Korrekte Software: Grundlagen und Methoden
Vorlesung 1 vom 13.04.21
Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Organisatorisches

- Veranstalter:

Serge Autexier
serge.autexier@dfki.de
Cartesium 1.49[1], Tel. 59834

Christoph Lüth
christoph.lueth@dfki.de
MZH 4186[1], Tel. 59830

- Termine:
  - Dienstag, 10 – 12
  - Donnerstag, 8 – 10**8:30 – 10:00**
- Webseite:

  `http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss21`

[1] Immer noch im Home-Office

Online-Konzept in Corona-Zeiten

- Keine **lange Vorlesung**, lieber **integrierte Veranstaltung**
- Kürzere **Vortragseinheiten**, dazwischen *Arbeitsfragen* (Kurzübungen)
  - Kein asynchrones Angebot (Aufzeichung der Meetings?)
- Wöchentliche **Übungsaufgaben** zur Vertiefung
- Technisch:
  - Nutzung von **Zoom**
  - Fragen/Kurzübungen in **HedgeDoc**: `http://hackmd.informatik.uni-bremen.de/`
  - Übungsblätter als **ausfüllbare PDFs**, Abgabe über `gitlab`.

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)

Übungsbetrieb

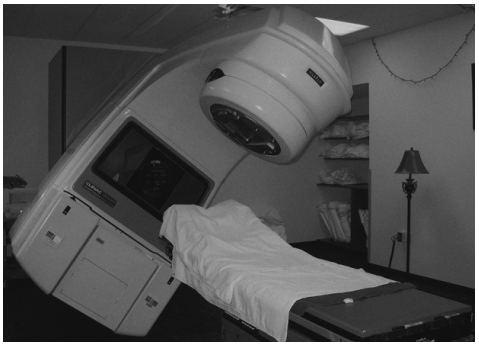
- Abgabe und Korrektur des Übungsbetriebs erfolgt über **gitlab**.
- Dazu legt ein **pro Gruppe** ein Repository an, und ladet uns (`clueth`, `autexier`) als Developer ein.
- Für jedes Übungsblatt:
  1. Das Übungsblatt ladet ihr von der Webseite herunter, und bearbeitet es **elektronisch**.
  2. Die Lösung liegt ihr als PDF ab (bitte Namen nicht verändern, `uebung-01.pdf`; Zusatzmaterial als `uebung-XX-...` wenn nötig), und ladet es **vor** dem Abgabezeitpunkt hoch (push).
  3. Nach der Abgabe laden wir die Änderungen herunter (pull), korrigieren direkt im PDF, fügen die Bewertung hinzu, und laden die Korrektur wieder hoch.

Arbeitsblatt 1.1: Jetzt seid ihr dran!

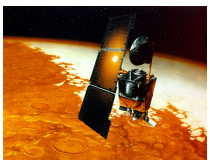
- Gruppiert euch in Gruppen zu drei Teilnehmenden! Nutzt dazu folgenden Doodle:
  `https://www.doodle.com/poll/3ha3dx4hzavhrucv`
- Zu jeder Gruppe gibt es ein Arbeitsblatt:
  `https://hackmd.informatik.uni-bremen.de/rfF0alFiS8y6nUtspD4YgA#`
- Auf diesem Arbeitsblatt bearbeitet ihr die Arbeitsfragen im Laufe des Kurses.
- Bitte nur in "eurem" Arbeitsblatt arbeiten
- Die Arbeitsblätter sind nicht notenrelevant.

I. 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: Ungeplantes Übergewicht

- „A software mistake caused a Tui flight to take off heavier than expected as female passengers using the title “Miss” were classified as children [...]“
- 38 erwachsene Passagiere als Kinder (35kg) statt als Erwachsene (69kg) klassifiziert.

$$38 \cdot (69\,kg - 35\,kg) = 1292\,kg$$

- Software „was programmed in an unnamed foreign country where the title “Miss” is used for a child and “Ms” for an adult female.“

Quelle: *Guardian, 09.04.2021.*

`https://www.theguardian.com/world/2021/apr/09/tui-plane-serious-incident-every-miss-on-board-child-weight-birmingham-majorca`

Arbeitsblatt 1.2: Jetzt seid ihr dran!

- Sucht im Netz nach weiteren Software-Disastern:
  1. Was ist passiert?
  2. Wie ist es passiert?
  3. Was war der Softwarefehler?
- Quellen: Suchmaschine nach Wahl (“software disasters”), The Risks Digest, `https://catless.ncl.ac.uk/Risks/`

II. 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
- Der Floyd-Hoare-Kalkül I
- Der Floyd-Hoare-Kalkül II: Invarianten
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Modellierung
- Spezifikation von Funktionen
- Referenzen und Speichermodelle
- Ausblick und Rückblick

III. Warum Semantik?

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 anhand der **Eigenschaften**

Arbeitsblatt 1.3: Maschinen und Funktionen

Was genau kann man sich unter "abstrakten Maschine" vorstellen?

Betrachtet als Beispiele:

- Eine Waschmaschine
- Einen Taschenrechner
- Ein Java-Programm, welches ein Array von Zahlen summiert

Was ist hier die Abstraktion?

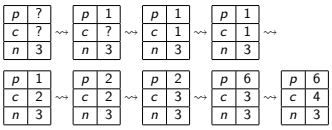
Unsere Sprache C0

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

Operationale Semantik

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

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



Arbeitsblatt 1.4: Operationale Semantik

Gegeben folgendes C0-Programm:

```
1 x= 0;
2 while (n > 0) {
3   x= x+ n*n;
4   n= n-1;
5   }
```

Entwickeln Sie die ersten zehn Schritte der operationalen Semantik wie im Beispiel oben für den initialen Zustand

n	4
x	?

$\rightsquigarrow \ldots$

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
```

$$[\![p_1]\!](\sigma) = \sigma[p \mapsto 1][c \mapsto 1]$$
$$[\![p_2]\!](\sigma) = \sigma[p \mapsto \sigma(p) * \sigma(c)][c \mapsto \sigma(c) + 1]$$
$$[\![p_3]\!](\sigma)[\![p_3]\!] = ??? \; fix(\Gamma([\![c \leq n]\!])[\![p_2]\!]))([\![p_1]\!](\sigma)) \; fix(\Gamma([\![c \leq n]\!])[\![p_2]\!])) \circ [\![$$

$$\Gamma([\![c \leq n]\!])([\![p_2]\!])(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } [\![c \leq n]\!](\sigma) = 0 \\ (\varphi \circ [\![p_2]\!])(\sigma) & \text{if } [\![c \leq n]\!](\sigma) = 1 \end{cases}$$

$$\Gamma(\beta)(\rho)(\varphi)(\sigma) = \begin{cases} \sigma & \text{if } \beta(\sigma) = 0 \\ (\varphi \circ \rho)(\sigma) & \text{if } \beta(\sigma) = 1 \end{cases}$$

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)
  }
// (6)
```

(1)
(2)
(3)
(4) $(p = 1 \wedge c = 1 \vee p = 1 \wedge c = 2 \vee p = 2 \wedge c = 3)\ p = (c-1)! \wedge c \leq n \wedge n = 3$
(5)
(6)

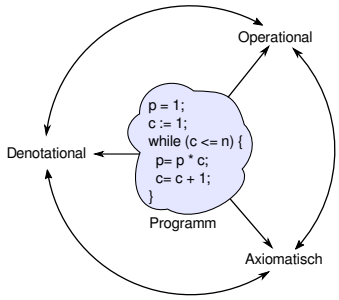
Arbeitsblatt 1.5: Zusicherungen

Betrachten Sie folgende Variation des Programms von oben:

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

- Welche der Zusicherungen (1) – (5) von oben gelten noch?
- Welche nicht? $(p = 1 \wedge c = 2 \vee p = 2 \wedge c = 3 \vee p = 6 \wedge c = 4)\ p = (c-1)! \wedge n = 3 \wedge n = 3$
- Was gilt stattdessen?
  $p = 6 \wedge c = 4 \wedge n = 3$

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 20.04.21

Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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

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

```
a = (3+y)*x+5*b;
a = ((3+y)*x)+(5*b);
a = 3+y*x+5*b;
```

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


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): $\mathbf{Loc} = \mathbf{Idt}$
- 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).

Partielle, endliche Abbildungen

Zustände sind **partielle, endliche Abbildungen** (finite partial maps)

$$f : X \rightharpoonup A$$

Notation:

- $f(x)$ für den Wert von x in f (*lookup*)
- $f(x) = \bot$ wenn x nicht in f (*undefined*)
- $f[x \mapsto n]$ für den Update an der Stelle x mit dem Wert n:

$$f[x \mapsto n](y) \stackrel{def}{=} \begin{cases} n & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$$

Partielle, endliche Abbildungen II

Zustände sind **partielle, endliche Abbildungen** (finite partial maps)

$$f : X \rightharpoonup A$$

Notation:

- $\langle x \mapsto n, y \mapsto m \rangle$ u.ä. für konkrete Abbildungen.
- $\langle\rangle$ ist die leere (überall undefinierte Abbildung):

$$\text{für alle } x \in X \text{ gilt: } \langle\rangle(x) = \bot$$

- Die Domäne eines Zustands sind alle Stellen, an denen er definiert ist:

$$Dom(f) \stackrel{def}{=} \{x \in X | f(x) \neq \bot\}$$

- Updates sind "linksassoziativ":

$$f[x \mapsto n][y \mapsto m] = (f[x \mapsto n])[y \mapsto m]$$

Arbeitsblatt 2.1: Jetzt seid ihr dran!

- In euren Gruppen-Arbeitsblättern unter `https://hackmd.informatik.uni-bremen.de/rfF0alFiS8y6nUtspD4YgA#` gebt folgendes an
- Wie sieht ein Zustand aus, der a den Wert 6 und c den Wert 2 zuweist.
- Welches sind Zustände, und welche nicht:
  - Ⓐ $\langle x \mapsto 1, a \mapsto 3 \rangle$
  - Ⓑ $\langle x \mapsto y, b \mapsto 6 \rangle$
  - Ⓒ $\langle x \mapsto 2, b \mapsto 6, x \mapsto 5 \rangle$
  - Ⓓ $\langle x \mapsto 3, b \mapsto 6, y \mapsto 5 \rangle$
- Update von Zuständen:
  - Ⓐ $\langle x \mapsto 1, a \mapsto 3 \rangle[y \mapsto 1] = ??$
  - Ⓑ $\langle x \mapsto 1, a \mapsto 3 \rangle[x \mapsto 3] = ??$
  - Ⓒ $\langle x \mapsto 1, a \mapsto 3 \rangle[x \mapsto 3][y \mapsto 1][x \mapsto 4] = ??$

Besprechung

- Wie sieht ein Zustand aus, der a den Wert 6 und c den Wert 2 zuweist: $\langle a \mapsto 6, c \mapsto 2 \rangle$
- Welches sind Zustände, und welche nicht:
  - Ⓐ $\langle x \mapsto 1, a \mapsto 3 \rangle$ +
  - Ⓑ $\langle x \mapsto y, b \mapsto 6 \rangle$ −
  - Ⓒ $\langle x \mapsto 2, b \mapsto 6, x \mapsto 5 \rangle$ −
  - Ⓓ $\langle x \mapsto 3, b \mapsto 6, y \mapsto 5 \rangle$ +
- Update von Zuständen:
  - Ⓐ $\langle x \mapsto 1, a \mapsto 3 \rangle[y \mapsto 1] = \langle x \mapsto 1, a \mapsto 3, y \mapsto 1 \rangle$
  - Ⓑ $\langle x \mapsto 1, a \mapsto 3 \rangle[x \mapsto 3] = \langle x \mapsto 3, a \mapsto 3 \rangle$
  - Ⓒ $\langle x \mapsto 1, a \mapsto 3 \rangle[x \mapsto 3][y \mapsto 1][x \mapsto 4] = \langle x \mapsto 4, y \mapsto 1, a \mapsto 3 \rangle$

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Regeln:

$$\frac{n \in \mathbf{Z}}{\langle n, \sigma \rangle \to_{Aexp} [\![n]\!]}$$

$$\frac{x \in \mathbf{Idt}, x \in Dom(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \to_{Aexp} v} \qquad \frac{x \in \mathbf{Idt}, x \notin Dom(\sigma)}{\langle x, \sigma \rangle \to_{Aexp} \bot}$$

Regelschreibweise vs. Funktionen

Sei $\mathrm{Int}+ = \mathrm{Int} \cup \{\bot\}$

```
AexpEval :: AExp -> (Zustand -> Int+)
AexpEval n :: Int s -> n
AexpEval x :: Loc s if Dom(s) contains x -> s(x)
AexpEval x :: Loc s if not(Dom(s) contains x) -> ⊥
```


Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma \rangle \to_{Aexp} n_2 \qquad n_i \in \mathbb{Z}, n \text{ Differenz von } n_1 \text{ und } n_2}{\langle a_1 - a_2, \sigma \rangle \to_{Aexp} n}$$

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

Regelschreibweise vs. Funktionen

Sei $\mathrm{Int}+ = \mathrm{Int} \cup \{\bot\}$

```
AexpEval :: AExp -> (Zustand -> Int+)
AexpEval n :: Int s -> n
AexpEval x :: Loc s if Dom(s) contains x -> s(x)
AexpEval x :: Loc s if not(Dom(s) contains x) -> ⊥
AExpEval (a1 + a2) s -> let n1 = AExpEval a1 s
                            n2 = AExpEval a2 s
                        in
                        if n1 :: Int and n2 :: Int then n1 + n2
                        if n1 == ⊥ or n2 == ⊥ then ⊥
```


Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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

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

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

Arbeitsblatt 2.2: Jetzt seid ihr dran!

- In euren Gruppen-Arbeitsblättern unter
  `https://hackmd.informatik.uni-bremen.de/rfF0alFiS8y6nUtspD4YgA#`
  vervollständigt die Funktion

```
AexpEval :: AExp -> (Zustand -> Int+)
AexpEval n :: Int s -> n
AexpEval x :: Loc s if Dom(s) contains x -> s(x)
AexpEval x :: Loc s if not(Dom(s) contains x) -> ⊥
AExpEval (a1 + a2) s -> let n1 = AExpEval a1 s
                            n2 = AExpEval a2 s
                        in
                        if n1 :: Int and n2 :: Int then n1 + n2
                        if n1 == ⊥ or n2 == ⊥ then ⊥
```

- Ergänzt dies für * und für /
- Für $\bot$ könnt ihr einfach \bot schreiben.

Beispiel-Ableitungen

Sei $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

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

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

Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \mid\mid b_2$
  $\langle b,\sigma\rangle \to_{Bexp} true \mid false \mid \bot$

Regeln:

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

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

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

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

Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \mid\mid b_2$
  $\langle b,\sigma\rangle \to_{Bexp} true \mid false \mid \bot$

Regeln:

$$\frac{\langle b,\sigma\rangle \to_{Bexp} true}{\langle !b,\sigma\rangle \to_{Bexp} false} \qquad \frac{\langle b,\sigma\rangle \to_{Bexp} false}{\langle !b,\sigma\rangle \to_{Bexp} true} \qquad \frac{\langle b,\sigma\rangle \to_{Bexp} \bot}{\langle !b,\sigma\rangle \to_{Bexp} \bot}$$

$$\frac{\langle b_1,\sigma\rangle \to_{Bexp} false}{\langle b_1 \,\&\&\, b_2,\sigma\rangle \to_{Bexp} false} \qquad \frac{\langle b_1,\sigma\rangle \to_{Bexp} \bot}{\langle b_1 \,\&\&\, b_2,\sigma\rangle \to_{Bexp} \bot}$$

$$\frac{\langle b_1,\sigma\rangle \to_{Bexp} true \quad \langle b_2,\sigma\rangle \to_{Bexp} t}{\langle b_1 \,\&\&\, b_2,\sigma\rangle \to_{Bexp} t}$$

Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \mid\mid b_2$
  $\langle b,\sigma\rangle \to_{Bexp} true \mid false \mid \bot$

Regeln:

$$\frac{\langle b_1,\sigma\rangle \to_{Bexp} true}{\langle b_1 \mid\mid b_2,\sigma\rangle \to_{Bexp} true} \qquad \frac{\langle b_1,\sigma\rangle \to_{Bexp} \bot}{\langle b_1 \mid\mid b_2,\sigma\rangle \to_{Bexp} \bot}$$

$$\frac{\langle b_1,\sigma\rangle \to_{Bexp} false \quad \langle b_2,\sigma\rangle \to_{Bexp} t}{\langle b_1 \mid\mid b_2,\sigma\rangle \to_{Bexp} t}$$

Operationale Semantik: Anweisungen

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

Beispiel:

$$\langle c,\sigma\rangle \to_{Stmt} \sigma' \mid \bot$$

$$\langle x = 5,\sigma\rangle \to_{Stmt} \sigma'\sigma[x \mapsto 5]$$

wobei $\sigma'(x) = 5$ und $\sigma'(y) = \sigma(y)$ für alle $y \neq x$
bzw. $\sigma' \stackrel{def}{=} \sigma[x \mapsto 5]$

Operationale Semantik: Anweisungen

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

Regeln:

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

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

$$\frac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \quad \langle c, \sigma\rangle \to_{Stmt} \sigma' \quad \langle \textbf{while } (b) \ c, \sigma'\rangle \to_{Stmt} \sigma''}{\langle \textbf{while } (b) \ c, \sigma\rangle \to_{Stmt} \sigma''}$$

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

Beispiel

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

$\sigma \stackrel{def}{=} \langle y \mapsto 2\rangle$

$$\frac{\langle 1, \sigma\rangle \to_{Aexp} 1}{\langle x = 1, \sigma\rangle \to_{Stmt} \sigma[x \mapsto 1] := \sigma_1} \quad \frac{\dfrac{\langle y, \sigma_1\rangle \to_{Aexp} 2}{\langle y! = 0, \sigma_1\rangle \to_{Bexp} \mathit{true}} \quad \dfrac{(A)}{\langle y = y - 1; x = 2 * x, \sigma_1\rangle \to_{Stmt} ?} \quad \dfrac{(B)}{\langle w, ?\rangle \to_{Stmt} ?}}{\langle \textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}, \sigma_1\rangle \to_{Stmt} ?}$$

$$\langle x = 1; \underbrace{\textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}}_{w}, \sigma\rangle \to_{Stmt} ?$$

(A)

$$\frac{\langle y - 1, \sigma_1\rangle \to_{Aexp} 1}{\langle y = y - 1, \sigma_1\rangle \to_{Stmt} \sigma_1[y \mapsto 1] := \sigma_2} \quad \frac{\langle 2 * x, \sigma_2\rangle \to_{Aexp} 2}{\langle x = 2 * x, \sigma_2\rangle \to_{Stmt} \sigma_2[x \mapsto 2] := \sigma_3}$$

$$\langle y = y - 1; x = 2 * x, \sigma_1\rangle \to_{Stmt} \sigma_3$$

$$\frac{\langle 1, \sigma\rangle \to_{Aexp} 1}{\langle x = 1, \sigma\rangle \to_{Stmt} \sigma_1} \quad \frac{\dfrac{\langle y, \sigma_1\rangle \to_{Aexp} 2}{\langle y! = 0, \sigma_1\rangle \to_{Bexp} \mathit{true}} \quad \dfrac{(A)}{\langle y = y - 1; x = 2 * x, \sigma_1\rangle \to_{Stmt} \sigma_3} \quad \dfrac{(B)}{\langle w, \sigma_3\rangle \to_{Stmt} ?}}{\langle \textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}, \sigma_1\rangle \to_{Stmt} ?}$$

$$\langle x = 1; \underbrace{\textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}}_{w}, \sigma\rangle \to_{Stmt} ?$$

