{\Downarrow P\}$ falls x Int-Variable
- $[\text{ass2}]_{\text{tot}}$: $\{P[x \to \mathcal{B}⟦E⟧]\}\ x := E\ \{\Downarrow P\}$ falls x Bool-Variable
- $[\text{skip}]_{\text{tot}}$: $\{P\}\ \text{skip}\ \{\Downarrow P\}$
- $[\text{comp}]_{\text{tot}}$: $\dfrac{\{P\}\ S1\ \{\Downarrow Q\}\ ,\ \{Q\}\ S2\ \{\Downarrow R\}}{\{P\}\ S1\ ;\ S2\ \{\Downarrow R\}}$
- $[\text{if}]_{\text{tot}}$: $\dfrac{\{P \wedge \mathcal{B}⟦b⟧\}\ S1\ \{\Downarrow Q\}\ ,\ \{P \wedge \neg\mathcal{B}⟦b⟧\}\ S2\ \{\Downarrow Q\}}{\{P\}\ \text{if } b \text{ then } S1 \text{ else } S2 \text{ end}\ \{\Downarrow Q\}}$
- $[\text{while}]_{\text{tot}}$: $\{P(z+1)\}\ S\ \{\Downarrow P(z)\}\quad (z \in \mathbf{N})$

  $\{\text{ex. } z: P(z)\}\ \text{while } b \text{ do } S \text{ end}\ \{\Downarrow P(0)\}$

  wobei $P(z+1) \Rightarrow \mathcal{B}⟦b⟧\ ,\ P(0) \Rightarrow \neg\mathcal{B}⟦b⟧$
- $[\text{cons}]_{\text{tot}}$: $\dfrac{\{P\}\ S\ \{\Downarrow Q\}}{\{P'\}\ S\ \{\Downarrow Q'\}}$ falls $P' \Rightarrow P \wedge Q \Rightarrow Q'$

Liberaleres Konzept: Abstiegsfunktion

- Zu jeder While-Schleife Term t mit nat. Zahl als Wert und
  - $\{t = k\}$ S $\{t < k\}$
  - $t = 0 \Rightarrow \neg\mathcal{B}$ $[\![b]\!]$

- Dann terminiert while b do S end

Terminierung: Fakultätsberechnung

y:=1; while x <> 1 do y:=y*x;x:=x-1 end $\{n>0 \wedge y=n!\}$

Abstiegsfunktion: $x - 1$ (Voraussetzung: $x > 0$)

Terminierung: ggt-Berechnung

```
while x<>y do
    if x>y then
        x := x –y
    else
        y := y – x
    end
end
```

Abstiegsfunktion: $|x - y|$

Terminierung: Maximum-Berechnung

```
i := 0;
        m := a[0];
        while i<n do
              i := i+1;
              if a[i] > m then
                   m := a[i]
              else
                   skip
              end
         end
```

Abstiegsfunktion: n - i

Terminierung: Dutch National Flag

```
b := 0; w:=0; r := n;
while w ≤ r do
        case a[w]
           blau:    swap(a[b],a[w]);
                    w := w+1; b:= b+1;
           weiss:   w:=w+1;
           rot:     swap(a[w],a[r]);
                    r := r-1;
        end
end
```

Abstiegsfunktion: $r + 1 - w$

Terminierung: Sattelsuche

```
iu := 0; io := m; ju := 0; jo := n;
while io – iu > 1 OR jo – ju > 1 do
  case a[io-1,ju]
    = k:   iu := io - 1; jo := ju+1;
    < k:   ju := ju + 1;
    > k:   io := io – 1;
  end
end
```

Abstiegsfunktion: (io-iu) +(jo-ju)

Terminierung: Größtes True-Quadrat

```
z := 0; s := 0;
while z < m do
       while s < n do
              case a[i,j]
                     F :     b[i,j] := 0;
                     T :     b[i,j] := MIN(b[i-1,j],b[i-1,j-1],b[i,j-1])+1
               end;
                s := s + 1;
        end
        z := z + 1; s := 0;
end
```

Abstiegsfunktion: mn – nz - s

Terminierung: Prozeduren

- nichtrekursiv:

- $[\text{call1}]_{par}$: $$\frac{\{P\}\ S\ \{\Downarrow Q\}}{\{P\}\ \text{call } p\ \{\Downarrow Q\}} \qquad \text{(proc } p \text{ is } S)$$

- rekursiv:
- $[\text{call2}]_{par}$: $$\frac{\{P(z)\}\ \text{call } p\ \{\Downarrow Q\} \vdash \{P(z+1)\}\ S\ \{\Downarrow Q\}}{\{\text{ex. } z: P(z)\}\ \text{call } p\ \{\Downarrow Q\}} \qquad \text{wobei } \neg P(0)$$

