# Korrekte Software: Grundlagen und Methoden
# Vorlesung 1 vom 04.04.18: Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Organisatorisches

- Veranstalter:


| Christoph Lüth | Serge Autexier |
| --- | --- |
| christoph.lueth@dfki.de | serge.autexier@dfki.de |
| MZH 4186, Tel. 59830 | Cartesium 2.11, Tel. 59834 |


- Termine:
  - Vorlesung: Dienstag, 12 – 14, MZH 6210
  - Übung: Donnerstag, 12 – 14, MZH 1110
- Webseite:

  http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss18

# Übungsbetrieb

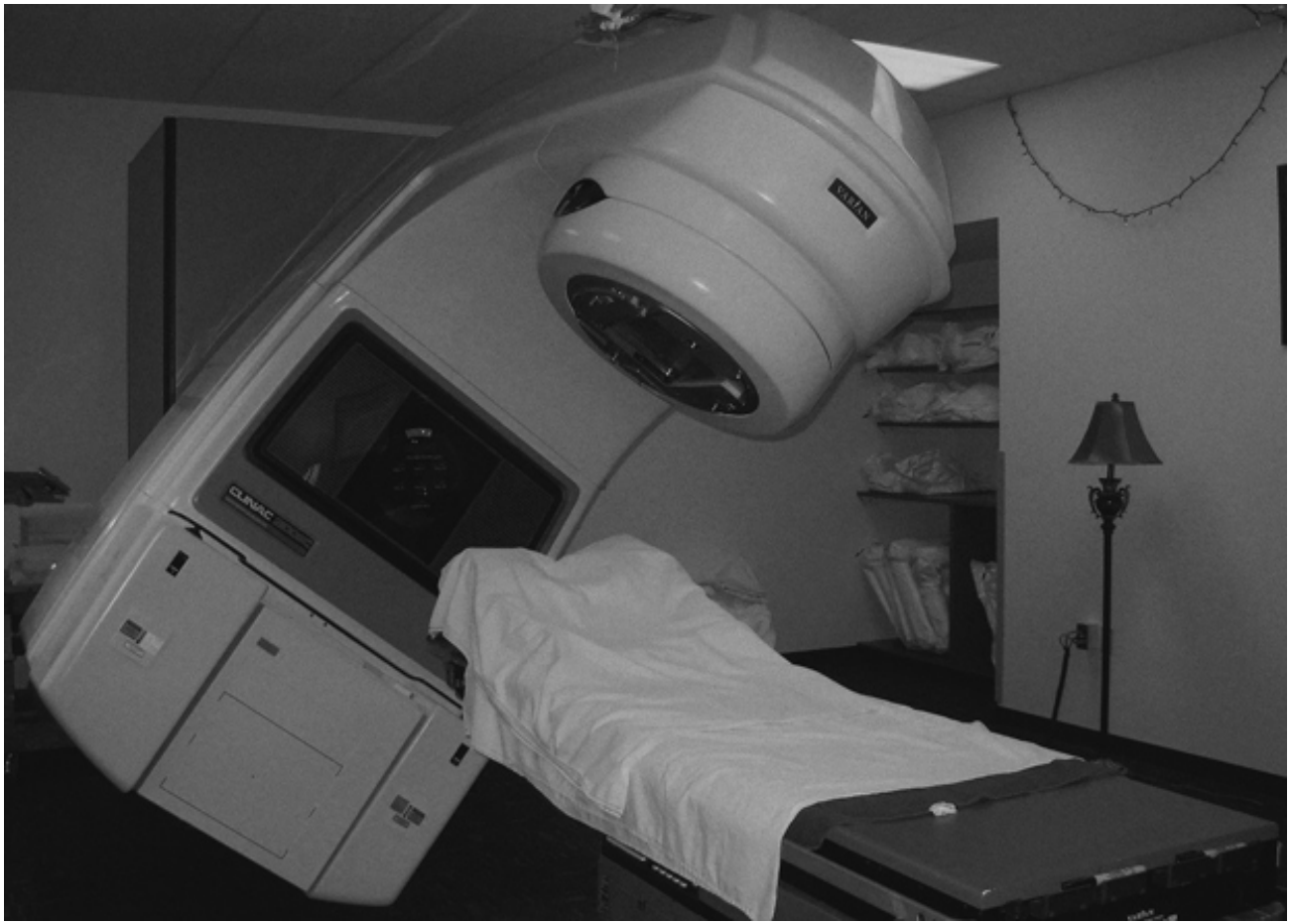
- "Leichtgewichtige" Übungsblätter, die **in der Übung** bearbeitet und **schnell** korrigiert werden können.
- Übungsblätter **vertiefen** Vorlesungsstoff, Bewertung gibt Feedback.
- Übungsbetrieb:
  - Gruppen bis zu drei Studierende
  - Ausgabe: Donnerstag in der Übung
  - Bearbeitung: in der Übung
  - Abgabe: Donnerstag abend

# Prüfungsform und Übungsbetrieb

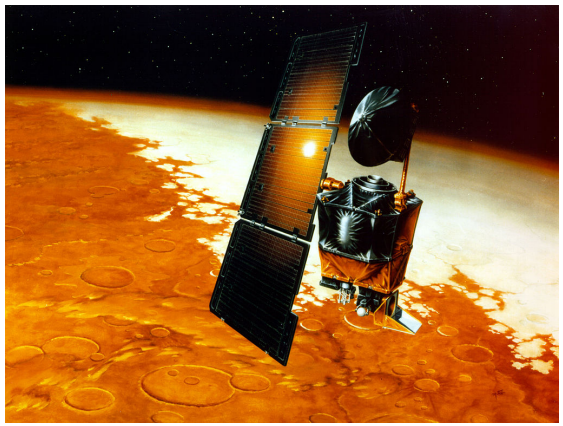
- 10 Übungsblätter (geplant)
- Bewertung:
  - A (sehr gut, 1.3) — nichts zu meckern, keine/kaum Fehler
  - B (gut, 2.3) — kleine Fehler, sonst gut
  - C (befriedigend, 3.3) — größere Fehler oder Mängel
  - Nicht bearbeitet — oder zu viele Fehler
- Prüfungsleistung:
  - Mündliche Prüfung
    - Einzelprüfung ca. 20– 30 Minuten
  - Übungsbetrieb (bis zu 20% Bonuspunkte, keine Voraussetzung)

# Warum Korrekte Software?

# Software-Disaster I: Therac-25



# Software-Disasters II: Space



Mariner 1 (27.08.1962), Mars Climate Orbiter (1999), Ariane 5 (04.06.1996)

# Software-Disaster III: AT&T (15.01.1990)

```
while (! empty(ring_rcv_buffer)
       && ! empty(side_buffer empty)) {
 initialize pointer to first message buffer;
 get copy of buffer;
 switch (message) {
   case (incoming_message):
     if (sender is out_of_service) {
       if (empty(ring_wrt_buffer)) {
         send "in service" to status map;
       } else {
         break;
       }
       process incoming message, set up pointers;
       break;
       }
   }
 do optional parameter work;
 }
```

# Software-Disaster IV: Airbus A400M

Sevilla, 09.05.2015

# Inhalt der Vorlesung

# Themen



Korrekte Software im Lehrbuch:

- Spielzeugsprache
- Wenig Konstrukte
- Kleine Beispiele

Korrekte Software im Einsatz:

- Richtige Programmiersprache
- Mehr als nur ganze Zahlen
- Skalierbarkeit — wie können große Programme verifiziert werden?

# Inhalt

- Grundlagen:
  - Beweis der **Korrektheit** von Programmen: der **Floyd-Hoare-Kalkül**
  - **Bedeutung** von Programmen: **Semantik**
- Betrachtete Programmiersprache: “C0” (erweiterte Untermenge von C)
- Erweiterung der Programmkonstrukte und des Hoare-Kalküls:
  1. Referenzen (Zeiger)
  2. Funktion und Prozeduren (Modularität)
  3. Reiche **Datenstrukturen** (Felder, `struct`)

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Warum Semantik?

# Idee

- Was wird hier berechnet?

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

# Idee

- Was wird hier berechnet? $p = n!$
- Warum? Wie können wir das **beweisen**?

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

# Idee

- Was wird hier berechnet? $p = n!$
- Warum? Wie können wir das **beweisen**?
- Wir berechnen symbolisch, welche Werte Variablen über den Programmverlauf annehmen.

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

# Semantik von Programmiersprachen

Drei wesentliche Möglichkeiten:

- **Operationale Semantik**:
  Ausführung auf einer **abstrakten** Maschine
- **Denotationale Semantik**:
  Abbildung in ein **mathematisches Objekt**
- **Axiomatische Semantik**:
  Beschreibung durch eines Programmes durch seine **Eigenschaften**

# Unsere Sprache C0

- C0 ist eine **Untermenge** der Sprache C
- C0-Programme sind **ausführbare** C-Programme
- Erste Ausbaustufe:
  - Zuweisungen, Fallunterscheidungen, Schleifen
  - Datentypen: ganze Zahlen mit Arithmetik
  - Relationen: Vergleich ($=$, $\leq$)
  - Boolsche Operatoren: Konjunktion, Disjunktion, Negation
- 1. Ausbaustufe: Funktionen und Prozeduren
- 2. Ausbaustufe: Felder und Strukturen
- 3. Ausbaustufe: Referenzen (Pointer)
- Fehlt: **union**, **goto**, ...

# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```

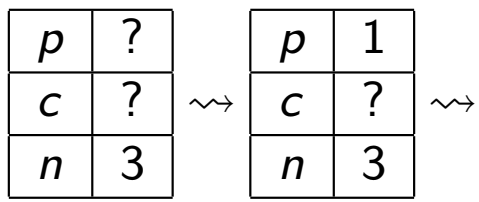
| | |
|---|---|
| $p$ | ? |
| $c$ | ? |
| $n$ | 3 |

$\rightsquigarrow$

# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

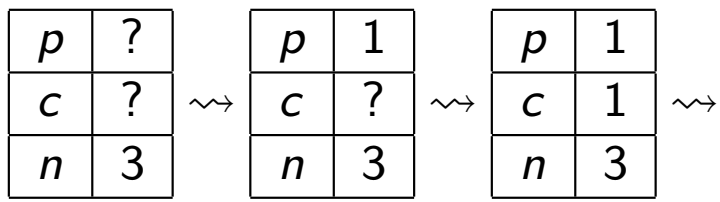
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

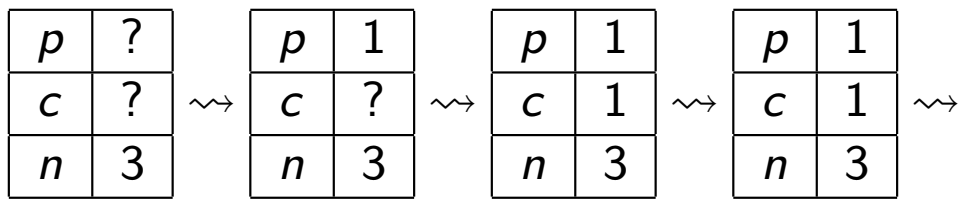
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

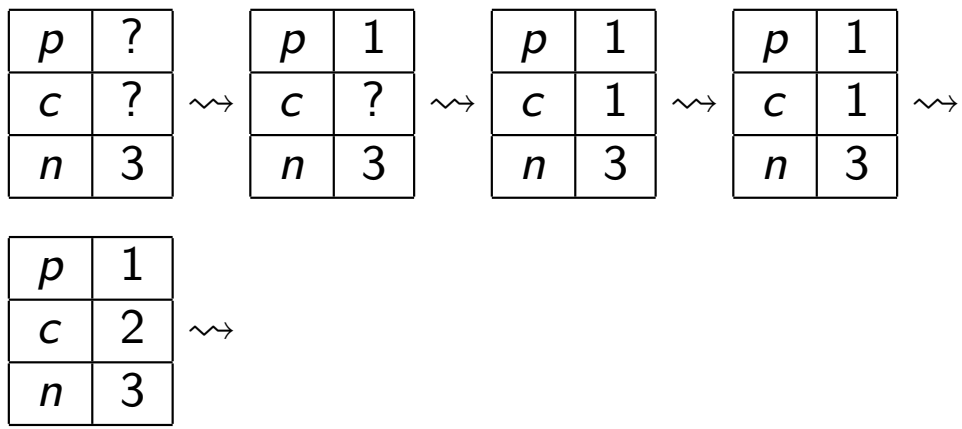
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

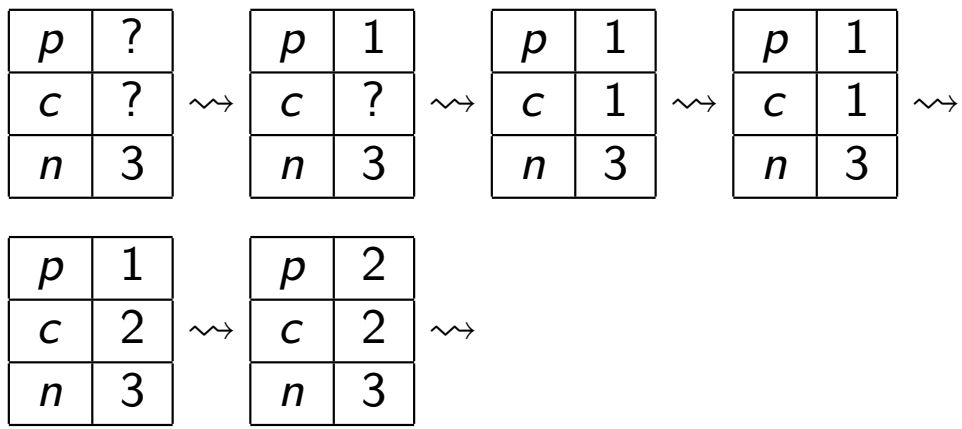
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

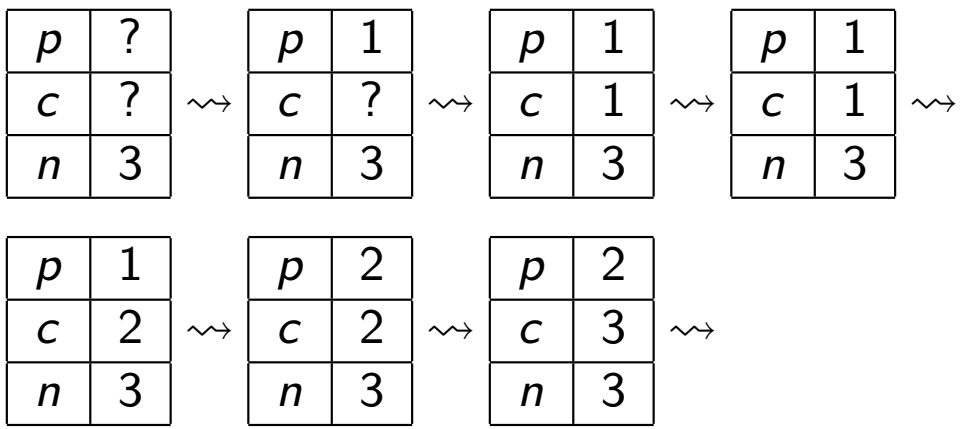
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

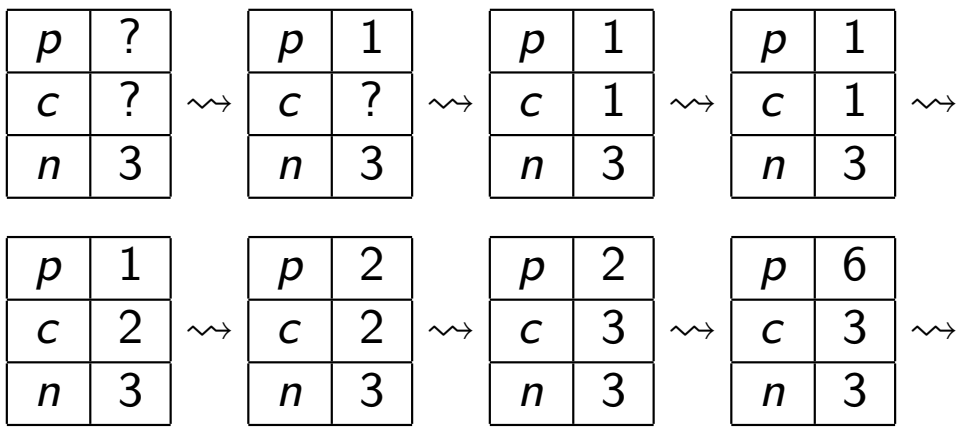
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

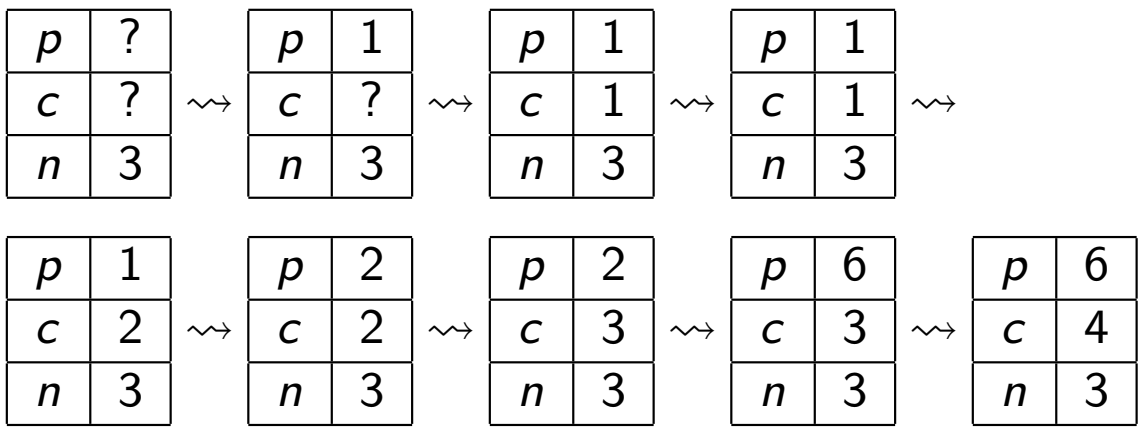
```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Operationale Semantik

- Kernkonzept: Zustandstandsübergänge einer abstrakten Maschine
- Abstrakte Maschine hat **impliziten Zustand**
- Zustand ordnet **Adressen** veränderliche **Werten** zu
- Konkretes Beispiel: $n \mapsto 3$, $p$ und $c$ undefiniert

```
p = 1;
c = 1;
while(c <= n){
   p = p * c;
   c = c + 1; }
```



# Denotationale Semantik

- Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- **Partielle** Funktionen zwischen Zuständen $[\![c]\!] : \sigma \rightharpoonup \sigma$
- Beispiel:

```
p = 1;
c = 1; // p1
while (c <= n) {
   p = p * c;
   c = c + 1; // p2
   }
// p3
```

$$[\![p_1]\!](\sigma) = \sigma(p \mapsto 1)(c \mapsto 1)$$
$$[\![p_2]\!](\sigma) = \sigma(p \mapsto \sigma(p) * \sigma(c))(c \mapsto \sigma(c) + 1)$$
$$[\![p_3]\!](\sigma) = ???$$

# Denotationale Semantik

- Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- **Partielle** Funktionen zwischen Zuständen $[\![c]\!] : \sigma \rightharpoonup \sigma$
- Beispiel:

```
p = 1;
c = 1; // p1
while (c <= n) {
   p = p * c;
   c = c + 1; // p2
   }
// p3
```

$$[\![p_1]\!](\sigma) = \sigma(p \mapsto 1)(c \mapsto 1)$$
$$[\![p_2]\!](\sigma) = \sigma(p \mapsto \sigma(p) * \sigma(c))(c \mapsto \sigma(c) + 1)$$
$$[\![p_3]\!](\sigma) = \mathit{fix}([\![p_2]\!])([\![p_1]\!](\sigma))$$

# Denotationale Semantik

- Kernkonzept: Abbildung von Programmen auf mathematisches Gegenstück (**Denotat**)
- **Partielle** Funktionen zwischen Zuständen $[\![c]\!] : \sigma \rightharpoonup \sigma$
- Beispiel:

```
p = 1;
c = 1; // p1
while (c <= n) {
   p = p * c;
   c = c + 1; // p2
   }
// p3
```

$$[\![p_1]\!](\sigma) = \sigma(p \mapsto 1)(c \mapsto 1)$$
$$[\![p_2]\!](\sigma) = \sigma(p \mapsto \sigma(p) * \sigma(c))(c \mapsto \sigma(c) + 1)$$
$$[\![p_3]\!] = \mathit{fix}([\![p_2]\!]) \circ [\![p_1]\!]$$

# Axiomatische Semantik

- Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- Zusicherungen sind zustandsabhängige Prädikate
- Beispiel (mit $n = 3$)

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
   // (4)
   p = p * c;
   c = c + 1; }
// (5)
```

(1) $n = 3$

(2) $p = 1 \wedge n = 3$

(3) $p = 1 \wedge c = 1 \wedge n = 3$

(4) ???

(5) $p = 6 \wedge c = 4 \wedge n = 3$

# Axiomatische Semantik

- Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- Zusicherungen sind zustandsabhängige Prädikate
- Beispiel (mit $n = 3$)

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
  // (4)
  p = p * c;
  c = c + 1; }
// (5)
```

(1) $n = 3$

(2) $p = 1 \wedge n = 3$

(3) $p = 1 \wedge c = 1 \wedge n = 3$

(4) $(p = 1 \wedge c = 1 \vee p = 1 \wedge c = 2 \vee$
$p = 2 \wedge c = 3 \vee p = 6 \wedge c = 4)$
$\wedge n = 3$

(5) $p = 6 \wedge c = 4 \wedge n = 3$

# Axiomatische Semantik

- Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- Zusicherungen sind zustandsabhängige Prädikate
- Beispiel (mit $n = 3$)

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
   // (4)
   p = p * c;
   c = c + 1; }
// (5)
```

(1) $n = 3$

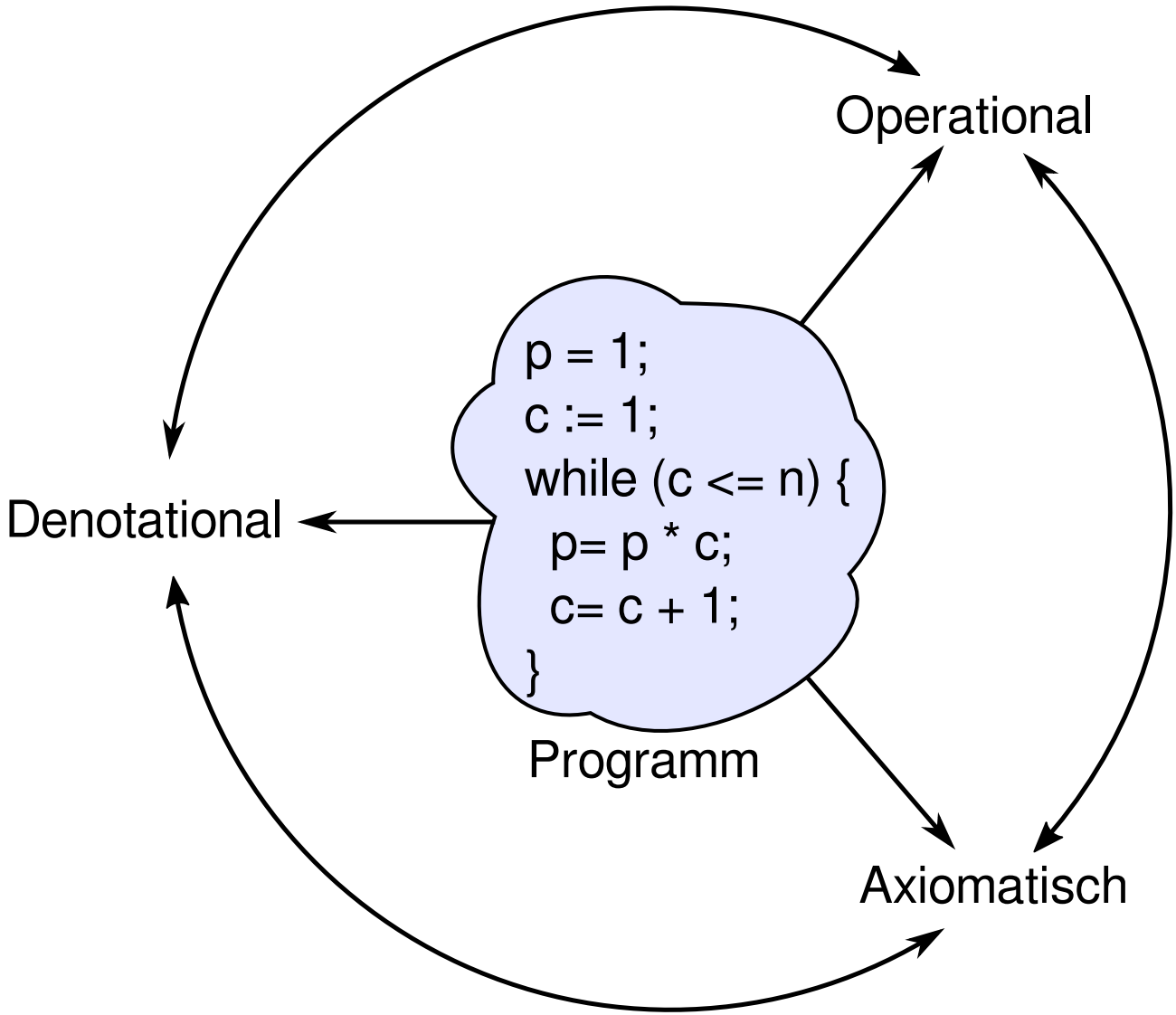
(2) $p = 1 \wedge n = 3$

(3) $p = 1 \wedge c = 1 \wedge n = 3$

(4) $p = (c - 1)! \wedge n = 3$

(5) $p = 6 \wedge c = 4 \wedge n = 3$

# Drei Semantiken — Eine Sicht



# Zusammenfassung

- Wir wollen die **Bedeutung** (Semantik) von Programmen beschreiben, um ihre Korrektheit beweisen zu können.
- Dazu gibt es verschiedene Ansätze, die wir betrachten werden.
- Nächste Woche geht es mit dem ersten los: **operationale** Semantik

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 2 vom 10.04.18: Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Zutaten

```
// GGT(A,B)
if (a == 0) r = b;
else {
  while (b != 0) {
    if (a <= b)
      b = b - a;
    else a = a - b;
  }
  r = a;
}
```

- Programme berechnen **Werte**
- Basierend auf
  - Werte sind **Variablen** zugewiesen
  - Evaluation von **Ausdrücken**
- Folgt dem Programmablauf

# Unsere Programmiersprache

Wir betrachten einen Ausschnitt der Programmiersprache **C** (**C0**).

Ausbaustufe 1 kennt folgende Konstrukte:

- Typen: **int**;
- Ausdrücke: Variablen, Literale (für ganze Zahlen), arithmetische Operatoren (für ganze Zahlen), Relationen ($==$, $<$, ...), boolsche Operatoren (&&, $||$);
- Anweisungen:
  - Fallunterscheidung (**if**...**else**...), Iteration (**while**), Zuweisung, Blöcke;
  - Sequenzierung und leere Anweisung sind implizit

# Semantik von C0

- Die (operationale) Semantik einer imperativen Sprache wie C0 ist ein **Zustandsübergang**: das System hat einen impliziten Zustand, der durch Zuweisung von **Werten** an **Adressen** geändert werden kann.

## Systemzustände

- Ausdrücke werten zu **Werten** $\mathbf{V}$ (hier ganze Zahlen) aus.
- Adressen $\mathbf{Loc}$ sind hier Programmvariablen (Namen)
- Ein **Systemzustand** bildet Adressen auf Werte ab: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$
- Ein Programm bildet einen Anfangszustand **möglicherweise** auf einen Endzustand ab (wenn es terminiert).
- Zusicherungen sind Prädikate über dem Systemzustand.

# C0: Ausdrücke und Anweisungen

$$
\begin{array}{rll}
\textbf{Aexp} & a ::= & \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \\
\textbf{Bexp} & b ::= & \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid ! \, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,||\, b_2 \\
\textbf{Exp} & e := & a \mid b \\
\textbf{Stmt} & c ::= & \mathbf{Idt} = \mathbf{Exp} \\
 & \mid & \textbf{if } (b) \; c_1 \; \textbf{ else } \; c_2 \\
 & \mid & \textbf{while } (b) \; c \\
 & \mid & c_1 ; c_2 \\
 & \mid & \{ \}
\end{array}
$$

NB: Nicht die **konkrete** Syntax.