(B)

$$\frac{\dfrac{\langle y, \sigma_3\rangle \to_{Aexp} 1}{\langle y! = 0, \sigma_3\rangle \to_{Bexp} \mathit{true}} \quad \dfrac{\langle y - 1, \sigma_3\rangle \to_{Aexp} 0 \quad \langle 2 * x, \sigma_4\rangle \to_{Aexp} 4}{\langle y = y - 1, \sigma_3\rangle \to_{Stmt} \sigma_3[y \mapsto 0] := \sigma_4 \quad \langle x = 2 * x, \sigma_4\rangle \to_{Stmt} \sigma_4[x \mapsto 4] := \sigma_5} \quad \dfrac{(C)}{\langle w, \sigma_5\rangle \to_{Stmt} \sigma_5}}{\langle y = y - 1; x = 2 * x, \sigma_3\rangle \to_{Stmt} \sigma_5}}{\langle w, \sigma_3\rangle \to_{Stmt} \sigma_5}$$

$$\left.\begin{array}{c}\dfrac{\langle y, \sigma_5\rangle \to_{Aexp} 0}{\langle y! = 0, \sigma_3\rangle \to_{Bexp} \mathit{false}} \\ \langle w, \sigma_5\rangle \to_{Stmt} \sigma_5\end{array}\right\} (C)$$

$$\underbrace{\textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}}_{w}$$

$$\cfrac{\cfrac{\cdots}{\langle x = 1; \underbrace{\textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}}_{w}, \sigma_1\rangle \to_{Stmt} \sigma_5} \quad \dfrac{\langle y, \sigma_1\rangle \to_{Aexp} 2}{\langle y! = 0, \sigma_1\rangle \to_{Bexp} \mathit{true}} \quad \dfrac{(A)}{\langle y = y - 1; x = 2 * x, \sigma_1\rangle \to_{Stmt} \sigma_3} \quad \dfrac{(B)}{\langle w, \sigma_3\rangle \to_{Stmt} \sigma_5}}{\langle x = 1; \underbrace{\textbf{while } (y! = 0) \ \{y = y - 1; x = 2 * x\}}_{w}, \sigma\rangle \to_{Stmt} \sigma_5}$$

$$\sigma_5 = \sigma_4[x \mapsto 4] = \sigma_3[y \mapsto 0][x \mapsto 4] = \sigma_2[x \mapsto 2][y \mapsto 0][x \mapsto 4]$$
$$= \sigma_1[y \mapsto 1][x \mapsto 2][y \mapsto 0][x \mapsto 4] = \langle y \mapsto 2\rangle[y \mapsto 1][x \mapsto 2][y \mapsto 0][x \mapsto 4]$$
$$= \langle y \mapsto 0, x \mapsto 4\rangle$$

und es gilt $\sigma_5(x) = 4 = 2^2 = 2^{\sigma_1(y)}$

Lineare, abgekürzte Schreibweise

```
// ⟨y ↦ 2⟩
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
```

Lineare, abgekürzte Schreibweise

```
// ⟨y ↦ 2⟩
x = 1;                          // Ableitung für x = 1
// ⟨y ↦ 2, x ↦ 1⟩
while(y!=0) // ⟨y! = 0, ⟨y ↦ 2, x ↦ 1⟩⟩ →Bexp true
|           y = y - 1;          // Ableitung für y = y − 1
|           // ⟨y ↦ 1, x ↦ 1⟩
|           x = 2 * x;          // Ableitung für x = 2 ∗ x
|           // ⟨y ↦ 1, x ↦ 2⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
```


Lineare, abgekürzte Schreibweise

```
// ⟨y ↦ 2⟩
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while(y!=0) // ⟨y! = 0, ⟨y ↦ 2, x ↦ 1⟩⟩ →Bexp true
|           y = y - 1;          // Ableitung für y = y − 1
|           // ⟨y ↦ 1, x ↦ 1⟩
|           x = 2 * x;          // Ableitung für x = 2 ∗ x
|           // ⟨y ↦ 1, x ↦ 2⟩
while(y!=0) // ⟨y! = 0, ⟨y ↦ 1, x ↦ 2⟩⟩ →Bexp true
|           y = y - 1;
|           // ⟨y ↦ 0, x ↦ 2⟩
|           x = 2 * x;
|           // ⟨y ↦ 0, x ↦ 4⟩
while(y!=0) // ⟨y! = 0, ⟨y ↦ 0, x ↦ 4⟩⟩ →Bexp false
// ⟨y ↦ 0, x ↦ 4⟩
```


Was haben wir gezeigt?

```
// ⟨y ↦ 2⟩
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
// ⟨y ↦ 0, x ↦ 4⟩
```

- Für **einen festen Anfangszustand** $\sigma_1 = \langle y \mapsto 2 \rangle$ gilt am Ende $x = 4 = 2^2 = 2^{\sigma_1(y)}$.
- Gilt das für alle?
- Für welche nicht?
- Wie kann man das für alle Anfangs-Zustände, für die es gilt, zeigen?

Was passiert hier?

```
// ⟨y ↦ −1⟩
x = 1;
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
```

- Ableitung terminiert nicht (Ableitungsbaum der Auswerttung der while-Schleife wächst unendlich)
- In linearer Schreibweise geht es immer wieder unten weiter.

Arbeitsblatt 2.3: Jetzt seid ihr dran!

- Werten Sie das nebenstehende Programm aus für den Anfangszustand $\langle x \mapsto 5, y \mapsto 2 \rangle$
- Geben Sie die Auswertung in abgekürzter Schreibweise an.
- Welche Beziehung gilt am Ende des Programs zwischen den Werten von x und y im Endzustand und im Anfangszustand?

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


Lineare, abgekürzte Schreibweise

```
while(y!=0) // ⟨x ↦ 5, y ↦ 2⟩                          σ1
|           // ⟨y! = 0, ⟨x ↦ 5, y ↦ 2⟩⟩ →Bexp true
|           x = x * x;
|           // ⟨x ↦ 25, y ↦ 2⟩
|           y = y - 1;
|           // ⟨x ↦ 25, y ↦ 1⟩
while(y!=0) // ⟨y! = 0, ⟨x ↦ 25, y ↦ 1⟩⟩ →Bexp true
|           x = x * x;
|           // ⟨x ↦ 625, y ↦ 1⟩
|           y = y - 1;
|           // ⟨x ↦ 625, y ↦ 0⟩                        σ5
while(y!=0) // ⟨y! = 0, ⟨x ↦ 625, y ↦ 0⟩⟩ →Bexp false
// ⟨x ↦ 625, y ↦ 0⟩
```

Und es gilt $625 = 5^4 = 5^{2^2}$ bzw. $\sigma_5(x) = \sigma_1(x)^{2^{\sigma_1(y)}}$

Äquivalenz arithmetischer Ausdrücke

Gegeben zwei Aexp a_1 and a_2

- Sind sie gleich?

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

```
(x*x) + 2*x*y + (y*y)      und      (x+y) * (x+y)
```

- Wann sind sie gleich?

$$\forall \sigma, n. \langle a_1, \sigma \rangle \to_{Aexp} n \Leftrightarrow \langle a_2, \sigma \rangle \to_{Aexp} n$$

```
x*x     und     8*x+9
x*x     und     x*x+1
```


Äquivalenz Boolscher Ausdrücke

Gegeben zwei Bexp-Ausdrücke b_1 and b_2

- Sind sie gleich?

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

```
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

Gegeben beliebiger Programmzustand σ. 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. $\langle b, \sigma\rangle \to_{Bexp} \bot$:

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

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

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

Beweis II

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

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

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

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

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 27.04.21
Denotationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Organisatorisches

- Übungsblätter: 17 Repositories.
- Organisation der Repos:
  - Flach (keine Unterverzeichnisse `Abgaben`, `ueb01` o.ä.)
  - Lösung in `uebung-XX.pdf` oder `uebung-XX.md`.
  - Abgabe auch in Markdown möglich (git-freundlicher).
  - Bewertungen in `00-BEWERTUNG.md` (fortgeschrieben).
- Feedback.

Fahrplan

Überblick

- Denotationale Semantik für C0
- Fixpunkte

Denotationale Semantik — Motivation

- **Operationale Semantik:**

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

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

- **Denotationale Semantik:**

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

  $$[\![c]\!]_C : \Sigma \rightharpoonup \Sigma$$

Denotationale Semantik — Motivation

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

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

oder

Zwei Programme sind äquivalent gdw. sie dieselbe partielle Funktion **denotieren**

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

Kompositionalität

- Semantik von zusammengesetzten Ausdrücken durch Kombination der Semantiken der Teilausdrücke
  - Bsp: Semantik einer Sequenz von Anweisungen durch Verknüpfung der Semantik der einzelnen Anweisungen
- Operationale Semantik ist **nicht** kompositional:

```
x= 3;
y= x+ 7; // (*)
z= x+ y;
```

  - Semantik von Zeile (*) ergibt sich aus der Ableitung davor
  - Kann nicht unabhängig abgeleitet werden
- Denotationale Semantik ist kompositional.
  - Wesentlicher Baustein: **partielle Funktionen**

Partielle Funktion

Definition (Partielle Funktion)

Eine **partielle Funktion** $f : X \rightharpoonup Y$ ist eine Relation $f \subseteq X \times Y$ so dass wenn $(x, y_1) \in f$ und $(x, y_2) \in f$ dann $y_1 = y_2$ (**Rechtseindeutigkeit**)

- Notation: für $f : X \rightharpoonup Y$, $(x, y) \in f \iff f(x) = y$.
- Wir benutzen beide Notationen, aber für die denotationale Semantik die Paar-Notation.
- Zustände sind partielle Abbildungen ($\longrightarrow$ letzte Vorlesung)
- Insbesondere **Systemzustände** $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

Beispiel

Als Beispiel betrachten wir die partielle Funktion $div3 : \{0 \ldots 10\} \to \mathbb{N}$

$$div3(x) = y \quad \text{g.d.w.} \quad 3 \cdot y = x$$

- Zuordnung:

$$\begin{array}{rcl} 0 & \mapsto & 0 \\ 1 & & \\ 2 & & \\ 3 & \mapsto & 1 \\ 4 & & \\ 5 & & \\ 6 & \mapsto & 2 \\ 7 & & \\ 8 & & \\ 9 & \mapsto & 3 \\ 10 & & \end{array}$$

- Notation als Relation (**Graph**):

$$div3 \stackrel{def}{=} \{(0,0),(3,1),(6,2),(9,3)\}$$

- Wir schreiben

$$\begin{array}{ll} div3(3) = 1 & \text{für } (3,1) \in div3 \\ div3(5) = \bot & \text{für es gibt kein } y \text{ mit } (5,y) \in div3 \\ div3(5) = \bot & \text{für } \forall y.(5,y) \notin div3 \end{array}$$

- Achtung, Partialität!

Arbeitsblatt 3.1: Relationen als Funktionen

Definiert wie im Beispiel eben die Funktion $sqrt : \{0, \ldots, 100\} \to \mathbb{N}$ mit

$$sqrt(x) = y \quad \text{g.d.w.} \quad y^2 = x$$

Was ist der Wert folgender Ausdrücke:

$$t_1 = 5 - sqrt(32) \quad t_2 = sqrt(49) + sqrt(0) \quad t_3 = \sqrt{3} \cdot sqrt(3) \quad t_4 = \frac{sqrt(64)}{0}$$

Denotierende Funktionen (Denotate)

- Arithmetische Ausdrücke: $a \in$ **Aexp** denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbb{Z}$
- Boolsche Ausdrücke: $b \in$ **Bexp** denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbb{B}$
- Anweisungen: $c \in$ **Stmt** denotiert eine partielle Funktion $\Sigma \rightharpoonup \Sigma$

Denotat von Aexp

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

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

Rechtseindeutigkeit

Lemma (Partielle Funktion)

$[\![-]\!]_{\mathcal{A}}$ *ist rechtseindeutig und damit eine* ***partielle Funktion****.*

Beweis.

z.z.: wenn $(\sigma, v_1) \in [\![a]\!]_{\mathcal{A}}, (\sigma, v_2) \in [\![a]\!]_{\mathcal{A}}$ dann $v_1 = v_2$.
Strukturelle Induktion über **Aexp**:

- Induktionsbasis sind $n \in \mathbf{Z}$ und $x \in$ **Idt**.
  Sei $a \equiv x$, dann $v_1 = \sigma(x) = v_2$.
- Induktionssschritt sind die anderen Klauseln.
  Sei $a \equiv a_1 + a_2$.
  Induktionsannahme ist: wenn $(\sigma, n_i) \in [\![a_i]\!]_{\mathcal{A}}, (\sigma, m_i) \in [\![a_i]\!]_{\mathcal{A}}$ dann $n_i = m_i$.
  Sei $v_1 = (\sigma, n_1 + n_2)$ mit $(\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, n_2) \in [\![a_2]\!]_{\mathcal{A}}$, und $v_2 = m_1 + m_2$ mit $(\sigma, m_1) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, m_2) \in [\![a_2]\!]_{\mathcal{A}}$.
  Aus der Annahme folgt $n_1 = m_1$ und $n_2 = m_2$, deshalb $v_1 = v_2$.

□

Kompositionalität und Striktheit

- Die Rechtseindeutigkeit erlaubt die Notation als partielle Funktion:

$$\begin{aligned} [\![3 * (x + y)]\!]_{\mathcal{A}}(\sigma) &= [\![3]\!]_{\mathcal{A}}(\sigma) \cdot ([\![x]\!]_{\mathcal{A}}(\sigma) + [\![y]\!]_{\mathcal{A}}(\sigma)) \\ &= 3 \cdot ([\![x]\!]_{\mathcal{A}}(\sigma) + [\![y]\!]_{\mathcal{A}}(\sigma)) \\ &= 3 \cdot (\sigma(x) + \sigma(y)) \end{aligned}$$

- Diese Notation versteckt die **Partialität**:

$$[\![1 + x/0]\!]_{\mathcal{A}}(\sigma) = 1 + \sigma(x)/0 = 1 + \bot = \bot$$

- Wenn ein Teilausdruck undefiniert ist, wird der gesamte Ausdruck undefiniert: $[\![-]\!]_{\mathcal{A}}$ ist **strikt** für alle arithmetischen Operatoren.

Arbeitsblatt 3.2: Semantik I

Hier üben wir noch einmal den Zusammenhang zwischen den beiden Notationen. Gegeben sei der Zustand $s = \langle x \mapsto 3, y \mapsto 4 \rangle$ und der Ausdruck $a = 7 * x + y$.
Berechnen Sie die Semantik zum einen als Relation (füllen Sie die Fragezeichen aus):

```
(s, ?) : [[7]]
(s, ?) : [[x]]
(s, ?) : [[7*x]]
(s, ?) : [[y]]
(s, ?) : [[7*x+ y]]
```

Berechnen Sie zum anderen die Semantik in der Funktionsnotation:

```
[[7*x+y]](s) = [[7*x]](s)+[[y]](s) = ... = ?
```

Ist das Ergebnis am Ende gleich?

Lösung

Denotat von Bexp

$$[\![a]\!]_{\mathcal{B}} : \textbf{Bexp} \to (\Sigma \rightharpoonup \mathbb{B})$$

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

Denotat von Bexp

$$[\![a]\!]_{\mathcal{B}} : \textbf{Bexp} \to (\Sigma \rightharpoonup \mathbb{B})$$

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

Kompositionalität und Striktheit

Lemma (Partielle Funktion)

$[\![-]\!]_{\mathcal{B}}$ *ist rechtseindeutig und damit eine* ***partielle Funktion****.*

- Beweis analog zu $[\![-]\!]_{\mathcal{A}}$.
- Ist $[\![-]\!]_{\mathcal{B}}$ strikt? Natürlich nicht:
- Sei $[\![b_1]\!]_{\mathcal{B}}(\sigma) = \mathit{false}$, dann $[\![b_1 \;\&\&\; b_2]\!]_{\mathcal{B}}(\sigma) = [\![b_1]\!]_{\mathcal{B}}(\sigma) = \mathit{false}$
- Wir können deshalb nicht so einfach schreiben $[\![b_1 \;\&\&\; b_2]\!]_{\mathcal{B}}(\sigma) = [\![b_1]\!]_{\mathcal{B}}(\sigma) \wedge [\![b_2]\!]_{\mathcal{B}}(\sigma)$
- Die normale zweiwertige Logik behandelt Definiertheit gar nicht. Bei uns müssen die logischen Operatoren links-strikt sein:

$$\begin{array}{lll}
\bot \wedge a = \bot & \mathit{false} \wedge a = \mathit{false} & \mathit{true} \wedge a = a \\
\bot \vee a = \bot & \mathit{true} \vee a = \mathit{true} & \mathit{false} \vee a = a
\end{array}$$

Arbeitsblatt 3.3: Semantik II

Wir üben noch einmal die Nichtstrikheit. Gegeben $s = \langle x \mapsto 7 \rangle$ und $b \equiv (7 == x) \;||\; (x/0 == 1)$
Berechnenen Sie die Semantik in den Notationen von oben:

```
(s, ?) : [[ (7 == x) || (x/0 == 1) ]]
...