  (proc p is S, $\mathcal{WB}(z) = \mathbf{N}$)

Beispiel: Quicksort

```
qs(0,n);

qs(von,bis) is
     b := von; w := von; r := bis;
     p := a[b];
    while w ≤ r do
           case
               a[w] < p:     swap(a[b],a[w]);
                             w := w+1; b:= b+1;
                a[w] = p:   w:=w+1;
               a[w] > p:    swap(a[w],a[r]);
                            r := r-1;
           end
    end
    if von < b-1 then qs(von,b-1) end;
    if r+1 < bis then qs(r+1,bis) end;
end
```

Abstiegsfunktion: bis - von

4.3 Korrektheit und Vollständigkeit

- stellen Beziehung her zwischen partieller Korrektheit und Natural Semantics:

- $\models$ {P} S {Q}   falls für alle Zustände s gilt:
  Wenn P(s) und <S,s>→s‘, so Q(s‘)

  „{P} S {Q} ist richtig“

- Zum Vergleich:
- $\vdash$ {P} S {Q}   falls sich diese Aussage mittels der Axiome und Regeln der partiellen Korrektheit herleiten lässt

  „{P} S {Q} ist beweisbar“

Korrektheit und Vollständigkeit

- Korrektheit: „Alles, was sich beweisen lässt, ist richtig“

Für alle P,Q,S: Wenn $\vdash$ {P} S {Q}, so $\models$ {P} S {Q}

- Vollständigkeit: „Alles, was richtig ist, lässt sich beweisen“

Für alle P,Q,S: Wenn $\models$ {P} S {Q}, so $\vdash$ {P} S {Q}

Korrektheit: Wenn $\vdash$ {P} S {Q}, so $\models$ {P} S {Q}

- Beweis durch Induktion über der Struktur einer Ableitung gemäß der Axiome und Regeln der partiellen Korrektheit:

- Anfang 1: Sei S = x := E und <S,s>→s‘. Also s‘ = s[x→$\mathcal{A}$ ⟦E⟧ (s)]. Wenn nun P[x→$\mathcal{A}$ ⟦E⟧](s), so gilt also auch P(s‘).
- Anfang 2: Sei S = skip -- trivial
- Schritt 1: Sei S = S1;S2 und <S1;S2,s>→s‘. Also ex. s“ mit <S1,s>→s“ und <S2,s“>→s‘. Sei nun $\vdash${P} S1 {Q} und $\vdash$ {Q} S2 {R}. Nach IV: $\models${P} S1 {Q} (*) und $\models${Q} S2 {R} (**). Wenn nun P(s) gilt, so ist wegen (*) Q(s“) und wegen (**) R(s‘).

Forts. Korrektheit: Wenn $\vdash$ {P} S {Q}, so $\models$ {P} S {Q}

- Schritt 2: Sei S = if b then S1 else S2 end und <S,s>→s‘.

  1. Fall: $\mathcal{B}[\![b]\!](s)$ = true. Also gilt $(P\wedge \mathcal{B}[\![b]\!])(s)$.

  Nach IV: $\models \{P\wedge \mathcal{B}[\![b]\!]\}$ S1 {Q} Also Q(s‘).

  2. Fall analog

- Schritt 3: Sei S = while b do S‘ end und <S,s> → s‘

  Zeigen per Induktion über Tiefe der Ableitung für <S,s>→s‘:

  $(P\wedge \neg\mathcal{B}[\![b]\!])(s')$.

  Anfang (Tiefe 0): Dann: $\mathcal{B}[\![b]\!](s)$ = false. Also ist s = s‘, also $(P\wedge \neg\mathcal{B}[\![b]\!])(s')$.

  Schritt: (Tiefe > 0) Dann: $\mathcal{B}[\![b]\!](s)$ = true. Also $(P\wedge \mathcal{B}[\![b]\!])(s)$. Dann gibt es

  s“ mit <S‘,s> → s“ und <S,s“> → s‘. Es gilt P(s“). Nach IV folgt $(P\wedge \neg\mathcal{B}[\![b]\!])(s')$.

- Schritt 4 (Konsequenzregel) kein Problem.

Vollständigkeit: Wenn $\models$ {P} S {Q}, so $\vdash$ {P} S {Q}

- Betrachten folgende Assertions:

  - {x = 2} x := x + 1 {x ungerade}
  - {x = 28} x := x + 1 {x ungerade}
  - {x durch 4 teilbar} x := x + 1 {x ungerade}
  - {x gerade} x := x +1 {x ungerade}

  - Alle richtig. Die letzte wohl am wertvollsten.

wpr(S,Q), die *schwächste Vorbedingung* für S und Q, ist dasjenige Prädikat P, wo P(s) gdw. für alle s‘ mit $\langle S,s\rangle \rightarrow s'$: Q(s‘).