# Eine Handvoll Beispiele

```
// {y = Y ∧ y ≥ 0}
x = 1;
while (y != 0) {
  y = y−1;
  x = 2∗x;
}
// {x = 2^Y}
```

```
// {a ≥ 0 ∧ b ≥ 0}
r = b;
q = 0;
while (b <= r) {
    r = r−a;
    q = q+1;
    }
// {a = b ∗ q + r ∧ r < b}
```

```
p = 1;
c = 1;
while (c<=n) {
    c = c+1;
    p = p∗c;
}
// {p = n!}
```

```
// {0 ≤ a}
t = 1;
s = 1;
i = 0;
while (s <= a) {
    t = t + 2;
    s = s + t;
    i = i + 1;
    }
// {i^2 ≤ a ∧ a < (i + 1)^2}
```

# Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck $a$ wertet unter gegebenen Zustand $\sigma$ zu einer ganzen Zahl $n$ (Wert) aus oder zu einem Fehler $\perp$.

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$
- Zustände bilden Adressen/Programmvariablen auf **Werte** ab ($\sigma$)

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n | \perp$$

# Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck $a$ wertet unter gegebenen Zustand $\sigma$ zu einer ganzen Zahl $n$ (Wert) aus oder zu einem Fehler $\perp$.

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$
- Zustände bilden Adressen/Programmvariablen auf **Werte** ab ($\sigma$)

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n | \perp$$

**Regeln:**

$$\frac{}{\langle n, \sigma \rangle \rightarrow_{Aexp} n}$$

# Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck $a$ wertet unter gegebenen Zustand $\sigma$ zu einer ganzen Zahl $n$ (Wert) aus oder zu einem Fehler $\perp$.

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$
- Zustände bilden Adressen/Programmvariablen auf **Werte** ab ($\sigma$)

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n | \perp$$

**Regeln:**

$$\frac{}{\langle n, \sigma \rangle \rightarrow_{Aexp} n}$$

$$\frac{x \in \mathbf{Loc}, x \in Dom(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \rightarrow_{Aexp} v} \qquad \frac{x \in \mathbf{Loc}, x \notin Dom(\sigma)}{\langle x, \sigma \rangle \rightarrow_{Aexp} \perp}$$

# Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$

$$\langle a, \sigma \rangle \to_{Aexp} n\bar{\perp}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbf{Z}, n \text{ Summe } n_1 \text{ und } n_2}{\langle a_1 + a_2, \sigma \rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad \text{falls } n_1 = \perp \text{ oder } n_2 = \perp}{\langle a_1 + a_2, \sigma \rangle \to_{Aexp} \perp}$$

# Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$

$$\langle a, \sigma \rangle \to_{Aexp} n\bar{\perp}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbf{Z}, n \text{ Summe } n_1 \text{ und } n_2}{\langle a_1 + a_2, \sigma \rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad \text{falls } n_1 = \perp \text{ oder } n_2 = \perp}{\langle a_1 + a_2, \sigma \rangle \to_{Aexp} \perp}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbf{Z}, n \text{ Diff. } n_1 \text{ und } n_2}{\langle a_1 - a_2, \sigma \rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad \text{falls } n_1 = \perp \text{ oder } n_2 = \perp}{\langle a_1 - a_2, \sigma \rangle \to_{Aexp} \perp}$$

# Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$

$$\langle a, \sigma \rangle \to_{Aexp} n | \bot$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbf{Z}, n \text{ Produkt } n_1 \text{ und } n_2}{\langle a_1 * a_2, \sigma \rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad \text{falls } n_1 = \bot \text{ oder } n_2 = \bot}{\langle a_1 * a_2, \sigma \rangle \to_{Aexp} \bot}$$

# Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$

$$\langle a, \sigma \rangle \to_{Aexp} n | \bot$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbf{Z}, n \text{ Produkt } n_1 \text{ und } n_2}{\langle a_1 * a_2, \sigma \rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad \text{falls } n_1 = \bot \text{ oder } n_2 = \bot}{\langle a_1 * a_2, \sigma \rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbf{Z}, n_2 \neq 0, n \text{ Quotient } n_1, n_2}{\langle a_1 / a_2, \sigma \rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad \text{falls } n_1 = \bot, n_2 = \bot \text{ oder } n_2 = 0}{\langle a_1 / a_2, \sigma \rangle \to_{Aexp} \bot}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\frac{}{\langle (x + y) * (x - y), \sigma \rangle \rightarrow_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\frac{\dfrac{}{\langle x + y, \sigma\rangle \to_{Aexp}} \qquad \dfrac{}{\langle x - y, \sigma\rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6}{\langle x + y, \sigma\rangle \to_{Aexp}} \qquad \dfrac{}{\langle x - y, \sigma\rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp}} \qquad \dfrac{}{\langle x - y, \sigma\rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{}{\langle x - y, \sigma\rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp} 11}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp} 11}$$

$$\dfrac{}{\langle (x * x) - (y * y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp} 11}$$

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle x, \sigma\rangle \to_{Aexp} 6}{\langle x * x, \sigma\rangle \to_{Aexp} 36}}{\langle (x * x) - (y * y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp} 11}$$

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle x, \sigma\rangle \to_{Aexp} 6}{\langle x * x, \sigma\rangle \to_{Aexp} 36} \qquad \dfrac{\langle y, \sigma\rangle \to_{Aexp} 5 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle y * y, \sigma\rangle \to_{Aexp} 25}}{\langle (x * x) - (y * y), \sigma\rangle \to_{Aexp}}$$

# Beispiel-Ableitungen

Sei $\sigma(x) = 6, \sigma(y) = 5$.

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x + y, \sigma\rangle \to_{Aexp} 11} \qquad \dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle x - y, \sigma\rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma\rangle \to_{Aexp} 11}$$

$$\dfrac{\dfrac{\langle x, \sigma\rangle \to_{Aexp} 6 \qquad \langle x, \sigma\rangle \to_{Aexp} 6}{\langle x * x, \sigma\rangle \to_{Aexp} 36} \qquad \dfrac{\langle y, \sigma\rangle \to_{Aexp} 5 \qquad \langle y, \sigma\rangle \to_{Aexp} 5}{\langle y * y, \sigma\rangle \to_{Aexp} 25}}{\langle (x * x) - (y * y), \sigma\rangle \to_{Aexp} 11}$$

# Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \mid\mid b_2$

**Regeln:**

$$\langle b, \sigma \rangle \to_{Bexp} 1|0|\bot$$

$$\frac{}{\langle \mathbf{1}, \sigma \rangle \to_{Bexp} \mathbf{1}} \qquad \frac{}{\langle \mathbf{0}, \sigma \rangle \to_{Bexp} \mathbf{0}}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 \text{ und } n_2 \text{ gleich}}{\langle a_1 == a_2, \sigma \rangle \to_{Bexp} 1}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 \text{ und } n_2 \text{ ungleich}}{\langle a_1 == a_2, \sigma \rangle \to_{Bexp} 0}$$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 == a_2, \sigma \rangle \to_{Bexp} \bot}$$

# Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \mid\mid b_2$

**Regeln:**

$$\langle b, \sigma \rangle \to_{Bexp} 1|0|\bot$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1}{\langle !b, \sigma \rangle \to_{Bexp} 0} \qquad \frac{\langle b, \sigma \rangle \to_{Bexp} 0}{\langle !b, \sigma \rangle \to_{Bexp} 1} \qquad \frac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle !b, \sigma \rangle \to_{Bexp} \bot}$$

$$\frac{\langle b_1, \sigma \rangle \to_{Bexp} t_1 \qquad \langle b_2, \sigma \rangle \to_{Bexp} t_2}{\langle b_1 \,\&\&\, b_2, \sigma \rangle \to_{Bexp} t}$$

wobei $t = 1$ wenn $t_1 = t_2 = 1$;
$t = 0$ wenn $t_1 = 0$ oder $(t_1 = 1$ und $t_2 = 0)$;
$t = \bot$ sonst

# Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \mid\mid b_2$

**Regeln:**

$$\langle b, \sigma \rangle \to_{Bexp} 1|0|\bot$$

$$\frac{\langle b_1, \sigma \rangle \to_{Bexp} t_1 \qquad \langle b_2, \sigma \rangle \to_{Bexp} t_2}{\langle b_1 \mid\mid b_2, \sigma \rangle \to_{Bexp} t}$$

wobei $t = 0$ wenn $t_1 = t_2 = 0$;
$t = 1$ wenn $t_1 = 1$ oder ($t_1 = 0$ und $t_2 = 1$);
$t = \bot$ sonst

# Operationale Semantik: Anweisungen

- **Stmt** $c ::= \textbf{Idt} = \textbf{Exp} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (b)\ c \mid c_1; c_2 \mid \{\}$

**Beispiel:**

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' | \bot$$

$$\langle x = 5, \sigma \rangle \rightarrow_{Stmt} \sigma'$$

wobei $\sigma'(x) = 5$ und $\sigma'(y) = \sigma(y)$ für alle $y \neq x$

# Operationale Semantik: Anweisungen

- **Stmt** $c ::= \textbf{Idt} = \textbf{Exp} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (b)\ c \mid c_1; c_2 \mid \{\}$

**Regeln:**

**Definiere:**

$$\sigma[m/x](y) := \begin{cases} m & \text{if } x = y \\ \sigma(y) & \text{sonst} \end{cases}$$

$$\langle x = 5, \sigma \rangle \rightarrow_{Stmt} \sigma[5/x]$$

**Es gilt:**

$$\begin{aligned} \forall \sigma, n, m, \forall x, y\ .\ x \neq y \Rightarrow \sigma[n/x][m/y] &= \sigma[m/y][n/x] \\ \forall \sigma, n, m, \forall x\ .\ \sigma[n/x][m/x] &= \sigma[m/x] \end{aligned}$$

# Operationale Semantik: Anweisungen

- **Stmt** $c ::= \textbf{Idt} = \textbf{Exp} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (b)\ c \mid c_1; c_2 \mid \{\}$

**Regeln:**

$$\langle \{\}, \sigma \rangle \to_{Stmt} \sigma$$

$$\frac{\langle a, \sigma \rangle \to_{Aexp} n \in \mathbf{Z}}{\langle x = a, \sigma \rangle \to_{Stmt} \sigma[n/x]} \qquad \frac{\langle a, \sigma \rangle \to_{Aexp} \bot}{\langle x = a, \sigma \rangle \to_{Stmt} \bot}$$

$$\frac{\langle c_1, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \qquad \langle c_2, \sigma' \rangle \to_{Stmt} \sigma'' \neq \bot}{\langle c_1; c_2, \sigma \rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle c_1, \sigma \rangle \to_{Stmt} \bot}{\langle c_1; c_2, \sigma \rangle \to_{Stmt} \bot}$$

$$\frac{\langle c_1, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \qquad \langle \{c_2\}, \sigma' \rangle \to_{Stmt} \bot}{\langle c_1; c_2, \sigma \rangle \to_{Stmt} \bot}$$

# Operationale Semantik: Anweisungen

- **Stmt** $c ::= \textbf{Idt} = \textbf{Exp} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (b)\ c \mid c_1; c_2 \mid \{\}$

**Regeln:**

$$\frac{\langle b, \sigma\rangle \to_{Bexp} 1 \qquad \langle c_1, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to_{Bexp} 0 \qquad \langle c_2, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to_{Bexp} \bot}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \bot}$$

# Operationale Semantik: Anweisungen

- **Stmt** $c ::= \textbf{Idt} = \textbf{Exp} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ (b)\ c \mid c_1; c_2 \mid \{\}$

**Regeln:**

$$\frac{\langle b, \sigma\rangle \to_{Bexp} 0}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \sigma}$$

$$\frac{\langle b, \sigma\rangle \to_{Bexp} 1 \qquad \langle c, \sigma\rangle \to_{Stmt} \sigma' \qquad \langle \textbf{while}\ (b)\ c, \sigma'\rangle \to_{Stmt} \sigma''}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle b, \sigma\rangle \to_{Bexp} 1 \qquad \langle c, \sigma\rangle \to_{Stmt} \bot}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \bot} \qquad \frac{\langle b, \sigma\rangle \to_{Bexp} \bot}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \bot}$$

# Beispiel

```
x = 1;
while (y != 0) {
  y = y - 1;
  x = 2 * x;
}
// x = 2^y
```

$\sigma(y) = 3$

# Äquivalenz arithmetischer Ausdrücke

Gegeben zwei Aexp $a_1$ and $a_2$

- Sind sie gleich?

$$a_1 \sim_{Aexp} a_2 \text{ gdw } \forall \sigma, n. \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow \langle a_2, \sigma \rangle \rightarrow_{Aexp} n$$

$$(x*x) + 2*x*y + (y*y) \qquad \text{und} \qquad (x+y) * (x+y)$$

- Wann sind sie gleich?

$$\exists \sigma, n. \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow \langle a_2, \sigma \rangle \rightarrow_{Aexp} n$$

$x*x$ und $9*x+22$
$x*x$ und $x*x+1$

# Äquivalenz Boolscher Ausdrücke

Gegeben zwei Bexp-Ausdrücke $b_1$ and $b_2$

- Sind sie gleich?

$$b_1 \sim_{Bexp} b_2 \text{ iff } \forall \sigma, b. \langle b_1, \sigma \rangle \rightarrow_{Bexp} b \Leftrightarrow \langle b_2, \sigma \rangle \rightarrow_{Bexp} b$$

A || (A && B) und A

# Beweisen

Zwei Programme $c_0, c_1$ sind äquivalent gdw. sie die gleichen Zustandsveränderungen bewirken. Formal definieren wir

**Definition**

$$c_0 \sim c_1 \text{ iff } \forall \sigma, \sigma'.\ \langle c_0, \sigma \rangle \to_{Stmt} \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \to_{Stmt} \sigma'$$

Ein einfaches Beispiel:

**Lemma**

*Sei* $w \equiv$ ***while*** $(b)\ c$ *mit* $b \in$ ***Bexp***, $c \in$ ***Stmt***.
*Dann gilt:* $w \sim$ ***if*** $(b)\ \{c; w\}$ ***else*** $\{\}$

Beweis an der Tafel

# Beweis

Gegeben beliebiger Programmzustand $\sigma$. Zu zeigen ist, dass sowohl $w$ also auch **if** $(b)$ $\{c; w\}$ **else** $\{\}$ zu dem selben Programmzustand auswerten oder beide zu einem Fehler. Der Beweis geht per Fallunterscheidung über die Auswertung von Teilausdrücken bzw. Teilprogrammen.

1. $\langle b, \sigma\rangle \to_{Bexp} 0$:

$$\langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma$$
$$\langle \textbf{if } (b)\ \{c; w\}\ \textbf{else}\ \{\}, \sigma\rangle \to_{Stmt} \langle \{\}, \sigma\rangle \to_{Stmt} \sigma$$

2. $\langle b, \sigma\rangle \to_{Bexp} 1$:
   1. $\langle c, \sigma\rangle \to_{Stmt} \sigma'$

$$\langle \overbrace{\textbf{while } (b)\ c}^{w}, \sigma\rangle \to_{Stmt} \langle c, \sigma\rangle \to_{Stmt} \sigma' \quad \langle w, \sigma'\rangle \to_{Stmt} \sigma''$$
$$\langle \textbf{if } (b)\ \{c; w\}\ \textbf{else}\ \{\}, \sigma\rangle \to_{Stmt} \langle \{c; w\}, \sigma\rangle \to_{Stmt} \langle c, \sigma\rangle \to_{Stmt} \sigma' \quad \langle w, \sigma'\rangle \to_{Stmt} \sigma''$$

# Beweis II

2. $\langle b, \sigma \rangle \to_{Bexp} 1$:

   2.2. $\langle c, \sigma \rangle \to_{Stmt} \bot$

$$\langle \overbrace{\textbf{while } (b)\ c}^{w}, \sigma \rangle \to_{Stmt} \langle c, \sigma \rangle \to_{Stmt} \bot$$

$$\langle \textbf{if } (b)\ \{c; w\}\ \textbf{else}\ \{\}, \sigma \rangle \to_{Stmt} \langle \{c; w\}, \sigma \rangle \to_{Stmt} \langle c, \sigma \rangle \to_{Stmt} \bot$$

3. $\langle b, \sigma \rangle \to_{Bexp} \bot$:

$$\langle \textbf{while } (b)\ c, \sigma \rangle \to_{Stmt} \bot$$

$$\langle \textbf{if } (b)\ \{c; w\}\ \textbf{else}\ \{\}, \sigma \rangle \to_{Stmt} \bot$$

# Zusammenfassung

- Operationale Semantik als ein Mittel zur Beschreibung der Semantik
- Auswertungsregeln arbeiten entlang der syntaktischen Struktur
- Werten Ausdrücke zu Werten aus und Programme zu Zuständen (zu gegebenen Zustand)
- Fragen zu Programmen: Gleichheit

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 3 vom 17.04.18: Denotationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Überblick

- Kleinster Fixpunkt
- Denotationale Semantik für C0

# Fixpunkt

- Sei $f : A \to A$ eine Funktion. Ein **Fixpunkt** von $f$ ist ein $a \in A$, so dass $f(a) = a$.
- Beispiele
  - Fixpunkte von $f(x) = \sqrt{x}$ sind 0 und 1; ebenfalls für $f(x) = x^2$.
  - Für die Sortierfunktion sind alle sortierten Listen Fixpunkte

# Regeln und Regelinstanzen

## Definition

Sei $R$ eine Menge von Regeln $\frac{x_1 \ \ldots x_n}{y}$, $n \geq 0$.
Die Anwendung einer Regel auf spezifische $a_1 \ldots a_n$ ist eine Regelinstanz

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Regelinstanzen sind

$$\frac{-}{4} \qquad \frac{-}{8} \qquad \frac{4 \quad 8}{32} \qquad \frac{4 \quad 4}{16}$$

$$\frac{16 \quad 32}{512} \qquad \frac{3 \quad 5}{15} \qquad \ldots$$

# Induktive Definierte Mengen

## Definition

Seit $R$ eine Menge von Regelinstanzen und $B$ eine Menge. Dann definieren wir

$$\hat{R}(B) = \{y \mid \exists x_1, \ldots, x_k \subseteq B.\ \frac{x_1, \ldots, x_k}{y} \in R\} \text{ und}$$

$$\hat{R}^0(B) = B \text{ und } \hat{R}^{i+1}(B) = \hat{R}(\hat{R}^i(B))$$

# Beispiel

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\begin{aligned}
\hat{R}^0(\emptyset) &= \emptyset \\
\hat{R}^1(\emptyset) &= \hat{R}(\emptyset) = \{4, 8\} \\
\hat{R}^2(\emptyset) &= ? \\
\hat{R}^3(\emptyset) &= ? \\
\hat{R}^{i+1}(\emptyset) &= ?
\end{aligned}$$

# Beispiel

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\begin{aligned}
\hat{R}^0(\emptyset) &= \emptyset \\
\hat{R}^1(\emptyset) &= \hat{R}(\emptyset) = \{4, 8\} \\
\hat{R}^2(\emptyset) &= \{16, 32, 64, 4, 8\} \\
\hat{R}^3(\emptyset) &= ? \\
\hat{R}^{i+1}(\emptyset) &= ?
\end{aligned}$$

# Beispiel

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\hat{R}^0(\emptyset) = \emptyset$$
$$\hat{R}^1(\emptyset) = \hat{R}(\emptyset) = \{4, 8\}$$
$$\hat{R}^2(\emptyset) = \{16, 32, 64, 4, 8\}$$
$$\hat{R}^3(\emptyset) = \{128, 256, 512, 1024, 2048, 4096, 16, 32, 64, 4, 8\}$$
$$\hat{R}^{i+1}(\emptyset) = ?$$

# Beispiel

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\begin{aligned}
\hat{R}^0(\emptyset) &= \emptyset \\
\hat{R}^1(\emptyset) &= \hat{R}(\emptyset) = \{4, 8\} \\
\hat{R}^2(\emptyset) &= \{16, 32, 64, 4, 8\} \\
\hat{R}^3(\emptyset) &= \{128, 256, 512, 1024, 2048, 4096, 16, 32, 64, 4, 8\} \\
\hat{R}^{i+1}(\emptyset) &= \{2^{2k+3l} | 1 \leq k + l \leq 2^i\}
\end{aligned}$$

# Induktive Definierte Mengen

## Definition

Seit $R$ eine Menge von Regelinstanzen und $B$ eine Menge. Dann definieren wir

$$\hat{R}(B) = \{y \mid \exists x_1, \ldots, x_k \subseteq B.\ \frac{x_1, \ldots, x_k}{y} \in R\} \text{ und}$$

$$\hat{R}^0(B) = B \text{ und } \hat{R}^{i+1}(B) = \hat{R}(\hat{R}^i(B))$$

## Definition (Abgeschlossen und Monoton)

- Eine Menge $S$ is **abgeschlossen unter $R$ ($R$-abgeschlossen)** gdw. $\hat{R}(S) \subseteq S$
- Eine Operation $f$ **ist monoton** gdw.

$$\forall A, B. A \subseteq B \Rightarrow f(A) \subseteq f(B)$$

# Kleinster Fixpunkt Operator

**Lemma**

*Für jede Menge von Regelinstanzen $R$ ist die induzierte Operation $\hat{R}$ monoton.*

**Lemma**

*Sei $A_i = \hat{R}^i(\emptyset)$ für alle $i \in \mathbb{N}$ und $A = \bigcup_{i \in \mathbb{N}} A_i$. Dann gilt*

(a) *$A$ ist $R$-abgeschlossen,*

(b) *$\hat{R}(A) = A$, und*

(c) *$A$ ist die kleinste $R$-abgeschlossene Menge.*

## Beweis von Lemma (a).

$A$ ist $R$-abgeschlossen:

Sei $\frac{x_1,\ldots,x_k}{y} \in R$ und $x_1, \ldots, x_k \subseteq A$.
Da $A = \bigcup_{i \in \mathbb{N}} A_i$ gibt es ein $j$ so dass $x_1, \ldots, x_k \subseteq A_j$.
Also auch:

$$\begin{aligned} y \in \hat{R}(A_j) &= \hat{R}(\hat{R}^j(\emptyset)) \\ &= \hat{R}^{j+1}(\emptyset) \\ &= A_{j+1} \subseteq A. \end{aligned}$$

## Beweis von Lemma (b): $\hat{R}(A) = A$.

- $\hat{R}(A) \subseteq A$:
  Da $A$ $R$-abgeschlossen gilt auch $\hat{R}(A) \subseteq A$.
- $A \subseteq \hat{R}(A)$:
  Sei $y \in A$. Dann $\exists n > 0.\ y \in A_n$ und $y \notin A_{n-1}$.
  Folglich muss es eine Regelinstanz $\frac{x_1,\ldots,x_k}{y} \in R$ geben mit $x_1, \ldots, x_k \subseteq A_{n-1} \subseteq A$.
  Da $\hat{R}$ monoton gilt $\hat{R}(A_{n-1}) \subseteq \hat{R}(A)$.
  Da $y \in A_n = \hat{R}(A_{n-1})$ folgt daraus $y \in \hat{R}(A)$.

## Beweis von Lemma (c).

$A$ ist die kleinste $R$-abgeschlossene Menge, d.h. für jede $R$-abgeschlossene Menge $B$ gilt $A \subseteq B$.

Beweis per Induktion über $n$ dass gilt $A_n \subseteq B$:

- Basisfall:
  $A_0 = \emptyset \subseteq B$
- Induktionsschritt:
  Da $B$ $R$-abgeschlossen ist gilt: $\hat{R}(B) \subseteq B$.
  Induktionsannahme: $A_n \subseteq B$.
  Dann gilt $A_{n+1} = \hat{R}(A_n) \subseteq \hat{R}(B) \subseteq B$ weil $\hat{R}$ monoton und $B$ ist $R$-abgeschlossen.

# Kleinster Fixpunkt Operator

**Definition**

$$fix(\hat{R}) = \bigcup_{n \in N} \hat{R}^n(\emptyset)$$

ist der **kleinste Fixpunkt**.

# Kleinster Fixpunkt

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\hat{R}^1(\emptyset) = \hat{R}(\emptyset) = \{4, 8\}$$
$$\hat{R}^2(\emptyset) = ?$$
$$\hat{R}^3(\emptyset) = ?$$
$$\hat{R}^{i+1}(\emptyset) = ?$$

# Kleinster Fixpunkt

- Betrachte folgende Regelmenge $R$

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\hat{R}^1(\emptyset) = \hat{R}(\emptyset) = \{4, 8\}$$
$$\hat{R}^2(\emptyset) = ?$$
$$\hat{R}^3(\emptyset) = ?$$
$$\hat{R}^{i+1}(\emptyset) = ?$$

- Wie sieht $\mathit{fix}(\hat{R})$ aus?