[[ (7 == x) || (x/0 == 1) ]](s) = ... ?
```

Hilfreiche Notation: $a \wedge b =$ `a /\ b`, $a \vee b =$ `a \/ b`

Lösung

```
(s, 7) : [[ 7 ]]
(s, 3) : [[ x ]]
(s, True) : [[ 7 == x ]]
(s, True) : [[ (7 == x) || (x/0 == 1) ]]
```

Alternativ:

```
[[ (7 == x) || (x/0 == 1) ]](s) =
[[ (7 == x) ]](s) \/ [[ (x/0 == 1 ]](s) =
([[7]](s) = [[x]](s) \/ [[x/0]](s) = [[1]](s)) =
(7 = 7  \/ \bot = 1) =
(True \/ \bot) =
True
```


Denotationale Semantik von Anweisungen

- Zuweisung: punktweise Änderung des Zustands σ zu $\sigma[x \mapsto n]$
- Sequenz: Komposition von Relationen

Definition (Komposition von Relationen)

Für zwei Relationen $R \subseteq X \times Y, S \subseteq Y \times Z$ ist ihre **Komposition**

$$R \circ S \stackrel{def}{=} \{(x, z) \mid \exists y \in Y.\, (x, y) \in R \wedge (y, z) \in S\}$$

Wenn R, S zwei partielle Funktionen sind, ist $R \circ S$ ihre Funktionskomposition.

- Leere Sequenz: Leere Funktion? Nein, Identität. Für Menge X,

$$\mathbf{Id}_X \stackrel{def}{=} X \times X = \{(x, x) \mid x \in X\} q$$

ist die **Identitätsfunktion** ($\mathbf{Id}_X(x) = x$).

Arbeitsblatt 3.4: Komposition von Relationen

Zur Übung: betrachten Sie folgende Relationen:

$$\begin{aligned}
R &= \{(1,7), (2,3), (3,9), (4,3)\} \\
cS &= \{(1,0), (2,0), (3,1), (3,5), (4,7), (5,9), (7,3), (8,15)\}
\end{aligned}$$

Berechnen Sie $R \circ S = \{(1, ?), \ldots\}$

Denotat von Stmt

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

$$\begin{aligned}
[\![x = a]\!]_{\mathcal{C}} &= \{(\sigma, \sigma[x \mapsto n]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in [\![a]\!]_{\mathcal{A}}\} \\
[\![c_1; c_2]\!]_{\mathcal{C}} &= [\![c_1]\!]_{\mathcal{C}} \circ [\![c_2]\!]_{\mathcal{C}} \\
[\![\{\,\}]\!]_{\mathcal{C}} &= \mathbf{Id}_{\Sigma} \\
[\![\textbf{if } (b) \; c_0 \; \textbf{else} \; c_1]\!]_{\mathcal{C}} &= \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_0]\!]_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}}\}
\end{aligned}$$

Aber was ist

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

Denotationale Semantik von while

- Sei $w \equiv$ **while** (b) c (und $\sigma \in \Sigma$). Operational gilt:

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

- Dann sollte auch gelten

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

- Das ist eine **rekursive** Definition von $[\![w]\!]_{\mathcal{C}}$:

$$x = F(x)$$

- Das ist ein **Fixpunkt**:

$$x = \mathit{fix}(F)$$

- Was ist das?

Fixpunkte

Definition (Fixpunkt)

Für $f : X \rightharpoonup X$ ist ein **Fixpunkt** ein $x \in X$ so dass $f(x) = x$.

- Hat jede Funktion $f : X \rightharpoonup X$ einen Fixpunkt? Nein
- Kann eine Funktion mehrere Fixpunkte haben? Ja — aber nur einen kleinsten.
- 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
  - Die Funktion $f(x) = x + 1$ hat keinen Fixpunkt in $\mathbb{Z}$
  - Die Funktion $f(X) = \mathbb{P}(X)$ hat überhaupt keinen Fixpunkt
- $\mathit{fix}(f)$ ist also der **kleinste Fixpunkt** von f.

Konstruktion des kleinsten Fixpunktes (Kurzversion)

- Gegeben Funktion Γ auf Denotaten $\Gamma : (\Sigma \rightharpoonup \Sigma) \rightharpoonup (\Sigma \rightharpoonup \Sigma)$
- Wir konstruieren eine Sequenz $\Gamma^i : \Sigma \rightharpoonup \Sigma$ (mit $i \in \mathbb{N}$) von Funktionen:

$$\Gamma^0(s) \stackrel{\mathit{def}}{=} \emptyset$$
$$\Gamma^{i+1}(s) \stackrel{\mathit{def}}{=} \Gamma(\Gamma^i)(s)$$

- Dann ist

$$\mathit{fix}(\Gamma) \stackrel{\mathit{def}}{=} \bigcup_{i \in \mathbb{N}} \Gamma^i$$

- Verkürzte Version — der Fixpunkt muss so nicht existieren (er tut es aber für alle Programme)

Denotationale Semantik für die Iteration

- Sei $w \equiv$ **while** (b) c
- Konstruktion: "Auffalten" der Schleife (f ist ein Denotat):

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

- b und c sind Parameter von Γ
- Dann ist

$$[\![w]\!]_{\mathcal{C}} = \mathit{fix}(\Gamma)$$

Denotation für Stmt

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

$$\begin{aligned}[\![x = a]\!]_{\mathcal{C}} &= \{(\sigma, \sigma[x \mapsto n]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in [\![a]\!]_{\mathcal{A}}\} \\ [\![c_1; c_2]\!]_{\mathcal{C}} &= [\![c_1]\!]_{\mathcal{C}} \circ [\![c_2]\!]_{\mathcal{C}} \\ [\![\{\}]\!]_{\mathcal{C}} &= \mathbf{Id}_{\Sigma} \\ [\![\textbf{if } (b) \ c_0 \ \textbf{else} \ c_1]\!]_{\mathcal{C}} &= \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_0]\!]_{\mathcal{C}}\} \\ &\quad \cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}}\} \\ [\![\textbf{while } (b) \ c]\!]_{\mathcal{C}} &= \mathit{fix}(\Gamma)\end{aligned}$$

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

Der Fixpunkt bei der Arbeit (I)

```
while (x < 0) {
  x= x+1;
  }
```

$$\Gamma(f)(\sigma) \stackrel{\mathit{def}}{=} \begin{cases}\sigma & \sigma(x) \geq 0 \\ f(\sigma[x \mapsto \sigma(x) + 1]) & \sigma(x) < 0\end{cases}$$

Wir betrachten den Zustand $s = \langle x \mapsto ? \rangle$ (nur eine Variable):

s	$\Gamma^0(s)$	$\Gamma^1(s)$	$\Gamma^2(s)$	$\Gamma^3(s)$
-2	$\bot$	$\Gamma^0(s[x \mapsto -1]) = \bot$	$\Gamma^1(s[x \mapsto -1]) = \bot$	$\Gamma^2(s[x \mapsto -1]) = 0$
-1	$\bot$	$\Gamma^0(s[x \mapsto 0]) = \bot$	$\Gamma^1(s[x \mapsto 0]) = 0$	$\Gamma^2(s[x \mapsto 0]) = 0$
0	$\bot$	0	0	0
1	$\bot$	1	1	1

Der Fixpunkt bei der Arbeit (II)

```
x= 0;
while (n > 0) {
  x= x+n;
  n= n-1;
  }
```

$$\Gamma(f)(\sigma) = \begin{cases}\sigma & \sigma(n) \leq 0 \\ f(\sigma[x \mapsto \sigma(x) + \sigma(n)][n \mapsto \sigma(n) - 1]) & \sigma(n) > 0\end{cases}$$

Wir betrachten Zustände $s = \langle x \mapsto ?, n \mapsto ? \rangle$ (zwei Variablen).
Der Wert von x im Initialzustand ist dabei unerheblich:

s	$\Gamma^0(s)$		$\Gamma^1(s)$		$\Gamma^2(s)$		$\Gamma^3(s)$		$\Gamma^4(s)$		$\Gamma^5(s)$	
n	x	n	x	n	x	n	x	n	x	n	x	n
-1	$\bot$	$\bot$	0	-1	0	-1	0	-1	0	-1	0	-1
0	$\bot$	$\bot$	0	0	0	0	0	0	0	0	0	0
1	$\bot$	$\bot$	$\bot$	$\bot$	1	0	1	0	1	0	1	0
2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	3	0	3	0	3	0
3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	6	0	6	0
4	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	10	0

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

```
x= 0;
while (n != 0) {
  x= x+n;
  n= n-1;
  }
```

$$\Gamma(f)(\sigma) = \begin{cases}\sigma & \sigma(n) = 0 \\ f(\sigma[x \mapsto \sigma(x) + \sigma(n)][n \mapsto \sigma(n) - 1]) & \text{sonst}\end{cases}$$

Jetzt ergibt sich:

s	$\Gamma^0(s)$		$\Gamma^1(s)$		$\Gamma^2(s)$		$\Gamma^3(s)$		$\Gamma^4(s)$	
n	x	n	x	n	x	n	x	n	x	n
-2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
-1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	$\bot$	0	0	0	0	0	0	0	0
1	$\bot$	$\bot$	$\bot$	$\bot$	1	0	1	0	1	0
2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	3	0	3	0
3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	6	0

Der Fixpunkt bei der Arbeit (IV)

```
while (1) {
  x= x+1;
  }
```

$$\Gamma(f)(\sigma) \stackrel{def}{=} f(\sigma[x \mapsto \sigma(x)+1])$$

Jetzt ergibt sich:

s	$\Gamma^0(s)$	$\Gamma^1(s)$	$\Gamma^2(s)$	$\Gamma^3(s)$
-2	$\bot$	$\bot$	$\bot$	$\bot$
-1	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	$\bot$	$\bot$	$\bot$
1	$\bot$	$\bot$	$\bot$	$\bot$
2	$\bot$	$\bot$	$\bot$	$\bot$
3	$\bot$	$\bot$	$\bot$	$\bot$

Arbeitsblatt 3.5: Semantik III

Wir betrachten das Beispielprogramm:

```
x= 1;
while (n > 0) {
  x= x*n;
  n= n-1;
  }
```

Berechnen Sie wie oben den Fixpunkt:

s	G^0	G^1	G^2	G^3	G^4
n	x n	x n	x n	x n	x n
0					
1					
2					
3					

Arbeitsblatt 3.5: Semantik III

Wir betrachten das Beispielprogramm:

```
x= 1;
while (n > 0) {
  x= x*n;
  n= n-1;
  }
```


Der Fixpunkt bei der Arbeit (V)

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

$$\Gamma(f)(\sigma) \stackrel{def}{=} \begin{cases} \sigma & \sigma(i) > \sigma(n) \\ f(\sigma[x \mapsto \sigma(x)+\sigma(i)][i \mapsto \sigma(i)+1]) & \text{sonst} \end{cases}$$

Wir betrachten nur die **while**-Schleife mit $s = \langle n \mapsto ?, i \mapsto ?, x \mapsto ? \rangle$.

s		$\Gamma^0(s)$			$\Gamma^1(s)$			$\Gamma^2(s)$			$\Gamma^3(s)$			$\Gamma^4(s)$		
n	i	n	i	x	n	i	x	n	i	x	n	i	x	n	i	x
0	0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	0	1	x	0	1	x	0	1	x
0	1	$\bot$	$\bot$	$\bot$	0	1	x	0	1	x	0	1	x	0	1	x
1	0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	1	2	$x+1$	1	2	$x+1$
1	1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	1	2	$x+1$	1	2	$x+1$	1	2	$x+1$
1	2	$\bot$	$\bot$	$\bot$	1	2	x	1	2	x	1	2	x	1	2	x
2	0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	2	3	$x+3$
2	1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	2	3	$x+3$	2	3	$x+3$
2	2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	2	3	$x+2$	2	3	$x+2$	2	3	$x+2$
2	3	$\bot$	$\bot$	$\bot$	2	3	x	2	3	x	2	3	x	2	3	x

Weitere Eigenschaften der denotationalen Semantik

Lemma (Partielle Funktion)

*$[\![-]\!]_C$ ist rechtseindeutig und damit eine **partielle Funktion**.*

- Beweis über strukturelle Induktion über $c \in$ **Stmt** und über **Fixpunktinduktion**:
  - Zu zeigen: wenn s rechtseindeutig, dann ist $\Gamma(s)$ rechtseindeutig
  - Dann ist $fix(\Gamma)$ rechtseindeutig.
- Eigenschaften der Iteration:
  - Sei $w \equiv$ **while** (b) c
  - Dann

$$[\![w]\!]_C = [\![\textbf{if } (b) \ \{c; w\} \ \textbf{else} \ \{\}]\!]_C \quad (1)$$
$$(\sigma, \sigma') \in [\![w]\!]_C \Longrightarrow (\sigma', \mathit{false}) \in [\![b]\!]_B \quad (2)$$

Beweis (1)

$$\begin{aligned}
[\![w]\!]_C &= fix(\Gamma) \\
&= \Gamma(fix(\Gamma)) \\
&= \Gamma([\![w]\!]_C) \\
&= \{(\sigma,\sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_B \wedge (\sigma,\sigma') \in [\![c]\!]_C \circ [\![w]\!]_C\} \\
&\quad \cup \{(\sigma,\sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]_B\} \\
&= \{(\sigma,\sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_B \wedge (\sigma,\sigma') \in [\![c;\ w]\!]_C\} \\
&\quad \cup \{(\sigma,\sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]_B \wedge (\sigma,\sigma) \in [\![\{\}]\!]_C\} \\
&= [\![\textbf{if } (b) \ \{c; w\} \ \textbf{else} \ \{\}]\!]_C
\end{aligned}$$

Note

$$fix(\Gamma) = \Gamma(fix(\Gamma))\Gamma(s) = \begin{array}{l}\{(\sigma,\sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_B \wedge (\sigma,\sigma') \in [\![c]\!]_C \circ s\} \\ \cup \{(\sigma,\sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]_B\}\end{array} [\![\textbf{if } (b) \ c_0 \ \textbf{else} \ c_1]\!]_C = \begin{array}{l}\{(\sigma, \\ \cup \{(\end{array}$$

Zusammenfassung

- Die denotationale Semantik bildet Programme (Ausdrücke) auf **partielle Funktionen** $\Sigma \rightharpoonup \Sigma$ ab.
- Zentral ist der Begriff des **kleinsten Fixpunktes**, der die Semantik der while-Schleife bildet.
- Undefiniertheit wird **implizit** behandelt (durch die Partialität von $\Sigma \rightharpoonup \Sigma$).
  - Nicht-Termination und Undefiniertheit sind semantisch äquivalent.
- Genaues Verhältnis zur **operationalen Semantik?** Nächste Vorlesung

Korrekte Software: Grundlagen und Methoden
Vorlesung 4 vom 4/6.05.21
Äquivalenz der Operationalen und Denotationalen Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma\rangle \to_{Aexp} n$	Denotational $[\![a]\!]_{\mathcal{A}}$
a_1/a_2	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad m \neq 0 \quad m, n \neq \bot}{\langle a_1 \circ a_2, \sigma\rangle \to_{Aexp} n \circ^{I} m}$	$\{(\sigma, n/m) \mid \sigma \in \Sigma, (\sigma, n) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}, m \neq 0\}$
	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad n = \bot, m = \bot \text{ oder } m = 0}{\langle a_1/a_2, \sigma\rangle \to_{Aexp} \bot}$	

Äquivalenz operationale und denotationale Semantik

- Für alle $a \in$ **Aexp**, für alle $n \in \mathbb{Z}$, für alle Zustände σ:

$$\langle a, \sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$$
$$\langle a, \sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin Dom([\![a]\!]_{\mathcal{A}})$$

- Beweis Prinzip?

Induktionsprinzip

Noether'sche Induktion

Sei $\succ$ eine **wohlfundierte Ordnung** über S und P eine Aussage über Elemente von S. Dann gilt

$$\frac{\forall v \in S.(\forall u \in S.v \succ u \wedge P(u)) \Rightarrow P(v)}{\forall x \in S.P(x)}$$

- Eine binäre Relation $\succ \subseteq S \times S$ ist eine Ordnung wenn gilt

$$\forall x \in S.\, x \not\succ x \qquad (irreflexiv)$$
$$\forall x, y \in S.\, x \succ y \Rightarrow y \not\succ x \qquad (asymmetrisch)$$
$$\forall x, y, z \in S.\, (x \succ y \wedge y \succ z) \Rightarrow x \succ z \qquad (transitiv)$$

- Eine Ordnung $\prec$ ist wohlfundiert, wenn es keine unendlich **absteigenden** Ketten gibt

$$a_1 \succ a_2 \succ a_3 \succ \ldots$$

	S	$\succ$
Mathematische Induktion	$\mathbb{N}$	$n \to n+1$
Strukturelle Induktion **Aexp**	**Aexp**	$a \succ a'$ genau dann, wenn a' ist Teilausdruck von a Bspw: x Teilausdruck von $(2 * x + 1)$ Ebenso $2 * x$ und 1

Arbeitsblatt 4.1: Übung zu struktureller Ordnung

Die strukturelle Ordnung auf arithmetischen Ausdrücken ist definiert als:

$$\forall a, a' \in \textbf{AExp}.a \succ a' \Leftrightarrow a' \text{ ist Teilausdruck von } a$$

Dabei ist "Teilausdruck" formalisiert als $\circ \in \{+, *, -, /\}$:

$$a \text{ Teilausdruck-von}(a_1 \circ a_2) \Leftrightarrow \left(\begin{array}{c} a = a_1 \vee a \text{ Teilausdruck-von } a_1 \vee \\ a = a_2 \vee a \text{ Teilausdruck-von } a_2 \end{array}\right)$$

- Argumentiert/beweist, dass die Relation "Teilausdruck-von"
  1. irreflexiv
  2. asymmetrisch und
  3. transitiv

  ist.

Äquivalenz operationale und denotationale Semantik

- Für alle $a \in$ **Aexp**, für alle $n \in \mathbb{Z}$, für alle Zustände σ:

$$\langle a, \sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$$
$$\langle a, \sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin Dom([\![a]\!]_{\mathcal{A}})$$

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

Beweis $\forall a \in \mathbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma.\quad \langle a,\sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma,n) \in [\![a]\!]_{\mathcal{A}}$
$\wedge\quad \langle a,\sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a]\!]_{\mathcal{A}})$

Induktionsanfänge

- $a \equiv m \in \mathbf{Z}$:

$$\left.\begin{array}{l}\langle m,\sigma\rangle \to_{Aexp} [\![m]\!] \\ [\![m]\!]_{\mathcal{A}} = \{(\sigma', [\![m]\!]) \mid \sigma' \in \Sigma\} \Rightarrow (\sigma, [\![m]\!]) \in [\![m]\!]_{\mathcal{A}}\end{array}\right] \Leftrightarrow$$

- $a \equiv X \in \mathbf{Loc}$:
  1. $X \in \mathit{Dom}(\sigma)$:

$$\left.\begin{array}{l}\langle X,\sigma\rangle \to_{Aexp} \sigma(X) \\ [\![X]\!]_{\mathcal{A}} = \{(\sigma', \sigma'(X)) | \sigma' \in \Sigma, X \in \mathit{Dom}(\sigma)\} \Rightarrow (\sigma, \sigma(X)) \in [\![X]\!]_{\mathcal{A}}\end{array}\right] \Leftrightarrow$$

  2. $X \notin \mathit{Dom}(\sigma)$:

$$\left.\begin{array}{l}\langle X,\sigma\rangle \to_{Aexp} \bot \\ [\![X]\!]_{\mathcal{A}} = \{(\sigma', \sigma'(X)) | \sigma' \in \Sigma, X \in \mathit{Dom}(\sigma)\} \Rightarrow \sigma \notin \mathit{Dom}([\![X]\!]_{\mathcal{A}})\end{array}\right] \Leftrightarrow$$

Beweis $\forall a \in \mathbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma.\quad \langle a,\sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma,n) \in [\![a]\!]_{\mathcal{A}}$
$\wedge\quad \langle a,\sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a]\!]_{\mathcal{A}})$

Induktionsschritte

- $a \equiv a_1 + a_2$:
  1. Fall: $m \neq \bot$ und $n \neq \bot$
     Es gilt

$$[\![a_1 + a_2]\!]_{\mathcal{A}} = \{(\sigma', u+v) | (\sigma', u) \in [\![a_1]\!]_{\mathcal{A}} \text{ und } (\sigma', v) \in [\![a_2]\!]_{\mathcal{A}}\}$$

     Induktionsannahme gilt für a_1 und a_2.

$$\langle a_1 + a_2, \sigma\rangle \to_{Aexp} m + n \overset{\text{(Def. } \langle ..\rangle \to_{Aexp}\text{)}}{\Longleftrightarrow} \langle a_1,\sigma\rangle \to_{Aexp} m \overset{\text{IA für } a_1}{\Longleftrightarrow} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\&\qquad\qquad \&$$
$$\langle a_2,\sigma\rangle \to_{Aexp} n \overset{\text{IA für } a_2}{\Longleftrightarrow} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$
$$\Updownarrow \text{(Def. } [\![.]\!]_{\mathcal{A}}\text{)}$$
$$(\sigma, m+n) \in [\![a_1 + a_2]\!]_{\mathcal{A}}$$

Beweis $\forall a \in \mathbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma.\quad \langle a,\sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma,n) \in [\![a]\!]_{\mathcal{A}}$
$\wedge\quad \langle a,\sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a]\!]_{\mathcal{A}})$

Induktionsschritte

- $a \equiv a_1 + a_2$: Induktionsannahme gilt für a_1 und a_2.
  2. Fall: $m = \bot$ oder $n = \bot$

$$\frac{\langle a_1,\sigma\rangle \to_{Aexp} n \qquad \langle a_2,\sigma\rangle \to_{Aexp} m \qquad m = \bot \text{ oder } n = \bot}{\langle a_1 + a_2,\sigma\rangle \to_{Aexp} \bot}$$

  - Fall $n = \bot$.
    Aus Induktionsannahme folgt, dass $\langle a_1,\sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a_1]\!]_{\mathcal{A}})$.
    Weiterhin gilt

$$[\![a_1 + a_2]\!]_{\mathcal{A}} = \{(\sigma', u+v) | (\sigma', u) \in [\![a_1]\!]_{\mathcal{A}} \text{ und } (\sigma', v) \in [\![a_2]\!]_{\mathcal{A}}\}$$

    Somit gilt $\sigma \notin \mathit{Dom}([\![a_1 + a_2]\!]_{\mathcal{A}})$.
  - Fall $n \neq \bot, m = \bot$: analog.

Beweis $\forall a \in \mathbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma.\quad \langle a,\sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma,n) \in [\![a]\!]_{\mathcal{A}}$
$\wedge\quad \langle a,\sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a]\!]_{\mathcal{A}})$

Induktionsschritte

- $a \equiv a_1 / a_2$:
  1. Fall: $m \neq \bot$ und $n \neq \bot$, $n \neq 0$
     Es gilt

$$[\![a_1/a_2]\!]_{\mathcal{A}} = \{(\sigma', u/v) | (\sigma', u) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', v) \in [\![a_2]\!]_{\mathcal{A}} \text{ und } v \neq 0\}$$

     Induktionsannahme gilt für a_1 und a_2.

$$\langle a_1/a_2,\sigma\rangle \to_{Aexp} m/n \overset{\text{(Def. } \langle ..\rangle \to_{Aexp}\text{)}}{\Longleftrightarrow} \langle a_1,\sigma\rangle \to_{Aexp} m \overset{\text{IA für } a_1}{\Longleftrightarrow} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\&\qquad\qquad \&$$
$$\langle a_2,\sigma\rangle \to_{Aexp} n \overset{\text{IA für } a_2}{\Longleftrightarrow} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$
$$\Updownarrow \text{(Def. } [\![.]\!]_{\mathcal{A}}\text{)}$$
$$(\sigma, m/n) \in [\![a_1/a_2]\!]_{\mathcal{A}}$$

Beweis $\forall a \in \mathbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma.\quad \langle a,\sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma,n) \in [\![a]\!]_{\mathcal{A}}$
$\wedge\quad \langle a,\sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a]\!]_{\mathcal{A}})$

Induktionsschritte

- $a \equiv a_1 / a_2$: Induktionsannahme gilt für a_1 und a_2.
  2. Fall:

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

  - Fall $n = 0$.
    Aus Induktionsannahme folgt, dass $\langle a_2,\sigma\rangle \to_{Aexp} 0 \Leftrightarrow (\sigma, 0) \in [\![a_2]\!]_{\mathcal{A}}$.
    Weiterhin gilt

$$[\![a_1/a_2]\!]_{\mathcal{A}} = \{(\sigma', u/v) | (\sigma', u) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', v) \in [\![a_2]\!]_{\mathcal{A}} \text{ und } v \neq 0\}$$

    Somit gilt $\sigma \notin \mathit{Dom}([\![a_1/a_2]\!]_{\mathcal{A}})$.
  - Fall $n = \bot, m = \bot$: analog wie bei $+$

q.e.d.

Operationale vs. denotationale Semantik

	Operational $\langle b,\sigma\rangle \to_{Bexp} \mathit{false} \mid \mathit{true} \mid \bot$	Denotational $[\![b]\!]_{\mathcal{B}}$
1	$\langle \mathbf{1},\sigma\rangle \to_{Bexp} \mathit{true}$	$\{(\sigma, \mathit{true}) \mid \sigma \in \Sigma\}$
0	$\langle \mathbf{0},\sigma\rangle \to_{Bexp} \mathit{false}$	$\{(\sigma, \mathit{false}) \mid \sigma \in \Sigma\}$

Operationale vs. denotationale Semantik

	Operat. $\langle b,\sigma\rangle \to_{Bexp} t$	Denotational $[\![b]\!]_{\mathcal{B}}$
$a_0 == a_1$	$\dfrac{\langle a_0,\sigma\rangle \to_{Aexp} n \quad \langle a_1,\sigma\rangle \to_{Aexp} m \quad n,m \neq \bot \quad n = m}{\langle a_0 == a_1,\sigma\rangle \to_{Bexp} \mathit{true}}$ $\dfrac{\langle a_0,\sigma\rangle \to_{Aexp} n \quad \langle a_1,\sigma\rangle \to_{Aexp} m \quad n,m \neq \bot \quad n \neq m}{\langle a_0 == a_1,\sigma\rangle \to_{Bexp} \mathit{false}}$ $\dfrac{\langle a_0,\sigma\rangle \to_{Aexp} n \quad \langle a_1,\sigma\rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_0 == a_1,\sigma\rangle \to_{Bexp} \bot}$	$\{(\sigma, \mathit{true}) \mid \sigma \in \Sigma, (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}, (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}, n_0 = n_1\}$ $\cup$ $\{(\sigma, \mathit{false}) \mid \sigma \in \Sigma, (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}, (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}, n_0 \neq n_1\}$
$a1 < a2$	analog	

Operationale vs. denotationale Semantik

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

Äquivalenz operationale und denotationale Semantik

- Für alle $b \in$ **Bexp**, für alle $t \in \mathbb{B}$, for alle Zustände σ:

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

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

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsanfänge

- $b \equiv \mathbf{0}$:

$$\left.\begin{array}{l} \langle \mathbf{0}, \sigma \rangle \rightarrow_{Bexp} false \\ [\![\mathbf{0}]\!]_{\mathcal{A}} = \{(\sigma', false) | \sigma' \in \Sigma\} \Rightarrow (\sigma, false) \in [\![b]\!]_{\mathcal{B}} \end{array}\right] \Leftrightarrow$$

- $b \equiv \mathbf{1}$:

$$\left.\begin{array}{l} \langle \mathbf{1}, \sigma \rangle \rightarrow_{Bexp} true \\ [\![\mathbf{1}]\!]_{\mathcal{A}} = \{(\sigma', true) | \sigma' \in \Sigma\} \Rightarrow (\sigma, true) \in [\![b]\!]_{\mathcal{B}} \end{array}\right] \Leftrightarrow$$

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

- $b \equiv b_1 \&\& b_2$:
  Es gilt

$$[\![b_1 \&\& b_2]\!]_{\mathcal{B}} = \{(\sigma', false) | (\sigma', false) \in [\![b_1]\!]_{\mathcal{B}}\} \cup \{(\sigma', t_2) | (\sigma', true) \in [\![b_1]\!]_{\mathcal{B}} \text{ und } (\sigma', t_2) \in [\![b_2]\!]_{\mathcal{B}}\}$$

  Induktionsannahme gilt für b_1 und b_2.
  - Fall $\langle b_1, \sigma \rangle \rightarrow_{Bexp} \bot$

$$\langle b_1 \&\& b_2, \sigma \rangle \rightarrow_{Bexp} \bot \overset{(\text{Def. } \langle .,. \rangle \rightarrow_{Bexp})}{\Longleftrightarrow} \langle b_1, \sigma \rangle \rightarrow_{Bexp} \bot \overset{\text{IA für } b_1}{\Longleftrightarrow} \sigma \notin Dom([\![b_1]\!]_{\mathcal{B}}) \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} \sigma \notin [\![b_1 \&\& b_2]\!]_{\mathcal{B}}$$

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

- $b \equiv b_1 \&\& b_2$:
  Es gilt

$$[\![b_1 \&\& b_2]\!]_{\mathcal{B}} = \{(\sigma', false) | (\sigma', false) \in [\![b_1]\!]_{\mathcal{B}}\} \cup \{(\sigma', t_2) | (\sigma', true) \in [\![b_1]\!]_{\mathcal{B}} \text{ und } (\sigma', t_2) \in [\![b_2]\!]_{\mathcal{B}}\}$$

  Induktionsannahme gilt für b_1 und b_2.
  - Fall $\langle b_1, \sigma \rangle \rightarrow_{Bexp} false$

$$\langle b_1 \&\& b_2, \sigma \rangle \rightarrow_{Bexp} false \overset{(\text{Def. } \langle .,. \rangle \rightarrow_{Bexp})}{\Longleftrightarrow} \langle b_1, \sigma \rangle \rightarrow_{Bexp} false \overset{\text{IA für } b_1}{\Longleftrightarrow} (\sigma, false) \in [\![b_1]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} (\sigma, false) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}}$$

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

- $b \equiv b_1 \&\& b_2$:

$$[\![b_1 \&\& b_2]\!]_{\mathcal{B}} = \{(\sigma', false) | (\sigma', false) \in [\![b_1]\!]_{\mathcal{B}}\} \cup \{(\sigma', t_2) | (\sigma', true) \in [\![b_1]\!]_{\mathcal{B}} \text{ und } (\sigma', t_2) \in [\![b_2]\!]_{\mathcal{B}}\}$$

  Induktionsannahme gilt für b_1 und b_2.
  - Fall $\langle b_1, \sigma \rangle \rightarrow_{Bexp} true, \langle b_2, \sigma \rangle \rightarrow_{Bexp} false$

$$\langle b_1 \&\& b_2, \sigma \rangle \rightarrow_{Bexp} false \overset{(\text{Def. } \langle .,. \rangle \rightarrow_{Bexp})}{\Longleftrightarrow} \langle b_1, \sigma \rangle \rightarrow_{Bexp} true \overset{\text{IA für } b_1}{\Longleftrightarrow} (\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$$
$$\& \qquad \&$$
$$\langle b_2, \sigma \rangle \rightarrow_{Bexp} false \overset{\text{IA für } b_2}{\Longleftrightarrow} (\sigma, false) \in [\![b_2]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} (\sigma, false) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}}$$

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

- $b \equiv b_1 \&\& b_2$:

$$[\![b_1 \&\& b_2]\!]_{\mathcal{B}} = \{(\sigma', false) | (\sigma', false) \in [\![b_1]\!]_{\mathcal{B}}\} \cup \{(\sigma', t_2) | (\sigma', true) \in [\![b_1]\!]_{\mathcal{B}} \text{ und } (\sigma', t_2) \in [\![b_2]\!]_{\mathcal{B}}\}$$

  Induktionsannahme gilt für b_1 und b_2.
  - Fall $\langle b_1, \sigma \rangle \rightarrow_{Bexp} true, \langle b_2, \sigma \rangle \rightarrow_{Bexp} true$

$$\langle b_1 \&\& b_2, \sigma \rangle \rightarrow_{Bexp} true \overset{(\text{Def. } \langle .,. \rangle \rightarrow_{Bexp})}{\Longleftrightarrow} \langle b_1, \sigma \rangle \rightarrow_{Bexp} true \overset{\text{IA für } b_1}{\Longleftrightarrow} (\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$$
$$\& \qquad \&$$
$$\langle b_2, \sigma \rangle \rightarrow_{Bexp} true \overset{\text{IA für } b_2}{\Longleftrightarrow} (\sigma, true) \in [\![b_2]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} (\sigma, true) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}}$$

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

- $b \equiv b_1 \&\& b_2$:

$$[\![b_1 \&\& b_2]\!]_{\mathcal{B}} = \{(\sigma', false) | (\sigma', false) \in [\![b_1]\!]_{\mathcal{B}}\} \cup \{(\sigma', t_2) | (\sigma', true) \in [\![b_1]\!]_{\mathcal{B}} \text{ und } (\sigma', t_2) \in [\![b_2]\!]_{\mathcal{B}}\}$$

  Induktionsannahme gilt für b_1 und b_2.
  - Fall $\langle b_1, \sigma \rangle \rightarrow_{Bexp} true, \langle b_2, \sigma \rangle \rightarrow_{Bexp} \bot$

$$\langle b_1 \&\& b_2, \sigma \rangle \rightarrow_{Bexp} \bot \overset{(\text{Def. } \langle .,. \rangle \rightarrow_{Bexp})}{\Longleftrightarrow} \langle b_1, \sigma \rangle \rightarrow_{Bexp} true \overset{\text{IA für } b_1}{\Longleftrightarrow} (\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$$
$$\& \qquad \&$$
$$\langle b_2, \sigma \rangle \rightarrow_{Bexp} \bot \overset{\text{IA für } b_2}{\Longleftrightarrow} \sigma \notin Dom([\![b_2]\!]_{\mathcal{B}}) \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} \sigma \notin Dom([\![b_1 \&\& b_2]\!]_{\mathcal{B}})$$

Beweis $\forall b \in \textbf{Bexp}. \forall t \in \mathbb{B}. \forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \land \langle b, \sigma \rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \textbf{Dom}([\![b]\!]_{\mathcal{B}})$

- $(\sigma, true) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} (\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$ und $(\sigma, true) \in [\![b_2]\!]_{\mathcal{B}}$
  - Siehe Folie 22
- $(\sigma, false) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} (\sigma, false) \in [\![b_1]\!]_{\mathcal{B}}$ oder $(\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$ und $(\sigma, false) \in [\![b_2]\!]_{\mathcal{B}}$
  - Siehe Folie 20 und 21
- $\sigma \notin Dom([\![b_1 \&\& b_2]\!]_{\mathcal{B}}) \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} \sigma \notin Dom([\![b_1]\!]_{\mathcal{B}})$ oder $\sigma \notin Dom([\![b_2]\!]_{\mathcal{B}})$
  - Siehe Folie 19 und 23

Somit gilt dann auch $\Leftrightarrow$ *q.e.d.*

Arbeitsblatt 4.2: Beweis Induktionsanfang

1. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{true} \Leftrightarrow (\sigma, \mathit{true}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
2. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{false} \Leftrightarrow (\sigma, \mathit{false}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
3. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a_1 == a_2]\!]_{\mathcal{B}})$

Beweist obige drei Aussagen unter Verwendung des für arithmetische Ausdrücke geltenden Lemmas

$$\forall a \in \mathbf{Aexp}.\, \forall n \in \mathbb{Z}.\, \forall \sigma.\quad \langle a, \sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma, n) \in [\![a]\!]_{\mathcal{A}} \\ \wedge\ \langle a, \sigma\rangle \to_{Bexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a]\!]_{\mathcal{A}})$$

Beweis
1. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{true} \Leftrightarrow (\sigma, \mathit{true}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
2. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{false} \Leftrightarrow (\sigma, \mathit{false}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
3. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a_1 == a_2]\!]_{\mathcal{B}})$

$$[\![a_1 == a_2]\!]_{\mathcal{B}} = \{(\sigma', \mathit{true}) | (\sigma', m) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', n) \in [\![a_2]\!]_{\mathcal{A}}, m = n\} \\ \cup \{(\sigma', \mathit{false}) | (\sigma', m) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', n) \in [\![a_2]\!]_{\mathcal{A}}, m \neq n\}$$

- Fall $\langle a_1, \sigma\rangle \to_{Bexp} m, \langle b_2, \sigma\rangle \to_{Bexp} n, m = n$