Aussagen über wpr

- Für alle S,Q: $\models$\{wpr(S,Q)\} S \{Q\}

  Beweis: Wenn wpr(S,Q)(s) und <S,s>→s‘, so per Def. von wpr: Q(s‘).

- Für alle P,S,Q: Wenn $\models$\{P\} S \{Q\}, so P $\Rightarrow$ wpr\{S,Q\}

  Beweis: Sei s Zst. mit P(s) und <S,s>→s‘. Wegen $\models$\{P\} S \{Q\} ist Q(s‘). Also wpr(S,Q)(s). Also P $\Rightarrow$ wpr\{S,Q\}

Vollständigkeit: Wenn $\models$ {P} S {Q}, so $\vdash$ {P} S {Q}

- Wir zeigen für alle S,Q: $\vdash$ {wpr(S,Q)} S {Q}. Originalaussage folgt dann mit den Aussagen der vorigen Folie

- Fall x := E. Offenbar wpr(x:=E,Q) = Q[x→$\mathcal{A}$ ⟦E⟧]
- Fall skip: Offenbar wpr(skip,Q) = Q.
- Fall S = S1;S2. Nach IV: $\vdash$ {wpr(S2,Q)} S2 {Q} und

$\vdash$ {wpr(S1,wpr(S2,Q))} S1 {wpr(S2,Q)}. Damit ist

$\vdash$ {wpr(S1,wpr(S2,Q))} S1;S2 {Q} Bleibt z.Z:

wpr(S1;S2,Q) $\Rightarrow$ wpr(S1,wpr(S2,Q))

Betrachten s mit wpr(S1;S2,Q)(s)

Sei <S1,s> →s“ und <S2,s“>→s‘ (wenn es s‘ bzw. s“ nicht gibt, wird die Aussage trivial). Also Q(s‘). Also wpr(S2,Q)(s“). Also

wpr(S1,wpr(S2,Q))(s).

Vollständigkeit: Wenn $\models \{P\}\ S\ \{Q\}$, so $\vdash \{P\}\ S\ \{Q\}$

- Wir zeigen für alle S,Q: $\vdash \{wpr(S,Q)\}\ S\ \{Q\}$.

- Fall S = if b then S1 else S2 end.

  Mit IV: $\vdash \{wpr(S1,Q)\}\ S1\ \{Q\}$ und $\vdash \{wpr(S2,Q)\}\ S2\ \{Q\}$.

  Sei $P = (\mathcal{B}[\![b]\!] \wedge wpr(S1,Q)) \vee (\neg\mathcal{B}[\![b]\!] \wedge wpr(S2,Q))$

  Mit $[cons]_P$: $\vdash \{\mathcal{B}[\![b]\!] \wedge P\}\ S1\ \{Q\}$ und $\vdash \{\neg\mathcal{B}[\![b]\!] \wedge P\}\ S2\ \{Q\}$.

  Also mit $[if]_P$: $\{P\}$ if b then S1 else S2 end $\{Q\}$. Bleibt z.Z.:

  wpr(if b then S1 else S2 end,Q) $\Rightarrow$ P.

  Sei $\langle S,s\rangle \rightarrow s'$ mit $Q(s')$. 1. Fall: $\mathcal{B}[\![b]\!](s)$. Also gilt linke Alternative in P. 2. Fall analog.

Vollständigkeit: Wenn $\models$ {P} S {Q}, so $\vdash$ {P} S {Q}

- Wir zeigen für alle S,Q: $\vdash$ {wpr(S,Q} S {Q}.

- Fall S = while b do S‘ end. Sei P = wpr(while b do S‘ end,Q).

1. Zeigen: $\neg\mathcal{B}⟦b⟧ \wedge P \Rightarrow Q$

   Sei s Zst. mit $(\neg\mathcal{B}⟦b⟧ \wedge P)(s)$. Also <while b do S‘ end,s>→s, also Q(s).

2. Zeigen: $\mathcal{B}⟦b⟧ \wedge P \Rightarrow$ wpr(S‘,P)

   Sei Sei s Zst. mit $(\mathcal{B}⟦b⟧ \wedge P)(s)$. Sei <S‘,s>→s‘. (Wenn s‘ nicht ex., ist Aussage trivial). Zeigen: P(s‘)

   1. Fall: Es gibt ein s“ mit <while b so S‘ end,s‘>→s“.

      Dann gilt auch <while b do S‘ end,s>→s“, und wegen Wahl von P auch Q(s“). Damit muss aber auch P(s‘) gelten

   2. Fall: Es gibt kein s“ mit <while b do S‘ end,s‘> →s“.

      Dann gilt P(s‘) trivialerweise.