# Denotationale Semantik - Motivation

- **Operationale Semantik**
  Eine Menge von Regeln, die einen Zustand und ein Programm in einen neuen Zustand oder Fehler überführen

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' | \bot$$

- **Denotationale Semantik**
  Eine Menge von Regeln, die ein Programm in eine $\underbrace{\textbf{partielle Funktion}}_{\text{Denotat}}$ von Zustand nach Zustand überführen

$$\mathcal{C}[\![c]\!] : \Sigma \rightharpoonup \Sigma$$

# Denotationale Semantik - Motivation

Zwei Programme sind äquivalent gdw. sie immer zum selben Zustand (oder Fehler) auswerten

$$c_0 \sim c_1 \text{ iff } (\forall \sigma, \sigma'. \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma')$$

oder
Zwei Programme sind äquivalent gdw. sie die selbe partielle Funktion **denotieren**

$$c_0 \sim c_1 \text{ iff } \{(\sigma, \sigma') | \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma'\} = \{(\sigma, \sigma') | \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma'\}$$

# Denotierte Funktionen

- jeder $a$ :**Aexp** denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbf{Z}$
- jeder $b$ :**Bexp** denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbf{T}$
- jedes $c$ :**Stmt** denotiert eine partielle Funktion $\Sigma \rightharpoonup \Sigma$

# Denotat von Aexp

$$\mathcal{A}[\![a]\!] : \mathbf{Aexp} \to (\Sigma \rightharpoonup \mathbf{Z})$$

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n) \mid \sigma \in \Sigma\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in \mathit{Dom}(\sigma)\} \\
\mathcal{A}[\![a_0 + a_1]\!] &= \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 - a_1]\!] &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 * a_1]\!] &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 / a_1]\!] &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge n_1 \neq 0\}
\end{aligned}$$

# Denotat von Bexp

$$\mathcal{B}[\![a]\!] : \mathbf{Bexp} \to (\Sigma \rightharpoonup \mathbf{T})$$

$$\begin{aligned}
\mathcal{B}[\![1]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma\} \\
\mathcal{B}[\![0]\!] &= \{(\sigma, 0) \mid \sigma \in \Sigma\} \\
\mathcal{B}[\![a_0 == a_1]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 = n_1\} \\
&\quad \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 \neq n_1\} \\
\mathcal{B}[\![a_0 < a_1]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 < n_1\} \\
&\quad \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 \geq n_1\}
\end{aligned}$$

# Denotat von Bexp

$$\mathcal{B}[\![a]\!] : \mathbf{Bexp} \to (\Sigma \rightharpoonup \mathbf{T})$$

$$\begin{aligned}
\mathcal{B}[\![!b]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&\quad \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[\![b]\!]\} \\
\mathcal{B}[\![b_1 \;\&\&\; b2]\!] &= \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[\![b_1]\!]\} \\
&\quad \cup \{(\sigma, t_2) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[\![b_1]\!], (\sigma, t_2) \in \mathcal{B}[\![b_2]\!]\} \\
\mathcal{B}[\![b_1 \;||\; b_2]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[\![b_1]\!]\} \\
&\quad \cup \{(\sigma, t_2) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[\![b_1]\!], (\sigma, t_2) \in \mathcal{B}[\![b_2]\!]\}
\end{aligned}$$

# Denotat von Stmt

$$\mathcal{C}[\![.]\!] : \textbf{Stmt} \to (\Sigma \rightharpoonup \Sigma)$$

$$\begin{aligned}
\mathcal{C}[\![x = a]\!] &= \{(\sigma, \sigma[n/x]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![c_1; c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \qquad \text{Komposition von Relationen} \\
\mathcal{C}[\![\{\,\}]\!] &= \textbf{Id} \qquad\qquad\qquad\qquad \textbf{Id} := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
\mathcal{C}[\![\textbf{if } (b) \ c_0 \ \textbf{ else } \ c_1]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}
\end{aligned}$$

# Denotat von Stmt

$$\mathcal{C}[\![.]\!] : \textbf{Stmt} \to (\Sigma \rightharpoonup \Sigma)$$

$$\begin{aligned}
\mathcal{C}[\![x = a]\!] &= \{(\sigma, \sigma[n/x]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![c_1; c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \qquad \text{Komposition von Relationen} \\
\mathcal{C}[\![\{\,\}]\!] &= \textbf{Id} \qquad\qquad\qquad\qquad \textbf{Id} := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
\mathcal{C}[\![\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}
\end{aligned}$$

Aber was ist

$$\mathcal{C}[\![\textbf{while } (b)\ c]\!] = ??$$

# Denotationale Semantik für while

Sei $w \equiv$ **while** $(b)$ $c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\}$$

$$\begin{aligned} \mathcal{C}[\![w]\!] \quad = \quad & \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![\{c; w\}]\!]\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \end{aligned}$$

# Denotationale Semantik für while

Sei $w \equiv$ **while** $(b)$ $c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![\{c; w\}]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{aligned}$$

# Denotationale Semantik für while

Sei $w \equiv$ **while** $(b)$ $c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![\{c; w\}]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![w]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{aligned}$$

# Denotationale Semantik von while

Sei $w \equiv \textbf{while}\ (b)\ c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\,\}$$

$$\mathcal{C}[\![w]\!]_0 \quad = \quad \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!](\sigma)\}$$

# Denotationale Semantik von while

Sei $w \equiv \textbf{while}\ (b)\ c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\,\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!]_0 &= \{(\sigma,\sigma) \mid (\sigma,0) \in \mathcal{B}[\![b]\!](\sigma)\} \\
\mathcal{C}[\![w]\!]_1 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge\ (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_0\}
\end{aligned}$$

# Denotationale Semantik von while

Sei $w \equiv \textbf{while}\ (b)\ c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\,\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!]_0 &= \{(\sigma,\sigma) \mid (\sigma,0) \in \mathcal{B}[\![b]\!](\sigma)\} \\
\mathcal{C}[\![w]\!]_1 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad\qquad \wedge\, (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_0\} \\
\mathcal{C}[\![w]\!]_2 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad\qquad \wedge\, (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_1\} \\
&\vdots
\end{aligned}$$

# Denotationale Semantik von while

Sei $w \equiv \textbf{while}\ (b)\ c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\,\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!]_0 &= \{(\sigma,\sigma) \mid (\sigma,0) \in \mathcal{B}[\![b]\!](\sigma)\} \\
\mathcal{C}[\![w]\!]_1 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge\ (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_0\} \\
\mathcal{C}[\![w]\!]_2 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge\ (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_1\} \\
&\ \ \vdots \\
\mathcal{C}[\![w]\!]_{i+1} &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge\ (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_i\}
\end{aligned}$$

# Denotationale Semantik von while

Sei $w \equiv \textbf{while}\ (b)\ c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\,\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!]_0 &= \{(\sigma,\sigma) \mid (\sigma,0) \in \mathcal{B}[\![b]\!](\sigma)\} \\
\mathcal{C}[\![w]\!]_1 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_0\} \\
\mathcal{C}[\![w]\!]_2 &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_1\} \\
&\vdots \\
\mathcal{C}[\![w]\!]_{i+1} &= \{(\sigma,\sigma') \mid \exists\sigma''.\ (\sigma,1) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge (\sigma'',\sigma') \in \mathcal{C}[\![w]\!]_i\}
\end{aligned}$$

$$\begin{aligned}
\Gamma(\varphi) &= \{(\sigma,\sigma') \mid \exists\sigma''.\ \mathcal{B}[\![b]\!](\sigma) = 1 \wedge (\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'',\sigma') \in \varphi\} \\
&\quad \cup \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!](\sigma) = 0\}
\end{aligned}$$

# Denotationale Semantik von while

Sei $w \equiv$ **while** $(b)$ $c$ (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\}$$

$$\begin{aligned} \Gamma(\psi) \quad = \quad & \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in \psi\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \end{aligned}$$

$\Gamma$ ist wie $\hat{R}$, wobei $R$ definiert ist wie folgt:

$$\begin{aligned} R \quad = \quad & \left\{ \frac{(\sigma'', \sigma')}{(\sigma, \sigma')} \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \right\} \\ & \cup \left\{ \frac{}{(\sigma, \sigma)} \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \right\} \end{aligned}$$

und die Semantik von $w$ ist der Fixpunkt von $\Gamma$, d.h. $\mathcal{C}[\![w]\!] = \mathit{fix}(\Gamma)$

# Denotation für Stmt

$$\mathcal{C}[\![.]\!] : \textbf{Stmt} \to (\Sigma \rightharpoonup \Sigma)$$

$$\begin{aligned}
\mathcal{C}[\![x = a]\!] &= \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![c_1; c_2]\!] &= \mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!] \qquad \text{Komposition von Relationen} \\
\mathcal{C}[\![\{\,\}]\!] &= \textbf{Id} \qquad\qquad\qquad\qquad \textbf{Id} := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
\mathcal{C}[\![\textbf{if } (b) \ c_0 \ \textbf{ else } \ c_1]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
\mathcal{C}[\![\textbf{while } (b) \ c]\!] &= \mathit{fix}(\Gamma)
\end{aligned}$$

mit

$$\begin{aligned}
\Gamma(\psi) &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \psi \circ \mathcal{C}[\![c]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{aligned}$$

# Weitere Intuition zur Fixpunkt Konstruktion

- Sei $w \equiv \textbf{while}\ (b)\ c$
- Zur Erinnerung: Wir haben begonnen mit $w \sim \textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\}$
- Dann müsste auch gelten

$$\mathcal{C}[\![w]\!] \stackrel{!}{=} \mathcal{C}[\![\textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\}]\!]$$

- Beweis an der Tafel

# Beweis $\mathcal{C}[\![w]\!] \stackrel{!}{=} \mathcal{C}[\![\textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\ \}]\!]$

$$
\begin{aligned}
\mathcal{C}[\![w]\!] &= \mathit{fix}(\Gamma) \\
&= \Gamma(\mathit{fix}(\Gamma)) \\
&= \Gamma(\mathcal{C}[\![w]\!]) \\
&= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c;\ w]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \mathcal{C}[\![\textbf{if } (b) \ \{c; w\} \ \textbf{else } \{\ \}]\!]
\end{aligned}
$$

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 4 vom 24.04.18: Äquivalenz der Operationalen und Denotationalen Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Operationale vs. denotationale Semantik

| | **Operational** $\langle a, \sigma \rangle \to_{Aexp} n$ | **Denotational** $\mathcal{A}[\![a]\!]$ |
|---|---|---|
| m$\in$ **Z** | $\langle m, \sigma \rangle \to_{Aexp} m$ | $\{(\sigma, m) \mid \sigma \in \Sigma\}$ |
| x$\in$ **Loc** | $\dfrac{x \in Dom(\sigma)}{\langle x, \sigma \rangle \to_{Aexp} \sigma(x)}$ | $\{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in Dom(\sigma)\}$ |
| $a_1 \circ a_2$ | $\dfrac{\langle a_1, \sigma \rangle \to_{Aexp} n \quad \langle a_2, \sigma \rangle \to_{Aexp} m \quad n, m \neq \bot}{\langle a_1 \circ a_2, \sigma \rangle \to_{Aexp} n \circ^I m}$ | $\{(\sigma, n \circ^I m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{A}[\![a_1]\!], (\sigma, m) \in \mathcal{A}[\![a_2]\!]\}$ |

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n \quad \langle a_2, \sigma \rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_1 \circ a_2, \sigma \rangle \to_{Aexp} \bot} \quad \circ \in \{+, *, -\}$$

# Operationale vs. denotationale Semantik

| | **Operational** $\langle a, \sigma \rangle \to_{Aexp} n$ | **Denotational** $\mathcal{A}[\![a]\!]$ |
|---|---|---|
| $a_1 / a_2$ | $\dfrac{\langle a_1, \sigma \rangle \to_{Aexp} n \quad \langle a_2, \sigma \rangle \to_{Aexp} m \quad m \neq 0 \quad m, n \neq \perp}{\langle a_1 \circ a_2, \sigma \rangle \to_{Aexp} n \circ' m}$ | $\{(\sigma, n/m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{A}[\![a_1]\!], (\sigma, m) \in \mathcal{A}[\![a_2]\!], m \neq 0\}$ |

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n \quad \langle a_2, \sigma \rangle \to_{Aexp} m \quad n = \perp, m = \perp \text{ oder } m = 0}{\langle a_1 / a_2, \sigma \rangle \to_{Aexp} \perp}$$

# Äquivalenz operationale und denotationale Semantik

- Für alle $a \in \mathbf{Aexp}$, für alle $n \in \mathbf{Z}$, für alle Zustände $\sigma$:

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in \mathcal{A}[\![a]\!]$$

$$\langle a, \sigma \rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{A}[\![a]\!])$$

- Beweis Prinzip?

# Äquivalenz operationale und denotationale Semantik

- Für alle $a \in \mathbf{Aexp}$, für alle $n \in \mathbf{Z}$, für alle Zustände $\sigma$:

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in \mathcal{A}[\![a]\!]$$

$$\langle a, \sigma \rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{A}[\![a]\!])$$

- Beweis per struktureller Induktion über $a$. (Warum?)

# Operationale vs. denotationale Semantik

| | **Operational** $\langle b, \sigma \rangle \to_{Bexp} 0\vert 1$ | **Denotational** $\mathcal{B}[\![b]\!]$ |
|---|---|---|
| 1 | $\langle 1, \sigma \rangle \to_{Bexp} 1$ | $\{(\sigma, 1) \vert \sigma \in \Sigma\}$ |
| 0 | $\langle 0, \sigma \rangle \to_{Bexp} 0$ | $\{(\sigma, 0) \vert \sigma \in \Sigma\}$ |

# Operationale vs. denotationale Semantik

**Operat.** $\langle b, \sigma \rangle \to_{Bexp} 0|1$ **Denotational** $\mathcal{B}[\![b]\!]$

$a_0 == a_1$

$$\frac{\langle a_0, \sigma \rangle \to_{Aexp} n \quad \langle a_1, \sigma \rangle \to_{Aexp} m \quad n, m \neq \bot \quad n = m}{\langle a_0 == a_1, \sigma \rangle \to_{Bexp} 1}$$

$$\frac{\langle a_0, \sigma \rangle \to_{Aexp} n \quad \langle a_1, \sigma \rangle \to_{Aexp} m \quad n, m \neq \bot \quad n \neq m}{\langle a_0 == a_1, \sigma \rangle \to_{Bexp} 0}$$

$$\frac{\langle a_0, \sigma \rangle \to_{Aexp} n \quad \langle a_1, \sigma \rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_0 == a_1, \sigma \rangle \to_{Bexp} \bot}$$

$$\{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 = n_1 \} \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 \neq n_1 \}$$

$a1 <= a2$ analog

# Operationale vs. denotationale Semantik

| | **Operational** $\langle a, \sigma \rangle \to_{Bexp} b$ | **Denotational** $\mathcal{B}[\![b]\!]$ |
|---|---|---|
| $b_1 \&\& b_0$ | $\dfrac{\langle b_1, \sigma \rangle \to_{Bexp} 0}{\langle b_1 \&\& b_2, \sigma \rangle \to 0}$ | $\{(\sigma, 0) \mid (\sigma, 0) \in \mathcal{B}[\![b_1]\!]\}$ |
| | $\dfrac{\langle b_1, \sigma \rangle \to_{Bexp} 1 \quad \langle b_2, \sigma \rangle \to_{Bexp} b}{\langle b_1 \&\& b_2, \sigma \rangle \to b}$ | $\{(\sigma, b) \mid (\sigma, 1) \in \mathcal{B}[\![b_1]\!], (\sigma, b) \in \mathcal{B}[\![b_2]\!]\}$ |
| | $\dfrac{\langle b_1, \sigma \rangle \to_{Bexp} \bot}{\langle b_1 \&\& b_2, \sigma \rangle \to \bot}$ | |
| $b_1 \mid\mid b_2$ | | analog |
| $!n$ | ... | |

# Äquivalenz operationale und denotationale Semantik

- Für alle $b \in \mathbf{Bexp}$, für alle $t \in \mathbf{B}$, for alle Zustände $\sigma$:

$$\langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in \mathcal{B}[\![b]\!]$$

$$\langle b, \sigma \rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{B}[\![b]\!])$$

- Beweis Prinzip?

# Äquivalenz operationale und denotationale Semantik

- Für alle $b \in \mathbf{Bexp}$, für alle $t \in \mathbf{B}$, for alle Zustände $\sigma$:

$$\langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in \mathcal{B}[\![b]\!]$$

$$\langle b, \sigma \rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{B}[\![b]\!])$$

- Beweis per struktureller Induktion über $b$ (unter Verwendung der Äquivalenz für AExp). (Warum?)

# Operationale vs. denotationale Semantik

| | Operational | Denotational $\mathcal{C}[\![c]\!]$ |
|---|---|---|
| $\{\}$ | $\dfrac{\langle c,\sigma\rangle \to_{Stmt} \sigma' \mid \bot}{\langle \{\},\sigma\rangle \to_{Stmt} \sigma}$ | $\mathcal{C}[\![\{\}]\!] = Id$ |
| $c_1; c_2$ | $\dfrac{\langle c_1,\sigma\rangle \to_{Stmt} \sigma' \neq \bot \quad \langle c_2,\sigma'\rangle \to_{Stmt} \sigma''}{\langle c_1; c_2,\sigma\rangle \to_{Stmt} \sigma''}$ <br> $\dfrac{\langle c_1,\sigma\rangle \to_{Stmt} \bot}{\langle c_1; c_2,\sigma\rangle \to_{Stmt} \bot}$ | $\mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!]$ |
| $x = a$ | $\dfrac{\langle a,\sigma\rangle \to_{Aexp} n}{\langle x = a,\sigma\rangle \to_{Stmt} \sigma[n/x]}$ <br> $\dfrac{\langle a,\sigma\rangle \to_{Aexp} \bot}{\langle x = a,\sigma\rangle \to_{Stmt} \bot}$ | $\{(\sigma,\sigma[n/x]) \mid (\sigma,n) \in \mathcal{A}[\![a]\!]\}$ |

# Operationale vs. denotationale Semantik

| | Operational $\langle c, \sigma\rangle \to_{Stmt} \sigma' \mid \bot$ | Denotational $\mathcal{C}[\![c]\!]$ |
|---|---|---|
| **if** $(b)$ $c_0$ | $\dfrac{\langle b, \sigma\rangle \to_{Bexp} 1 \quad \langle c_0, \sigma\rangle \to_{Stmt} \sigma'}{\langle c, \sigma\rangle \to_{Stmt} \sigma'}$ $\dfrac{\langle b, \sigma\rangle \to_{Bexp} \bot}{\langle c, \sigma\rangle \to_{Stmt} \bot}$ | $\{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\}$ |
| **else** $c_1$ | $\dfrac{\langle b, \sigma\rangle \to_{Bexp} 0 \quad \langle c_1, \sigma\rangle \to_{Stmt} \sigma'}{\langle c, \sigma\rangle \to_{Stmt} \sigma'}$ | $\{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}$ |

# Operationale vs. denotationale Semantik

**Operational** $\langle c, \sigma \rangle \to_{Stmt} \sigma' | \bot$ **Denotational** $\mathcal{C}[\![c]\!]$

$\underbrace{\textbf{while } (b)\ c}_{w}$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 0}{\langle w, \sigma \rangle \to_{Stmt} \sigma} \qquad \frac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle w, \sigma \rangle \to_{Stmt} \bot}$$

$fix(\Gamma)$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \quad \langle w, \sigma' \rangle \to_{Stmt} \sigma''}{\langle w, \sigma \rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c, \sigma \rangle \to_{Stmt} \bot}{\langle w, \sigma \rangle \to_{Stmt} \bot}$$

mit

$$\begin{aligned} \Gamma(\varphi) \quad = \quad & \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \varphi \circ \mathcal{C}[\![c]\!]\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \end{aligned}$$

# Äquivalenz operationale und denotationale Semantik

- Für alle $c \in \mathbf{Stmt}$, für alle Zustände $\sigma, \sigma'$:

$$\langle c, \sigma \rangle \to_{Stmt} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[\![c]\!]$$

$$\langle c, \sigma \rangle \to_{Stmt} \bot \Rightarrow \sigma \notin Dom(\mathcal{C}[\![c]\!])$$

- $\Rightarrow$ Beweis Prinzip?
- $\Leftarrow$ Beweis Prinzip?

# Äquivalenz operationale und denotationale Semantik

- Für alle $c \in \textbf{Stmt}$, für alle Zustände $\sigma, \sigma'$:

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[\![c]\!]$$

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \bot \Rightarrow \sigma \notin Dom(\mathcal{C}[\![c]\!])$$

- $\Rightarrow$ Beweis per Induktion über die Ableitung in der operationalen Semantik (Warum?)

- $\Leftarrow$ Beweis Prinzip?

# Äquivalenz operationale und denotationale Semantik

- Für alle $c \in \mathbf{Stmt}$, für alle Zustände $\sigma, \sigma'$:

$$\langle c, \sigma \rangle \to_{Stmt} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[\![c]\!]$$

$$\langle c, \sigma \rangle \to_{Stmt} \bot \Rightarrow \sigma \notin Dom(\mathcal{C}[\![c]\!])$$

- $\Rightarrow$ Beweis per Induktion über die Ableitung in der operationalen Semantik (Warum?)

- $\Leftarrow$ Beweis per struktureller Induktion über $c$ (Verwendung der Äquivalenz für arithmetische und boolsche Ausdrücke). Für die While-Schleife Rückgriff auf Definition des Fixpunkts und Induktion über die Teilmengen $\Gamma^i(\emptyset)$ des Fixpunkts. (Warum?)

# Äquivalenz operationale und denotationale Semantik

- Für alle $c \in \mathbf{Stmt}$, für alle Zustände $\sigma, \sigma'$:

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[\![c]\!]$$

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \bot \Rightarrow \sigma \notin Dom(\mathcal{C}[\![c]\!])$$

- $\Rightarrow$ Beweis per Induktion über die Ableitung in der operationalen Semantik (Warum?)
- $\Leftarrow$ Beweis per struktureller Induktion über $c$ (Verwendung der Äquivalenz für arithmetische und boolsche Ausdrücke). Für die While-Schleife Rückgriff auf Definition des Fixpunkts und Induktion über die Teilmengen $\Gamma^i(\emptyset)$ des Fixpunkts. (Warum?)
  - Gegenbeispiel für $\Leftarrow$ in der zweiten Aussage: wähle $c \equiv \mathit{while}(1)\{\}$: $\mathcal{C}[\![c]\!] = \emptyset$ aber $\langle c, \sigma \rangle \rightarrow_{Stmt} \bot$ gilt nicht (sondern?).

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 5 vom 03.05.18: Die Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

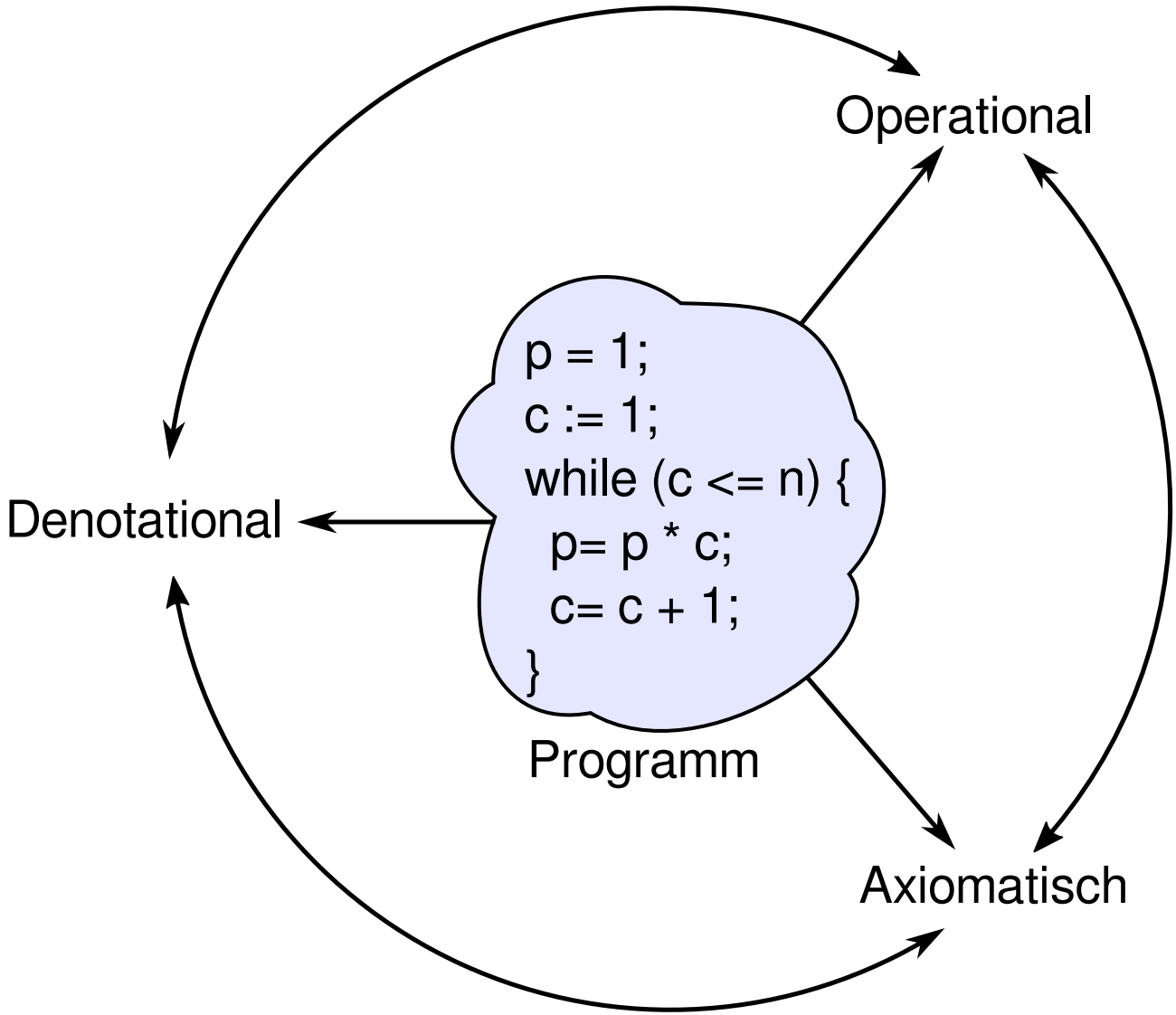
Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Drei Semantiken — Eine Sicht



# Floyd-Hoare-Logik: Idee

- Was wird hier berechnet?

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

# Floyd-Hoare-Logik: Idee

- Was wird hier berechnet? $p = n!$
- Warum? Wie können wir das **beweisen**?

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

# Floyd-Hoare-Logik: Idee

- Was wird hier berechnet? $p = n!$
- Warum? Wie können wir das **beweisen**?
- Wir berechnen symbolisch, welche Werte Variablen über den Programmverlauf annehmen.

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

# Floyd-Hoare-Logik: Idee

- Was wird hier berechnet? $p = n!$
- Warum? Wie können wir das **beweisen**?

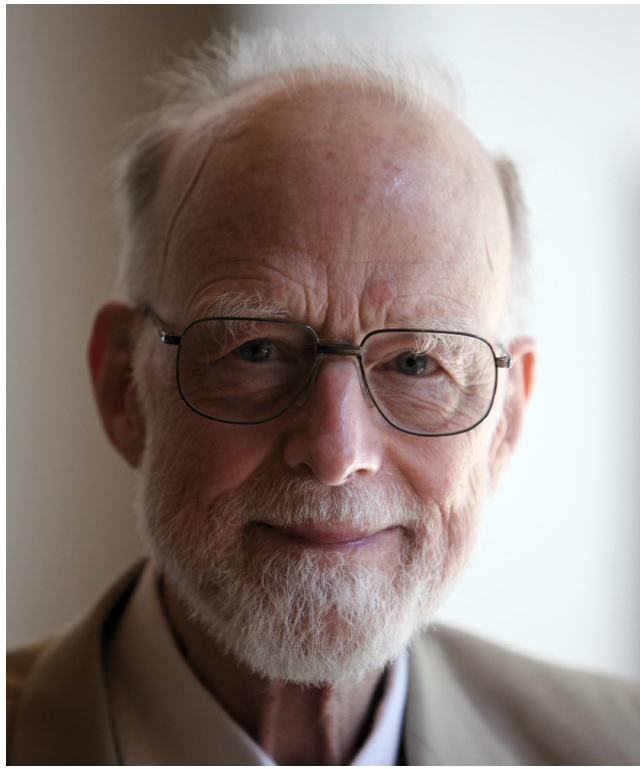
```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

- Operationale/denotionale Semantik nicht für **Korrektheitsbeweise** geeignet: Ausdrücke werden zu groß, skaliert nicht.
- **Abstraktion** nötig.
- Grundidee: **Zusicherungen** über den Zustand an bestimmten Punkten im Programmablauf.

# Bob Floyd und Tony Hoare

Bildquelle: Stanford University

Robert Floyd
1936 – 2001



Bildquelle: Wikipedia

Sir Anthony Charles Richard Hoare
* 1934

# Grundbausteine der Floyd-Hoare-Logik

- **Zusicherungen** über den Zustand
- Beispiele:
  - (B): Hier gilt $p = c = 1$
  - (D): Hier ist $c$ ist um eines größer als der Wert von $c$ an Punkt (C)
- Gesamtaussage: Wenn am Punkt(A) der Wert von $n \geq 0$, dann ist am Punkt (E) $p = n!$.