$$\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{true} \overset{\text{(Def. } \langle ..,.. \rangle \to_{Bexp})}{\Longleftrightarrow} \langle a_1, \sigma\rangle \to_{Bexp} m \overset{\text{IA für } a_1}{\Longleftrightarrow} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\&$$
$$\langle a_2, \sigma\rangle \to_{Bexp} m \overset{\text{IA für } a_2}{\Longleftrightarrow} (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}$$
$$\text{Def. } [\![.]\!]_{\mathcal{B}} \Updownarrow$$
$$(\sigma, \mathit{true}) [\![a_1 == a_2]\!]_{\mathcal{B}}$$

Beweis
1. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{true} \Leftrightarrow (\sigma, \mathit{true}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
2. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{false} \Leftrightarrow (\sigma, \mathit{false}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
3. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a_1 == a_2]\!]_{\mathcal{B}})$

$$[\![a_1 == a_2]\!]_{\mathcal{B}} = \{(\sigma', \mathit{true}) | (\sigma', m) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', n) \in [\![a_2]\!]_{\mathcal{A}}, m = n\} \\ \cup \{(\sigma', \mathit{false}) | (\sigma', m) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', n) \in [\![a_2]\!]_{\mathcal{A}}, m \neq n\}$$

- Fall $\langle a_1, \sigma\rangle \to_{Bexp} m, \langle b_2, \sigma\rangle \to_{Bexp} n, m \neq n$

$$\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{false} \overset{\text{(Def. } \langle ..,.. \rangle \to_{Bexp})}{\Longleftrightarrow} \langle a_1, \sigma\rangle \to_{Aexp} m \overset{\text{Lemma für } a_1}{\Longleftrightarrow} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\&$$
$$\langle a_2, \sigma\rangle \to_{Aexp} n \overset{\text{Lemma für } a_2}{\Longleftrightarrow} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$
$$\text{Def. } [\![.]\!]_{\mathcal{B}} \Updownarrow$$
$$(\sigma, \mathit{false}) [\![a_1 == a_2]\!]_{\mathcal{B}}$$

Beweis
1. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{true} \Leftrightarrow (\sigma, \mathit{true}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
2. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{false} \Leftrightarrow (\sigma, \mathit{false}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
3. $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot \Leftrightarrow \sigma \notin \mathit{Dom}([\![a_1 == a_2]\!]_{\mathcal{B}})$

$$[\![a_1 == a_2]\!]_{\mathcal{B}} = \{(\sigma', \mathit{true}) | (\sigma', m) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', n) \in [\![a_1]\!]_{\mathcal{A}}, m = n\} \\ \cup \{(\sigma', \mathit{false}) | (\sigma', m) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma', n) \in [\![a_2]\!]_{\mathcal{A}}, m \neq n\}$$

- Fall $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot$:

$$\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot \overset{\text{(Def. } \langle ..,.. \rangle \to_{Bexp})}{\Longleftrightarrow} \langle a_1, \sigma\rangle \to_{Aexp} \bot \overset{\text{Lemma für } a_1}{\Longleftrightarrow} \sigma \notin \mathit{Dom}([\![a_1]\!]_{\mathcal{A}})$$
$$\vee \qquad\qquad \vee$$
$$\langle a_2, \sigma\rangle \to_{Aexp} \bot \overset{\text{Lemma für } a_2}{\Longleftrightarrow} \sigma \notin \mathit{Dom}([\![a_2]\!]_{\mathcal{A}})$$
$$\text{Def. } [\![.]\!]_{\mathcal{B}} \Updownarrow$$
$$\sigma \notin \mathit{Dom}([\![a_1 == a_2]\!]_{\mathcal{B}})$$

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

	Operational $\langle c, \sigma\rangle \to_{Stmt} \sigma' \mid \bot$	Denotational $[\![c]\!]_{\mathcal{C}}$
	$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \bot}{\langle c, \sigma\rangle \to_{Stmt} \bot}$	
if (b) c_0	$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \quad \langle c_0, \sigma\rangle \to_{Stmt} \sigma'}{\langle c, \sigma\rangle \to_{Stmt} \sigma'}$	$\{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}, (\sigma, \sigma') \in [\![c_0]\!]_{\mathcal{C}}\}$
else c_1	$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{false} \quad \langle c_1, \sigma\rangle \to_{Stmt} \sigma'}{\langle c, \sigma\rangle \to_{Stmt} \sigma'}$	$\{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}, (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}}\}$

Operationale vs. denotationale Semantik

	Operational $\langle c, \sigma\rangle \to_{Stmt} \sigma' \mid \bot$	Denotational $[\![c]\!]_{\mathcal{C}}$
$\underbrace{\textbf{while } (b)\ c}_{w}$	$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{false}}{\langle w, \sigma\rangle \to_{Stmt} \sigma} \quad \dfrac{\langle b, \sigma\rangle \to_{Bexp} \bot}{\langle w, \sigma\rangle \to_{Stmt} \bot}$	$\mathit{fix}(\Gamma)$
	$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \quad \langle c, \sigma\rangle \to_{Stmt} \sigma' \neq \bot \quad \langle w, \sigma'\rangle \to_{Stmt} \sigma''}{\langle w, \sigma\rangle \to_{Stmt} \sigma''}$	
	$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \quad \langle c, \sigma\rangle \to_{Stmt} \bot}{\langle w, \sigma\rangle \to_{Stmt} \bot}$	

mit

$$\Gamma(\varphi) = \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}, (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}} \circ \varphi\} \\ \cup \{(\sigma, \sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}\}$$

Äquivalenz operationale und denotationale Semantik

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

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

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

Operationale Semantik: C0 Programme

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

Regeln:

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

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

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

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

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

Operationale Semantik: C0 Programme

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

Regeln:

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

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

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

Operationale Semantik: C0 Programme

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

Regeln:

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

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

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

Ableitungstiefe für Programme

- Die Ableitungstiefe einer Programmauswertung mittels Regeln der operationaler Semantik ist die **Anzahl der Regelanwendungen** mit Conclusion der Form $\langle .,. \rangle \to_{Stmt} ..$

$$\frac{\vdots \quad \cdots \quad \vdots}{\begin{array}{c} Prämisse_1 \quad \cdots \quad Prämisse_n \\ Conclusion \end{array}}$$

Operationale Semantik: C0 Programme

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

Regeln: **Programmstruktur** **Ableitungstiefe**

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

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

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

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

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

Äquivalenz operationale und denotationale Semantik

- Für alle $c \in$ **Stmt**, für alle Zustände σ, σ':

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

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

Beweis $\forall c \in$ **Stmt**.$\forall \sigma, \sigma'$. 1. $\langle c, \sigma\rangle \to_{Stmt} \sigma' \Rightarrow (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}}$
2. $\langle c, \sigma\rangle \to_{Stmt} \bot \Rightarrow \sigma \notin$ **Dom**$([\![c]\!]_{\mathcal{C}})$

Induktionsanfang – Ableitungstiefe 1

- Fall $c \equiv x = a$:

$$[\![x = a]\!]_{\mathcal{C}} = \{(\sigma, \sigma[x \mapsto m]) | (\sigma, m) \in [\![a]\!]_{\mathcal{A}}\}$$

- Fall $\langle a, \sigma\rangle \to_{Aexp} m \in \mathbb{Z}$

$$\begin{array}{ccc} \langle x = a, \sigma\rangle \to_{Stmt} \sigma[x \mapsto m] & & \\ \Updownarrow \text{(Def. } \langle ..,.. \rangle \to_{Stmt} .\text{)} & & \\ \langle a, \sigma\rangle \to_{Aexp} m \in \mathbb{Z} & \xLeftrightarrow{\text{Lemma für } a} & (\sigma, m) \in [\![a]\!]_{\mathcal{A}} \\ & & \Downarrow \text{Def. } [\![.]\!]_{\mathcal{C}} \\ & & (\sigma, \sigma[x \mapsto m]) \in [\![x = a]\!]_{\mathcal{C}} \end{array}$$

- Fall $\langle a, \sigma\rangle \to_{Aexp} \bot$:

$$\begin{array}{ccc} \langle x = a, \sigma\rangle \to_{Stmt} \bot & & \\ \Updownarrow \text{(Def. } \langle ..,.. \rangle \to_{Stmt} .\text{)} & & \\ \langle a, \sigma\rangle \to_{Aexp} \bot & \xLeftrightarrow{\text{Lemma für } a} & \sigma \notin Dom([\![a]\!]_{\mathcal{A}}) \\ & & \Downarrow \text{Def. } [\![.]\!]_{\mathcal{C}} \\ & & \sigma \notin Dom([\![x = a]\!]_{\mathcal{C}}) \end{array}$$

Beweis $\forall c \in$ **Stmt**.$\forall \sigma, \sigma'$. 1. $\langle c, \sigma\rangle \to_{Stmt} \sigma' \Rightarrow (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}}$
2. $\langle c, \sigma\rangle \to_{Stmt} \bot \Rightarrow \sigma \notin$ **Dom**$([\![c]\!]_{\mathcal{C}})$

Induktionsschritt:

- Fall $c \equiv$ **if**(b) c_1 **else** c_2:

$$[\![\textbf{if}(b) \ c_1 \ \textbf{else} \ c_2]\!]_{\mathcal{C}} = \{(\sigma, \sigma') | (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}}, (\sigma, true) \in [\![b]\!]_{\mathcal{B}}\} \cup \{(\sigma, \sigma') | (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}}, (\sigma, false) \in [\![b]\!]_{\mathcal{B}}\}$$

- Fall $\langle \sigma, b\rangle \to_{Bexp} true, \langle c_1, \sigma\rangle \to_{Stmt} \sigma'$:

$$\begin{array}{ccccc} \langle \textbf{if}(b) \ c_1 \ \textbf{else} \ c_2, \sigma\rangle & \xrightarrow{\text{Def. } \langle ..,.. \rangle \to_{Stmt} .} & \langle b, \sigma\rangle \to_{Bexp} true & \xLeftrightarrow{\text{Lemma für } b} & (\sigma, true) \in [\![b]\!]_{\mathcal{B}} \\ & & \& & & \& \\ & & \langle c_1, \sigma\rangle \to_{Stmt} \sigma' & \xLeftrightarrow{\text{IH für } c_1} & (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}} \\ & & & & \Downarrow \text{Def. } [\![.]\!]_{\mathcal{C}} \\ & & & & (\sigma, \sigma') \in [\![\textbf{if}(b) \ c_1 \ \textbf{else} \ c_2]\!]_{\mathcal{C}} \end{array}$$

- Fall $\langle \sigma, b\rangle \to_{Bexp} false, \langle c_2, \sigma\rangle \to_{Stmt} \sigma'$:

$$\begin{array}{ccccc} \langle \textbf{if}(b) \ c_1 \ \textbf{else} \ c_2, \sigma\rangle & \xrightarrow{\text{Def. } \langle ..,.. \rangle \to_{Stmt} .} & \langle b, \sigma\rangle \to_{Bexp} false & \xLeftrightarrow{\text{Lemma für } b} & (\sigma, false) \in [\![b]\!]_{\mathcal{B}} \\ & & \& & & \& \\ & & \langle c_2, \sigma\rangle \to_{Stmt} \sigma' & \xLeftrightarrow{\text{IH für } c_2} & (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}} \end{array}$$

Beweis $\forall c \in \mathbf{Stmt}.\forall \sigma,\sigma'.$ 1. $\langle c,\sigma\rangle \to_{Stmt} \sigma' \Rightarrow (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}}$
2. $\langle c,\sigma\rangle \to_{Stmt} \bot \Rightarrow \sigma \notin \mathit{Dom}([\![c]\!]_{\mathcal{C}})$

Induktionsschritt:

- Fall $c \equiv \mathbf{while}(b)\ c$: $\quad [\![\mathbf{while}(b)\ c]\!]_{\mathcal{C}} = \mathit{fix}(\Gamma)$
  - Fall $\langle b,\sigma\rangle \to_{Bexp} \mathit{true}$, $\langle c,\sigma\rangle \to_{Stmt} \sigma'$, $\langle \mathbf{while}(b)\ c,\sigma'\rangle \to_{Stmt} \sigma''$

$$\langle \mathbf{while}(b)\ c,\sigma\rangle \to_{Stmt} \sigma'' \overset{\text{(Def. } \langle\ldots\rangle\to_{Stmt}\text{)}}{\Longleftrightarrow} \langle b,\sigma\rangle \to_{Bexp} \mathit{true} \overset{\text{Lemma für } b}{\Longleftrightarrow} (\sigma,\mathit{true}) \in [\![b]\!]_{\mathcal{B}}$$

$$\& \qquad\qquad \&$$

$$\langle c,\sigma\rangle \to_{Stmt} \sigma' \overset{\text{IH für } \langle c,\sigma\rangle\to_{Stmt}\sigma'}{\Longleftrightarrow} (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}}$$

$$\& \qquad\qquad \&$$

$$\langle \mathbf{while}(b)\ c,\sigma'\rangle \to_{Stmt} \sigma'' \overset{\text{IH für } \langle \mathbf{while}(b)\ c,\sigma'\rangle\to_{Stmt}\sigma''}{\Longleftrightarrow} (\sigma',\sigma'') \in [\![\mathbf{while}(b)\ c]\!]_{\mathcal{C}}$$

$$\Big\Downarrow \text{Def. } [\![\cdot]\!]_{\mathcal{C}}$$

$$(\sigma,\sigma'') \in [\![\mathbf{while}(b)\ c]\!]_{\mathcal{C}}$$

Äquivalenz operationale und denotationale Semantik

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

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

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

Beweis $\forall c \in \mathbf{Stmt}.\forall \sigma,\sigma'.(\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c,\sigma\rangle \to_{Stmt} \sigma'$

Induktionsanfang:

- Fall $c \equiv x = a$:

$$[\![x = a]\!]_{\mathcal{C}} = \{(\sigma'',\sigma''[x \mapsto t])|(\sigma'',t) \in [\![a]\!]_{\mathcal{A}}\}$$

$$\begin{array}{rl} & (\sigma,\sigma') \in \{(\sigma'',\sigma''[x \mapsto t])|(\sigma'',t) \in [\![a]\!]_{\mathcal{A}}\} \\ \overset{\text{Def. } [\![\cdot]\!]_{\mathcal{C}}..}{\Longrightarrow} & \underbrace{(\sigma,t) \in [\![a]\!]_{\mathcal{A}}} \wedge \sigma' = \sigma[x \mapsto t] \\ \overset{\text{Lemma AExp}}{\Longrightarrow} & \langle a,\sigma\rangle \to_{Aexp} t \wedge \sigma' = \sigma[x \mapsto t] \\ \overset{\text{Def. } \langle\ldots\rangle\to_{Stmt}.}{\Longrightarrow} & \langle x = a,\sigma\rangle \to_{Stmt} \sigma[x \mapsto t] \wedge \sigma' = \sigma[x \mapsto t] \\ \Longrightarrow & \langle x = a,\sigma\rangle \to_{Stmt} \sigma' \end{array}$$

- Fall $c \equiv \{\}$

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

$$\begin{array}{rl} & (\sigma,\sigma') \in \{(\sigma'',\sigma'')|\sigma'' \in \Sigma\} \\ \overset{\text{Def. } [\![\cdot]\!]_{\mathcal{C}}..}{\Longrightarrow} & \sigma = \sigma' \\ \overset{\text{Def. } \langle\ldots\rangle\to_{Stmt}.}{\Longrightarrow} & \langle\{\},\sigma\rangle \to_{Stmt} \sigma \wedge \sigma = \sigma' \\ \Longrightarrow & \langle\{\},\sigma\rangle \to_{Stmt} \sigma' \end{array}$$

Beweis $\forall c \in \mathbf{Stmt}.\forall \sigma,\sigma'.(\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c,\sigma\rangle \to_{Stmt} \sigma'$

Induktionsschritt:

- Fall $\mathbf{while}\ (b)\ c$:

$$[\![\mathbf{while}\ (b)\ c]\!]_{\mathcal{C}} = \mathit{fix}(\Gamma)$$
$$\text{mit } \Gamma(s) = \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \circ s\}$$
$$\cup\ \{(\sigma,\sigma) \mid (\sigma,\mathit{false}) \in [\![b]\!]_{\mathcal{B}}\}$$

Induktionshypothese gilt für c

$$\begin{array}{rl} & (\sigma,\sigma') \in [\![\mathbf{while}\ (b)\ c]\!]_{\mathcal{C}} \\ \overset{\text{Def. } [\![\cdot]\!]_{\mathcal{C}}..}{\Longrightarrow} & (\sigma,\sigma') \in \mathit{fix}(\Gamma) \end{array}$$

Beweis $\forall c \in \mathbf{Stmt}.\forall \sigma,\sigma'.(\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c,\sigma\rangle \to_{Stmt} \sigma'$

Induktionsschritt:

- Fall $\mathbf{while}\ (b)\ c$:

$$[\![\mathbf{while}\ (b)\ c]\!]_{\mathcal{C}} = \mathit{fix}(\Gamma)$$
$$\text{mit } \Gamma(s) = \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \circ s\}$$
$$\cup\ \{(\sigma,\sigma) \mid (\sigma,\mathit{false}) \in [\![b]\!]_{\mathcal{B}}\}$$

Induktionshypothese gilt für c

$$\begin{array}{rll} (\sigma,\sigma') \in [\![\mathbf{while}\ (b)\ c]\!]_{\mathcal{C}} & \overset{\text{Def. } [\![\cdot]\!]_{\mathcal{C}}..}{\Longrightarrow} & (\sigma,\sigma') \in \mathit{fix}(\Gamma) \\ & \overset{\text{Def. } \mathit{fix}(\Gamma)}{\Longrightarrow} & (\sigma,\sigma') \in \bigcup_{i\in\mathbb{N}} \Gamma^i(\emptyset) \end{array}$$

Unterbeweis: $\quad \forall i \in \mathbb{N}.(\sigma,\sigma') \in \Gamma^i(\emptyset) \Rightarrow \langle \mathbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma'$ (UB)

Woraus dann folgt, dass $\quad (\sigma,\sigma') \in \bigcup_{i\in\mathbb{N}} \Gamma^i(\emptyset) \Rightarrow \langle \mathbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma'$ (1)

$\forall i \in \mathbb{N}.(\sigma,\sigma') \in \Gamma^i(\emptyset) \Rightarrow \langle \mathbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma'$ (UB)

Es gilt nach wie vor die Induktionshypothese für dieses c, dass

$$\forall \sigma'',\sigma'''.(\sigma'',\sigma''') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c,\sigma''\rangle \to_{Stmt} \sigma''' \quad (IB)$$

Beweis per Induktion über i:
Induktionsanfang

- $i = 0$:

$$\begin{array}{rl} (\sigma,\sigma') \in \Gamma^0(\emptyset) \Rightarrow & (\sigma,\sigma') \in \emptyset \\ \Rightarrow & \mathit{false} \end{array}$$

Implikation trivialerweise erfüllt da $\mathit{false} \Rightarrow F$ immer wahr
Induktionsschritt $i \to i+1$:
Induktionsannahme (UB) gilt für i

$$\begin{array}{rl} & (\sigma,\sigma') \in \Gamma^{i+1}(\emptyset) \\ \Longrightarrow & (\sigma,\sigma') \in \Gamma(\Gamma^i(\emptyset)) \\ \overset{\text{Def. } \Gamma}{\Rightarrow} & (\sigma,\sigma') \in \{(\sigma'',\sigma''') \mid (\sigma'',\mathit{true}) \in [\![b]\!]_{\mathcal{B}}, (\sigma'',\sigma'''') \in [\![c]\!]_{\mathcal{C}}, \\ & \qquad (\sigma'''',\sigma''') \in \Gamma^i(\emptyset)\} \\ & \qquad \cup \{(\sigma'',\sigma'') \mid (\sigma'',\mathit{false}) \in [\![b]\!]_{\mathcal{B}}\} \end{array}$$

Fallunterscheidung über Zugehörigkeit zu welcher Teilmenge

$\forall i \in \mathbb{N}.(\sigma,\sigma') \in \Gamma^i(\emptyset) \Rightarrow \langle \mathbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma'$ (UB)

Es gilt nach wie vor die Induktionshypothese für dieses c, dass

$$\forall \sigma'',\sigma'''.(\sigma'',\sigma''') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c,\sigma''\rangle \to_{Stmt} \sigma''' \quad (IB)$$

Beweis per Induktion über i:
Induktionsschritt $i \to i+1$:
Induktionsannahme (UB) gilt für i

- **Fall** $(\sigma,\sigma') \in \{(\sigma'',\sigma''') \mid (\sigma'',\mathit{true}) \in [\![b]\!]_{\mathcal{B}}, (\sigma'',\sigma'''') \in [\![c]\!]_{\mathcal{C}}, (\sigma'''',\sigma''') \in \Gamma^i(\emptyset)\}$

$$\begin{array}{rl} & (\sigma,\sigma') \in \Gamma(\Gamma^i(\emptyset)) \\ \overset{\text{Def. } \Gamma}{\Rightarrow} & (\sigma,\sigma') \in \{(\sigma'',\sigma''') \mid (\sigma'',\mathit{true}) \in [\![b]\!]_{\mathcal{B}}, (\sigma'',\sigma'''') \in [\![c]\!]_{\mathcal{C}}, \\ & \qquad (\sigma'''',\sigma''') \in \Gamma^i(\emptyset)\} \\ & \qquad \cup \{(\sigma'',\sigma'') \mid (\sigma'',\mathit{false}) \in [\![b]\!]_{\mathcal{B}}\} \\ \overset{Fall}{\Longrightarrow} & \underbrace{(\sigma,\mathit{true}) \in [\![b]\!]_{\mathcal{B}}}_{\text{Lemma BExp}} \wedge \underbrace{(\sigma,\sigma'') \in [\![c]\!]_{\mathcal{C}}}_{\text{IH (IB)}} \wedge \underbrace{(\sigma'',\sigma') \wedge \Gamma^i(\emptyset)}_{\text{IH (UB) für } i} \\ \Longrightarrow & \langle b,\sigma\rangle \to_{Bexp} \mathit{true} \wedge \langle c,\sigma\rangle \to_{Stmt} \sigma'' \wedge \langle \mathbf{while}\ (b)\ c,\sigma''\rangle \to_{Stmt} \sigma' \\ \overset{\langle\ldots\rangle\to_{Stmt}.}{\Longrightarrow} & \langle \mathbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma' \end{array}$$

- **Fall** $(\sigma,\sigma') \in \{(\sigma'',\sigma'') \mid (\sigma'',\mathit{false}) \in [\![b]\!]_{\mathcal{B}}\}$

$$\begin{array}{rl} & (\sigma,\sigma') \in \Gamma(\Gamma^i(\emptyset)) \\ \overset{\text{Def. } \Gamma}{\Rightarrow} & (\sigma,\sigma') \in \{(\sigma'',\sigma''') \mid (\sigma'',\mathit{true}) \in [\![b]\!]_{\mathcal{B}}, (\sigma'',\sigma'''') \in [\![c]\!]_{\mathcal{C}}, \\ & \qquad (\sigma'''',\sigma''') \in \Gamma^i(\emptyset)\} \\ & \qquad \cup \{(\sigma'',\sigma'') \mid (\sigma'',\mathit{false}) \in [\![b]\!]_{\mathcal{B}}\} \\ \overset{Fall}{\Longrightarrow} & \underbrace{(\sigma,\mathit{false}) \in [\![b]\!]_{\mathcal{B}}}_{\text{Lemma BExp}} \wedge \underbrace{\sigma = \sigma'}_{\text{IA für } c_1} \\ \Longrightarrow & \langle b,\sigma\rangle \to_{Bexp} \mathit{false} \wedge \sigma = \sigma' \\ \overset{\langle\ldots\rangle\to_{Stmt}.}{\Longrightarrow} & \langle \mathbf{while}\ (b)\ c,\sigma\rangle \to_{Stmt} \sigma' \end{array}$$

Beweis $\forall c \in \mathbf{Stmt}.\forall \sigma,\sigma'.(\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c,\sigma\rangle \rightarrow_{Stmt} \sigma'$

Induktionsschritt:

- Fall **while** (b) c:

$$[\![\textbf{while } (b) \ c]\!]_{\mathcal{C}} = fix(\Gamma)$$
$$\text{mit } \Gamma(s) = \{(\sigma,\sigma') \mid (\sigma, true) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \circ s\}$$
$$\cup \{(\sigma,\sigma) \mid (\sigma, false) \in [\![b]\!]_{\mathcal{B}}\}$$

Induktionshypothese gilt für c

$$(\sigma,\sigma') \in [\![\textbf{while } (b) \ c]\!]_{\mathcal{C}} \overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} (\sigma,\sigma') \in fix(\Gamma)$$
$$\overset{\text{Def. } fix(\Gamma)}{\Longrightarrow} (\sigma,\sigma') \in \bigcup_{i\in\mathbb{N}} \Gamma^i(\emptyset)$$

Unterbeweis: $\forall i \in \mathbb{N}.(\sigma,\sigma') \in \Gamma^i(\emptyset) \Rightarrow \langle \textbf{while } (b) \ c,\sigma\rangle \rightarrow_{Stmt} \sigma'$ (UB)

Woraus dann folgt, dass $(\sigma,\sigma') \in \bigcup_{i\in\mathbb{N}} \Gamma^i(\emptyset) \Rightarrow \langle \textbf{while } (b) \ c,\sigma\rangle \rightarrow_{Stmt} \sigma'$ (1)

Äquivalenz operationale und denotationale Semantik

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

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

  - Gegenbeispiel für $\Leftarrow$ in der zweiten Aussage: wähle $c \equiv while(1)\{\}$: $[\![c]\!]_{\mathcal{C}} = \emptyset$ aber $\langle c,\sigma\rangle \rightarrow_{Stmt} \perp$ gilt nicht (sondern?).

Fahrplan

Korrekte Software: Grundlagen und Methoden
Vorlesung 5 vom 11/18.05.21
Die Floyd-Hoare-Logik I

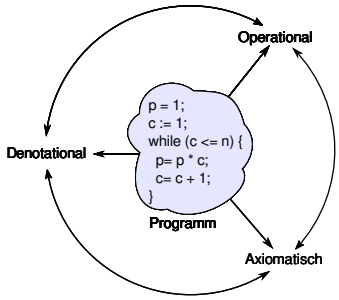
Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Drei Semantiken — Eine Sicht



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.

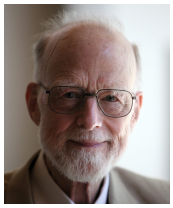
```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