Mit IV: $\vdash$\{wpr(S‘,P)\} S‘ {P}. Mit (2): $\vdash\{P \wedge \mathcal{B}⟦b⟧\}$ S‘ {P}. Mit $[\text{while}]_P$:

$\vdash$\{P\} while b do S‘ end $\{\neg\mathcal{B}⟦b⟧ \wedge P\}$. Mit [cons] und (1):

$\vdash$\{P\} while b do S‘ end {Q}.

Gödelscher Unvollständigkeitssatz

„In jedem hinreichend ausdrucksstarken formalen System gibt es Aussagen, die richtig, aber nicht beweisbar sind“

hinreichend ausdrucksstark: im wesentlichen Rechnen mit natürlichen Zahlen und etwas Logik

Formales System = Aussagen formulieren mittels Sprache; Beweisschritte mittels Ersetzungsregeln

Gödels Beweisidee 1: Übersetzung in Zahlentheorie

- Aussagen der Sprache übersetzen in Zahlen:

Sprache hat Alphabet: Zeichen → Zahl

Satz der Sprach ist Folge von Zeichen.

Kodierung: $2^{\text{erster Bu.}}\ 3^{\text{zweiter Bu}}\ 5^{\text{dritter Bu}}$

Anwendung einer Regel: Rechnen mit Zahlen

Gödels Beweisidee 2: Selbstaussage

- Lügnerparadoxon: Ein Kreter sagt: „Alle Kreter lügen“
- Gödels Selbstaussage:
- H = „H ist nicht beweisbar“
  - 1. Fall: beweisbar, dann falsch, also System nicht korrekt
  - 2. Fall: nicht beweisbar, dann aber richtig!

Gödels Argument trägt nicht in unserem Setting

- Grund: Lassen beliebige Prädikate zu.
- Haben: Abzählbar viele Variablen
- →überabzählbar viele Prädikate
- →es können nicht alle Prädikate in einer Sprache formuliert werden
- → Voraussetzung für Gödels Satz treffen nicht zu

Zusammenfassung Semantik

- Drei Konzepte, entsprechen drei grundlegenden Herangehensweisen an Modellbildung in der Informatik
  - Operationell: Nutzen abstrakte Maschinenmodelle; im Mittelpunkt steht der *Schritt*
  - Denotationell: Nutzen mathematische Formalismen wie Funktionen, Relationen, Gleichungen
  - Logisch: Nutzen logische Aussagen und Beweiskalküle

Zusammenfassung Semantik

- All diese Konzepte finden sich auch in Ansätzen zur (semi-) formalen Spezifikation
  - Operationell: State Charts, Activity diagrams, Petrinetze, Message sequence diagrams, ...
  - Denotationell: Z, μ-Kalkül
  - Logisch: Larch, temporale Logik

Zusammenfassung Semantik

- Operationelle Semantik:
  - relativ leicht zu definieren
  - relativ schwer auszuwerten
  - momentan Hauptmethode für verteilte bzw. interaktive Systeme (Protokolle, verteilte Algorithmen, GUI,...)
  - Anwendung: Vergleich von Systemen, Model Checking

Zusammenfassung Semantik

- Denotationelle Semantik:
  - relativ schwer zu definieren
  - reichhaltige Auswertungsmöglichkeiten
  - verlangt mathematische Fertigkeiten
  - Anwendung: Statische Analyse
    - kompositional

Zusammenfassung Semantik

- Axiomatische Semantik:
  - mittelschwer zu definieren
  - leichte Auswertung
  - verlangt logische Fertigkeiten
  - Anwendung: Programmverifikation
    - kompositional

Modellbildung in der Informatik

- Informatik = Übergang von informalen Ideen zu formalen Konstrukten
- vernünftiger Zwischenschritt: Modelle

- konsequente Umsetzung: Modellbasierte Softwareentwicklung

- State of the art: Informelle/semiformelle Modellierungskonzepte (vornehmlich UML)

Vorteile formaler Semantik/formaler Modelle

- Missverständnisarm
- Möglichkeit bedeutungstreuer Übersetzung
- Verfügbarkeit z.T. automatisierter Methoden zur Auswertung, zur Fehlererkennung
- Konstruktionsmethodik („correct by construction")
- Proof-carrying code
- tieferes Verständnis

Nachteile formaler Modelle

- Erstellungsaufwand
- Schwer lesbar für Quereinsteiger
- Aber:
  - Aufwand zahlt sich oft aus
  - Tools verstecken Teil der Schwierigkeiten
  - sichere Jobs für Experten
  - aktuelle Forschung: Komplexitätsreduktion