```
// (A)
p= 1;
c= 1;
// (B)
while (c <= n) {
   p= p * c;
   // (C)
   c= c + 1;
   // (D)
}
// (E)
```

# Grundbausteine der Floyd-Hoare-Logik

- **Logische Variablen** (zustandsfrei) und **Programmvariablen**
- **Zusicherungen** mit logischen und Programmvariablen
- **Floyd-Hoare-Tripel** $\{P\}\, c\, \{Q\}$
  - Vorbedingung $P$ (Zusicherung)
  - Programm $c$
  - Nachbedingung $Q$ (Zusicherung)
- Floyd-Hoare-Logik abstrahiert Programme durch logische Formeln.

# Zusicherungen (Assertions)

- Erweiterung von **Aexp** and **Bexp** durch
  - **Logische** Variablen **Var** $v := N, M, L, U, V, X, Y, Z$
  - Definierte Funktionen und Prädikate über **Aexp** $n!, \sum_{i=1}^{n} i, \ldots$
  - Implikation und Quantoren $b_1 \Rightarrow b_2, \textbf{forall}\, v.\, b, \textbf{exists}\, v.\, b$
- Formal:

$$\begin{aligned} \textbf{Aexpv}\ a ::= {} & \textbf{Z} \mid \textbf{Idt} \mid \textbf{Var} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\ & \mid f(e_1, \ldots, e_n) \end{aligned}$$

$$\begin{aligned} \textbf{Assn}\ b ::= {} & \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2 \\ & \mid !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\ & \mid b_1 \Rightarrow b_2 \mid p(e_1, \ldots, e_n) \mid \textbf{forall}\, v;\ b \mid \textbf{exists}\, v;\ b \end{aligned}$$

# Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand $\sigma$?
  - Auswertung (denotationale Semantik) ergibt *true*
  - **Aber:** was ist mit den logischen Variablen?

# Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand $\sigma$?
  - Auswertung (denotationale Semantik) ergibt *true*
  - **Aber:** was ist mit den logischen Variablen?
- **Belegung** der logischen Variablen: $I : \mathbf{Var} \to \mathbf{T}$
- Semantik von $b$ unter der Belegung $I$: $\mathcal{B}_v[\![b]\!]^I, \mathcal{A}_v[\![a]\!]^I$

**Erfülltheit von Zusicherungen**

$b \in \mathbf{Assn}$ ist in Zustand $\sigma$ mit Belegung $I$ erfüllt ($\sigma \models^I b$), gdw

$$\mathcal{B}_v[\![b]\!]^I(\sigma) = \mathit{true}$$

# Floyd-Hoare-Tripel

## Partielle Korrektheit ($\models \{P\}\, c\, \{Q\}$)

$c$ ist **partiell korrekt**, wenn für alle Zustände $\sigma$, die $P$ erfüllen, gilt: **wenn** die Ausführung von $c$ mit $\sigma$ in $\sigma'$ terminiert, **dann** erfüllt $\sigma'$ $Q$.

$$\models \{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I P \land \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \implies \sigma' \models^I Q$$

- Gleiche Belegung der logischen Variablen für $P$ und $Q$.

## Totale Korrektheit ($\models [P]\, c\, [Q]$)

$c$ ist **total korrekt**, wenn für alle Zustande $\sigma$, die $P$ erfüllen, die Ausführung von $c$ mit $\sigma$ in $\sigma'$ terminiert, und $\sigma'$ erfüllt $Q$.

$$\models [P]\, c\, [Q] \iff \forall I.\, \forall \sigma.\, \sigma \models^I P \implies \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \land \sigma' \models^I Q$$

- Folgendes **gilt**: $\models \{\mathbf{1}\}\ \mathbf{while}(1)\{\ \}\ \{\mathbf{1}\}$
- Folgendes gilt **nicht**: $\models [\mathbf{1}]\ \mathbf{while}(1)\{\ \}\ [\mathbf{1}]$

# Regeln der Floyd-Hoare-Logik

- Die Floyd-Hoare-Logik erlaubt es, Zusicherungen der Form $\vdash \{P\}\, c\, \{Q\}$ syntaktisch **herzuleiten**.

- Der **Kalkül** der Logik besteht aus sechs Regeln der Form

$$\frac{\vdash \{P_1\}\, c_1\, \{Q_1\} \ldots \vdash \{P_n\}\, c_n\, \{Q_n\}}{\vdash \{P\}\, c\, \{Q\}}$$

- Für jedes Konstrukt der Programmiersprache gibt es eine Regel.

# Regeln des Floyd-Hoare-Kalküls: Zuweisung

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Eine Zuweisung x=e ändert den Zustand so dass an der Stelle $x$ jetzt der Wert von $e$ steht. Damit **nachher** das Prädikat $P$ gilt, muss also **vorher** das Prädikat gelten, wenn wir $x$ durch $e$ ersetzen.
- Es ist völlig normal (aber dennoch falsch) zu denken, die Substitution gehöre eigentlich in die Nachbedingung.
- Beispiele:

x = 5
$\{x < 10\}$

x = x+ 1

# Regeln des Floyd-Hoare-Kalküls: Zuweisung

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Eine Zuweisung x=e ändert den Zustand so dass an der Stelle $x$ jetzt der Wert von $e$ steht. Damit **nachher** das Prädikat $P$ gilt, muss also **vorher** das Prädikat gelten, wenn wir $x$ durch $e$ ersetzen.
- Es ist völlig normal (aber dennoch falsch) zu denken, die Substitution gehöre eigentlich in die Nachbedingung.
- Beispiele:

$\{5 < 10 \iff (x < 10)[5/x]\}$
x = 5
$\{x < 10\}$

x = x+ 1

# Regeln des Floyd-Hoare-Kalküls: Zuweisung

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Eine Zuweisung x=e ändert den Zustand so dass an der Stelle $x$ jetzt der Wert von $e$ steht. Damit **nachher** das Prädikat $P$ gilt, muss also **vorher** das Prädikat gelten, wenn wir $x$ durch $e$ ersetzen.
- Es ist völlig normal (aber dennoch falsch) zu denken, die Substitution gehöre eigentlich in die Nachbedingung.
- Beispiele:

$\{5 < 10 \iff (x < 10)[5/x]\}$
x = 5
$\{x < 10\}$

x = x+ 1
$\{x < 10\}$

# Regeln des Floyd-Hoare-Kalküls: Zuweisung

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Eine Zuweisung x=e ändert den Zustand so dass an der Stelle $x$ jetzt der Wert von $e$ steht. Damit **nachher** das Prädikat $P$ gilt, muss also **vorher** das Prädikat gelten, wenn wir $x$ durch $e$ ersetzen.
- Es ist völlig normal (aber dennoch falsch) zu denken, die Substitution gehöre eigentlich in die Nachbedingung.
- Beispiele:

$\{5 < 10 \iff (x < 10)[5/x]\}$
x = 5
$\{x < 10\}$

$\{x < 9 \iff x + 1 < 10\}$
x = x+ 1
$\{x < 10\}$

# Regeln des Floyd-Hoare-Kalküls: Fallunterscheidung

$$\frac{\vdash \{A \,\&\&\, b\}\, c_0\, \{B\} \qquad \vdash \{A \,\&\&\, \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

- In der Vorbedingung des **if**-Zweiges gilt die Bedingung $b$, und im **else**-Zweig gilt die Negation $\neg b$.

- Beide Zweige müssem mit derselben Nachbedingung enden.

# Regeln des Floyd-Hoare-Kalküls: Iteration

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

- Iteration korrespondiert zu **Induktion**.
- Bei (natürlicher) Induktion zeigen wir, dass die **gleiche** Eigenschaft $P$ für 0 gilt, und dass wenn sie für $P(n)$ gilt, daraus folgt, dass sie für $P(n+1)$ gilt.
- Analog dazu benötigen wir hier eine **Invariante** $A$, die sowohl **vor** als auch **nach** dem Schleifenrumpf gilt.
- In der Vorbedingung des Schleifenrumpfes können wir die Schleifenbedingung $b$ annehmen.
- Die **Vorbedingung** der **Schleife** ist die Invariante $A$, und die **Nachbedingung** der **Schleife** ist $A$ und die Negation der Schleifenbedingung $b$.

# Regeln des Floyd-Hoare-Kalküls: Sequenzierung

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

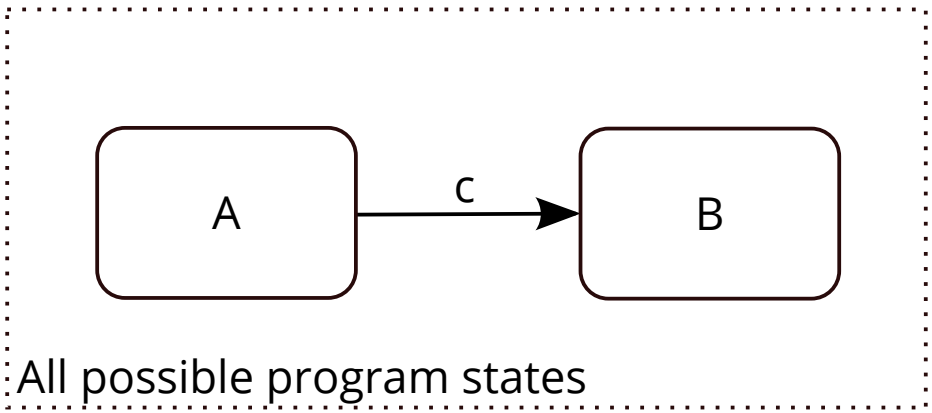
- Hier wird eine Zwischenzusicherung $B$ benötigt.

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}}$$

- Trivial.

# Regeln des Floyd-Hoare-Kalküls: Weakening

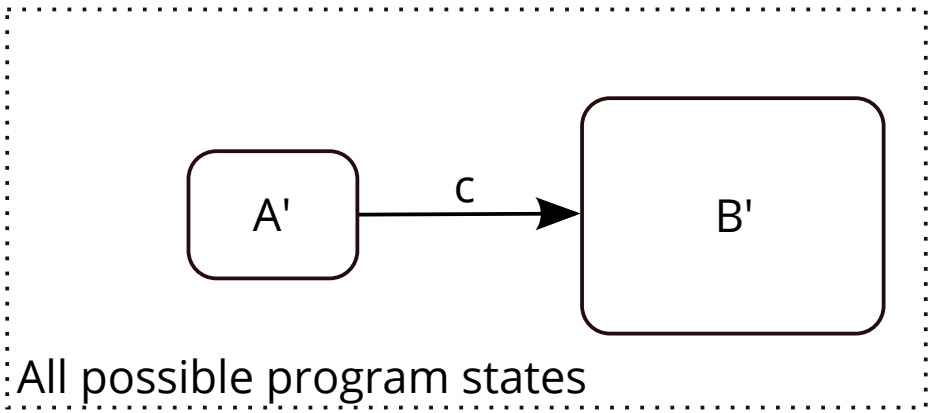
$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$




- $\models \{A\}\, c\, \{B\}$: Ausführung von $c$ startet in Zustand, in dem $A$ gilt, und endet (ggf) in Zustand, in dem $B$ gilt.
- Zustandsprädikate beschreiben Mengen von Zuständen: $P \subseteq Q$ gdw. $P \Longrightarrow Q$.

# Regeln des Floyd-Hoare-Kalküls: Weakening

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$




- $\models \{A\}\, c\, \{B\}$: Ausführung von $c$ startet in Zustand, in dem $A$ gilt, und endet (ggf) in Zustand, in dem $B$ gilt.
- Zustandsprädikate beschreiben Mengen von Zuständen: $P \subseteq Q$ gdw. $P \Longrightarrow Q$.
- Wir können $A$ zu $A'$ einschränken ($A' \subseteq A$ oder $A' \Longrightarrow A$), oder $B$ zu $B'$ vergrößern ($B \subseteq B'$ oder $B \Longrightarrow B'$), und erhalten $\models \{A'\}\, c\, \{B'\}$.

# Überblick: die Regeln des Floyd-Hoare-Kalküls

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{}{\vdash \{A\}\, \{\,\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

# Einfache Beispiele

Sei $p$:

```
z= x;
x= y;
y= z;
```

Zu zeigen:

- $p$ vertauscht $x$ und $y$

Sei $q$:

```
if (x < y) {
  z = x;
  } else {
  z= y;
  }
```

Zu zeigen:

# Einfache Beispiele

Sei $p$:

```
z= x;
x= y;
y= z;
```

Zu zeigen:

- $p$ vertauscht $x$ und $y$
- $\vdash \{x = X \wedge y = Y\}$
  $p$
  $\{x = Y \wedge y = X\}$

Sei $q$:

```
if (x < y) {
  z = x;
  } else {
  z= y;
  }
```

Zu zeigen:

# Einfache Beispiele

Sei $p$:

```
z= x;
x= y;
y= z;
```

Zu zeigen:

- $p$ vertauscht $x$ und $y$
- $\vdash \{x = X \wedge y = Y\}$
  $p$
  $\{x = Y \wedge y = X\}$

Sei $q$:

```
if (x < y) {
  z = x;
  } else {
  z= y;
  }
```

Zu zeigen:

- $q$ berechnet in $z$ das Minimum von $x$ und $y$

# Einfache Beispiele

Sei $p$:

```
z= x;
x= y;
y= z;
```

Zu zeigen:

- $p$ vertauscht $x$ und $y$
- $\vdash \{x = X \wedge y = Y\}$
  $p$
  $\{x = Y \wedge y = X\}$

Sei $q$:

```
if (x < y) {
  z = x;
  } else {
  z= y;
  }
```

Zu zeigen:

- $q$ berechnet in $z$ das Minimum von $x$ und $y$
- $\vdash \{true\}$
  $q$
  $\{z \leq x \wedge z \leq y\}$

# Wie wir Floyd-Hoare-Beweise aufschreiben

```
// {P}
// {P_1}
x= e;
// {P_2}
// {P_3}
while (x< n) {
    // {P_3 ∧ x < n}
    // {P_4}
    z= a;
    // {P_3}
    }
// {P_3 ∧ ¬(x < n)}
// {Q}
```

- Beispiel zeigt: $\vdash \{P\}\, c\, \{Q\}$
- Programm wird mit gültigen Zusicherungen annotiert.
- Vor einer Zeile steht die Vorbedingung, danach die Nachbedingung.
- Implizite Anwendung der Sequenzenregel.
- Weakening wird notiert durch mehrere Zusicherungen, und muss **bewiesen** werden.
  - Im Beispiel: $P \Longrightarrow P_1$, $P_2 \Longrightarrow P_3$, $P_3 \wedge x < n \Longrightarrow P_4$, $P_3 \wedge \neg(x < n) \Longrightarrow Q$.

# Das einfache Beispiel in neuer Notation

$// \ \{x = X \wedge y = Y\}$
$// \ \{y = Y \wedge x = X\}$
z= x ;
$// \ \{y = Y \wedge z = X\}$
x= y ;
$// \ \{x = Y \wedge z = X\}$
y= z ;
$// \ \{x = Y \wedge y = X\}$

# Das Fakultätsbeispiel

$// \ \{1 = 0! \wedge \wedge 0 \leq n\}$
$// \ \{1 = (1-1)! \wedge 1 \leq 1 \wedge 1 - 1 \leq n\}$
p= 1;
$// \ \{p = (1-1)! \wedge 1 \leq 1 \wedge 1 - 1 \leq n\}$
c= 1;
$// \ \{p = (c-1)! \wedge 1 \leq c \wedge c - 1 \leq n\}$
**while** (c<= n) {
$// \ \{p = (c-1)! \wedge 1 \leq c \wedge c - 1 \leq n \wedge c \leq n\}$
$// \ \{p * c = (c-1)! * c \wedge 1 \leq c \wedge c \leq n\}$
$// \ \{p * c = c! \wedge 1 \leq c \wedge c \leq n\}$
$// \ \{p * c = ((c+1)-1)! \wedge 1 \leq c+1 \wedge (c+1) - 1 \leq n\}$
p= p*c;
$// \ \{p = ((c+1)-1)! \wedge 1 \leq c+1 \wedge (c+1) - 1 \leq n\}$
c= c+1;
$// \ \{p = (c-1)! \wedge 1 \leq c \wedge c - 1 \leq n\}$
}
$// \ \{p = (c-1)! \wedge 1 \leq c \wedge c - 1 \leq n \wedge \neg(c \leq n)\}$
$// \ \{p = (c-1)! \wedge 1 \leq c \wedge c - 1 \leq n \wedge c > n\}$
$// \ \{p = (c-1)! \wedge 1 \leq c \wedge c - 1 = n\}$
$// \ \{p = n!\}$

# Das Fakultätsbeispiel (Quellcode)

```
/** { 1 == factorial(0) && 0 <= n } */
/** { 1 == factorial(1- 1) && 1 <= 1 && 1-1 <= n } */
p= 1;
/** { p == factorial(1- 1) && 1 <= 1 && 1-1 <= n } */
c= 1;
/** { p == factorial(c- 1) && 1 <= c && c-1 <= n } */
while (c<= n) {
  /** { p == factorial(c-1) && 1<= c && c-1 <= n && c <= n } */
  /** { p*c == factorial(c-1)* c && 1 <= c && c <= n } */
  /** { p*c == factorial(c) && 1 <= c && c <= n } */
  /** { p*c == factorial((c+1)- 1) && 1 <= c+1 && (c+1)-1 <= n
  p= p*c;
  /** { p == factorial((c+1)-1) && 1<= c+1 && (c+1)-1 <= n } */
  c= c+1;
  /** { p == factorial(c-1) && 1 <= c && c-1 <= n } */
  }
/** { p == factorial(c-1) && 1<= c && c-1 <= n && factorial(c <=
/** { p == factorial(c-1) && 1<= c && c-1 <= n && c > n } */
/** { p == factorial(c-1) && 1<= c && c-1 == n } */
/** { p == factorial(n) } */
```

# Zusammenfassung Floyd-Hoare-Logik

- Die Logik abstrahiert über konkrete Systemzustände durch Zusicherungen (Hoare-Tripel $\{P\}\, c\, \{Q\}$).
- Zusicherungen sind boolsche Ausdrücke, angereichert durch logische Variablen.
- Semantische **Gültigkeit** von Hoare-Tripeln: $\models \{P\}\, c\, \{Q\}$.
- Syntaktische **Herleitbarkeit** von Hoare-Tripeln: $\vdash \{P\}\, c\, \{Q\}$
- Zuweisungen werden durch Substitution modelliert, d.h. die Menge der gültigen Aussagen ändert sich.
- Für Iterationen wird eine **Invariante** benötigt (die **nicht** hergeleitet werden kann).

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 6 vom 15.05.18: Invarianten und die Korrektheit des Floyd-Hoare-Kalküls

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Invarianten

# Invarianten Finden: die Fakultät

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

Invariante:

$$p = (c-1)!$$

- Kern der Invariante: Fakultät bis $c-1$ berechnet.

# Invarianten Finden: die Fakultät

```
p= 1;
c= 1;
while (c <= n) {
   p = p * c;
   c = c + 1;
}
```

Invariante:

$$p = (c-1)! \wedge c - 1 \leq n$$

- Kern der Invariante: Fakultät bis $c-1$ berechnet.
- Invariante impliziert Nachbedingung

# Invarianten Finden: die Fakultät

```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

Invariante:

$$p = (c-1)! \wedge c - 1 \leq n \wedge c > 0$$

- Kern der Invariante: Fakultät bis $c-1$ berechnet.
- Invariante impliziert Nachbedingung
- Nebenbedingung für Weakening innerhalb der Schleife.

# Invarianten finden

1. Initiale Invariante: momentaner Zustand der Berechnung

2. Invariante und negierte Schleifenbedingung muss Nachbedingung implizieren; ggf. Invariante verstärken.

3. Beweise innerhalb der Schleife benötigen ggf. weiter Nebenbedingungen; Invariante verstärken.

# Zählende Schleifen

- Fakultät ist Beispiel für zählende Schleife (**for**).
- Für Nachbedigung $\psi$ ist Invariante:

$$\psi[i-1/n] \wedge i - 1 \leq n$$

- Ggf. weitere Nebenbedingungen erforderlich

```
for (i= 0; i<= n; i++) {
  ...
}
```

ist syntaktischer Zucker für

```
i= 0;
while (i<= n) {
  ...
  i= i+1;
}
```

# Beispiel 1

```
1  // {0 ≤ y}
2  x= 1;
3  c= 1;
4  while (c <= y) {
5    x= 2*x;
6    c= c+1;
7  }
8  // {x = 2^y}
```

- Invariante:

# Beispiel 1

```
1  // {0 ≤ y}
2  x= 1;
3  c= 1;
4  while (c <= y) {
5    x= 2*x;
6    c= c+1;
7  }
8  // {x = 2^y}
```

- Invariante:

$$x = 2^{c-1}$$

# Beispiel 1

```
1  // {0 ≤ y}
2  x= 1;
3  c= 1;
4  while (c <= y) {
5    x= 2*x;
6    c= c+1;
7  }
8  // {x = 2^y}
```

- Invariante:

$$x = 2^{c-1} \wedge c - 1 \leq y$$

# Beispiel 2

```
1  // {0 ≤ y}
2  x= 1;
3  c= 1;
4  while (c < y) {
5    c= c+1;
6    x= 2*x;
7  }
8  // {x = 2^y}
```

- Invariante:

# Beispiel 2

```
1  // {0 ≤ y}
2  x= 1;
3  c= 1;
4  while (c < y) {
5    c= c+1;
6    x= 2*x;
7  }
8  // {x = 2^y}
```

- Invariante:

$$x = 2^c$$

# Beispiel 2

```
1  // {0 ≤ y}
2  x= 1;
3  c= 1;
4  while (c < y) {
5    c= c+1;
6    x= 2*x;
7  }
8  // {x = 2^y}
```

- Invariante:

$$x = 2^c \wedge c \leq y$$

# Beispiel 2

```
1  // {0 ≤ y}
2  x= 1;
3  c= 0;
4  while (c < y) {
5    c= c+1;
6    x= 2*x;
7  }
8  // {x = 2^y}
```

- Invariante:

$$x = 2^c \wedge c \leq y$$

# Beispiel 3

```
1  // {y = Y ∧ 0 ≤ y}
2  x= 1;
3  while (y != 0) {
4    x= 2*x;
5    y= y-1;
6  }
7  // {x = 2^Y}
```

- Invariante:

$$x = 2^{Y-y}$$

# Beispiel 4

```
1  // {0 ≤ a}
2  r= a;
3  q= 0;
4  while (b <= r) {
5      r= r-b;
6      q= q+1;
7      }
8  // {a = b * q + r ∧ 0 ≤ r ∧ r < b}
```

Invariante:

# Beispiel 4

```
1  // {0 ≤ a}
2  r= a;
3  q= 0;
4  while (b <= r) {
5      r= r-b;
6      q= q+1;
7      }
8  // {a = b * q + r ∧ 0 ≤ r ∧ r < b}
```

Invariante:

$$a = b \cdot q + r \wedge 0 \leq r$$

- Spezieller Fall: letzter Teil der Nachbedingung ist genau negierte Schleifeninvariante

# Beispiel 5

```
1  // {0 ≤ a}
2  t= 1;
3  s= 1;
4  i= 0;
5  while (s <= a) {
6     t= t+ 2;
7     s= s+ t;
8     i= i+ 1;
9     }
10 // {i² ≤ a ∧ a < (i + 1)²}
```

- Was berechnet das?
  Ganzzahlige Wurzel von $a$.
- Invariante:

  $s-t \leq a \wedge t = 2 \cdot i+1 \wedge s = i^2+t$

- Nachbedingung 1: aus
  $s - t \leq a, s = i^2 + t$ folgt
  $i^2 \leq a$.
- Nachbedingung 2: aus
  $s = i^2 + t, t = 2 \cdot i + 1$ und
  $a < s$ folgt $a < (i + 1)^2$.

# Korrektheit des Floyd-Hoare-Kalküls

# Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

- Definition von letzter Woche: $P, Q \in \mathbf{Assn}, c \in \mathbf{Stmt}$

  $\models \{P\}\, c\, \{Q\}$ "Hoare-Tripel gilt" (semantisch)

  $\vdash \{P\}\, c\, \{Q\}$ "Hoare-Tripel herleitbar" (syntaktisch)

- **Frage**: $\vdash \{P\}\, c\, \{Q\} \overset{?}{\rightsquigarrow} \models \{P\}\, c\, \{Q\}$

# Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

- Definition von letzter Woche: $P, Q \in \mathbf{Assn}, c \in \mathbf{Stmt}$

  $\models \{P\}\, c\, \{Q\}$ "Hoare-Tripel gilt" (semantisch)

  $\vdash \{P\}\, c\, \{Q\}$ "Hoare-Tripel herleitbar" (syntaktisch)

- **Frage**: $\vdash \{P\}\, c\, \{Q\} \overset{?}{\rightsquigarrow} \models \{P\}\, c\, \{Q\}$
- **Korrektheit**: $\vdash \{P\}\, c\, \{Q\} \overset{?}{\Longrightarrow} \models \{P\}\, c\, \{Q\}$
  - Wir können nur gültige Eigenschaften von Programmen herleiten.
- **Vollständigkeit**: $\models \{P\}\, c\, \{Q\} \overset{?}{\Longrightarrow} \vdash \{P\}\, c\, \{Q\}$
  - Wir können alle gültigen Eigenschaften auch herleiten.

# Korrektheit des Floyd-Hoare-Kalküls

**Der Floyd-Hoare-Kalkül ist korrekt.**

Wenn $\vdash \{P\}\, c\, \{Q\}$, dann $\models \{P\}\, c\, \{Q\}$.

Beweis:

- Definition von $\models \{P\}\, c\, \{Q\}$:

$$\models \{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I P \land \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \implies \sigma' \models^I Q$$

- Beweis durch **Regelinduktion** über der **Herleitung** von $\vdash \{P\}\, c\, \{Q\}$.
- Bsp: Zuweisung, Sequenz, Weakening, While.
  - Zuweisung benötigt Lemma: $\sigma \models^I B[e/x] \iff \sigma[\mathcal{A}[\![e]\!](\sigma)/x] \models^I B$

# Vollständigkeit der Floyd-Hoare-Logik

**Floyd-Hoare-Logik ist vollständig modulo weakening.**

Wenn $\models \{P\}\, c\, \{Q\}$, dann $\vdash \{P\}\, c\, \{Q\}$ bis auf die Bedingungen der Weakening-Regel.

- Beweis durch Konstruktion einer schwächsten Vorbedingung $\mathrm{wp}(c, Q)$.
  - Problemfall: while-Schleife.
- Wenn wir eine gültige Zusicherung nicht herleiten können, liegt das nur daran, dass wir eine Beweisverpflichtung nicht beweisen können.
- Logik erster Stufe ist unvollständig, also **können** wir gar nicht besser werden.

# Zusammenfassung

- Invarianten finden in **drei Schritten**,
- Floyd-Hoare-Logik ist **korrekt**, wir können nur gültige Zusicherungen herleiten.
- Floyd-Hoare-Logik ist **vollständig** bis auf das Weakening.

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 7 vom 22.05.16: Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Motivation

- Immer nur ganze Zahlen ist doch etwas langweilig.
- Weitere Basisdatentypen von C (Felder, Zeichenketten, Strukturen)
- Noch rein funktional, keine Referenzen
- Nicht behandelt, aber nur syntaktischer Zucker: **enum**
- Prinzipiell: keine **union**

# Arrays

- Beispiele:

```
int six[6] = {1,2,3,4,5,6};
int a[3][2];
int b[][] = { {1, 0},
              {3, 7},
              {5, 8} }; /* Ergibt Array [3][2] */
```

  - b [2][1] liefert 8, b [1][0] liefert 3

- Index startet mit 0, *row-major order*
- In C0: Felder als echte Objekte (in C: Felder $\cong$ Zeiger)
- Allgemeine Form:

```
typ name[groesse1][groesse2]...[groesseN] =
    { ... }
```

  - Alle Felder haben **feste Größe**.

# Zeichenketten

- Zeichenketten sind in C (und C0) Felder von **char**, die mit einer Null abgeschlossen werden.
- Beispiel:

```
char hallo[5] = {'h', 'a', 'l', 'l', 'o', '\0'}
```

- Nützlicher syntaktischer Zucker:

```
char hallo[] = "hallo";
```

- Auswertung: hallo[4] liefert o

# Strukturen

- Strukturen haben einen *structure tag* (optional) und Felder:

```
struct Vorlesung {
  char dozenten[2][30];
  char titel[30];
  int cp;
} ksgm;

struct Vorlesung pi3;
```

- Zugriff auf Felder über Selektoren:

```
int i = 0;
char name1[] = "Serge Autexier";
while (i < strlen(name1)) {
  ksgm.dozenten[0][i] = name1[i];
  i = i + 1;
}
```

- Rekursive Strukturen nur über Zeiger erlaubt (kommt noch)

# C0: Erweiterte Ausdrücke

- **Lexp** beschreibt L-Werte (l-values), abstrakte Speicheradressen
- Neuer Basisdatentyp **C** für Zeichen
- Erweiterte Grammatik:

$$\textbf{Lexp}\ \ l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt}$$

$$\textbf{Aexp}\ \ a ::= \textbf{Z} \mid \textbf{C} \mid \textbf{Lexp} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$$

$$\textbf{Bexp}\ \ b ::= \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,||\, b_2$$

$$\textbf{Exp}\ \ e := \textbf{Aexp} \mid \textbf{Bexp}$$

# Werte und Zustände

- Zustände bilden **strukturierte** Adressen auf Werte (wie vorher) ab.