- Operationale/denotionale Semantik nicht für **Korrektheitsbeweise** geeignet: Ausdrücke werden zu groß, skaliert nicht — **Abstraktion** nötig.
- Grundprinzip:
  1. Zustandsabhängige **Zusicherungen** für bestimmte Punkte im Programmablauf.
  2. Berechnung der Gültigkeit dieser Zusicherungen durch **zustandsfreie Regeln**.

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 um eines größer als der Wert von c an Punkt (C)
- Gesamtaussage: Wenn bei (A) der Wert von $n \geq 0$ ist, dann ist bei (E) $p = n!$
- Beobachtung:
  - n ist ein „Eingabevariable“, der Wert am Anfang des Programmes ist relevant;
  - p ist eine „Ausgabevariable“, der Wert am Ende des Programmes ist relevant;
  - c ist eine „Arbeitsvariable“, der Wert am Anfang und Ende ist irrelevant;

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


Arbeitsblatt 5.1: Was berechnet dieses Programm?

```
// (A)
x= 1;
c= 1;
// (B)
while (c <= y) {
  // (C)
  x= 2*x;
  c= c+1;
  // (D)
}
// (E)
```

Betrachtet nebenstehendes Programm.
Analog zu dem Beispiel auf der vorherigen Folie:

1. Was berechnet das Programm?
2. Welches sind „Eingabevariablen“, welches „Ausgabevariablen“, welches sind „Arbeitsvariablen“?
3. Welche Zusicherungen und Zusammenhänge gelten zwischen den Variablen an den Punkten (A) bis (E)?

Auf dem Weg zur Floyd-Hoare-Logik

- Kern der Floyd-Hore-Logik sind **zustandsabhängige Aussagen**
- Aber: wie können wir Aussagen **jenseits** des Zustandes treffen?
- Einfaches Beispiel:

```
x = x+ 1;
```

  - Der Wert von x wird um 1 erhöht
  - Der Wert von x ist hinterher größer als vorher
- Wir benötigen **zustandsfreie** Aussagen, um von Zuständen **vergleichen** zu können.
- Die Logik **abstrahiert** den Effekt von Programmen.

Grundbausteine der Floyd-Hoare-Logik

- **Logische Variablen** (zustandsfrei) und **Programmvariablen** (zustandsabhängig)
- **Zusicherungen** mit logischen und Programmvariablen
- **Floyd-Hoare-Tripel** $\{P\}\, c\, \{Q\}$
  - Vorbedingung P (Zusicherung)
  - Programm c
  - Nachbedingung Q (Zusicherung)
- Floyd-Hoare-Logik abstrahiert von Programmen zu logischen 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** $n!, x^y, \ldots$
  - Implikation und Quantoren $b_1 \longrightarrow b_2, \forall v..b, \exists v..b$
- Formal:

$$\begin{aligned}
\textbf{Aexpv}\ a ::= {} & \mathbf{Z} \mid \mathbf{Idt} \mid \mathbf{Var} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid a_1/a_2 \\
& \mid f(e_1, \ldots, e_n) \\
\textbf{Assn}\ b ::= {} & \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \\
& \mid !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\
& \mid b_1 --> b_2 \mid p(e_1, \ldots, e_n) \mid \backslash\mathbf{forall}\ v.\, b \mid \backslash\mathbf{exists}\ v.\, b \\
\textbf{Assn}\ b ::= {} & \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \\
& \mid \neg b \mid b_1 \wedge b2 \mid b_1 \vee b_2 \\
& \mid b_1 \longrightarrow b_2 \mid p(e_1, \ldots, e_n) \mid \forall v.\, b \mid \exists v.\, b
\end{aligned}$$

Denotationale Semantik von Zusicherungen

- Erste Näherung: Funktion

$$[\![a]\!]_{\mathcal{A}} : \mathbf{Aexpv} \to (\Sigma \rightharpoonup \mathbb{Z})$$
$$[\![b]\!]_{\mathcal{B}} : \mathbf{Assn} \to (\Sigma \rightharpoonup \mathbb{B})$$

  - **Konservative** Erweiterung von $[\![a]\!]_{\mathcal{A}} : \mathbf{Aexp} \to (\Sigma \rightharpoonup \mathbb{Z})$
- Aber: was ist mit den logischen Variablen?
- Zusätzlicher Parameter **Belegung** der logischen Variablen $I : \mathbf{Var} \to \mathbb{Z}$

$$[\![a]\!]_{\mathcal{A}} : \mathbf{Aexpv} \to (\mathbf{Var} \to \mathbb{Z}) \to (\Sigma \rightharpoonup \mathbb{Z})$$
$$[\![b]\!]_{\mathcal{B}} : \mathbf{Assn} \to (\mathbf{Var} \to \mathbb{Z}) \to (\Sigma \rightharpoonup \mathcal{B})$$

Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand σ?
  - Auswertung (denotationale Semantik) ergibt *true*
  - Belegung ist zusätzlicher Parameter

Erfülltheit von Zusicherungen

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

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

Arbeitsblatt 5.2: Zusicherungen

Betrachte folgende Zusicherung:

$$a \quad \equiv \quad \underbrace{2 \cdot x = X}_{p} \longrightarrow \underbrace{x \leq X}_{q}$$

Gegeben folgende Belegungen $I_1, \ldots, I_3$ und Zustände $s_1, \ldots, s_3$:

$$s_1 = \langle x \mapsto 0 \rangle, s_2 = \langle x \mapsto 1 \rangle, s_3 = \langle x \mapsto 5 \rangle$$
$$I_1 = \langle X \mapsto 0 \rangle, I_2 = \langle X \mapsto 2 \rangle, I_3 = \langle X \mapsto 10 \rangle$$

Unter welchen Belegungen und Zuständen ist a wahr?

	I_1			I_2			I_3		
	p	q	a	p	q	q	p	q	a
s_1									
s_2									
s_3									

Wie kann man a so ändern, dass a für **alle** Belegungen und Zustände wahr ist?

Floyd-Hoare-Tripel

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

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

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

- Gleiche Belegung der logischen Variablen in P und Q erlaubt **Vergleich** zwischen Zuständen

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

c ist **total korrekt**, wenn für alle Zustande σ, die P erfüllen, die Ausführung von c mit σ in τ terminiert, und τ erfüllt Q.

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

Beispiele

- Folgendes **gilt**:

$$\models \{\mathit{true}\}\ \mathbf{while}(1)\{\ \}\ \{\mathit{true}\}$$

- Folgendes gilt **nicht**:

$$\models [\mathit{true}]\ \mathbf{while}(1)\{\ \}\ [\mathit{true}]$$

- Folgende **gelten**:

$$\models \{\mathit{false}\}\ \mathbf{while}\ (\mathbf{1})\ \{\ \}\ \{\mathit{true}\}$$
$$\models [\mathit{false}]\ \mathbf{while}\ (\mathbf{1})\ \{\ \}\ [\mathit{true}]$$

Wegen *ex falso quodlibet*: $\mathit{false} \Longrightarrow \phi$

Arbeitsblatt 5.3: Gültigkeit

Welche dieser Hoare-Tripel ist semantisch gültig?

```
// {x = X ∧ x ≥ 3}
x= x−3;
if (x < 0) x= 0;
x= x+3;
// {x = X}
```

```
// {b = B}
b= b−a;
x= a+b;
// {x = a + B}
```

```
// {x = X ∧ y = Y}
x= x+y;
y= x−y;
x= x−y;
// {x = Y ∧ y = X}
```

Gültigkeit und Herleitbarkeit

- **Semantische Gültigkeit:** $\models \{P\}\, c\, \{Q\}$
  - Definiert durch denotationale Semantik:
    $$\models \{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I P \land \exists \tau.\, (\sigma, \tau) \in [\![c]\!]_C \implies \tau \models^I Q$$
  - Problem: müssten Semantik von c ausrechnen
- **Syntaktische Herleitbarkeit:** $\vdash \{P\}\, c\, \{Q\}$
  - Durch **Regeln** definiert
  - Kann **hergeleitet** werden
  - Muss **korrekt** bezüglich semantischer Gültigkeit gezeigt werden
- Generelles Vorgehen in der Logik

Regeln des Floyd-Hoare-Kalküls

- Der Floyd-Hoare-Kalkül 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 < 10)[5/x] ⟺ 5 < 10}
x = 5
// {x < 10}
```

```
//{x + 1 < 10 ⟺ x < 9}
x = x+ 1
//{x < 10}
```


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.

Ein allererstes Beispiel

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

- Was berechnet dieses Programm?
- Die Werte von x und y werden vertauscht.
- Wie spezifizieren wir das?
- $\vdash \{x = X \land y = Y\}\, p\, \{y = X \land x = Y\}$

Herleitung:

$$\dfrac{\dfrac{\dfrac{\vdash \{x = X \land y = Y\}\ z = x;\ \{?z = X \land y = Y\} \qquad \vdash \{?z = X \land y = Y\}\ x = y;\ \{z = X \land x = Y\}}{\vdash \{x = X \land y = Y\}\ z = x; x = y;\ \{?z = X \land x = Y\}} \qquad \vdash \{?z = X \land x = Y\}\ y = z;\ \{y = X \land x = Y\}}{\vdash \{x = X \land y = Y\}\ z = x; x = y; y = z;\ \{y = X \land x = Y\}}$$

Vereinfachte Notation für Sequenzen

```
// {y = Y ∧ x = X}
z= x;
// {y = Y ∧ z = X}
x= y;
// {x = Y ∧ z = X}
y= z;
// {x = Y ∧ y = X}
```

- Die **gleiche** Information wie der Herleitungsbaum
- aber **kompakt** dargestellt
- Beweis erfolgt **rückwärts** (von der letzten Zuweisung ausgehend)

Arbeitsblatt 5.4: Ein erster Beweis

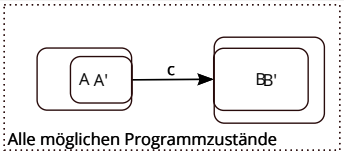
Betrachte den Rumpf des Fakultätsprogramms:

```
// (B)
p= p* c;
// (A)
c= c+ 1;
// {p = (c − 1)!}
```

- Welche Zusicherungen gelten
  1. an der Stelle (A)?
  2. an der Stelle (B)?

Regeln des Floyd-Hoare-Kalküls: Weakening

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



- $\models \{A\}\, c\, \{B\}$: 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. $P \implies Q$.
- Wir können A zu A' einschränken ($A' \subseteq A$ oder $A' \implies A$), oder B zu B' vergrößern ($B \subseteq B'$ oder $B \implies B'$), und erhalten $\models \{A'\}\, c\, \{B'\}$.

Arbeitsblatt 5.5: Ein zweiter Beweis

Wir betrachten noch einmal das Vertauschen ohne Hilfsvariable:

```
// {x = X ∧ y = Y}
// (A)
x= x+y;
// (B)
y= x−y;
// (C)
x= x−y;
// {y = X ∧ x = Y}
```

- Welche Zusicherungen gelten an den Stellen (A), (B), (C) und wie werden sie so vereinfacht, dass die Vorbedingung entsteht?
  1. (C)?
  2. (B)?
  3. (A)?

Regeln des Floyd-Hoare-Kalküls: Fallunterscheidung

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

- In der Vorbedingung des **if**-Zweiges gilt die Bedingung b, und im **else**-Zweig gilt die Negation $\neg b$.
- Beide Zweige müssen mit derselben Nachbedingung enden.

Arbeitsblatt 5.6: Dreimal ist Bremer Recht

Betrachte folgendes Programm:

```
// (F)
if (x < y) {
  // (E)
  // ...
  z = x;
  // (C)
} else {
  // (D)
  // ...
  z= y;
  // (B)
}
// (A)
```

- Was berechnet dieses Programm?
- Wie spezifizieren wir das?
- Welche Zusicherungen müssen an den Stellen (A) – (F) gelten?
- Wo müssen wir welche logische Umformungen nutzen?

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.

Wie wir Floyd-Hoare-Beweise aufschreiben

```
// {P}
// {P2[e/x]}
x= e;
// {P3}
while (x< n) {
  // {P3 ∧ x < n}
  // {P3[a/z]}
  z= a;
  // {P3}
}
// {P3 ∧ ¬(x < n)}
// {Q}
```

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

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Zusammenfassung Floyd-Hoare-Logik

- Die Logik abstrahiert über konkrete Systemzustände durch **Zusicherungen**
- Zusicherungen sind boolsche Ausdrücke, angereichert durch logische Variablen.
- **Hoare-Tripel** $\{P\}\, c\, \{Q\}$ abstrahieren die Semantik von c
  - 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 20.05.21

Floyd-Hoare-Logik II: Invarianten

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Die Floyd-Hoare-Logik bis hierher

- **Hoare-Tripel** $\{P\}\, c\, \{Q\}$ spezifizieren was c berechnet (**Korrektheit**)
  - 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).

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Invarianten finden: die Fakultät

```
1  p= 1;
2  c= 1;
3  // {I}
4  while (c <= n) {
5    // {I ∧ c ≤ n}
6    p = p * c;
7    c = c + 1;
8    // {I}
9  }
10 // {I ∧ ¬(c ≤ n)}
11 // {p = n!}
```

Invariante:

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

- Kern der Invariante: Fakultät bis $c-1$ berechnet.
- Invariante impliziert Nachbedingung $p = n! = (c-1)!$
  - $\neg(c \leq n) \Leftrightarrow c - 1 \geq n$ — was fehlt?
- Nebenbedingung für Weakening innerhalb der Schleife.
  - $c! = c * (c-1)!$ gilt nur für $c > 0$.

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 Nachbedingung $\psi[n]$ ist Invariante:

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

- Ggf. weitere Nebenbedingungen erforderlich
- Variante: $i = 0, \ldots, n-1$

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

ist syntaktischer Zucker für

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


Arbeitsblatt 6.1: Summe I

```
1 // {0 ≤ n}
2 x= 0;
3 c= 1;
4 while (c <= n) {
5   x= x+c;
6   c= c+1;
7 }
8 // {x = sum(0, n)}
```

1. Was ist die initiale Invariante?
2. Was fehlt, um aus der initialen Invariante die Nachbedingung zu schließen?
3. Was fehlt, damit der Schleifenrumpf die Invariante erhält?

Annotiert das Programm mit den Korrektheitszusicherungen!

Hierbei ist $sum(a, b)$ die Summe der Zahlen von a bis b, mit folgenden Eigenschaften:

$$a > b \Longrightarrow sum(a, b) = 0$$
$$a \leq b \Longrightarrow sum(a, b) = a + sum(a+1, b)$$
$$a \leq b \Longrightarrow sum(a, b) = sum(a, b-1) + b$$

Lösungsblatt 6.1: Summe I

```
// {0 ≤ n}
// {0 = sum(0,0) ∧ 0 < 1 ∧ 0 ≤ n}
// {0 = sum(0,1−1) ∧ 0 < 1 ∧ 1−1 ≤ n}
x= 0;
// {x = sum(0,1−1) ∧ 0 < 1 ∧ 1−1 ≤ n}
c= 1;
// {x = sum(0,c−1) ∧ 0 < c ∧ c−1 ≤ n}
while (c <= n) {
 // {x = sum(0,c−1) ∧ 0 < c ∧ c−1 ≤ n ∧ c ≤ n}
 // {x+c = sum(0,c−1)+c ∧ 0 < c ∧ c ≤ n}
 // {x+c = sum(0,c) ∧ 0 < c ∧ c ≤ n}
 x= x+c;
 // {x = sum(0,c) ∧ 0 < c ∧ c ≤ n}
 // {x = sum(0,(c+1)−1) ∧ 0 < c+1 ∧ (c+1)−1 ≤ n}
 c= c+1;
 // {x = sum(0,c−1) ∧ 0 < c ∧ c−1 ≤ n}
}
// {x = sum(0,c−1) ∧ 0 < n ∧ c−1 ≤ n ∧ ¬(c ≤ n)}
// {x = sum(0,c−1) ∧ c−1 ≤ n ∧ c−1 ≥ n}
// {x = sum(0,n)}
```

Invariante:

$$x = sum(0, c-1) \\ \wedge\ c-1 \leq n \\ \wedge\ 0 < c$$

Benötigte Eigenschaften von *sum*:

$$sum(a,a) = a$$
$$a \leq b \implies sum(a,b) = sum(a,b-1) + b$$

Variante der zählenden Schleife

```
// {0 ≤ y}
// {0 = sum(0,0) ∧ 0 ≤ y ∧ 0 ≤ 0}
x= 0;
// {x = sum(0,0) ∧ 0 ≤ y ∧ 0 ≤ 0}
c= 0;
// {x = sum(0,c) ∧ c ≤ y ∧ 0 ≤ c}
while (c < y) {
 // {x = sum(0,c) ∧ c ≤ y ∧ 0 ≤ c ∧ c < y}
 // {x+(c+1) = sum(0,c)+(c+1) ∧ c < y ∧ 0 ≤ c}
 // {x+c+1 = sum(0,c+1) ∧ c+1 ≤ y ∧ 0 ≤ c+1}
 c= c+1;
 // {x+c = sum(0,c) ∧ c ≤ y ∧ 0 ≤ c}
 x= x+c;
 // {x = sum(0,c) ∧ c ≤ y ∧ 0 ≤ c}
}
// {x = sum(0,c) ∧ c ≤ y ∧ 0 ≤ c ∧ ¬(c < y)}
// {x = sum(0,c) ∧ c ≤ y ∧ c ≥ y}
// {x = sum(0,y)}
```

- Was ist hier die Invariante?

$$x = sum(0,c) \wedge c \leq y \wedge 0 \leq c$$

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Arbeitsblatt 6.2: Summe II

```
// {n = N ∧ 0 ≤ n}
x= 0;
while (n != 0) {
 x= x+n;
 n= n−1;
}
// {x = sum(0,N)}
```

- Was ist der erste Teil der Invariante?
- Der Rest ist wie vorher?
- Annotiert das Programm mit dem Korrektheitszusicherungen.

Lösungsblatt 6.2: Summe II

```
// {n = N ∧ 0 ≤ n}
// {0 = 0 ∧ n = N}
// {0 = sum(n+1,N) ∧ n = N}
// {0 = sum(n+1,N) ∧ n ≤ N}
x= 0;
// {x = sum(n+1,N) ∧ n ≤ N}
while (n != 0) {
 // {x = sum(n+1,N) ∧ n ≤ N ∧ n ≠ 0}
 // {x+n = n+sum(n+1,N) ∧ n ≤ N}
 // {x+n = sum(n,N) ∧ n ≤ N}
 x= x+n;
 // {x = sum(n,N) ∧ n ≤ N}
 // {x = sum((n−1)+1,N) ∧ n−1 ≤ N}
 n= n−1;
 // {x = sum(n+1,N) ∧ n ≤ N}
}
// {x = sum(n+1,N) ∧ n ≤ N ∧ ¬(n ≠ 0)}
// {x = sum(n+1,N) ∧ n = 0}
// {x = sum(1,N)}
// {x = sum(0,N)}
```

Invariante:

$$sum(0,n) + x = sum(0,N)$$
$$x + sum(0,n) = sum(0,n) + sum(n+1,N)$$
$$x = sum(n+1,N)$$

wenn $n \leq N$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

```
// {n = N ∧ 0 ≤ n}
// {n! = N! ∧ n = N ∧ 0 ≤ n}
// {n! · 1 = N! ∧ n ≤ N ∧ 0 ≤ n}
p= 1;
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
while (0 < n) {
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ 0 < n}
 // {n! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n−1)! · n · p = N! ∧ n ≤ N ∧ 0 < n}
 p= n*p;
 // {(n−1)! · p = N! ∧ n ≤ N ∧ 0 < n}
 // {(n−1)! · p = N! ∧ n−1 ≤ N ∧ 0 ≤ n−1}
 n= n−1;
 // {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n}
}
// {n! · p = N! ∧ n ≤ N ∧ 0 ≤ n ∧ ¬(0 < n)}
// {n! · p = N! ∧ 0 ≤ n ∧ n ≥ 0}
// {p = N!}
```

$$n! \cdot p = N! \\ \wedge\ 0 \leq n \\ \wedge\ n \leq n$$

Arbeitsblatt 6.3: Nicht-zählende Schleife

```
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}
```

Was ist hier die Invariante?

- Hinweis: es ist ganz einfach.

Lösungsblatt 6.3: Nicht-zählende Schleife

```
// {0 ≤ a}
// {a = b·0+a ∧ 0 ≤ a}
r= a;
// {a = b·0+r ∧ 0 ≤ r}
q= 0;
// {a = b·q+r ∧ 0 ≤ r}
while (b <= r) {
   // {a = b·q+r ∧ 0 ≤ r ∧ b ≤ r}
   // {a = b·q+r ∧ b ≤ r}
   // {a = b·q+b+(r−b) ∧ 0 ≤ r−b}
   r= r−b;
   // {a = b·q+b+r ∧ 0 ≤ r}
   // {a = b·(q+1)+r ∧ 0 ≤ r}
   q= q+1;
   // {a = b·q+r ∧ 0 ≤ r}
   }
// {a = b·q+r ∧ 0 ≤ r ∧ ¬(b ≤ r)}
// {a = b·q+r ∧ 0 ≤ r ∧ r < b}
```

- Der Schlüssel ist die Beobachtung, dass der dritte Teil der Nachbedingung genau die negierte Schleifenbedingung ist.
- Der Rest ist also die Invariante.

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

```
// {0 ≤ a}
// {1 − 1 ≤ a ∧ 1 = 2 · 0 + 1 ∧ 1 = 0² + 1}
t= 1;
// {1 − t ≤ a ∧ t = 2 · 0 + 1 ∧ 1 = 0² + t}
s= 1;
// {s − t ≤ a ∧ t = 2 · 0 + 1 ∧ s = 0² + t}
i= 0;
// {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t}
while (s <= a) {
  // {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t ∧ s ≤ a}
  // {t = 2 · i + 1 ∧ s = i² + t ∧ s ≤ a}
  // {s ≤ a ∧ t + 2 = 2 · i + 3 ∧ s = i² + 2 · i + 1}
  // {s + (t + 2) − (t + 2) ≤ a ∧ t + 2 = 2 · (i + 1) + 1 ∧ s + (t + 2) = (i + 1)² + (t + 2)}
  t= t+ 2;
  // {s + t − t ≤ a ∧ t = 2 · (i + 1) + 1 ∧ s + t = (i + 1)² + t}
  s= s+ t;
  // {s − t ≤ a ∧ t = 2 · (i + 1) + 1 ∧ s = (i + 1)² + t}
  i= i+ 1;
  // {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t}
  }
// {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i² + t ∧ ¬(s ≤ a)}
// {?i² ≤ a ∧ a < (i + 1)²}
```


Zum Abschluss etwas Magie

Fast Inverse Square Root
(Quake III, John Cormack)

```
float Q_rsqrt( float number )
{
 long i;
 float x2, y;
 const float threehalfs = 1.5F;

 x2 = number* 0.5F;
 y  = number;
 i  = * (long *) &y;
 i  = 0x5f3759df - ( i >> 1 );
 y  = * (float *) &i;
 y  = y * (threehalfs - (x2 * y * y));
// y = y * (threehalfs - (x2 * y * y));
 return y;
}
```

- Verkürztes **Newton-Verfahren**
- „Evil floating-point bit-level hacking“
- Nicht zu verifizieren (nicht standard-konform)

Zusammenfassung

- Der schwierigste Teil bei Korrektheitsbeweisen mit dem Floyd-Hoare-Kalkül sind die while-Schleifen.
- Die Regel für die while-Schleife braucht eine **Invariante**, die nicht aus der Anwendung erschlossen werden kann.
- Wir können die Invariante in drei Stufen konstruieren:
  1. Algorithmischer Kern: was wird bis hier berechnet?
  2. Ist die Invariante **stark** genug, um die Nachbedingung zu implizieren?
  3. Wird die Invariante durch die Schleife erhalten? Werden noch Nebenbedingungen benötigt?
- Vereinfachender Sonderfall: **zählende** Schleifen (for-Schleifen)

Korrekte Software: Grundlagen und Methoden
Vorlesung 7 vom 25.05.21
Korrektheit des Floyd-Hoare-Kalküls

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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.

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Korrektheit des Floyd-Hoare-Kalküls

Der Floyd-Hoare-Kalkül ist korrekt.
Wenn $\vdash \{P\}\, c\, \{Q\}$, dann $\models \{P\}\, c\, \{Q\}$.

Beweis:

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

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

- Beweis durch **Regelinduktion** über der **Herleitung** von $\vdash \{P\}\, c\, \{Q\}$.
- Bsp: Zuweisung, Sequenz, Weakening, While.
  - While-Schleife erfordert Induktion über Fixpunkt-Konstruktion

Korrektheit der Zuweisung

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

Zu zeigen: $\models \{P[e/x]\}\, x = e\, \{P\}$

$\iff \forall I.\, \forall \sigma.\, \sigma \models^I P[e/x] \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![x = e]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I P$

$\iff \forall I.\, \forall \sigma.\, \sigma \models^I P[e/x] \Longrightarrow \sigma([x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)]) \models^I P$
with $(\sigma, \sigma([x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)])) \in [\![x = e]\!]_{\mathcal{C}}$

Wir benötigen folgende **Lemma** (Beweis durch strukturelle Induktion über B und a):

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B$$
$$[\![a[e/x]]\!]^I_{\mathcal{A}v}(\sigma) = [\![a]\!]^I_{\mathcal{A}v}(\sigma[x \mapsto [\![e]\!]^I_{\mathcal{A}v}(\sigma)])$$

Korrektheit der Sequenzenregel

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

Annahmen:

$(A1)$ $\models \{A\}\, c_1\, \{B\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I B$

$(A2)$ $\models \{B\}\, c_2\, \{C\} \iff \forall I.\, \forall \sigma.\, \sigma \models^I B \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I C$

Zu zeigen:

$\models \{A\}\, c_1; c_2\, \{C\} \Longleftrightarrow \forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1; c_2]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I C$

$(\sigma, \sigma') \in [\![c_1; c_2]\!]_{\mathcal{C}} \iff (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}} \circ [\![c_2]\!]_{\mathcal{C}}$
$\iff \exists \rho.\, (\sigma, \rho) \in [\![c_1]\!]_{\mathcal{C}} \wedge (\rho, \sigma') \in [\![c_2]\!]_{\mathcal{C}}$

Aus $\sigma \models^I A$ und $\exists \rho.\, (\sigma, \rho) \in [\![c_1]\!]_{\mathcal{C}}$ folgt mit (A1) $\rho \models^I B$

Aus $\rho \models^I B$ und $\exists \sigma'.\, (\rho, \sigma') \in [\![c_2]\!]_{\mathcal{C}}$ folgt mit (A2) $\sigma' \models^I C$ □

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.

Vollständigkeitsbeweis

- Zu Zeigen:

$$\forall c \in \textbf{Stmt}.\forall Q \in \textbf{Assn}.\exists \mathrm{wp}(c,Q).\forall I.\forall \sigma.\sigma \models^I \mathrm{wp}(c,Q) \Rightarrow [\![c]\!]_c\sigma \models^I Q$$

- Beweis per struktureller Induktion über c:
  - $c \equiv \{\}$: Wähle $\mathrm{wp}(\{\},Q) := Q$
  - $c \equiv X = a$: wähle $\mathrm{wp}(X = a, Q) := Q[a/x]$
  - $c \equiv c_0; c_1$: Wähle $\mathrm{wp}(c_0; c_1, Q) := \mathrm{wp}(c_0, \mathrm{wp}(c_1, Q))$
  - $c \equiv$ **if** b c_0 **else** c_1: Wähle $\mathrm{wp}(c,Q) := (b \wedge \mathrm{wp}(c_0,Q)) \vee (\neg b \wedge \mathrm{wp}(c_1,Q))$
  - $c \equiv$ **while** (b) c_0: ??

Vollständigkeitsbeweis: while

- $c \equiv$ **while** (b) c_0:
  Wie müssen eine Formel finden ($\mathrm{wp}($**while** (b) $c_0, Q)$) die alle σ charakterisiert, so dass
  $\sigma \models^I \mathrm{wp}($**while** (b) $c_0, Q)$
  $\longleftrightarrow \forall k \geq 0 \forall \sigma_0, \ldots, \sigma_k.$ $\sigma = \sigma_0$
  $\forall 0 \leq i < k.(\sigma_i \models^I b \wedge \underbrace{[\![c_0]\!]_c\sigma_i = \sigma_{i+1}}_{c_0 \text{ terminiert auf } \sigma_i \text{ in } \sigma_{i+1}}$
  $\sigma_k \models^I b \vee Q$
- Es gibt so eine Formel ausdrückbar in **Assn**, die im Wesentlichen darauf aufbaut, dass
  1. jede Sequenz an Werten, die die Programmvariablen $\overline{X}$ in b und c_0 annehmen, mittels einer Formel beschrieben werden kann (β-Prädikat)
  2. $\mathrm{wp}(c_0, \overline{X} = \overline{\sigma_{i+1}(X)})$ die Formel beschreibt, was vor c_0 gelten muss, damit hinterher die Programmvariablen $\overline{X}$ die Werte $\overline{\sigma_{i+1}(X)}$ haben
  3. $\neg \mathrm{wp}(c_0, \mathit{false})$ beschreibt was vor c_0 nicht gelten darf, damit c_0 nicht terminiert.

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.
- Vollständigkeit (relativ):

$$\models \{P\}\, c\, \{Q\} \Leftrightarrow P \Rightarrow \mathrm{wp}(c,Q)$$

- 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

- Die Floyd-Hoare-Logik ist **korrekt**, wir können nur gültige Zusicherungen herleiten.
- Beweis durch Struktur über der Ableitung: wir beweisen jede Regel als korrekt.
- Die Floyd-Hoare-Logik ist **vollständig** bis auf das Weakening.

Korrekte Software: Grundlagen und Methoden
Vorlesung 8 vom 27.5.21 / 1.6.2021 / 3.6.2021
Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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 ≅ 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[6] = {'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:

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

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

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

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

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: $\mathbf{V} = \mathbb{Z} \uplus \mathbf{C}$
- Zustände: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

- Wir betrachten nur Zugriffe vom Typ **Z** oder **C** (**elementare Typen**)
- Nützliche Abstraktion des tatsächlichen 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

$x[0].c[0] \mapsto 1$	$x[1].c[0] \mapsto 3$
$x[0].c[1] \mapsto 2$	$x[1].c[1] \mapsto 4$
$x[0].b.name[0] \mapsto$ 'n'	$x[1].b.name[0] \mapsto$ 'n'
$x[0].b.name[1] \mapsto$ 'a'	$x[1].b.name[1] \mapsto$ 'a'
$x[0].b.name[2] \mapsto$ 'm'	$x[1].b.name[2] \mapsto$ 'm'
$x[0].b.name[3] \mapsto$ 'e'	$x[1].b.name[3] \mapsto$ 'e'
$x[0].b.name[4] \mapsto$ '1'	$x[1].b.name[4] \mapsto$ '2'
$x[0].b.name[5] \mapsto$ '\0'	$x[1].b.name[5] \mapsto$ '\0'

Operationale Semantik: L-Werte

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

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

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

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

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

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

Operationale Semantik: Ausdrücke

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

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

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

- Auswertung für **C**:

$$\frac{}{\langle c :: \mathbf{C}, \sigma\rangle \to_{Aexp} \mathrm{Ord(c)}}$$

wobei Ord : **C** $\to$ **Z** eine bijektive Funktion ist, die jedem Character eine Ordinalzahl zuweist (zum Beispiel ASCII Wert).

Operationale Semantik: Zuweisungen

- Zuweisungen sind nur definiert für elementare Typen:

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

In allen anderen Fällen ($\bot$, keine/unterschiedliche elementare Typen)

$$\langle m = e, \sigma\rangle \to_{Stmt} \bot$$

- Die restlichen Regeln bleiben

Denotationale Semantik

- Denotation für **Lexp**:

$$[\![-]\!]_{\mathcal{L}} : \textbf{Lexp} \to (\Sigma \rightharpoonup \textbf{Loc})$$

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

- Denotation für **Characters** $c \in \mathbf{C}$:

$$[\![c]\!]_{\mathcal{A}} = \{(\sigma, \mathrm{Ord}(c)) | \sigma \in \Sigma\}$$

- Denotation für **Zuweisungen**:

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

Floyd-Hoare-Kalkül

- Die Regeln des Floyd-Hoare-Kalküls berechnen geltende Zusicherungen
- Nötige Änderung: Substitution in Zusicherungen wird zur Ersetzung von **Lexp**-Ausdrücken

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

- Jetzt werden **Lexp** ersetzt, keine **Idt**

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

Anmerkung: l und e enthalten **keine** logischen Variablen.

- Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
- Problem: Feldzugriffe

Von der Substitution zur Ersetzung

$$\textbf{Assn}\ b ::= \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid p(e_1, \ldots, e_n) \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid b_1 \longrightarrow b_2 \mid \forall v.\, b \mid \exists v.\, b \qquad \textbf{(Literale)}$$

$$\mathit{true}[e/l] := \mathit{true}$$
$$\mathit{false}[e/l] := \mathit{false}$$
$$(a_1 = a_2)[e/l] := (a_1[e/l] = a_2[e/l])$$
$$(b_1 \wedge b_2)[e/l] := (b_1[t/x] \wedge b_2[e/l])$$
$$(\forall v.\, b)[e/l] := \forall v.\, (b[e/l])$$

$$n[e/l] := n \qquad (n \in \mathbf{Z} \uplus \mathbf{C})$$

$$l'[e/l] := l'[e/l] \begin{cases} e & \text{falls } l = l' \\ l' & \text{sonst} \end{cases} \qquad (l' \in \textbf{Lexp})$$

$$(a_1 + a_2)[e/l] := a_1[e/l] + a_2[e/l]$$
$$\ldots$$

Beispiel Problemsituationen:
$(c[i].x[0])[5/c[1].x[0]] = ?$
$(c[1].x[0])[8/c[1].x[j]] = ?$
$(c[i].x[0])[8/c[1].x[j]] = ?$

Beispiel

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

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

Beispiel: Problem