## Systemzustände

- **Locations:** $\mathbf{Loc} ::= \mathbf{Idt} \mid \mathbf{Loc}[\mathbb{Z}] \mid \mathbf{Loc}.\mathbf{Idt}$
- Werte: $\mathbf{V} = \mathbb{Z}$
- Zustände: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

- Wir betrachten nur Zugriffe vom Typ $\mathbf{Z}$ oder $\mathbf{C}$ (**elementare Typen**)

- Nützliche Abstraktion des tatsächliche C-Speichermodells

# Beispiel

## Programm

```
struct A {
 int c[2];
 struct B {
  char name[20];
 } b;
};

struct A x[] = {
 {{1,2},
  {{'n','a','m','e','1','\0'}}
 },
 {{3,4},
  {{'n','a','m','e','2','\0'}}
 }
};
```

## Zustand

$$x[0].c[0] \mapsto 1 \qquad x[1].c[0] \mapsto 3$$
$$x[0].c[1] \mapsto 2 \qquad x[1].c[1] \mapsto 4$$
$$x[0].b.name[0] \mapsto \text{`}n' \qquad x[1].b.name[0] \mapsto \text{`}n'$$
$$x[0].b.name[1] \mapsto \text{`}a' \qquad x[1].b.name[1] \mapsto \text{`}a'$$
$$x[0].b.name[2] \mapsto \text{`}m' \qquad x[1].b.name[2] \mapsto \text{`}m'$$
$$x[0].b.name[3] \mapsto \text{`}e' \qquad x[1].b.name[3] \mapsto \text{`}e'$$
$$x[0].b.name[4] \mapsto \text{`}1' \qquad x[1].b.name[4] \mapsto \text{`}2'$$
$$x[0].b.name[5] \mapsto \text{`}\backslash 0' \qquad x[1].b.name[5] \mapsto \text{`}\backslash 0'$$

# Operationale Semantik: L-Werte

- **Lexp** $m$ wertet zu **Loc** $l$ aus: $\langle m, \sigma \rangle \to_{Lexp} l \mid \bot$

$$\frac{x \in \mathbf{Idt}}{\langle x, \sigma \rangle \to_{Lexp} x}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad \langle a, \sigma \rangle \to_{Aexp} i \neq \bot}{\langle m[a], \sigma \rangle \to_{Lexp} l[i]}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l \qquad \langle a, \sigma \rangle \to_{Aexp} \bot}{\langle m[a], \sigma \rangle \to_{Lexp} \bot}$$

$$\frac{\langle m, \sigma \rangle \to_{Lexp} l}{\langle m.i, \sigma \rangle \to_{Lexp} l.i}$$

# Operationale Semantik: Ausdrücke und Zuweisungen

- Ein L-Wert als Ausdruck wird ausgewertet, indem er ausgelesen wird:

$$\frac{\langle m, \sigma\rangle \to_{Lexp} l \qquad l \in Dom(\sigma)}{\langle m, \sigma\rangle \to_{Aexp} \sigma(l)}$$

$$\frac{\langle m, \sigma\rangle \to_{Lexp} l \qquad l \notin Dom(\sigma)}{\langle , \sigma\rangle \to_{Aexp} \bot}$$

- Zuweisungen sind nur definiert für elementare Typen:

$$\frac{\langle m :: \tau, \sigma\rangle \to_{Lexp} l \qquad \langle e :: \tau, \sigma\rangle \to v \qquad \tau \text{ elementarer Typ}}{\langle m = e, \sigma\rangle \to_{Stmt} \sigma[v/l]}$$

- Die restlichen Regeln bleiben

# Denotationale Semantik

- Denotation für **Lexp**:

$$\begin{aligned}\mathcal{L}[\![x]\!] &= \{(\sigma, x) \mid \sigma \in \Sigma\} \\ \mathcal{L}[\![m[a]]\!] &= \{(\sigma, l[i]) \mid (\sigma, l) \in \mathcal{L}[\![m]\!], (\sigma, i) \in \mathcal{A}[\![a]\!]\} \\ \mathcal{L}[\![m.i]\!] &= \{(\sigma, m.i) \mid (\sigma, l) \in \mathcal{L}[\![m]\!]\}\end{aligned}$$

- Denotation für **Zuweisungen**:

$$\mathcal{C}[\![m = e]\!] = \{(\sigma, \sigma[v/l]) \mid (\sigma, l) \in \mathcal{L}[\![m]\!], (\sigma, v) \in \mathcal{A}[\![e]\!]\}$$

# Floyd-Hoare-Kalkül

- Die Regeln des Floyd-Hoare-Kalküls berechnen geltende Zusicherungen
- Nötige Änderung: Substitution in Zusicherungen
  - Jetzt werden **Lexp** ersetzt, keine **Idt**
  - Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
  - Problem: Feldzugriffe

# Beispiel

```
int a[3];
/** { 1 } */
/** { 3 = 3 and 3 = 3 } */
a[2] = 3;
/** { a[2] = 3 and a[2] = 3 } */
/** { 4 = 4 and a[2] = 3 and 4 * a[2] = 12 } */
a[1] = 4;
/** { a[1] = 4 and a[2] = 3 and a[1] * a[2] = 12 } */
/** { 5 = 5 and a[1] = 4 and a[2] = 3 and
      5 * a[1] * a[2] = 60 } */
a[0] = 5;
/** { a[0] = 5 and a[1] = 4 and a[2] = 3 and
      a[0] * a[1] * a[2] = 60 } */
```

# Beispiel: Problem

```
int a[3];
int i;
/** { 0 <= i< 2 } */
a[0] = 3;
a[1] = 7;
a[2] = 9;
a[i] = -1;
/** { a[1] == 7 }
```

# Erstes Beispiel: Ein Feld initialisieren

```
1  // {n ≤ 0}
2  i= 0;
3  while (i< n) {
4     a[i]= i;
5     i= i+1;
6     // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
7     }
8  // {∀j.0 ≤ j < n ⟶ a[j] = j}
```

# Erstes Beispiel: Ein Feld initialisieren

```
1  // {n ≤ 0}
2  i= 0;
3  while (i< n) {
4     a[i]= i;
5     i= i+1;
6     // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
7     }
8  // {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

$$(\forall j.\, 0 \leq j < n \longrightarrow P[j]) \wedge P[n] \Longrightarrow \forall j.\, 0 \leq j < n+1 \longrightarrow P[j]$$

# Längeres Beispiel: Suche nach dem maximalen Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i< n) {
5    if (a[r] < a[i]) {
6      r= i;
7      }
8    else {
9      }
0    i= i+1;
1    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
2    }
3  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  i= 0;
3  r= -1;
4  while (i< n) {
5    if (a[i] == 0) {
6      r= i;
7      }
8    else {
9      }
10   i= i+1;
11   // {(r ≠ -1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
12   }
13 // {r ≠ -1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

# Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  i= 0;
3  r= -1;
4  while (i< n) {
5    if (a[i] == 0) {
6      r= i;
7      }
8    else {
9      }
10   i= i+1;
11   // {(r ≠ -1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
12   }
13 // {r ≠ -1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

- Spezifikation zu schwach: wann ist $r = -1$?

# Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  i= 0;
3  r= -1;
4  while (i< n) {
5    if (a[i] == 0) {
6      r= i;
7      }
8    else {
9      }
10   i= i+1;
11   // {(r ≠ -1 ⟶ 0 ≤ r < i ∧ a[r] = 0)
         ∧(r = -1 ⟶ ∀j. 0 ≤ j < i ⟶ a[j] ≠ 0)
         ∧0 ≤ i ≤ n}
12   }
13 // {(r ≠ -1 ⟶ 0 ≤ r < n ∧ a[r] = 0)
       ∧(r = -1 ⟶ ∀j. 0 ≤ j < n ⟶ a[j] ≠ 0)}
```

# Längeres Beispiel: Suche nach dem ersten Null-Element

```
1  // {0 ≤ n}
2  i= 0;
3  r= -1;
4  while (i< n) {
5    if (r== -1 && a[i] == 0) {
6      r= i;
7      }
8    else {
9      }
0    i= i+1;
1    // {(r ≠ -1 ⟶ 0 ≤ r < i ∧ a[r] = 0 ∧ (∀j.0 ≤ j < i ⟶ a[j] ≠ 0))
         ∧(r = -1 ⟶ ∀j. 0 ≤ j < i ⟶ a[j] ≠ 0)
         0 ≤ i ≤ n}
2    }
3  // {(r ≠ -1 ⟶ 0 ≤ r < n ∧ a[r] = 0 ∧ (∀j.0 ≤ j < r ⟶ a[j] ≠ 0))
       ∧(r = -1 ⟶ ∀j. 0 ≤ j < n ⟶ a[j] ≠ 0)}
```

# Zusammenfassung

- Strukturierte Datentypen (Felder und Structs) erfordern strukturierte Adressen
- Abstraktion über „echtem“ Speichermodell
- Änderungen in der Semantik und im Floyd-Hoare-Kalkül überschaubar
- . . . aber mit erheblichen Konsequenzen: Substitution

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 8 vom 29.05.18: Modellierung und Spezifikation

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Erstes Beispiel: Ein Feld initialisieren

```
1  // {n ≤ 0}
2  i= 0;
3  while (i< n) {
4     a[i]= i;
5     i= i+1;
6     // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
7     }
8  // {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

$$(\forall j.\, 0 \leq j < n \longrightarrow P[j]) \wedge P[n] \longrightarrow \forall j.\, 0 \leq j < n+1 \longrightarrow P[j]$$

# Längeres Beispiel: Suche nach dem maximalen Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i< n) {
5     if (a[r] < a[i]) {
6        r= i;
7        }
8     else {
9        }
10    i= i+1;
11    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
12    }
13 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

# Sortierte Arrays?

- Wie formulieren wir, dass ein Array sortiert ist? Ggf. bis zu einem bestimmten Punkt $n$ sortiert ist?

```
int a[8];
```
// $\{\forall 0 \leq j \leq n < 6.a[j] \leq a[j+1]\}$

- Alternativ würden man auch gerne ein Prädikat definieren können

// $\{\forall a.sorteduntil(a, 0) \longleftrightarrow true\}$
// $\{\forall a.\forall i.i \geq 0 \longrightarrow (sorteduntil(a, i+1) \longleftrightarrow (a[i] \leq a[i+1] \wedge sorteduntil(a, i))$

# Formelsprache mit Quantoren

- Wir brauchen Programmausdrücken wie **Aexp**
- Wir müssen neue Funktionen verwendenn können
  - Etwa eine Fakultätsfunkion
- Wir müssen neue Prädikate definieren können
  - Etwa einen `sorteduntil`
- Wir müssen Formeln bilden können
  - Analog zu **Bexp**
  - Zusätzlich mit Implikation $\longrightarrow$, Äquivalenz $\longleftrightarrow$
  - Zusätzlich Quantoren über logische Variablen wie in

$$(\forall j.\, 0 \leq j < n \longrightarrow P[j]) \wedge P[n] \longrightarrow \forall j.\, 0 \leq j < n+1 \longrightarrow P[j]$$

$$\forall i.i \geq 0 \longrightarrow (\mathit{sorteduntil}(a, i+1) \longleftrightarrow (a[i] \leq a[i+1] \wedge \mathit{sorteduntil}(a, i)))$$

# Was brauchen wir?

- Definiere Terme als Variablen und Funktionen besimmter Stelligkeit
- Definiere Literale und Formeln
- Interpretation von Formeln
  - mit und ohne Programmvariablen

# Zusicherungen (Assertions)

- Erweiterung von **Aexp** and **Bexp** durch
  - **Logische** Variablen **Var** $v := N, M, L, U, V, X, Y, Z$
  - Definierte Funktionen und Prädikate über **Aexp** $n!, \sum_{i=1}^{n} i, \ldots$
  - Implikation, Äquivalenzen und Quantoren
    $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b, \exists v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b$
- Formal:

$$
\begin{array}{rrl}
\textbf{Lexp} & l ::= & \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \\
\textbf{Aexpv} & a ::= & \textbf{Z} \mid \textbf{Idt} \mid \textbf{Var} \mid \textbf{C} \mid \textbf{Lexp} \\
 & & \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
 & & \mid f(e_1, \ldots, e_n) \\
\textbf{Assn} & b ::= & \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2 \\
 & & \mid !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\
 & & \mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \ldots, e_n) \\
 & & \mid \forall v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b \mid \exists v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b
\end{array}
$$

# Zusicherungen (Assertions)

- Erweiterung von **Aexp** and **Bexp** durch
  - **Logische** Variablen **Var** $v := N, M, L, U, V, X, Y, Z$
  - ~~Definierte Funktionen und Prädikate über **Aexp**~~ ~~$n!, \sum_{i=1}^{n} i, \ldots$~~
  - Implikation, Äquivalenzen und Quantoren
    $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b, \exists v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b$
- Formal:

$$
\begin{array}{rrl}
\textbf{Lexp} & l ::= & \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \\
\textbf{Aexpv} & a ::= & \textbf{Z} \mid \textbf{Idt} \mid \textbf{Var} \mid \textbf{C} \mid \textbf{Lexp} \\
 & & \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
 & & \mid f(e_1, \ldots, e_n) \\
\textbf{Assn} & b ::= & \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2 \\
 & & \mid !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\
 & & \mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \ldots, e_n) \\
 & & \mid \forall v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b \mid \exists v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b
\end{array}
$$

# Zusicherungen (Assertions)

- Erweiterung von **Aexp** and **Bexp** durch
  - **Logische** Variablen **Var** $\quad v := N, M, L, U, V, X, Y, Z$
  - Funktionen und Prädikate selbst definieren
  - Implikation, Äquivalenzen und Quantoren
    $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b, \exists v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b$
- Formal:

$$
\begin{array}{rrl}
\textbf{Lexp} & l ::= & \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \\
\textbf{Aexpv} & a ::= & \textbf{Z} \mid \textbf{Idt} \mid \textbf{Var} \mid \textbf{C} \mid \textbf{Lexp} \\
 & & \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
 & & \mid f(e_1, \ldots, e_n) \\
\textbf{Assn} & b ::= & \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2 \\
 & & \mid !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\
 & & \mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \ldots, e_n) \\
 & & \mid \forall v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b \mid \exists v_{(\mathbf{Z}|\mathbf{C}|\mathrm{Array})}.\, b
\end{array}
$$

# Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand $\sigma$?
  - Auswertung (denotationale Semantik) ergibt *true*
- **Belegung** der logischen Variablen: $I : \mathbf{Var} \to (\mathbf{Z} \cup \mathbf{C} \cup \mathrm{Array})$
- Semantik von $b$ unter der Belegung $I$: $\mathcal{B}_v[\![b]\!]^I, \mathcal{A}_v[\![a]\!]^I$

$$\mathcal{A}_v[\![I]\!]^I = \{(\sigma, \sigma(i) \mid (\sigma, i) \in \mathcal{L}_v[\![I]\!]^I, i \in Dom(\sigma)\}$$

# Denotationale Semantik

- Denotation für **Lexp**unter der Belegung $I$:

$$\begin{aligned}
\mathcal{L}_v[\![x]\!]^I &= \{(\sigma, x) \mid \sigma \in \Sigma\} \\
\mathcal{L}_v[\![m[a]]\!]^I &= \{(\sigma, I[i]) \mid (\sigma, I) \in \mathcal{L}_v[\![m]\!]^I, (\sigma, i) \in \mathcal{A}[\![a]\!]^I\} \\
\mathcal{L}_v[\![m.i]\!]^I &= \{(\sigma, m.i) \mid (\sigma, I) \in \mathcal{L}_v[\![m]\!]^I\}
\end{aligned}$$

- Denotation für **Aexp**unter der Belegung $I$:

$$\begin{aligned}
\mathcal{A}_v[\![I]\!]^I &= \{(\sigma, \sigma(i) \mid (\sigma, i) \in \mathcal{L}_v[\![I]\!]^I, i \in Dom(\sigma)\} && I \in \textbf{Lexp} \\
\mathcal{A}_v[\![v]\!]^I &= \{(\sigma, I(v))\} && v \in \textbf{Var}
\end{aligned}$$

# Grenzen der Denotationalen Semantik

- Problem mit selbst-definierten Funktionen: $\mathcal{A}_v[\![f(a_1,\ldots,a_n)]\!]^I = ?$
- Problem mit selbst-definierten Prädikaten: $\mathcal{B}_v[\![p(a_1,\ldots,a_n)]\!]^I = ?$
- Betrachte Gleichung:

$$\forall v_{\mathbf{Z}}.f(v) = t$$

so dass all in $t$ vorkommenden Operationen eine denotationale Semantik haben (und $f$ nicht in $t$ vorkommt).

  - Dies gibt der Funktion $f$ eine Semantik, nämlich f.a. Programmzustände $\sigma$:

$$\mathcal{A}_v[\![f(a)]\!]^I(\sigma) = \mathcal{A}_v[\![t]\!]^{I'}(\sigma) \text{ mit } I' := I[\mathcal{A}_v[\![a]\!]^I(\sigma)/v]$$

  - Was ist wenn es mehrere solche Gleichungen gibt? Was ist wenn das nicht terminiert?
    $\Longrightarrow$ Nur eindeutig definierte, terminierende Definition (konservative Erweiterungen)
    $\Longrightarrow$ Wird nicht weiter vertieft in dieser Vorlesung; alle im Folgenden verwendete Definition sind derart.
  - Analog für Prädikate

# Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand $\sigma$?
  - Auswertung (denotationale Semantik) ergibt *true*
- **Belegung** der logischen Variablen: $I : \mathbf{Var} \to (\mathbf{Z} \cup \mathbf{C} \cup \text{Array})$
- Semantik von $b$ unter der Belegung $I$:

$$\begin{aligned}
\mathcal{B}_v[\![\forall v_{\mathbf{Z}}.b]\!]^I = \{&(\sigma,1) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma,1) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \\
\cup \{&(\sigma,0) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma,0) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \\
\mathcal{B}_v[\![\exists v_{\mathbf{Z}}.b]\!]^I = \{&(\sigma,1) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma,1) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \\
\cup \{&(\sigma,0) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma,0) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\}
\end{aligned}$$

Analog für $\forall v_{\mathbf{C}}.b$, $\exists v_{\mathbf{C}}.b$, $\forall v_{\text{Array}}.b$ und $\exists v_{\text{Array}}.b$

# Erfülltheit von Zusicherungen

## Erfülltheit von Zusicherungen

$b \in$ **Assn** ist in Zustand $\sigma$ mit Belegung $I$ erfüllt ($\sigma \models^I b$), gdw

$$\mathcal{B}_v[\![b]\!]^I(\sigma) = \mathit{true}$$

## Denotation für **Zuweisungen**

$$\mathcal{C}_v[\![m = e]\!]^I = \{(\sigma, \sigma[v/l]) \mid (\sigma, l) \in \mathcal{L}_v[\![m]\!]^I, (\sigma, v) \in \mathcal{A}_v[\![e]\!]^I\}$$

# Formeln ohne Programmvariablen, ohne Arrays, ohne Strukturen

- Eine Formel $b \in$ **Assn** ist pur, wenn sie weder Programmvariablen, noch Strukturen, noch Felder enthält (also keine Teilterme aus **Lexp** und **Idt**.
- Eine Formel ist geschlossen, wenn sie pur ist und keine freien logischen Variablen enthält.
- Sei $\mathbf{Assn}^c \subseteq \mathbf{Assn}$ die Menge der geschlossenen Formeln

**Lemma**

*Für eine geschlossene Formel b ist der Wahrheitswert $\mathcal{B}_v[\![b]\!]^I(\sigma)$ von b unabhängig von I und $\sigma$.*

- Sei $\Gamma$ eine endliche Menge von Formeln, dann definieren wir

$$\bigwedge \Gamma := \begin{cases} b_1 \,\&\&\, \ldots \,\&\&\, b_n & \text{für alle } b_i \in \Gamma, \Gamma \neq \emptyset \\ 1 & \text{falls } \Gamma = \emptyset \end{cases}$$

# Erfülltheit von Zusicherungen unter Kontext

## Erfülltheit von Zusicherungen unter Kontext

Sei $\Gamma \subseteq \mathbf{Assn}^c$ eine endliche Menge und $b \in \mathbf{Assn}$. Im Kontext $\Gamma$ ist $b$ in Zustand $\sigma$ mit Belegung $I$ erfüllt ($\Gamma, \sigma \models^I b$), gdw

$$\mathcal{B}_v[\![\Gamma \longrightarrow b]\!]^I(\sigma) = \mathit{true}$$

# Floyd-Hoare-Tripel mit Kontext

- Sei $\Gamma \in \mathbf{Assn}^c$ und $P, Q \subseteq \mathbf{Assn}$

**Partielle Korrektheit unter Kontext ($\Gamma \models \{P\}\, c\, \{Q\}$)**

$c$ ist **partiell korrekt**, wenn für alle Zustände $\sigma$ und alle Belegungen $I$ die unter Kontext $\Gamma$ $P$ erfüllen, gilt:
**wenn** die Ausführung von $c$ mit $\sigma$ in $\sigma'$ terminiert, **dann** erfüllen $\sigma'$ und $I$ im Kontext $\Gamma$ auch $Q$.

$$\Gamma \models \{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \Gamma, \sigma \models^I P \wedge \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \implies \Gamma, \sigma' \models^I Q$$

# Floyd-Hoare-Kalkül mit Kontext

$$\frac{}{\Gamma \vdash \{P[e/x]\}\, x = e\, \{P\}}$$

# Floyd-Hoare-Kalkül mit Kontext

$$\frac{}{\Gamma \vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{\Gamma \vdash \{A \mathbin{\&\&} b\}\, c_0\, \{B\} \qquad \Gamma \vdash \{A \mathbin{\&\&} \neg b\}\, c_1\, \{B\}}{\Gamma \vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

# Floyd-Hoare-Kalkül mit Kontext

$$\frac{}{\Gamma \vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{\Gamma \vdash \{A \,\&\&\, b\}\, c_0\, \{B\} \qquad \Gamma \vdash \{A \,\&\&\, \neg b\}\, c_1\, \{B\}}{\Gamma \vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\Gamma \vdash \{A \wedge b\}\, c\, \{A\}}{\Gamma \vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

# Floyd-Hoare-Kalkül mit Kontext

$$\frac{}{\Gamma \vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{\Gamma \vdash \{A \mathbin{\&\&} b\}\, c_0\, \{B\} \qquad \Gamma \vdash \{A \mathbin{\&\&} \neg b\}\, c_1\, \{B\}}{\Gamma \vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\Gamma \vdash \{A \wedge b\}\, c\, \{A\}}{\Gamma \vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{\Gamma \vdash \{A\}\, c_1\, \{B\} \qquad \Gamma \vdash \{B\}\, c_2\, \{C\}}{\Gamma \vdash \{A\}\, c_1; c_2\, \{C\}}$$

# Floyd-Hoare-Kalkül mit Kontext

$$\frac{\Gamma \longrightarrow (A' \longrightarrow A) \qquad \Gamma \vdash \{A\}\, c\, \{B\} \qquad \Gamma \longrightarrow (B \longrightarrow B')}{\Gamma \vdash \{A'\}\, c\, \{B'\}}$$

und es muss gezeigt werden für alle Zustände $\sigma$ und Belegungen $I$ dass $\Gamma \longrightarrow (A' \longrightarrow A)$ wahr bzw. dass

$$\mathcal{B}_v[\![\Gamma \longrightarrow (A' \longrightarrow A)]\!]^I(\sigma) = 1$$

(Analog für $\Gamma \longrightarrow (B \longrightarrow B')$).

# Floyd-Hoare-Kalkül mit Kontext

$$\frac{\Gamma \longrightarrow (A' \longrightarrow A) \qquad \Gamma \vdash \{A\}\, c\, \{B\} \qquad \Gamma \longrightarrow (B \longrightarrow B')}{\Gamma \vdash \{A'\}\, c\, \{B'\}}$$

und es muss gezeigt werden für alle Zustände $\sigma$ und Belegungen $I$ dass $\Gamma \longrightarrow (A' \longrightarrow A)$ wahr bzw. dass

$$\mathcal{B}_v[\![\Gamma \longrightarrow (A' \longrightarrow A)]\!]^I(\sigma) = 1$$

(Analog für $\Gamma \longrightarrow (B \longrightarrow B')$).

**Problem**

$\mathcal{B}_v[\![.]\!]^I(\sigma)$ *im Allgemeinen nicht berechenbar wegen*

$$\mathcal{B}_v[\![\forall_{\mathbf{Z}} v.b]\!]^I = \{(\sigma, 1) \mid \textit{für alle } i \in \mathbf{Z} \textit{ gilt } (\sigma, 1) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\}$$
$$\cup \{(\sigma, 0) \mid \textit{für ein } i \in \mathbf{Z} \textit{ gilt } (\sigma, 0) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\}$$

# Spezifikation von Funktionen

- Verwendung von geschlossenen Formeln zur Spezifikation von Funktionen und Prädikaten als Kontext
- Fakultät: $n!$ aka factorial($n$):

$$\begin{aligned} \text{factorial}(0) &== 1 \\ \forall n_{\mathbf{Z}}.(n > 0) &\longrightarrow \text{factorial}(n+1) == (n+1) \times \text{factorial}(n) \end{aligned}$$

- $\sum_{i=0}^{n}$ aka sumup($n$):

$$\begin{aligned} \text{sumup}(0) &== 0 \\ \forall x_{\mathbf{Z}}.(x \geq 0) &\longrightarrow \text{sumup}(x+1) == (x+1) + \text{sumup}(x) \end{aligned}$$

# Spezifikation von Prädikaten

- Gerade und Ungerade:

$$\forall n_{\mathbf{Z}}.\, \text{Gerade}(n) \longleftrightarrow \exists q_{\mathbf{Z}}.n == 2 \times q$$
$$\forall n_{\mathbf{Z}}.\, \text{UnGerade}(n) \longleftrightarrow \exists q_{\mathbf{Z}}.n == 2 \times q + 1$$

- Sortierte Arrays:

$$\forall a_{\text{Array}[\mathbf{Z}]}.\, \text{sorteduntil}(a, 0) \longleftrightarrow 1$$
$$\forall a_{\text{Array}[\mathbf{Z}]}.\forall i_{\mathbf{Z}}.i \geq 0 \longrightarrow (\text{sorteduntil}(a, i+1) \longleftrightarrow (a[i] \leq a[i+1] \wedge \text{sorteduntil}(a, i)))$$

# Das Fakultätsbeispiel

$\Gamma = \{$Def. von factorial$\}$:

```
/** { 1 == factorial(0) && 0 <= n } */
/** { 1 == factorial(1- 1) && 1 <= 1 && 1-1 <= n } */
p= 1;
/** { p == factorial(1- 1) && 1 <= 1 && 1-1 <= n } */
c= 1;
/** { p == factorial(c- 1) && 1 <= c && c-1 <= n } */
while (c<= n) {
  /** { p == factorial(c-1) && 1<= c && c-1 <= n && c <= n } */
  /** { p*c == factorial(c-1)* c && 1 <= c && c <= n } */
  /** { p*c == factorial(c) && 1 <= c && c <= n } */
  /** { p*c == factorial((c+1)- 1) && 1 <= c+1 && (c+1)-1 <= n
  p= p*c;
  /** { p == factorial((c+1)-1) && 1<= c+1 && (c+1)-1 <= n } */
  c= c+1;
  /** { p == factorial(c-1) && 1 <= c && c-1 <= n } */
  }
/** { p == factorial(c-1) && 1<= c && c-1 <= n &&
      factorial(c <= n) } */
/** { p == factorial(c-1) && 1<= c && c-1 <= n && c > n } */
/** { p == factorial(c-1) && 1<= c && c-1 == n } */
/** { p == factorial(n) } */
```

# Zurück zum Problem

- Man braucht Rechenverfahren, um $\Gamma \longrightarrow (A' \longrightarrow A)$ etc. zu beweisen
- Man braucht Regeln um Wahre Aussagen herleiten zu können
  - Analog zu operationaler Semantik, aber für logisches Schließen
  - Regelmenge K (Kalkülregeln) die Formeln aus anderen Formeln ableiten.

# Natürliches Schließen — Die Regeln

$$\frac{\phi \quad \psi}{\phi \,\&\&\, \psi}\ \&\&I \qquad \frac{\phi \,\&\&\, \psi}{\phi}\ \&\&E_L \qquad \frac{\phi \,\&\&\, \psi}{\psi}\ \&\&E_R$$

$$\frac{\begin{array}{c}[\phi]\\ \vdots \\ \psi\end{array}}{\phi \longrightarrow \psi}\ \longrightarrow I \qquad \frac{\phi \quad \phi \longrightarrow \psi}{\psi}\ \longrightarrow E$$

$$\frac{\mathbf{0}}{\phi}\ \mathbf{0} \qquad \frac{\begin{array}{c}[\phi \longrightarrow \mathbf{0}]\\ \vdots \\ \mathbf{0}\end{array}}{\phi}\ \text{raa}$$

# Die fehlenden Schlußregeln

$$\frac{\begin{array}{c}[\phi]\\ \vdots\\ \mathbf{0}\end{array}}{\neg\phi}\ \neg I \qquad \frac{\phi \quad \neg\phi}{\mathbf{0}}\ \neg E$$

$$\frac{\phi}{\phi \,||\, \psi}\ ||I_L \qquad \frac{\psi}{\phi \,||\, \psi}\ ||I_R \qquad \frac{\phi \,||\, \psi \qquad \begin{array}{c}[\phi]\\ \vdots\\ \sigma\end{array} \qquad \begin{array}{c}[\psi]\\ \vdots\\ \sigma\end{array}}{\sigma}\ ||E$$

$$\frac{\phi \longrightarrow \psi \quad \psi \longrightarrow \phi}{\phi \longleftrightarrow \psi}\ \longleftrightarrow I \qquad \frac{\phi \quad \phi \longleftrightarrow \psi}{\psi}\ \longleftrightarrow E_L \qquad \frac{\psi \quad \phi \longleftrightarrow \psi}{\phi}\ \longleftrightarrow E_R$$

# Regeln für die Gleichheit

- **Reflexivität**, **Symmetrie**, **Transitivität**:

$$\frac{}{x == x}\ \text{refl} \qquad \frac{x == y}{y == x}\ \text{sym} \qquad \frac{x == y \quad y == z}{x == z}\ \text{trans}$$

- **Kongruenz**:

$$\frac{x_1 == y_1, \ldots, x_n == y_n}{f(x_1, \ldots, x_n) == f(y_1, \ldots, y_n)}\ \text{cong}$$

- **Substitutivität**:

$$\frac{x_1 = y_1, \ldots, x_m = y_m \quad P(x_1, \ldots, x_m)}{P(y_1, \ldots, y_m)}\ \text{subst}$$

# Prädikatenlogik mit Induktion

- Sei $\mathrm{fix}(\hat{K})$ die Menge aller mittels Kalkülregeln ableitbaren Formeln.

# Prädikatenlogik mit Induktion

- Sei $\mathrm{fix}(\hat{K})$ die Menge aller mittels Kalkülregeln ableitbaren Formeln.
- $K$ ist korrekt:

$$\text{f.a. } F \in \mathrm{fix}(K) \text{ gilt: f.a. } \sigma,\ I.\ \mathcal{B}_v[\![F]\!]^I(\sigma) = 1.$$

# Prädikatenlogik mit Induktion

- Sei $\mathrm{fix}(\hat{K})$ die Menge aller mittels Kalkülregeln ableitbaren Formeln.
- $K$ ist korrekt:

$$\text{f.a. } F \in \mathrm{fix}(K) \text{ gilt: f.a. } \sigma, I.\ \mathcal{B}_v[\![F]\!]^I(\sigma) = 1.$$

- $K$ ist vollständig:

$$\text{f.a. } F \text{ mit f.a. } \sigma, I.\ \mathcal{B}_v[\![F]\!]^I(\sigma) = 1 \text{ gilt: } F \in \mathrm{fix}(K).$$

- Eine Logik $L$ ist entscheidbar, wenn für alle Formeln $F$ entschieden werden kann ob $\mathcal{B}_v[\![F]\!]^I(\sigma) = 1$ oder $\mathcal{B}_v[\![F]\!]^I(\sigma) = 0$

# Prädikatenlogik mit Induktion

- Sei $\mathrm{fix}(\hat{K})$ die Menge aller mittels Kalkülregeln ableitbaren Formeln.
- $K$ ist korrekt:

$$\text{f.a. } F \in \mathrm{fix}(K) \text{ gilt: f.a. } \sigma, I.\ \mathcal{B}_v[\![F]\!]^I(\sigma) = 1.$$

- $K$ ist vollständig:

$$\text{f.a. } F \text{ mit f.a. } \sigma, I.\ \mathcal{B}_v[\![F]\!]^I(\sigma) = 1 \text{ gilt: } F \in \mathrm{fix}(K).$$

- Eine Logik $L$ ist entscheidbar, wenn für alle Formeln $F$ entschieden werden kann ob $\mathcal{B}_v[\![F]\!]^I(\sigma) = 1$ oder $\mathcal{B}_v[\![F]\!]^I(\sigma) = 0$
- Für unsere Belange brauchen wir als Logik Prädikatenlogik mit Gleichheit und Induktion (wegen Rekursion):
  - Es gibt für diese Logik korrekte Kalküle, aber keine vollständigen (und sie ist auch nicht entscheidbar).

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 9 vom 05.06.18: Verifikationsbedingungen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;

y = x;

x = z;
// {X = y ∧ Y = x}
```

# Idee

- Hier ist ein einfaches Programm:

$// \{X = x \wedge Y = y\}$
z = y ;

y = x ;
$// \{X = y \wedge Y = z\}$
x = z ;
$// \{X = y \wedge Y = x\}$

# Idee

- Hier ist ein einfaches Programm:

$// \{X = x \wedge Y = y\}$
z = y ;
$// \{X = x \wedge Y = z\}$
y = x ;
$// \{X = y \wedge Y = z\}$
x = z ;
$// \{X = y \wedge Y = x\}$

# Idee

- Hier ist ein einfaches Programm:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

- Wir sehen:

  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Verifikation kann **berechnet** werden.

# Idee

- Hier ist ein einfaches Programm:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

- Wir sehen:

  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Verifikation kann **berechnet** werden.

- Geht das immer?

# Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — $P$ passt auf jede beliebige Nachbedingung.

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

# Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — $P$ passt auf jede beliebige Nachbedingung.

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Was ist mit den anderen Regeln?

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

# Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — $P$ passt auf jede beliebige Nachbedingung.

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Was ist mit den anderen Regeln? Nur **while** macht Probleme!

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

# Berechnung von Vorbedingungen

- Die Rückwärtsrechnung von einer gegebenen Nachbedingung entspricht der Berechnung einer Vorbedingung.
- Gegeben C0-Programm $c$, Prädikat $Q$, dann ist
  - $\mathrm{wp}(c, Q)$ die **schwächste Vorbedingung** $P$ so dass $\models \{P\}\, c\, \{Q\}$;
  - Prädikat $P$ **schwächer** als $P'$ wenn $P' \Longrightarrow P$
- Semantische Charakterisierung:

**Schwächste Vorbedingung**

Gegeben Zusicherung $Q \in \mathbf{Assn}$ und Programm $c \in \mathbf{Stmt}$, dann

$$\models \{P\}\, c\, \{Q\} \Longleftrightarrow P \Longrightarrow \mathrm{wp}(c, Q)$$

- Wie können wir $\mathrm{wp}(c, Q)$ berechnen?

# Berechnung von $\mathrm{wp}(c, Q)$

- Einfach für Programme ohne Schleifen:

$$
\begin{aligned}
\mathrm{wp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{wp}(x = e, P) &\stackrel{def}{=} P[e/x] \\
\mathrm{wp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wp}(c_1, \mathrm{wp}(c_2, P)) \\
\mathrm{wp}(\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{wp}(c_0, P)) \vee (\neg b \wedge \mathrm{wp}(c_1, P))
\end{aligned}
$$

- Für Schleifen: nicht entscheidbar.
  - "Cannot in general compute a **finite** formula" (Mike Gordon)
- Wir können rekursive Formulierung angeben:

$$
\mathrm{wp}(\textbf{while } (c)\ , P) \stackrel{def}{=} (\neg b \wedge P) \vee (b \wedge \mathrm{wp}(c, \mathrm{wp}(\textbf{while } (b)\ c, P)))
$$

  - Hilft auch nicht weiter...

# Lösung: Annotierte Programme

- Wir helfen dem Rechner weiter und **annotieren** die Schleifeninvariante am Programm.
- Damit berechnen wir:
  - die **approximative** schwächste Vorbedingung $\mathsf{awp}(c, Q)$
  - zusammen mit einer Menge von **Verifikationsbedingungen** $\mathsf{wvc}(c, Q)$
- Die Verifikationsbedingungen treten dort auf, wo die Weakening-Regel angewandt wird.
- Es gilt:

$$\bigwedge \mathsf{wvc}(c, Q) \Longrightarrow \models \{\mathsf{awp}(c, Q)\}\, c\, \{Q\}$$

# Approximative schwächste Vorbedingung

- Für die **while**-Schleife:

$$\begin{aligned}
\text{awp}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ \ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\text{wvc}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ \ i\ {*}/\ c, P) &\stackrel{def}{=} \text{wvc}(c, i, \\
&\quad ) \cup \{i \wedge b \longrightarrow \text{awp}(c, i)\} \\
&\quad \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}$$

- Entspricht der **while**-Regel (1) mit Weakening (2):

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}} \quad (1)$$

$$\frac{A \wedge b \Longrightarrow C \qquad \vdash \{C\}\, c\, \{A\} \qquad A \wedge \neg b \Longrightarrow B}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{B\}} \quad (2)$$

# Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(x = e, P) &\stackrel{def}{=} P[e/x] \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(\textbf{while } (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} i
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(x = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(\textbf{while } (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \\
&\qquad \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}
$$

$$
wvc(\{P\}\, c\, \{Q\}) \stackrel{def}{=} \{P \longrightarrow \mathrm{awp}(c, Q)\} \cup \mathrm{wvc}(c, Q)
$$

# Beispiel: das Fakultätsprogramm

- In der Praxis sind Vorbedingung gegeben, und nur die Verifikationsbedingungen relevant.
- Sei $F$ das annotierte Fakultätsprogramm:

```
1  // {0 ≤ n}
2  p= 1;
3  c= 1;
4  while (c <= n) /** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
6     c = c + 1;
7     }
8  // {p = n!}
```

- Berechnung der Verifikationsbedingungen zur Nachbedingung.

# Notation für Verifikationsbedingungen

```
1  // {0 ≤ n}
2  p= 1;
3  c= 1;
4  while (c <= n) /** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
6     c = c + 1;
7     }
8  // {p = n!}
```

**AWP**

| | |
|---|---|
| 6 | $p = ((c+1)-1)! \wedge ((c-1)+1) \leq n$ |
| 5 | $p \cdot c = ((c+1)-1)! \wedge ((c-1)+1) \leq n$ |
| 3 | $p = (1-1)! \wedge (1-1) \leq n$ |
| 2 | $1 = (1-1)! \wedge (1-1) \leq n$ |

**VC**

| | |
|---|---|
| 4 | $p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n) \longrightarrow$ $p \cdot c = ((c+1)-1)! \wedge ((c-1)+1) \leq n$ |
| 4 | $p = (c-1)! \wedge c-1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$ |
| 1 | $0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n$ |

# Vereinfachung von Verifikationsbedingungen

Wir nehmen folgende **strukturellen Vereinfachungen** an den generierten Verifikationsbedingungen vor:

1. Auswertung konstanter arithmetischer Ausdrücke, einfache arithmetische Gesetze
   - Bsp. $(x + 1) - 1 \rightsquigarrow x$, $1 - 1 \rightsquigarrow 0$
2. Normalisierung der Relationen (zu $<$, $\leq$, $=$, $\neq$) und Vereinfachung
   - Bsp: $\neg(x \leq y) \rightsquigarrow x > y \rightsquigarrow y < x$
3. Konjunktionen in der Konklusion werden zu einzelnen Verifikationsbedingungen
   - Bsp: $A_1 \wedge A_2 \wedge A_3 \longrightarrow P \wedge Q \rightsquigarrow A_1 \wedge A_2 \wedge A_3 \longrightarrow P, A_1 \wedge A_2 \wedge A_3 \longrightarrow Q$
4. Alle Bedingungen mit einer Prämisse *false* oder einer Konklusion *true* sind trivial erfüllt.

# Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i!= n) /** inv (∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n */
5   {
6    if (a[r] < a[i]) {
7       r= i;
8       }
9    else {
10      }
11   i= i+1;
12   }
13 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

- Sehr lange Verifikationsbedingungen (u.a. wegen Fallunterscheidung)
- Wie können wir das beheben?

# Spracherweiterung: Explizite Spezifikationen

- Erweiterung der Sprache C0 um Invarianten für Schleifen und **explizite Zusicherung**

$$\begin{array}{lll} \textbf{Assn} & a ::= \ldots & \text{— Zusicherungen} \\ \textbf{Stmt} & c ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\ & \mid & \textbf{while}\ (b)\ /{*}{*}\,\textbf{inv}\ a\ {*}/\ c \\ & \mid & /{*}{*}\{a\}\ {*}/ \end{array}$$

- Zusicherungen haben **keine Semantik** (Kommentar!), sondern erzwingen eine neue Vorbedingung.
- Dazu vereinfachte Regel für Fallunterscheidung:

$$\text{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, \stackrel{def}{=})(b \wedge \text{awp}(c_0, P)) \vee (\neg b \wedge \text{awp}(c_1, P))$$

Wenn $\text{awp}(c_0, P) = b \wedge P_0$, $\text{awp}(c_1, P) = \neg b \wedge P_0$, dann gilt

$$(b \wedge b \wedge P_0) \vee (\neg b \wedge \neg b \wedge P_0) = (b \wedge P_0) \vee (\neg b \wedge P_0) = (b \vee \neg b) \wedge P_0 = P_0$$

# Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(x = e, P) &\stackrel{def}{=} P[e/x] \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} Q \quad \text{wenn} \quad \mathrm{awp}(c_0, P) = b \wedge Q,\ \mathrm{awp}(c_1, P) = \neg b \wedge Q \\
\mathrm{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(/{**}\{q\}\ {*}/, P) &\stackrel{def}{=} \mathrm{q} \\
\mathrm{awp}(\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\\
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(x = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(/{**}\{q\}\ {*}/, P) &\stackrel{def}{=} \{q \longrightarrow P\} \\
\mathrm{wvc}(\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}
$$

# Maximales Element mit Zusicherung

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i!= n) /** inv (∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n */
5   {
6    if (a[r] < a[i]) {
7      /** {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ a[r] < a[i]} */
8      r= i;
9      }
10   else {
11     /** {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])} */
12     }
13   i= i+1;
14   }
15 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

- Explizite Zusicherungen verkleinern Verifikationsbedingung

# Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls lassen sich, weitgehend schematisch, rückwärts (vom Ende her) anwenden — nur Schleifen machen Probleme.
- Wir **annotieren** daher die Invarianten an Schleifen, und können dann die schwächste Vorbedingung und Verifikationsbedingungen automatisch berechnen.
  - Dabei sind die **Verifikationsbedingungen** das interessante.
- Um die Verifikationsbedingungen zu vereinfachen führen wir **explizite Zusicherungen** in C0 ein
- Die Generierung von Verifikationsbedingungen korrespondiert zur relativen Vollständigkeit der Floyd-Hoare-Logik.
- Nächste Woche: warum eigentlich immer **rückwärts**?

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 10 vom 12.06.18: Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Idee

- Hier ist ein einfaches Programm:

$// \{X = x \wedge Y = y\}$
z = y;

y = x;

x = z;
$// \{X = y \wedge Y = x\}$

# Idee

- Hier ist ein einfaches Programm:

$// \{X = x \wedge Y = y\}$
z = y ;

y = x ;
$// \{X = y \wedge Y = z\}$
x = z ;
$// \{X = y \wedge Y = x\}$

# Idee

- Hier ist ein einfaches Programm:

$// \{X = x \wedge Y = y\}$
z = y ;
$// \{X = x \wedge Y = z\}$
y = x ;
$// \{X = y \wedge Y = z\}$
x = z ;
$// \{X = y \wedge Y = x\}$

# Idee

- Hier ist ein einfaches Programm:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

- Wir haben gesehen:

  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Verifikation kann **berechnet** werden.

# Idee

- Hier ist ein einfaches Programm:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

- Wir haben gesehen:

  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Verifikation kann **berechnet** werden.

- Muss das rückwärts sein? Warum nicht vorwärts? Was ist der Vorteil?

# Nachteile der Rückwärtsberechnung

```
// {i ≠ 3}
.
.  /* 400 Zeilen, die
.     i nicht verändern */
.
a[i] = 5;
// {a[3] = 7}
```

Errechnete Vorbedingung (AWP):

$$(a[3] = 7)[5/a[i]]$$

- Kann nicht vereinfacht werden, weil wir nicht wissen, ob $i \neq 3$
- AWP wird **sehr groß**.
- Das Problem wächst mit der Länge der Programme.

# Der Floyd-Hoare-Kalkül Vorwärts

# Vorwärtsanwendung der Regeln

- Zuweisungsregel kann nicht vorwärts angewandt werden, weil die Vorbedingung keine offene Variable ist:

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

# Vorwärtsanwendung der Regeln

- Zuweisungsregel kann nicht vorwärts angewandt werden, weil die Vorbedingung keine offene Variable ist:

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Die anderen Regeln passen:

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

# Zuweisungregel Vorwärts

- Alternative Zuweisungsregel (nach Floyd):

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = e[V/x] \wedge P[V/x]\}}$$

  - $FV(P)$ sind die **freien** Variablen in $P$.

- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden

- Gilt auch für die anderen Regeln.

# Vorwärtsverkettung

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = (e[V/x]) \wedge P[V/x]\}}$$

```
// {0 ≤ x}
x= 2*y ;
// {∃V1. 0 ≤ V1 ∧ x = 2 · y}
x= x+1;
// {∃V2. (∃V1. 0 ≤ V1 ∧ x = 2 · y)[V2/x] ∧ x = (x + 1)[V2/x]}
```

- **Vereinfachung** der letzten Nachbedingung:

$$\exists V_2.\, (\exists V_1.0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x]$$

# Vorwärtsverkettung

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = (e[V/x]) \wedge P[V/x]\}}$$

// $\{0 \leq x\}$
x= 2*y ;
// $\{\exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y\}$
x= x+1;
// $\{\exists V_2.\, (\exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x]\}$

- **Vereinfachung** der letzten Nachbedingung:

$$\begin{aligned} & \exists V_2.\, (\exists V_1.0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x] \\ \Longleftrightarrow\ & \exists V_2.\, (\exists V_1.\, 0 \leq V_1 \wedge V_2 = 2 \cdot y) \wedge x = V_2 + 1 \end{aligned}$$

# Vorwärtsverkettung

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = (e[V/x]) \wedge P[V/x]\}}$$

// $\{0 \leq x\}$
x= 2*y ;
// $\{\exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y\}$
x= x+1;
// $\{\exists V_2.\, (\exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x]\}$

- **Vereinfachung** der letzten Nachbedingung:

$$\begin{aligned} & \exists V_2.\, (\exists V_1.0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x] \\ \Longleftrightarrow\ & \exists V_2.\, (\exists V_1.\, 0 \leq V_1 \wedge V_2 = 2 \cdot y) \wedge x = V_2 + 1 \\ \Longleftrightarrow\ & \exists V_2.\, \exists V_1.\, 0 \leq V_1 \wedge x = V_2 + 1 \wedge V_2 = 2 \cdot y \end{aligned}$$

# Vorwärtsverkettung

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.\, x = (e[V/x]) \wedge P[V/x]\}}$$

// $\{0 \leq x\}$
x= 2*y ;
// $\{\exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y\}$
x= x+1;
// $\{\exists V_2.\, (\exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x]\}$

- **Vereinfachung** der letzten Nachbedingung:

$$\begin{aligned}
& \exists V_2.\, (\exists V_1.0 \leq V_1 \wedge x = 2 \cdot y)[V_2/x] \wedge x = (x+1)[V_2/x] \\
\iff & \exists V_2.\, (\exists V_1.\, 0 \leq V_1 \wedge V_2 = 2 \cdot y) \wedge x = V_2 + 1 \\
\iff & \exists V_2.\, \exists V_1.\, 0 \leq V_1 \wedge x = V_2 + 1 \wedge V_2 = 2 \cdot y \\
\iff & \exists V_1.\, 0 \leq V_1 \wedge x = 2 \cdot y + 1
\end{aligned}$$

# Regeln der Vorwärtsverkettung

1. Wenn $x$ nicht in Vorbedingung auftritt, dann $P[V/x] \equiv P$.
2. Wenn $x$ nicht in rechter Seite $e$ auftritt, dann $e[V/x] \equiv e$.
3. Wenn beides der Fall ist, kann der Existenzquantor wegfallen:
$$V \notin FV(P) \Longrightarrow \exists V.\, P \equiv P$$
4. Wenn $x$ vorher zugewiesen wurde, Vereinfachung mit
$$\exists V.\, P[V] \wedge V = t \Longrightarrow P[t/V]$$

# Vorwärtsverkettung

- Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- Vorteil: Vorbedingung bleibt kleiner
- Nachteil: in der Anwendung **umständlicher**
- Vereinfachung benötigt Lemma: $\exists x.P(x) \wedge x = t \iff P(t)$

**Zwischenfazit: Der Floyd-Hoare-Kalkül ist symmetrisch**

Es gibt zwei Zuweisungsregeln, eine für die **Rückwärtsanwendung** von Regeln, eine für die **Vorwärtsanwendung**

# Vorwärtsberechnung von Verifikationsbedingungen

# Stärkste Nachbedingung

- Vorwärtsberechnung von Verifikationsbedingungen: Nachbedingung
- Gegeben C0-Programm $c$, Prädikat $P$, dann ist
  - $\mathrm{sp}(P, c)$ die **stärkste Nachbedingung** $Q$ so dass $\models \{P\}\, c\, \{Q\}$
  - Prädikat $Q$ **stärker** als $Q'$ wenn $Q \Longrightarrow Q'$.
- Semantische Charakterisierung:

**Stärkste Nachbedingung**

Gegeben Zusicherung $P \in \mathbf{Assn}$ und Programm $c \in \mathbf{Stmt}$, dann

$$\models \{P\}\, c\, \{Q\} \Longleftrightarrow \mathrm{sp}(P, c) \Longrightarrow Q$$

- Wie können wir $\mathrm{sp}(P, c)$ berechnen?

# Berechnung von Nachbedingungen

- Wir berechnen die **approximative** stärkste Nachbedingung.
- Viele Klauseln sind ähnlich der schwächsten Vorbedingung.
- Ausnahmen:
  - While-Schleife: andere Verifikationsbedingungen
  - If-Anweisung: Weakening eingebaut
  - **Zuweisung**: Vorwärtsregel
- Nach jeder Zuweisung Nachbedingung **vereinfachen**

# Überblick: Approximative stärkste Nachbedingung

$$\mathsf{asp}(P)\{\,\} \stackrel{def}{=} P$$

# Überblick: Approximative stärkste Nachbedingung

$$\begin{aligned} \mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\ \mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \end{aligned}$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{*}{*}\{q\}\ {*}/ &\stackrel{def}{=} q
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathsf{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathsf{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)x = e &\stackrel{def}{=} \emptyset
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathsf{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)x = e &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{svc}(P)c_1 \cup \mathsf{svc}(\mathsf{asp}(c_1)P)c_2
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathsf{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)x = e &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{svc}(P)c_1 \cup \mathsf{svc}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{svc}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{svc}(P \wedge b)c_0 \cup \mathsf{svc}(P \wedge \neg b)c_1
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathsf{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)x = e &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{svc}(P)c_1 \cup \mathsf{svc}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{svc}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{svc}(P \wedge b)c_0 \cup \mathsf{svc}(P \wedge \neg b)c_1 \\
\mathsf{svc}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} \{P \longrightarrow q\}
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathsf{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathsf{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathsf{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{asp}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{asp}(b \wedge P)c_0 \vee \mathsf{asp}(\neg b \wedge P)c_1 \\
\mathsf{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathsf{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathsf{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)x = e &\stackrel{def}{=} \emptyset \\
\mathsf{svc}(P)c_1; c_2 &\stackrel{def}{=} \mathsf{svc}(P)c_1 \cup \mathsf{svc}(\mathsf{asp}(c_1)P)c_2 \\
\mathsf{svc}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathsf{svc}(P \wedge b)c_0 \cup \mathsf{svc}(P \wedge \neg b)c_1 \\
\mathsf{svc}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} \{P \longrightarrow q\} \\
\mathsf{svc}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} \mathsf{svc}(i \wedge b)c \cup \{P \longrightarrow i\} \\
&\qquad\qquad \cup \{\mathsf{asp}(i \wedge b)c \longrightarrow i\}
\end{aligned}
$$

# Überblick: Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathrm{asp}(P)\{\,\} &\stackrel{def}{=} P \\
\mathrm{asp}(P)x = e &\stackrel{def}{=} \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\mathrm{asp}(P)c_1; c_2 &\stackrel{def}{=} \mathrm{asp}(\mathrm{asp}(c_1)P)c_2 \\
\mathrm{asp}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathrm{asp}(b \wedge P)c_0 \vee \mathrm{asp}(\neg b \wedge P)c_1 \\
\mathrm{asp}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} q \\
\mathrm{asp}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} i \wedge \neg b \\
\\
\mathrm{svc}(P)\{\,\} &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P)x = e &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P)c_1; c_2 &\stackrel{def}{=} \mathrm{svc}(P)c_1 \cup \mathrm{svc}(\mathrm{asp}(c_1)P)c_2 \\
\mathrm{svc}(P)\textbf{if } (b)\ c_0\ \textbf{else}\ c_1 &\stackrel{def}{=} \mathrm{svc}(P \wedge b)c_0 \cup \mathrm{svc}(P \wedge \neg b)c_1 \\
\mathrm{svc}(P)/{**}\{q\}\ {*}/ &\stackrel{def}{=} \{P \longrightarrow q\} \\
\mathrm{svc}(P)\textbf{while } (b)\ /{**}\,\textbf{inv}\ i\ {*}/\ c &\stackrel{def}{=} \mathrm{svc}(i \wedge b)c \cup \{P \longrightarrow i\} \\
&\qquad\qquad \cup \{\mathrm{asp}(i \wedge b)c \longrightarrow i\} \\
\\
\mathit{svc}(\{P\}\,c\,\{Q\}) &\stackrel{def}{=} \{\mathrm{asp}(P)c \longrightarrow Q\} \cup \mathrm{svc}(P)c
\end{aligned}
$$

# Beispiel: Fakultät

```
1  // {0 ≤ n}
2  p= 1;
3  c= 1;
4  while (c <= n) /** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
6     c = c + 1;
7     }
8  // {p = n!}
```

# Fakultät: Stärkste Vorbedingung

Notation: $asp_x$ = Stärkste Nachbedingung **nach** Zeile $x$.

$$
\begin{aligned}
asp_2 = \;& \exists V.\, 0 \leq n[V/p] \wedge p = (1[V/p]) \\
& \rightsquigarrow 0 \leq n \wedge p = 1 \\
asp_3 = \;& \exists V.\, (0 \leq n \wedge p = 1)[V/c] \wedge c = (1[V/c]) \\
& \rightsquigarrow 0 \leq n \wedge p = 1 \wedge c = 1 \\
asp_4 = \;& \neg(c \leq n) \wedge p = (c-1)! \wedge c - 1 \leq n \\
asp_5 = \;& \exists V_1.\, (p = (c-1)! \wedge (c-1) \leq n \wedge c \leq n)[V_1/p] \\
& \quad \wedge p = (p \cdot c)[V_1/p] \\
& \rightsquigarrow \exists V_1.\, (V_1 = (c-1)! \wedge (c-1) \leq n \wedge c \leq n) \wedge p = (V_1 \cdot c) \\
& \rightsquigarrow c - 1 \leq n \wedge c \leq n \wedge p = (c-1)! \cdot c \\
asp_6 = \;& \exists V_2.\, (c - 1 \leq n \wedge c \leq n \wedge p = (c-1)! \cdot c)[V_2/c] \\
& \quad \wedge c = (c+1)[V_2/c] \\
& \rightsquigarrow \exists V_2.\, (V_2 - 1 \leq n \wedge V_2 \leq n \wedge p = (V_2 - 1)! \cdot V_2) \wedge c = (V_2 + 1) \\
& \rightsquigarrow c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1)
\end{aligned}
$$

# Fakultät: Verifikationsbedingungen

Notation: $vc_x$ = in Zeile $x$ generierte Verifikationsbedingung

$$\begin{aligned} vc_4 = \quad & \{asp_3 \longrightarrow p = (c-1)! \wedge c - 1 \leq n, \\ & \quad asp_6 \longrightarrow p = (c-1)! \wedge c - 1 \leq n\} \\ vc_8 = \quad & asp_4 \longrightarrow p = n! \end{aligned}$$

Vereinfachung: $vc_4 \Longleftrightarrow vc_{4.1} \wedge vc_{4.2} \wedge vc_{4.3} \wedge vc_{4.4}$

$$\begin{aligned} vc_{4.1} \quad & = 0 \leq n \wedge p = 1 \wedge c = 1 \longrightarrow p = (c-1)! \\ & = 0 \leq n \longrightarrow 1 = (1-1)! \\ vc_{4.2} \quad & = 0 \leq n \wedge p = 1 \wedge c = 1 \longrightarrow c - 1 \leq n \\ & = 0 \leq n \longrightarrow 0 \leq n \\ vc_{4.3} \quad & = c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1) \longrightarrow p = (c-1)! \\ & = c - 1 \leq n \wedge p = (c-2)! \cdot (c-1) \longrightarrow p = (c-1)! \\ vc_{4.4} \quad & = c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1) \longrightarrow c - 1 \leq n \\ & = c - 1 \leq n \longrightarrow c - 1 \leq n \\ vc_8 \quad & = n \leq c - 1 \wedge c - 1 \leq n \wedge p = (c-1)! \longrightarrow p = n! \end{aligned}$$

# Beispiel: Suche nach dem Maximalen Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i!= n) /** inv (∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r <
   n */ {
5     if (a[r] < a[i]) {
6        r= i;
7        }
8     else {
9        }
10    i= i+1;
11    }
12 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

- Problem: wir müssen u.a. zeigen

$$(\exists V_1.\,(\forall j.\,0 \le j < i-1 \longrightarrow a[j] \le a[V_1]) \wedge i-1 \neq n \wedge a[V_1] < a[i-1] \wedge r = i-1) \longrightarrow 0 \le r < n$$

Deshalb: Invariante **verstärken**!

# Beispiel: Suche nach dem Maximalen Element

Verstärkte Invariante (und Schleifenbedingung):

```
1   // {0 < n}
2   i= 0;
3   r= 0;
4   while (i< n) /** inv  (∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r])   */
                          ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n
5   {
6     if (a[r] < a[i]) {
7       r= i;
8       }
9     else {
10      }
11    i= i+1;
12    }
13  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$(\exists V_1.\,(\forall j.\,0 \le j < i-1 \longrightarrow a[j] \le a[V_1]) \wedge \\ 0 \le i-1 < n \wedge a[V_1] < a[i-1] \wedge r = i-1) \longrightarrow 0 \le r < n$$

# Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls sind **symmetrisch**: die Zuweisungsregel gibt es“rückwärts” und “vorwärts”.
- Dual zu Beweis und Verifikationsbedingung rückwärts gibt es Regel und Verifkationsbedingungen vorwärts. automatisch prüfen.
- Kern der Vorwärtsberechnung ist die Zuweisungsregel nach Floyd.
- Vorwärtsberechnung erzeugt kleinere Terme, ist aber umständlicher zu handhaben.
- Rückwärtsberechnung ist einfacher zu handhaben, erzeugt aber (tendenziell sehr) große Terme.

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 11 vom 19.06.18: Funktionen und Prozeduren

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Funktionen & Prozeduren

- **Funktionen** sind das zentrale Modularisierungskonzept von C
  - Kleinste Einheit
  - NB. Prozeduren sind nur Funktionen vom Typ **void**
- In objektorientierten Sprachen: Methoden
  - Funktionen mit (implizitem) erstem Parameter **this**
- Wie behandeln wir Funktionen?

# Modellierung und Spezifikation von Funktionen

Wir brauchen:

(1) Von Anweisungen zu Funktionen: Deklarationen und Parameter

# Modellierung und Spezifikation von Funktionen

Wir brauchen:

(1) Von Anweisungen zu Funktionen: Deklarationen und Parameter

(2) Semantik von Funktionsdefinition und Funktionsaufruf

# Modellierung und Spezifikation von Funktionen

Wir brauchen:

(1) Von Anweisungen zu Funktionen: Deklarationen und Parameter

(2) Semantik von Funktionsdefinition und Funktionsaufruf

(3) Spezifikation von Funktionen

# Modellierung und Spezifikation von Funktionen

Wir brauchen:

(1) Von Anweisungen zu Funktionen: Deklarationen und Parameter

(2) Semantik von Funktionsdefinition und Funktionsaufruf

(3) Spezifikation von Funktionen

(4) Beweisregeln für Funktionsdefinition und Funktionsaufruf

# Von Anweisungen zu Funktionen

- Erweiterung unserer Kernsprache um Funktionsdefinition und Deklarationen:

$$
\begin{aligned}
\textbf{FunDef} &::= \textbf{FunHeader}\ \textbf{FunSpec}^{+}\ \textbf{Blk} \\
\textbf{FunHeader} &::= \textbf{Type}\ \textbf{Idt}(\textbf{Decl}^{*}) \\
\textbf{Decl} &::= \textbf{Type}\ \textbf{Idt} \\
\textbf{Blk} &::= \{\textbf{Decl}^{*}\ \textbf{Stmt}\} \\
\textbf{Type} &::= \textbf{char} \mid \textbf{int} \mid \textbf{Struct} \mid \textbf{Array} \\
\textbf{Struct} &::= \textbf{struct}\ \ \textbf{Idt}^{?}\ \{\textbf{Decl}^{+}\} \\
\textbf{Array} &::= \textbf{Type}\ \textbf{Idt}[\textbf{Aexp}]
\end{aligned}
$$

- Abstrakte Syntax (konkrete Syntax mischt **Type** und **Idt**, Kommata bei Argumenten, ...)
- Größe von Feldern: **konstanter** Ausdruck
- **FunSpec** später

# Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe

- Return-Anweisung

$$
\begin{array}{rll}
\textbf{Aexp} & a ::= & \mathbf{Z} \mid \mathbf{C} \mid \mathbf{Lexp} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \\
& \mid & \mathbf{Idt}(\mathbf{Exp}^*) \\
\textbf{Bexp} & b ::= & \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,||\, b_2 \\
\textbf{Exp} & e := & \mathbf{Aexp} \mid \mathbf{Bexp} \\
\textbf{Stmt} & c ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \mathbf{if}\ (b)\ c_1\ \mathbf{else}\ c_2 \\
& \mid & \mathbf{while}\ (b)\ /{**}\ \mathbf{inv}\ \ a\ {*}/\ c \mid /{**}\{a\}\ {*}/ \\
& \mid & \mathbf{Idt}(a^*) \\
& \mid & \mathbf{return}\ \ a^?
\end{array}
$$

# Rückgabewerte

- Problem: **return** bricht sequentiellen Kontrollfluss:

```
if (x == 0) return -1;
y = y / x;      // Wird nicht immer erreicht
```

- Lösung 1: verbieten!
  - MISRA-C (Guidelines for the use of the C language in critical systems):

**Rule 14.7 (required)**

A function shall have a single point of exit at the end of the function.

  - Nicht immer möglich, unübersichtlicher Code...

- Lösung 2: Erweiterung der Semantik von $\Sigma \rightharpoonup \Sigma$ zu $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$

# Erweiterte Semantik

- Denotat einer Anweisung: $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$
- Abbildung von Ausgangszustand $\Sigma$ auf:
  - Sequentieller Folgezustand, oder
  - Rückgabewert und Rückgabezustand
- Was ist mit **void**?

# Erweiterte Semantik

- Denotat einer Anweisung: $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$
- Abbildung von Ausgangszustand $\Sigma$ auf:
  - Sequentieller Folgezustand, oder
  - Rückgabewert und Rückgabezustand
- Was ist mit **void**?
  - Erweiterte Werte: $\mathbf{V}_U \stackrel{def}{=} \mathbf{V} + \{*\}$
- Komposition zweier Anweisungen $f, g : \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$:

$$g \circ_S f(\sigma) \stackrel{def}{=} \begin{cases} g(\sigma') & f(\sigma) = \sigma' \\ (\sigma', v) & f(\sigma) = (\sigma', v) \end{cases}$$

# Semantik von Anweisungen

$$\mathcal{C}[\![.]\!] : \mathbf{Stmt} \to \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$$

$$\mathcal{C}[\![x = e]\!] = \{(\sigma, \sigma[a/x]) \mid (\sigma, a) \in \mathcal{A}[\![e]\!]\}$$

$$\mathcal{C}[\![c_1; c_2]\!] = \mathcal{C}[\![c_1]\!] \circ_S \mathcal{C}[\![c_2]\!] \qquad \text{Komposition wie oben}$$

$$\mathcal{C}[\![\{\ \}]\!] = \mathbf{Id}_\Sigma \qquad \mathbf{Id}_\Sigma := \{(\sigma, \sigma) | \sigma \in \Sigma\}$$

$$\begin{aligned}\mathcal{C}[\![\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1]\!] = &\{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \\ &\cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\ &\text{mit } \sigma' \in \Sigma \cup (\Sigma \times \mathbf{V}_U)\end{aligned}$$

$$\mathcal{C}[\![\textbf{return }\ e]\!] = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in \mathcal{A}[\![e]\!]\}$$

$$\mathcal{C}[\![\textbf{return}]\!] = \{(\sigma, (\sigma, *))\}$$

$$\mathcal{C}[\![\textbf{while } (b)\ c]\!] = \mathit{fix}(\Gamma)$$

$$\begin{aligned}\Gamma(\psi) \stackrel{\mathit{def}}{=} &\{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \psi \circ_S \mathcal{C}[\![c]\!]\} \\ &\cup \{(\sigma, \sigma) \mid (\sigma, \mathit{false}) \in \mathcal{B}[\![b]\!]\}\end{aligned}$$

# Semantik von Funktionsdefinitionen

$$\mathcal{D}_{fd}[\![.]\!] : \mathbf{FunDef} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$

Das Denotat einer Funktion ist eine Anweisung, die über den tatsächlichen Werten für die Funktionsargumente parametriert ist.

$$\begin{aligned}\mathcal{D}_{fd}[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!] = \\ \lambda v_1, \ldots, v_n.\, \{(\sigma, (\sigma', v)) \mid \\ (\sigma, (\sigma', v)) \in \mathcal{D}_{blk}[\![blk]\!] \circ_S \{(\sigma, \sigma[v_1/p_1, \ldots, v_n/p_n])\}\}\end{aligned}$$

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.
- Von $\mathcal{D}_{blk}[\![blk]\!]$ sind nur **Rückgabezustände** interessant.
  - Kein „fall-through“

# Semantik von Blöcken und Deklarationen

$$\mathcal{D}_{blk}[\![.]\!] : \mathbf{Blk} \rightharpoonup \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$$
$$\mathcal{D}_{d}[\![.]\!] : \mathbf{Decl} \rightharpoonup \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$$

Blöcke bestehen aus Deklarationen und einer Anweisung:

$$\mathcal{D}_{blk}[\![decls\ stmts]\!] = \mathcal{C}[\![stmts]\!] \circ_S \mathcal{D}_d[\![decls]\!]$$
$$\mathcal{D}_d[\![t\ i]\!] = \{(\sigma, \sigma[\perp / i])\}$$

- Verallgemeinerung auf Sequenz von Deklarationen

# Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $\mathcal{D}_{fd}[\![f]\!]$
- Call by name, call by value, call by reference. . . ?
  - C kennt nur call by value (C-Standard 99, §6.9.1. (10))
  - Was ist mit **Seiteneffekten?** Wie können wir Werte **ändern**?
    - Erst mal gar nicht. . .

# Funktionsaufrufe

- Um eine Funktion $f$ aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von $f$ dem Bezeichner $f$ zuordnen.

- Deshalb brauchen wir eine **Umgebung** (Environment):

$$\begin{aligned}\mathbf{Env} &= Id \rightharpoonup [\![\mathbf{FunDef}]\!] \\ &= Id \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)\end{aligned}$$

- Das Environment ist **zusätzlicher Parameter** für alle Definitionen

# Semantik von Funktionsaufrufen

$$\mathcal{A}[\![f(t_1,\ldots,t_n)]\!]\Gamma = \{(\sigma,v) \mid \exists \sigma', v.\, (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \\ \wedge (\sigma,v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\}$$

$$\mathcal{C}[\![f(t_1,\ldots,t_n)]\!]\Gamma = \{(\sigma,\sigma') \mid \exists \sigma'.\, (\sigma,(\sigma',*)) \in \Gamma(f)(v_1,\ldots,v_n) \\ \wedge (\sigma,v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\}$$

- Aufruf einer nicht-definierten Funktion $f$ oder mit falscher Anzahl $n$ von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- Aufruf von Funktion $\mathcal{A}[\![f(t_1,\ldots,t_n)]\!]$ ignoriert Endzustand
- Aufruf von Prozedur $\mathcal{C}[\![f(t_1,\ldots,t_n)]\!]$ ignoriert Rückgabewert

# Semantik von Funktionsaufrufen

$$\mathcal{A}[\![f(t_1,\ldots,t_n)]\!]\Gamma = \{(\sigma,v) \mid \exists \sigma', v.\, (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \wedge (\sigma,v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\}$$

$$\mathcal{C}[\![f(t_1,\ldots,t_n)]\!]\Gamma = \{(\sigma,\sigma') \mid \exists \sigma'.\, (\sigma,(\sigma',*)) \in \Gamma(f)(v_1,\ldots,v_n) \wedge (\sigma,v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\}$$

$$\mathcal{C}[\![x = f(t_1,\ldots,t_n)]\!]\Gamma = \{(\sigma,\sigma'[v/x]) \mid \exists \sigma', v.\, (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \wedge (\sigma,v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\}$$

- Aufruf einer nicht-definierten Funktion $f$ oder mit falscher Anzahl $n$ von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- Aufruf von Funktion $\mathcal{A}[\![f(t_1,\ldots,t_n)]\!]$ ignoriert Endzustand
- Aufruf von Prozedur $\mathcal{C}[\![f(t_1,\ldots,t_n)]\!]$ ignoriert Rückgabewert
- Besser: Kombination mit Zuweisung

# Spezifikation von Funktionen

- Wir **spezifizieren** Funktionen durch **Vor-** und **Nachbedingungen**
  - Ähnlich den Hoare-Tripeln, aber vereinfachte Syntax
  - **Behavioural specification**, angelehnt an JML, OCL, ACSL (Frama-C)
- Syntaktisch:

$$\textbf{FunSpec} ::= /** \ \textbf{pre}\ \textbf{Bexp}\ \textbf{post}\ \textbf{Bexp}\ */$$

| | | |
|---|---|---|
| Vorbedingung | **pre** sp; | $\Sigma \to \mathbb{B}$ |
| Nachbedingung | **post** sp; | $\Sigma \times (\Sigma \times \mathbf{V}_U) \to \mathbb{B}$ |
| | **old**(e) | Wert von $e$ im **Vorzustand** |
| | **result** | **Rückgabewert** der Funktion |

# Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!;
 */
{
 int p;
 int c;

 p= 1;
 c= 1;
 while (c<= n) /** inv p == (c − 1)! ∧ c ≤ n + 1 ∧ 0 < c */ {
    p= p*c;
    c= c+1;
    }
 return p;
}
```

# Beispiel: Suche

```
int findmax(int a[], int a_len)
  /** pre  \array(a, a_len); */
  /** post ∀i. 0 ≤ i < a_len ⟶ a[i] ≤ \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /** inv (∀i. 0 ≤ i < j ⟶ a[i] ≤ x) ∧ j ≤ a_len; */
    {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```

# Semantik von Spezifikationen

- Vorbedingung: Auswertung als $\mathcal{B}[\![sp]\!]\,\Gamma$ über dem Vorzustand
- Nachbedingung: Erweiterung von $\mathcal{B}[\![.]\!]$ und $\mathcal{A}[\![.]\!]$
  - Ausdrücke können in Vor- oder Nachzustand ausgewertet werden.
  - **result**kann nicht in Funktionen vom Typ **void** auftreten.

$$\mathcal{B}_{sp}[\![.]\!] : \mathbf{Env} \to \mathbf{Bexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B}$$

$$\mathcal{A}_{sp}[\![.]\!] : \mathbf{Env} \to \mathbf{Aexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}$$

$$\mathcal{B}_{sp}[\![!b]\!]\,\Gamma = \{((\sigma,(\sigma',v)),1) \mid ((\sigma,(\sigma',v)),\mathit{false}) \in \mathcal{B}_{sp}[\![b]\!]\Gamma\} \cup \{((\sigma,(\sigma',v)),0) \mid ((\sigma,(\sigma',v)),\mathit{true}) \in \mathcal{B}_{sp}[\![b]\!]\Gamma$$

$$\ldots$$

$$\mathcal{B}_{sp}[\![\backslash\mathbf{old}(e)]\!]\,\Gamma = \{((\sigma,(\sigma',v)),b) \mid (\sigma,b) \in \mathcal{B}[\![e]\!]\,\Gamma\}$$

$$\mathcal{A}_{sp}[\![\backslash\mathbf{old}(e)]\!]\,\Gamma = \{((\sigma,(\sigma',v)),a) \mid (\sigma,a) \in \mathcal{A}[\![e]\!]\,\Gamma\}$$

$$\mathcal{A}_{sp}[\![\backslash\mathbf{result}]\!]\,\Gamma = \{((\sigma,(\sigma,v)),v)\}$$

$$\mathcal{B}_{sp}[\![\mathbf{pre}\ p\ \mathbf{post}\ q]\!]\,\Gamma = \{(\sigma,(\sigma',v)) \mid \sigma \in \mathcal{B}[\![p]\!]\,\Gamma \wedge (\sigma',(\sigma,v)) \in \mathcal{B}_{sp}[\![p]\!]\,\Gamma\}$$

# Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre } p \textbf{ post } q \models \mathit{FunDef}$$
$$\iff \forall v_1, \ldots, v_n. \mathcal{D}_{fd}[\![\mathit{FunDef}]\!]\,\Gamma \in \mathcal{B}_{sp}[\![\textbf{pre } p \textbf{ post } q]\!]\,\Gamma$$

  - $\Gamma$ enthält globale Definitionen, insbesondere andere Funktionen.

- Wie passt das zu $\models \{P\}\, c\, \{Q\}$ für Hoare-Tripel?

- Wie **beweisen** wir das?

# Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre } p \textbf{ post } q \models \mathit{FunDef}$$
$$\iff \forall v_1, \ldots, v_n. \mathcal{D}_{fd}[\![\mathit{FunDef}]\!]\,\Gamma \in \mathcal{B}_{sp}[\![\textbf{pre } p \textbf{ post } q]\!]\,\Gamma$$

  - $\Gamma$ enthält globale Definitionen, insbesondere andere Funktionen.

- Wie passt das zu $\models \{P\}\, c\, \{Q\}$ für Hoare-Tripel?

- Wie **beweisen** wir das? **Erweiterung** des Hoare-Kalküls

# Erweiterung des Floyd-Hoare-Kalküls

$$\mathcal{C}[\![.]\!] : \mathbf{Stmt} \to \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$$

Hoare-Tripel: zusätzliche Spezifikation für **Rückgabewert**.

**Partielle Korrektheit ($\models \{P\}\, c\, \{Q|Q_R\}$)**

$c$ ist **partiell korrekt**, wenn für alle Zustände $\sigma$, die $P$ erfüllen:

- die Ausführung von $c$ mit $\sigma$ in $\sigma'$ regulär terminiert, so dass $\sigma'$ die Spezifikation $Q$ erfüllt,
- oder die Ausführung von $c$ in $\sigma'$ mit dem Rückgabewert $v$ terminiert, so dass $(\sigma', v)$ die Rückgabespezifikation $Q_R$ erfüllt.

$$\begin{aligned}
&\models \{P\}\, c\, \{Q|Q_R\} \iff \\
&\quad \forall \sigma.\, \sigma \models \mathcal{B}[\![P]\!]\Gamma \implies \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!]\Gamma \wedge \sigma' \models \mathcal{B}[\![Q]\!]\Gamma \\
&\qquad\qquad\qquad\qquad \vee \\
&\qquad\qquad\qquad\qquad \exists \sigma', v.\, (\sigma, (\sigma', v)) \in \mathcal{C}[\![c]\!]\Gamma \wedge (\sigma', v) \models \mathcal{B}[\![Q_R]\!]\Gamma
\end{aligned}$$

# Kontext

- Wir benötigen ferner einen **Kontext** $\Gamma$, der Funktionsbezeichnern ihre **Spezifikation** (Vor/Nachbedingung) zuordnet.
- $\Gamma(f) = \forall x_1, \ldots, x_n. (P, Q)$, für Funktion $f(x_1, \ldots, x_n)$ mit Vorbedingung $P$ und Nachbedingung $Q$.
- Notation: $\Gamma \models \{P\}\, c\, \{Q|Q_R\}$ und $\Gamma \vdash \{P\}\, c\, \{Q|Q_R\}$
- Korrektheit gilt immer nur im **Kontext**, dadurch kann jede Funktion separat verifiziert werden (**Modularität**)

# Erweiterung des Floyd-Hoare-Kalküls: return

$$\frac{}{\Gamma \vdash \{Q\}\ \textbf{return}\ \{P|Q\}} \qquad \frac{}{\Gamma \vdash \{Q[e/\backslash\textbf{result}]\}\ \textbf{return}\ e\ \{P|Q\}}$$

- Bei **return** wird die Rückgabespezifikation $Q$ zur Vorbedingung, die reguläre Nachfolgespezifikation wird ignoriert, da die Ausführung von **return** kein Nachfolgezustand hat.
- **return** ohne Argument darf nur bei einer Nachbedingung $Q$ auftreten, die kein **result** enthält.
- Bei **return** mit Argument ersetzt der Rückgabewert den **result** in der Rückgabespezifikation.

# Erweiterung des Floyd-Hoare-Kalküls: Spezifikation

$$\frac{P \Longrightarrow P'[y_i/\,\backslash\mathbf{old}(y_i)] \quad \Gamma[f \mapsto \forall x_1, \ldots, x_n.\,(P, Q)] \vdash \{P'\}\, blk\, \{Q|Q\}}{\Gamma \vdash f(x_1, \ldots, x_n)/{**}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ blk\}}$$

- Die Parameter $x_i$ werden per Konvention nur als $x_i$ referenziert, aber es ist immer der Wert im **Vorzustand** gemeint (eigentlich $\backslash\mathbf{old}(x_i)$).
- Variablen unterhalb von $\backslash\mathbf{old}(y)$ werden bei der Substitution (Zuweisungsregel) **nicht ersetzt!**
- $\backslash\mathbf{old}(y)$ wird beim Weakening von der Vorbedingung $P$ ersetzt

# Erweiterung des Floyd-Hoare-Kalküls: Aufruf

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\,(P, Q), f \text{ vom Typ } \textbf{void}}{\begin{array}{l}\Gamma \vdash \{Y_j = y_j \;\&\&\; P[t_i/x_i]\} \\ \quad f(t_1, \ldots, t_n) \\ \quad \{Q[t_i/x_i][Y_j/\,\backslash\textbf{old}(y_j)] | Q_R\}\end{array}}$$

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\,(P, Q)}{\begin{array}{l}\Gamma \vdash \{Y_j = y_j \;\&\&\; P[t_i/x_i]\} \\ \quad x = f(t_1, \ldots, t_n) \\ \quad \{Q[t_i/x_i][Y_j/\,\backslash\textbf{old}(y_j)][x/\,\backslash\textbf{result}] | Q_R\}\end{array}}$$

- $\Gamma$ muss $f$ mit der Vor-/Nachbedingung $P, Q$ enthalten
- In $P$ und $Q$ werden Parameter $x_i$ durch Argumente $t_i$ ersetzt.
- $y_1, \ldots, y_m$ sind die als $\backslash\textbf{old}(y_j)$ in $Q$ auftretenden Variablen
- $Y_1, \ldots, Y_m$ dürfen nicht in $P$ oder $Q$ enthalten sein
- Im ersten Fall (Aufruf als Prozedur) enthält $Q$ kein $\backslash\textbf{result}$

# Erweiterter Floyd-Hoare-Kalkül I

$$\frac{}{\Gamma \vdash \{P\}\,\{\}\,\{P|Q_R\}} \qquad \frac{\Gamma \vdash \{P\}\,c_1\,\{R|Q_R\} \quad \Gamma \vdash \{R\}\,c_2\,\{Q|Q_R\}}{\Gamma \vdash \{P\}\,c_1; c_2\,\{Q|Q_R\}}$$

$$\frac{}{\Gamma \vdash \{Q[e/x]\}\,x = e\,\{Q|Q_R\}} \qquad \frac{\Gamma \vdash \{P \wedge b\}\,c\,\{P|Q_R\}}{\Gamma \vdash \{P\}\ \textbf{while}\ (b)\ c\,\{P \wedge \neg b|Q_R\}}$$

$$\frac{\Gamma \vdash \{P \wedge b\}\,c_1\,\{Q|Q_R\} \quad \Gamma \vdash \{P \wedge \neg b\}\,c_2\,\{Q|Q_R\}}{\Gamma \vdash \{P\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\,\{Q|Q_R\}}$$

$$\frac{P \longrightarrow P' \quad \Gamma \vdash \{P'\}\,c\,\{Q'|R'\} \quad Q' \longrightarrow Q \quad R' \longrightarrow R}{\Gamma \vdash \{P\}\,c\,\{Q|R\}}$$

# Erweiterter Floyd-Hoare-Kalkül II

$$\frac{}{\Gamma \vdash \{Q\}\ \textbf{return}\ \{P|Q\}} \qquad \frac{}{\Gamma \vdash \{Q[e/\backslash\textbf{result}]\}\ \textbf{return}\ e\ \{P|Q\}}$$

$$\frac{\begin{array}{l}\Gamma[f \mapsto (P,Q)] \vdash \{X_i = x_i \wedge Y_j = y_j \wedge P\} \\ \qquad\qquad blk \\ \qquad\qquad \{Q[X_i/x_i][y_j/\backslash\textbf{old}(y_j)]|Q[X_i/x_i][y_j/\backslash\textbf{old}(y_j)]\}\end{array}}{\Gamma \vdash f(x_1,\ldots,x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}{*}/\ \{ds\ blk\}}$$

$$\frac{\Gamma(f) = \forall x_1,\ldots,x_n.(P,Q), f\ \text{vom Typ}\ \textbf{void}}{\begin{array}{l}\Gamma \vdash \{Y_j = y_j \wedge P[t_i/x_i]\} \\ \qquad f(t_1,\ldots,t_n) \\ \qquad \{Q[t_i/x_i][Y_j/\backslash\textbf{old}(y_j)]|Q_R\}\end{array}}$$

$$\frac{\Gamma(f) = \forall x_1,\ldots,x_n.(P,Q)}{\begin{array}{l}\Gamma \vdash \{Y_j = y_j \wedge P[t_i/x_i]\} \\ \qquad x = f(t_1,\ldots,t_n) \\ \qquad \{Q[t_i/x_i][Y_j/\backslash\textbf{old}(y_j)][x/\backslash\textbf{result}]|Q_R\}\end{array}}$$

# Beispiel: die Fakultätsfunktion, rekursiv

```
int fac(int x)
/** pre 0 ≤ x;
    post \result = x! */
{
 int r = 0;

 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r* x;
}
```

# Beobachtungen

- Der Aufruf einer Funktion **ersetzt** die momentane Nachbedingung — das ist ein Problem
- Wir brauchen keine Invariante mehr — ist durch die Nachbedingung gegeben
- Rekursion benötigt keine Extrabehandlung
  - Termination von rekursiven Funktionen wird extra gezeigt

# Zusammenfassung

- Funktionen sind **zentrales Modularisierungskonzept**
- Wir müssen Funktionen **modular** verifizieren können
- Erweiterung der **Semantik:**
  - Semantik von Deklarationen und Parameter — straightforward
  - Semantik von **Rückgabewerten** — Erweiterung der Semantik
  - **Funktionsaufrufe** — Environment, um Funktionsbezeichnern eine Semantik zu geben
- Erweiterung der **Spezifikationen:**
  - Spezifikation von Funktionen: **Vor-/Nachzustand** statt logischer Variablen
- Erweiterung des Hoare-Kalküls:
  - Environment, um andere Funktionen zu nutzen
  - Gesonderte Nachbedingung für Rückgabewert/Endzustand

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 12 vom 26.06.18: Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Motivation

- Warum Referenzen?
  - Nötig für *call by reference*
  - Funktion können sonst nur **globale** Seiteneffekte haben
  - Effizienz
- Kurze Begriffsklärung:
  - Referenzen: getypt, eingeschränkte Arithmetik
  - Zeiger: ungetypt, Zeigerarithmetik

# Refenzen in C

- Pointer in C ("pointer type"):
  - Schwach getypt (**void** $*$ kompatibel mit allen Zeigertypen, Typumwandlung)
  - Eingeschränkte Zeigerarithmetik (Addition, Subtraktion)
  - Felder werden durch Zeigerarithmetik implementiert
- Pointer sind *first-class-values*
- C-Standard läßt das Speichermodell relativ offen
  - Repräsentation von Objekten

# Referenzen in anderen Sprachen

- Java:
  - Alles ist eine Referenz
  - Schwach getypt (Subtyping und Typumwandlung)
- Haskell, SML, OCaml:
  - Stark getypt (typsicher)
- Scriptsprachen (Python, Ruby):
  - Ähnlich Java

# Ausdrücke

- Neue Operatoren: Addressoperator ($\&a$) und Derefenzierung ($*l$)

$$
\begin{array}{rl}
\textbf{Lexp} & l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \mid *a \\
\textbf{Aexp} & a ::= \textbf{Z} \mid \textbf{C} \mid \textbf{Lexp} \mid \&l \\
 & \quad \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid \textbf{Idt}(\textbf{Exp}^*) \\
\textbf{Bexp} & b ::= \ldots \\
\textbf{Exp} & e := \textbf{Aexp} \mid \textbf{Bexp} \\
\textbf{Stmt} & c ::= \ldots \\
\textbf{Type} & t ::= \textbf{char} \mid \textbf{int} \mid *t \mid \textbf{struct}\ \ \textbf{Idt}^?\ \{\textbf{Decl}^+\} \mid t\ \ \textbf{Idt}[a]
\end{array}
$$

# Das Problem mit Zeigern

- Bisheriges Speichermodell: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$
- **Aliasing:**
  Verschiedene Bezeichner (**Lexp**) für die gleiche Lokation $l \in \mathbf{Loc}$

```
int a;
int *p;

p= &a;
a= 0;
// {a = 0}
*p= 7;
// {a = 7}
```

- Wert von a ändert sich **ohne dass a erwähnt** wird.
- Großes Problem für Semantik und Hoare-Kalkül.

# Erweiterung des Zustandmodells

- Bisheriger Zustand $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$ mit
  - **Locations:** $\mathbf{Loc} ::= \mathbf{Idt} \mid \mathbf{Loc}[\mathbb{Z}] \mid \mathbf{Loc}.\mathbf{Idt}$
  - Werte: $\mathbf{V} = \mathbb{Z}$
- Ansatz reicht nicht mehr:

  (i) Werte müssen auch Locations sein: $\mathbf{V} \stackrel{def}{=} \mathbb{Z} + \mathbf{Loc}$

  (ii) **Idt**als Location nicht ausreichend für Referenzen und Funktionen
- Man kann den Zustand **modellbasiert** (wie bisher) oder **axiomatisch** beschreiben.

# Axiomatisches Zustandsmodell

- Der Zustand ist ein abstrakter Datentyp $\Sigma$ (und **Loc**) mit zwei Operationen und folgenden Gleichungen:

$$\mathit{read} : \Sigma \to \mathbf{Loc} \rightharpoonup \mathbf{V}$$
$$\mathit{upd} : \Sigma \to \mathbf{Loc} \to \mathbf{V} \rightharpoonup \Sigma$$
$$\mathbf{V} \stackrel{\mathit{def}}{=} \mathbb{Z} + \mathbf{Loc}$$

$$\mathit{read}(\mathit{upd}(\sigma, l, v), l) = v$$
$$l \neq m \Longrightarrow \mathit{read}(\mathit{upd}(\sigma, l, v), m) = \mathit{read}(\sigma, m)$$
$$\mathit{upd}(\mathit{upd}(\sigma, l, v), l, w) = \mathit{upd}(\sigma, l, w)$$
$$l \neq m \Longrightarrow \mathit{upd}(\mathit{upd}(\sigma, l, v), m, w) = \mathit{upd}(\mathit{upd}(\sigma, m, w), l, v)$$

- Diese Gleichungen sind **vollständig.**

# Axiomatisches Speichermodell

- Es gibt einen **leeren** Speicher, und neue ("frische") Adressen:

$$\begin{aligned} \mathit{empty} &: \Sigma \\ \mathit{fresh} &: \Sigma \to \mathbf{Loc} \\ \mathit{rem} &: \Sigma \to \mathbf{Loc} \to \Sigma \end{aligned}$$

  - *fresh* modelliert **Allokation**, *rem* modelliert **Deallokation**
- *dom* beschreibt den **Definitionsbereich**:

$$\begin{aligned} \mathit{dom}(\sigma) &= \{l \mid \exists v.\, \mathit{read}(\sigma, l) = v\} \\ \mathit{dom}(\mathit{empty}) &= \emptyset \end{aligned}$$

- Eigenschaften von *empty*, *fresh* und *rem*:

$$\begin{aligned} \mathit{fresh}(\sigma) &\notin \mathit{dom}(\sigma) \\ \mathit{dom}(\mathit{rem}(\sigma, l)) &= \mathit{dom}(\sigma) \setminus \{l\} \\ l \neq m \implies \mathit{read}(\mathit{rem}(\sigma, l), m) &= \mathit{read}(\sigma, m) \end{aligned}$$

# Zeigerarithmetik

- Ziegerarithmetik: Rechnen mit Zeigern
  - Implementiert Felder und Strukturen
  - Wir betrachten keine **Differenz** von Zeigern

$$add : \mathbf{Loc} \to \mathbf{Z} \to \mathbf{Loc}$$

$$add(l, 0) = l$$
$$add(add(l, a), b) = add(l, a + b)$$
$$add(l, a) = l \Longrightarrow a = 0$$
$$add(l, a) = add(l, b) \Longrightarrow a = b$$

# Erweiterung der Semantik

- Problem: **Loc** haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
  - x = x+1 — Links: Addresse der Variablen, rechts: Wert an dieser Adresse
- Lösung in C: “Except when it is (. . . ) the operand of the unary & oprerator, the left operand of the . operator or an assigment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)”
  *C99 Standard*, §6.3.2.1 (2)
- Nicht spezifisch für C

# Umgebung

- Für Funktionen brauchten wir eine **Umgebung** (Environment):

$$\begin{aligned}\mathbf{Env} &= \mathit{Id} \rightharpoonup [\![\mathbf{FunDef}]\!] \\ &= \mathit{Id} \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)\end{aligned}$$

- Diese muss erweitert werden für Variablen:

$$\mathbf{Env} = \mathit{Id} \rightharpoonup ([\![\mathbf{FunDef}]\!] \uplus \mathbf{Loc})$$

  - Insbesondere: gleicher Namensraum für Funktionen und Variablen (*C99 Standard*, §6.2.3)

# Erweiterung der Semantik: Lexp

$$\mathcal{L}[\![-]\!] : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$\begin{aligned}
\mathcal{L}[\![x]\!]\ \Gamma &= \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\} \\
\mathcal{L}[\![lexp[a]]\!]\ \Gamma &= \{(\sigma, add(l, i \cdot sizeof(\tau))) \mid (\sigma, l) \in \mathcal{L}[\![lexp]\!]\ \Gamma, (\sigma, i) \in \mathcal{A}[\![a]\!]\ \Gamma\} \\
&\qquad type(\Gamma, lexp) = \tau \text{ ist der Basistyp des Feldes} \\
\mathcal{L}[\![lexp.f]\!]\ \Gamma &= \{(\sigma, l.f) \mid (\sigma, add(l, fld_off(\tau, f))) \in \mathcal{L}[\![lexp]\!]\ \Gamma\} \\
&\qquad type(\Gamma, lexp) = \tau \text{ ist der Typ der Struktur} \\
\mathcal{L}[\![*e]\!]\ \Gamma &= \mathcal{A}[\![e]\!]\ \Gamma
\end{aligned}$$

- $type(\Gamma, e)$ ist der **Typ** eines Ausdrucks
- $fld\_off(\tau, f)$ ist der **Offset** des Feldes $f$ in der Struktur $\tau$
- $sizeof(\tau)$ ist die **Größe** von Objekten des Typs $\tau$

# Erweiterung der Semantik: Aexp(1)

$$\mathcal{A}[\![-]\!] : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
\mathcal{A}[\![n]\!]\ \Gamma &= \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbb{N} \\
\mathcal{A}[\![e]\!]\ \Gamma &= \{(\sigma, \mathit{read}(\sigma, l)) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\qquad e \in \mathbf{Lexp} \text{ und } \mathit{type}(\Gamma, e) \text{ kein Array-Typ} \\
\mathcal{A}[\![e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\qquad e \in \mathbf{Lexp} \text{ und } \mathit{type}(\Gamma, e) \text{ Array-Typ} \\
\mathcal{A}[\![\&e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\}
\end{aligned}$$

# Erweiterung der Semantik: Aexp(2)

$$\mathcal{A}[\![-]\!] : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
\mathcal{A}[\![a_0 + a_1]\!]\ \Gamma &= \{(\sigma, n_0 + n_1 \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 - a_1]\!]\ \Gamma &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 * a_1]\!]\ \Gamma &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 / a_1]\!]\ \Gamma &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma \\
&\qquad \wedge\ n_1 \neq 0\}
\end{aligned}$$

# Und jetzt?

- Zustand erweitert, so dass wir Zeiger modellieren können.
- Semantik entsprechend erweitert.
- Was machen wir mit dem Hoare-Kalkül, speziell der **Zuweisung**?

# Und jetzt?

- Zustand erweitert, so dass wir Zeiger modellieren können.
- Semantik entsprechend erweitert.
- Was machen wir mit dem Hoare-Kalkül, speziell der **Zuweisung**?
- Vorherige Modellierung — Zuweisung durch Substitution modelliert — nicht mehr ausreichend.
- Daher: **explizite Zustandsprädikate**

# Explizite Zustandsprädikate

- Zusicherungenen (**Assn**) sind zustandsabhängige Prädikate
  - Mit anderen Worten, Prädikate über Programmvariablen.
- Axiomatische Beschreibung des Zustandes erforderte neue Modellierung auf der Ebene der Prädikate
- Explizite Zustandsprädikate modellieren die Zustandsoperationen *read* und *upd* **explizit**

# Explizite Zustandsprädikate

- Erweiterung von **Aexpv** um *read*, neue Sorte **State** mit Operation *upd*:

$$\begin{aligned}
\mathbf{Lexp}_s &\quad l ::= \ldots \mid \cancel{*a} \\
\mathbf{Assn}_s &\quad b ::= \ldots \\
\mathbf{Aexp}_s &\quad a ::= \mathit{read}(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid l \mid \cancel{\&l} \mid \ldots \mid \backslash\mathbf{old}(e) \mid \ldots \\
\mathbf{State} &\quad S ::= \mathit{StateVar} \mid \mathit{upd}(S, l, e)
\end{aligned}$$

- Zustandsvariablen *StateVar*: Aktueller Zustand $\sigma$, Vorzustand $\rho$
- Explizite Zustandsprädikate enthalten kein $*$ oder &
- Damit Semantik:

$$\begin{aligned}
\mathcal{B}_{sp}[\![.]\!] &: \mathbf{Env} \to \mathbf{Assn}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B} \\
\mathcal{A}_{sp}[\![.]\!] &: \mathbf{Env} \to \mathbf{Aexp}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}
\end{aligned}$$

# Hoare-Triple

$$\Gamma \models \{P\}\, c\, \{Q|R\}$$

- $P, Q, R \in \mathbf{Assn}_s$ sind **explizite Zustandsprädikate**
- Deklarationen (**Decl**) allozieren für jede Variable eine Location (*fresh*), und ordnen diese in $\Gamma$ dem Namen zu.
- Gültigkeit von Hoare-Tripeln (partielle, totale Korrektheit) wie vorher

# Floyd-Hoare-Kalkül mit expliziten Zustandsprädikaten

$$\frac{}{\Gamma \vdash \{Q[upd(\sigma, x, e)/\sigma]\}\, x = e\, \{Q|R\}}$$

- Ein **Lexp** $l$ auf der rechten Seite $e$ wird durch $read(\sigma, l)$ ersetzt.[1]
- & dient lediglich dazu, diese Konversion zu verhindern.
- $*$ erzwingt diese Konversion, auch auf der linken Seite $x$.
- Beispiel: $*$a $=$ $*$&b;

[1] Außer $l$ ist ein Array-Typ.

# Formal: Konversion in Zustandsprädikate

$$(-)^\dagger : \mathbf{Lexp} \to \mathbf{Lexp}_s$$

$$i^\dagger = i \quad (i \in \mathbf{Idt})$$

$$I.id^\dagger = I^\dagger.id$$

$$I[e]^\dagger = I^\dagger[e^\#]$$

$$*I^\dagger = I^\#$$

$$(-)^\# : \mathbf{Aexp} \to \mathbf{Aexp}_s$$

$$e^\# = read(\sigma, e^\dagger) \quad (e \in \mathbf{Lexp})$$

$$n^\# = n$$

$$v^\# = v \quad (v \text{ logische Variable})$$

$$\&e^\# = e^\dagger$$

$${e_1 + e_2}^\# = {e_1}^\# + {e_2}^\#$$

$$\backslash\mathbf{result}^\# = \backslash\mathbf{result}$$

$$\backslash\mathbf{old}(e)^\# = \backslash\mathbf{old}(e)$$

$$\frac{}{\Gamma \vdash \{Q[upd(\sigma, x^\dagger, e^\#)/\sigma]\}\, x = e\, \{Q|R\}}$$

# Zwei kurze Beispiele

```
void foo(){
 int x, y, z;
 // {true}
 z= x;
 x= 0;
 z= 5;
 y= x;
 // {y = 0}
}
```

```
void foo(){
 int x, y, *z;
 // {true}
 z= &x;
 x= 0;
 *z= 5;
 y= x;
 // {y = 5}
```

# Ein problematisches Beispiel

```
void foo(int *p)
{
  int x;

  // {true}
  x= 7;
  *p= 99;
  // {x = 7}
}
```

# Ein problematisches Beispiel

```
void foo(int *p)
{
  int x;

  // {true}
  x= 7;
  *p= 99;
  // {x = 7}
}
```

- Können **weder** beweisen, dass $*p = x$ **noch** $*p \neq x$
- Erfordert Spezifikation: wenn $*p$ auf ein **gültiges** Objekt zeigt, dann $*p \neq x$ da $x$ **lokale** Variable.
- Generelles Problem — was ist mit

  ```
  void foo(int *p, int *q)
  { ... }
  ```

- Können weder beweisen, dass $*p = *q$ noch $*p \neq *q$

# Weitere Beispiele: Felder

```
#include <limits.h>
#define N 10
int a[N];

int findmax(int a[], int a_len)
  /** pre  \array(a, a_len); */
  /** post ∀i. 0 ≤ i < a_len ⟶ a[i] ≤ \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /** inv (∀i. 0 ≤ i < j ⟶ a[i] ≤ x) ∧ j ≤ a_len; */
    {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```

# Felder und Zeiger revisited

- In C sind Zeiger und Felder schwach spezifiziert
- Insbesondere:
  - a[j]= $*$(a+j) für a Array-Typ
  - Derefenzierung von $*$x nur definiert, wenn x "gültig" ist (d.h. auf ein Objekt zeigt) *C99 Standard*, §6.5.3.2(4)
- Bisher in den Hoare-Regeln ignoriert — **partielle** Korrektheit.
- Ist das sinnvoll? Nein, bekannte Fehlerquelle

# Spezifikation von Zeigern und Feldern

## Das Prädikat \valid(x)

$\backslash\mathbf{valid}(x) \iff \mathit{read}(\sigma, x^\dagger)$ ist definiert

- Insbesondere: $\backslash\mathbf{valid}(*x) \iff \mathit{read}(\sigma, \mathit{read}(\sigma, x))$ ist definiert.
- Felder als Parameter werden Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger "in Wirklichkeit" ein Feld ist.
- $\backslash\mathbf{array}(a, n)$ bedeutet: a ist ein Feld der Länge n, d.h.

$$\backslash\mathbf{array}(a, n) \iff (\forall i.\, 0 \leq i < n \implies \backslash\mathbf{valid}(a[i]))$$

- Validität kann abgeleitet werden:

$$\frac{\text{x= \&e}}{\backslash\text{valid(*x)}} \qquad \frac{\backslash\text{array(a, n)} \quad 0 \leq i \quad i < n}{\backslash\text{valid(a[i])}}$$

# Zusammenfassung

- Um Referenzen (Pointer) in C behandeln zu können, benötigen wir ein erweitertes **Zustandsmodell**
- Referenzen werden zu Werten wie Zahlen oder Zeichen.
  - Arrays und Strukturen sind **keine** first-class values.
  - Großes Problem: **aliasing**
- Erweiterung der Semantik und der Hoare-Tripel nötig:
  - Vor/Nachbedingungen werden zu **expliziten Zustandsprädikaten**.
  - Zuweisung wird zu **Zustandsupdate**.
  - Problem:
    - Zustände werden **sehr groß**
    - Rückwärtsrechnung erzeugt schnell sehr große „unbestimmte“ Zustände, die nicht vereinfacht werden können
    - Hier ist Vorwärtsrechnung vorteilhaft

# Korrekte Software: Grundlagen und Methoden
# Vorlesung 13 vom 03.07.18: Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

# Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und Rückblick

# Was gibt's heute?

- Rückblick
- Ausblick
- Feedback

# Rückblick

# Semantik

- Operational — Auswertungsrelation $\langle c, \sigma \rangle \to \sigma'$
- Denotational — Partielle Funktion $[\![c]\!] : \Sigma \rightharpoonup \Sigma$
- Axiomatisch — Floyd-Hoare-Logik
- Welche Semantik wofür?
- Beweis: Äquivalenz von operationaler und denotationaler Semantik

# Floyd-Hoare-Logik

- Floyd-Hoare-Logik: partiell und total
- $\vdash \{P\}\, c\, \{Q\}$ vs. $\models \{P\}\, c\, \{Q\}$: Vollständigkeit, Korrektheit
- Die sechs Basisregeln
- Zuweisungsregel: vorwärts (Floyd) vs. rückwärts (Hoare)
- VCG: Schwächste Vorbedingung und stärkste Nachbedingung
- Beweis: Korrektheit und Vollständigkeit der Floyd-Hoare-Logik

# Erweiterung der Programmiersprache

- Für jede Erweiterung:
  - Wie modellieren wir semantisch?
  - Wie ändern sich die Regeln der Logik?
- Strukturen und Felder
  - Lokationen: strukturierte Werte **Lexp**
  - Erweiterte Substitution in Zuweisungsregel

# Erweiterung der Programmiersprache

- Prozeduren und Funktionen
  - Modellierung von **return**: Erweiterung zu $\Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$
  - Spezifikation von Funktionen durch Vor-/Nachbedingungen
  - Spezifikation der Funktionen muss im Kontext stehen
- Referenzen
  - Konversion zwischen **Lexp**und **Aexp**
  - Lokationen nicht mehr symbolisch (Variablennamen), sondern abstrakt $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}, \mathbf{V} = \mathbb{Z} + \mathbf{Loc}$
  - Zustand als **abstrakter Datentyp** mit Operationen *read* und *upd*
  - Zuweisung nicht mehr mit Substitution, sondern explizit durch *upd*
  - Spezifikationen sind **explizite Zustandsprädikate**, Konversion $(-)^\dagger, (-)^\#$

# Ausblick

# Was geht noch?

- Die Sprache C
- Andere Programmiersprachen
- Logik und Spezifikation
- Success Stories

# Die Sprache C: Was haben wir ausgelassen?

**Semantik**:

- Nichtdeterministische Semantik: Seiteneffekte, Sequence Points
  $\longrightarrow$ Umständlich zu modellieren, Effekt zweitrangig

- Implementationsabhängiges, unspezifiziertes und undefiniertes Verhalten
  $\longrightarrow$ Genauere Unterscheidung in der Semantik

**Kontrollstrukturen**:

- **switch** $\longrightarrow$ Ist im allgemeinen Fall ein **goto**

- **goto**, setjmp/longjmp
  $\longrightarrow$ Allgemeinfall: tiefe Änderung der Semantik (*continuations*)

# Die Sprache C: Was haben wir ausgelassen?

**Typen**:

- Funktionszeiger $\longrightarrow$ Für "saubere" Benutzung gut zu modellieren
- Weitere Typen: **short**/**long int**, **double**/**float**, wchar_t, und Typkonversionen $\longrightarrow$ Fleißarbeit
- Fließkommazahlen $\longrightarrow$ Spezifikation nicht einfach
- **union** $\longrightarrow$ Kompliziert das Speichermodell
- **volatile** $\longrightarrow$ Bricht read/update-Gleichungen
- **typedef** $\longrightarrow$ Ärgernis für Lexer/Parser, sonst harmlos

# Die Sprache C: Was haben wir ausgelassen?

Für **realistische C-Programme**:

- Compiler-Erweiterungen (`gcc`, `clang`)
- Büchereien (Standardbücherei, Posix, . . . )
- Nebenläufigkeit

# Andere Sprachen: Wie modelliert man Java?

- Die **Kernsprache** ist ähnlich zu C0.
- Java hat erschwerend:
  - dynamische Bindung,
  - Klassen mit gekapselten Zustand und Invarianten,
  - Nebenläufigkeit, und
  - Reflektion.
- Java hat dafür aber
  - ein einfacheres Speichermodell, und
  - eine wohldefinierte Ausführungsumgebung (die JVM).

# Andere Sprachen: Wie modelliert man C++?

- Sehr **vorsichtig** (konservativ)
- Viele Features, fehlende formale Semantik, ...
- Mehrfachvererbung theoretisch anspruchsvoll
- Es gibt **keine** Formalismen/Werkzeuge, die C++ voll unterstützen
- Ansätze: Übersetzung nach C/LLVM, Behandlung dort

# Andere Sprachen: Wie modelliert man PHP?

Gar nicht.

# Logik und Spezifikation

- Wir **generieren** Verifikationsbedingungen, wie kann man sie **beweisen**?
- **Automatische Beweiser:**
  - **SAT-Checker** lösen Erfüllbarkeitsproblem der Aussagenlogik (MiniSAT, Chaff)
  - **SMT-Beweiser** beweisen Aussagen der Prädikatenlogik mit linearer Arithmetik, Funktionen und Induktion (Z3, Yices, CVC)
- **Interaktive Beweiser:**
  - Beweisführung durch Benutzer, **Überprüfung** durch Beweiser
  - Sehr **mächtige** Logiken, aber nicht vollautomatisch (Isabelle, Coq)

# Beispiel: Z3

- SMT-Beweiser versuchen Gegenbeweis zu konstruieren
- Daher: um $\phi$ zu beweisen, versuchen wir $\not\Phi$ zu widerlegen

Beweis einer VC:

$$x \geq 0 \wedge y > 0 \Longrightarrow x = 0 * y + x$$

Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
   (not (=> (and (>= x 0) (> y 0))
            (= x (+ (* 0 y) x))))
)
(check-sat)
```

Antwort:

`unsat`

Unerfüllbare VC:

$$x \geq 0 \wedge y > 0 \Longrightarrow x \geq y$$

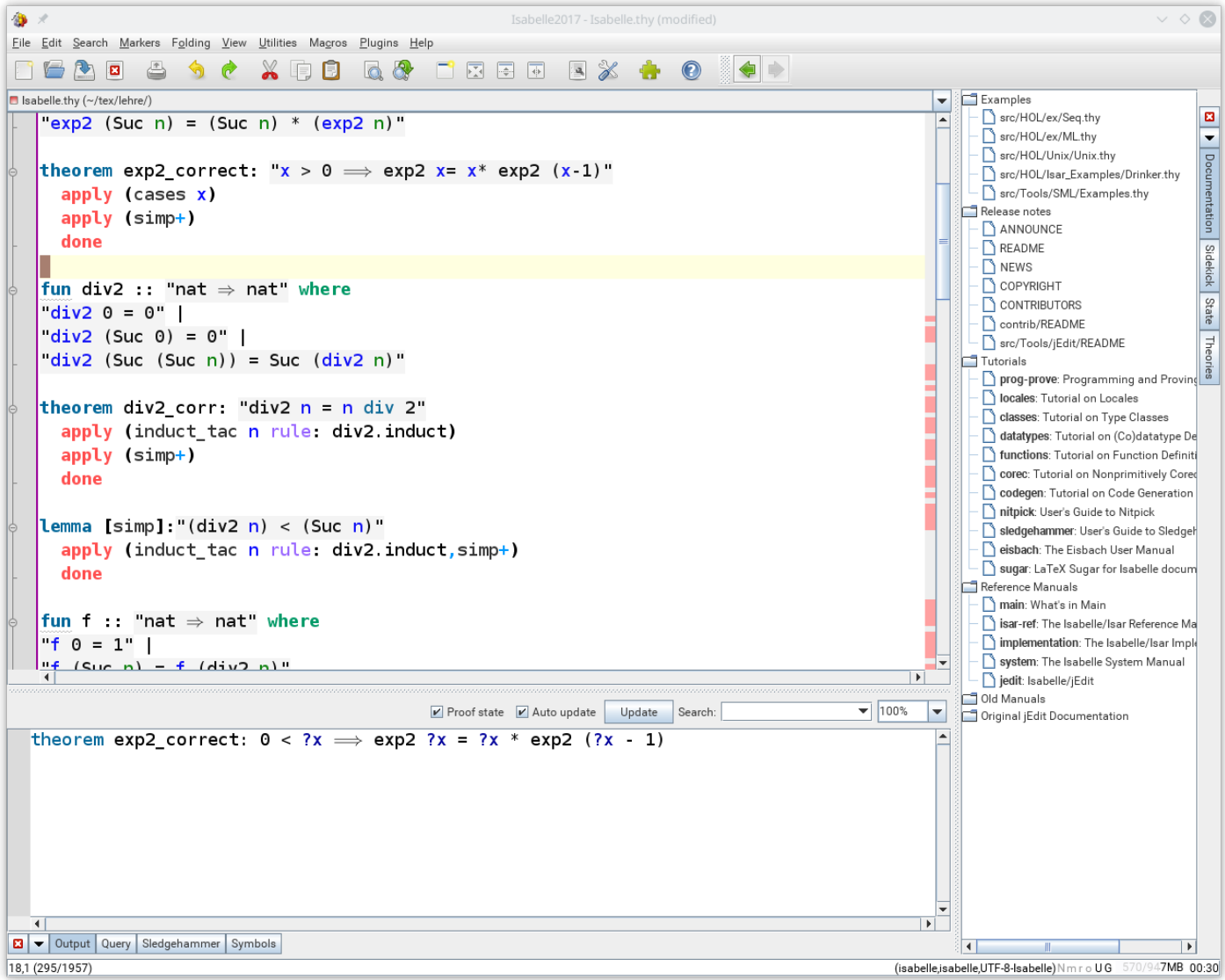
Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
   (not (=> (and (>= x 0) (> y 0))
            (>= x y)))
)
(check-sat)
```

Antwort:

`sat`

# Beispiel: Isabelle



# Korrekte Software in der Industrie

- Meist in speziellen Anwendungsgebieten: Luft-/Raumfahrt, Automotive, sicherheitskritische Systeme, Betriebssysteme
- Ansätze:
  1. Vollautomatisch: **statische Analyse** (Abstrakte Interpretation) für spezielle Aspekte: Freiheit von Ausnahmen und Unter/Überläufen, Programmsicherheit, Laufzeitverhalten (WCET) (nicht immer korrekt, meist vollständig)
     - Werkzeuge: `absint`
  2. Halbautomatisch: **Korrektheitsannotationen**, Überprüfung automatisch
     - Werkzeuge: Spark (ADA), Frama-C (C), JML (ESC/Java, Krakatao; Java), Boogie und Why (generisches VCG), VCC (C)
  3. Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
     - Beispiele: L4.verified, CompCert, SAMS

# Feedback

# Deine Meinung zählt

- Was war gut, was nicht?
- Arbeitsaufwand?
- Mehr **Theorie** oder mehr **Praxis**?
- Programmieraufgaben?
- Leichtgewichtiger Übungsbetrieb — mehr oder weniger?
- Bitte auch die **Evaluation** auf stud.ip beantworten!

Tschüß!