```
int a[3];
int i;
// {0 ≤ i < 2}
// ↯
// {i ≠ 1}
a[0] = 3;
// {i ≠ 1}
// {i ≠ 1 ∧ 7 = 7)}
a[1] = 7;
// {i ≠ 1 ∧ a[1] = 7}
a[2] = 9;
// {i ≠ 1 ∧ a[1] = 7}
// {(i = 1 ∧ 7 = −1) ∨ (i ≠ 1 ∧ a[1] = 7}{a[1] = 7}
a[i] = −1;
// {a[1] = 7}
```

$$\overline{\vdash \{P[e/l]\}\ l = e\ \{P\}}$$

Arbeitsblatt 8.1: Jetzt seid ihr dran

Annotiert die beiden folgenden Programme:

```
int a[2];
int b[2];
// {0 ≤ n ∧ 0 ≤ m ∧ n ≤ m}
a[0] = m;
//
b[0] = a[0] − n;
//
b[1] = a[0] + n
//
a[1] = b[0] ∗ b[1];
// {a[1] = m² − n²}
```

```
int a[3];
int i;
// {0 ≤ n}
i = 2;
//
a[i] = 3;
//
a[0] = n;
//
a[2] = a[2] ∗ a[0];
// {a[2] = 3 ∗ n}
```


Erstes Beispiel: Ein Feld initialisieren

```
1  // {0 ≤ n}
2  // {∀j.0 ≤ j < 0 ⟶ a[j] = j ∧ 0 ≤ n}
3  i= 0;
4  // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
5  while (i< n) {
6    // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
7    // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i + 1 ≤ n}
8    // {∀j.0 ≤ j < i ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
9    //      ∧((i = i ∧ i = i) ∨ (i ≠ i ∧ a[i] = i)) ∧ i + 1 ≤ n}
10   // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
11   //      ∧ i + 1 ≤ n}
12   a[i]= i;
13   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
14   i= i+1;
15   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
16   }
17 // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
18 // {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

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

Beispiel: Suche nach dem maximalen Element

```
1  // {0 < n}
2  // {(∀j.0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 ∧ 0 ≤ 0 < n}
3  i= 0;
4  // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ∧ 0 ≤ 0 < n}
5  r= 0;
6  // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
7  while (i< n) {
8    // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
9    // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
10   if (a[r] < a[i]) {
11     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
12     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ a[i] ≤ a[i] ∧ 0 ≤ i+1 ≤ n ∧ 0 ≤ i < n}
13     // {(∀j.0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
14     r= i;
15     // {(∀j.0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
16     }
17   else {
18     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
19     // {(∀j.0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
20     }
21   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
22   i= i+1;
23   // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
24 }
25 // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
26 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```


Vorgehensweise

```
1 // {I}
2 while (b) {
3   // {I ∧ b}
4   c
5   // {I}
6 }
7 // {I ∧ ¬b}
8 // {Φ}
```

1. Finde/rate/formuliere Invariante I
2. Beweise $(I \wedge \neg b) \longrightarrow \Phi$
3. Zeige mittels Floyd-Hoare-Regeln, dass Invariante durch Schleifenrumpf c erhalten bleibt
4. Setze Beweis mit Floyd-Hoare Regeln vor der Schleife fort

Längeres Beispiel: Suche nach einem Null-Element

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

Merkt euch folgende korrekten logischen Umformungen:

- $(F \wedge H) \vee (G \wedge H)$ ist äquivalent zu $(F \vee G) \wedge H$
- $\neg F \vee G$ ist äquivalent zu $F \longrightarrow G$

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  // {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
3  i= 0;
4  // {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10   if (a[i] == 0) {
11     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
12     // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
13     // {(i = −1 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0) ∨ (0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0)}
14     // {(i = −1 ∨ (0 ≤ i < i + 1 ∧ a[i] = 0)) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
15     // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
16     r= i;
17     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18     }
19   else {
20     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```


Benutzte Logische Umformungen

- Zeilen 11-12:
  - $[D \wedge C] \Rightarrow [C]$ und
  - Erweiterung von C auf $B(i) \wedge C$, weil $C \vdash B(i)$ gilt.
- $[\varphi] \Rightarrow [\psi \vee \varphi]$ in der Form

$$[(B(i) \wedge C)] \Rightarrow [(\neg A(i) \wedge C) \vee (B(i) \wedge C))]$$

- DeMorgan:

$$[(\neg A(i) \wedge C) \vee (B(i) \wedge C))] \Rightarrow [(\neg A(i) \vee B(i)) \wedge C]$$

- Klassische Implikation:

$$[\neg U \vee V] \Leftrightarrow [U \Rightarrow V]$$

Längeres Beispiel: Suche nach einem Null-Element

```
10  /** { 0 ≤ n } */
11  /** { 0 ≤ 0 ≤ n } */
12  i= 0;
13  /** { 0 ≤ i ≤ n } */
14  /** { (−1 ≠ −1 ⟶ 0 ≤ −1< i ∧ a[−1] == 0) ∧ 0 ≤ i ≤ n } */
15  r= −1;
16  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
17  while (i< n) {
18    /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i < n } */
19    /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
20    if (a[i] == 0) {
21      /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
22      /** { 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
23      /** { (i ≠ −1 ⟶ 0 ≤ i< i+1 ∧ a[i] == 0) ∧ 0≤ i+1≤ n } */
24      r= i;
25      /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0≤ i+1≤ n } */
26      }
27    else {
28       /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] ≠ 0 } */
29      /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
30      }
31    /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
32    i= i+1;
33    /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
34    }
35  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n) } */
36  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n } */
37  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ i == n } */
38  /** { r ≠ −1 ⟶ 0 ≤ r< n ∧ a[r] == 0 } */
```


Allgemeine Regel bei Ersetzungen?

Wie sieht nun die allgemeine Regel aus für

$$\overline{\vdash \{P[e/l]\}\ l = e\,\{P\}}$$

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

```
int a[3];
int i;
i = 8;
a[0] = 3;
a[1] = i;
a[2] = 9;
a[a[2]−a[1]] = −1;
// {a[1] = −1}
```


Allgemeine Regel bei Ersetzungen (Nur Arrays)

Wie sieht nun die allgemeine Regel aus für

$$\overline{\vdash \{P[e/l]\}\ l = e\,\{P\}}$$

1. Wenn l Programmvariable ist, wie gewohnt substituieren
2. Wenn $l = a[s]$:
   1. Vorkommen der Form $a[t]$ in **Literalen** $L(a[t])$ und s und t beide in $\mathbb{Z}$ oder **ldt**,
      - dann ersetze $L(a[t])$ durch $L(e)$, falls $s = t$
   2. Vorkommen der Form $a[t]$ in **Literalen** $L(a[t])$ und s oder t sind nicht aus $\mathbb{Z}$,
      - dann ersetze $L(a[t])$ durch $(t = s \wedge L(e)) \vee (t \neq s \wedge L(a[t]))$

2.2 könnt ihr immer machen, 2.1 ist eine Optimierung

- Das ist jetzt immer noch nicht die ganz allgemeine Form, aber für unsere Belange reicht das.

Arbeitsblatt 8.2: Längeres Beispiel: Suche nach dem ersten Null-Element

Ausgehend von dem vorherigem Beispiel, annotiert folgendes

```
1  // {0 ≤ n}
2  i= 0;
3  r= −1;
4  /* −− beforeloop −− */
5  while (i< n) {
6    /* −− startloop −− */
7    if (r== −1 && a[i] == 0) {
8      r= i;
9      }
10   else {
11   }
12   /* −− afterif −− */
13   i= i+1;
14   /* −− endloop −− */
15 }
16 /* −− afterloop −− */
17 /** {(r ≠ −1 ⟶ (0 ≤ r< n ∧ a[r] == 0 ∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0)))
18      ∧ (r==−1 ⟶ (∀ int 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 wird zur Ersetzung
  - Anwendung der Zuweisungsregel führt i.A. zu großen Formeln

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül I
- Der Floyd-Hoare-Kalkül II: Invarianten
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Modellierung
- Spezifikation von Funktionen
- Referenzen und Speichermodelle
- Ausblick und Rückblick

Korrekte Software: Grundlagen und Methoden
Vorlesung 9 vom 08.06.21
Verifikationsbedingungen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;
// {X = x ∧ Y = z}
y = x;
// {X = y ∧ Y = z}
x = z;
// {X = y ∧ Y = x}
```

- Wir sehen:
  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Verifikation kann **berechnet** werden.
- Geht das immer?

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?

Rückwärtsanwendung der Regeln

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

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

- Was ist mit den anderen Regeln?

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

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

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

Arbeitsblatt 9.1: Eine Kleine Fallunterscheidung

Berechnet die Vorbedingung für folgendes Programm:

```
// ?
if (y == 7) {
  //
  x= 3;
  //
  }
else {
  y= 0;
  x= 10;
  //
  }
// x+ y == 10
```


Rückwärtsanwendung: if

```
// ?
if (b) {
  // {P1}
  ...
  // {Q}
  }
else {
  // {P2}
  ...
  // {Q}
  }
// {Q}
```

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

Regel in der Form nicht geeignet. Besser:

$$A \stackrel{def}{=} (P_1 \wedge b) \vee (P_2 \wedge \neg b)$$

$$(P_1 \wedge b) \vee (P_2 \wedge \neg b) \wedge b \Longleftrightarrow (P_1 \wedge b) \vee \mathit{false} \Longleftrightarrow P_1 \wedge b$$

$$(P_1 \wedge b) \vee (P_2 \wedge \neg b) \wedge \neg b \Longleftrightarrow \mathit{false} \vee (P_2 \wedge \neg b) \Longleftrightarrow P_2 \wedge \neg b$$

Kombiniert mit Weakening ergibt neue Regel:

$$\frac{P_1 \wedge b \Longrightarrow P_1 \quad \vdash \{P_1\}\, c_0\, \{B\} \qquad P_2 \wedge \neg b \Longrightarrow P_2 \quad \vdash \{P_2\}\, c_0\, \{B\}}{\vdash \{(P_1 \wedge b) \vee (P_2 \wedge \neg b)\}\ \mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1\, \{B\}}$$

Neue Regeln

- Wir können aus dem Hoare-Kalkül **neue Regeln** ableiten, in dem wir
  1. Existierende Regeln **instantiieren**, oder
  2. existierende Regeln **verknüpfen**.
- Wir benötigen das hier, um die Regeln des Hoare-Kalkül in eine Form zu bringen, welche die Rückwärtsrechnung ermöglicht.

Das Hinzufügen abgeleiteter Regeln ist eine **konservative Erweiterung** — es lassen sich damit nicht mehr oder weniger Hoare-Tripel $\vdash \{P\}\, c\, \{Q\}$ herleiten.

Regeln für die Rückwärtsrechnung

1. **Nachbedingung** der **Konklusion** ist von der Form $\{Q\}$ (**offene** Meta-Variable)
2. Alle **Vorbedingungen** der **Prämissen** ist von der Form $\{P_i\}$ (**unterschiedliche** P_i)
3. Alle Variablen in den Vorbedingungen der Konklusion, den Weakenings und Nachbedingungen der Prämisse sind **determiniert**[1].

Welche Regeln passen noch nicht? **while**-Regel passt noch nicht ...

[1] **Entweder** in der Nachbedingung oder dem Programmausdruck der Konklusion, **oder** den Vorbedingungen der Prämisse enthalten.

Regeln für die Rückwärtsrechnung: while

- **while**-Regel (1) wird mit Weakening zu (2):

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

$$\frac{I \wedge b \Longrightarrow R \qquad \vdash \{R\}\, c\, \{I\} \qquad I \wedge \neg b \Longrightarrow Q}{\vdash \{I\}\ \textbf{while}\ (b)\ c\ \{Q\}} \quad (2)$$

- Implikationen $I \wedge b \Longrightarrow R$, $I \wedge \neg b \Longrightarrow Q$ werden zu **Beweisverpflichtungen**
- Bedingung I (**Invariante**) muss **vorgegeben** werden.

Übersicht: Regeln für den Hoare-Kalkül Rückwärts

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}} \qquad \frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{\vdash \{A_0\}\, c_0\, \{B\} \quad \vdash \{A_1\}\, c_1\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

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

Annotierte Programme

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

$$\bigwedge \mathrm{wvc}(c, Q) \Longrightarrow \vdash \{\mathrm{awp}(c, Q)\}\, c\, \{Q\}$$

Überblick: Approximative schwächste Vorbedingung

$$\begin{aligned}
\mathrm{awp}(\{\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(I = e, P) &\stackrel{def}{=} P[e/I] \qquad (\textbf{Genauer}: \text{Folie 24 letzte VL}) \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\mathrm{wvc}(\{\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(I = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\} \\
\mathrm{wvc}(\{P\}\, c\, \{Q\}) &\stackrel{def}{=} \{P \longrightarrow \mathrm{awp}(c, Q)\} \cup \mathrm{wvc}(c, Q)
\end{aligned}$$

Berechnung der Verifikationsbedingungen

Programmkorrektheit

- Gegeben: Annotiertes Programm c mit Vorbedingung P und Nachbedingung Q.
- Gesucht: $\mathrm{wvc}(\{P\}\, c\, \{Q\})$

1. Rekursiv von der Nachbedigung ausgehend berechnen wir für jede Zeile des Programmes die gültige approximative Vorbedingung $\mathrm{awp}(c, -)$.
2. Dabei notieren wir alle auftretenden Verifikationsbedingungen $\mathrm{wvc}(c, -)$
3. Dabei werden **keine** Vereinfachungen vorgenommen.

Beispiel: das Fakultätsprogramm

- 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 $\mathrm{wvc}(\{0 \le n\}\, F\, \{p = n!\})$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  // {1 = (1 − 1)! ∧ 1 − 1 ≤ n}
3  p= 1;
4  // {p = (1 − 1)! ∧ 1 − 1 ≤ n}
5  c= 1;
6  // {p = (c − 1)! ∧ c − 1 ≤ n}
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n */ {
9    // {p · c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
10   p = p * c;
11   // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
12   c = c + 1;
13   // {p = (c − 1)! ∧ c − 1 ≤ n}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1	$p = (c-1)! \wedge c - 1 \le n \wedge \neg(c \le n)$ $\longrightarrow p = n!$
2	$p = (c-1)! \wedge c - 1 \le n \wedge c \le n$ $\longrightarrow p \cdot c = ((c+1)-1)! \wedge$ $(c+1) - 1 \le n$
3	$0 \le n \longrightarrow 1 = (1-1)! \wedge (1-1) \le n$

Vereinfachung von Verifikationsbedingungen

Wir nehmen folgende **strukturelle Vereinfachungen** vor:

1. Konjunktionen in der Konklusion werden zu einzelnen Verifikationsbedingungen
   - Bsp: $A_1 \wedge A_2 \wedge A_3 \longrightarrow P \wedge Q \rightsquigarrow A_1 \wedge A_2 \wedge A_3 \longrightarrow P, A_1 \wedge A_2 \wedge A_3 \longrightarrow Q$
2. Auswertung konstanter arithmetischer Ausdrücke, einfache arithmetische Gesetze
   - Bsp. $(x+1)-1 \rightsquigarrow x$, $1-1 \rightsquigarrow 0$
3. Normalisierung der Relationen (zu $<, \leq, =, \neq$) und Vereinfachung
   - Bsp: $\neg(x \leq y) \rightsquigarrow x > y \rightsquigarrow y < x$, $x \leq x \rightsquigarrow \mathit{true}$, $4 \leq 5 \rightsquigarrow \mathit{true}$
4. Alle Bedingungen mit einer Prämisse *false* oder einer Konklusion *true* sind trivial erfüllt.

Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge n < c \longrightarrow p = n!$
- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1) - 1 \leq n$
  $\rightsquigarrow$ $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$
- 3: $0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n$
  $\rightsquigarrow$ $0 \leq n \longrightarrow 1 = 0!$
  $0 \leq n \longrightarrow 0 \leq n \rightsquigarrow \mathit{true}$

Es bleibt zu zeigen:

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge n < c \longrightarrow p = n!$
  Aus $n > c$ folgt $n \geq c - 1$, also $c - 1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.
- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge c \leq n \longrightarrow p \cdot c = c!$
  Aus $p = (c-1)!$ folgt $p \cdot c = c \cdot (c-1)!$, und mit $c \cdot (c-1)! = c!$ folgt die Behauptung.
- 3: $1 = 0!$ folgt direkt aus der Definition der Fakultät.

Arbeitsblatt 9.2: Da summt was...

```
1 // {0 ≤ n ∧ n = N}
2 p= 0;
3 while (n>0) //** inv p= sum(n+1,N); */
4 { p = p + n;
5   n = n - 1;
6   }
7 // {p = sum(1, N)}
```

1. Berechnet zuerst die **unvereinfachten** VCs (für sind die AWPs nötig)
2. Danach vereinfacht die VCs **schematisch** wie oben beschrieben.
3. Welche VCs sind beweisbar?

Dabei gilt: $\mathit{sum}(i,j) = \begin{cases} 0 & i > j \\ i + \mathit{sum}(i+1, j) & i \leq j \end{cases}$

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 < i} */
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}
```


Maximales Element (Schleifenrumpf)

```
1  while (i!= n)
2    /** inv {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
3  {
4    // {(a[r] < a[i] ∧ φ(i+1, i)) ∨ (¬(a[r] < a[i]) ∧ φ(i+1, r))}
5    if (a[r] < a[i]) {
6      // {φ(i+1, i)}
7      r= i;
8      // {φ(i+1, r)}
9      }
10   else {
11     // {φ(i+1, r)}
12   }
13   // {φ(i+1, r)}
14   i= i+1;
15   // {φ(i, r)}
16   }
17 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

VC:

1. $\varphi(i,r) \wedge \neg(i \neq n) \longrightarrow \varphi(n,r)$
2. $\varphi(i,r) \wedge i \neq n \longrightarrow$
   $(a[r] < a[i] \wedge \varphi(i+1, i))$
   $\vee$
   $(\neg(a[r] < a[i]) \wedge \varphi(i+1, r))$

Maximales Element (Initialisierung)

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

VC:

1. $\varphi(i,r) \wedge \neg(i \neq n) \longrightarrow \varphi(n,r)$
2. $\varphi(i,r) \wedge i \neq n \longrightarrow$
   $(a[r] < a[i] \wedge \varphi(i+1, i))$
   $\vee$
   $(\neg(a[r] < a[i]) \wedge \varphi(i+1, r))$
3. $0 \leq n \longrightarrow \varphi(0,0)$

Maximales Element (Verifikationsbedingungen)

Unvereinfacht:

1 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge \neg(i \neq n) \longrightarrow$
$(\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq n$

2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge i \neq n \longrightarrow$
$((a[r] < a[i] \wedge (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i \leq i+1) \vee$
$(\neg(a[r] < a[i]) \wedge (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i+1))$

3 $0 \leq n \longrightarrow (\forall j.0 \leq j < 0 \longrightarrow a[j] \leq a[0]) \wedge 0 \leq 0 \leq 0$

1.1 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge i = n \longrightarrow \forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]$

1.2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge i = n \longrightarrow 0 \leq r \leq n$

2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i \wedge i \neq n \longrightarrow$
$((a[r] < a[i] \wedge (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i \leq i+1) \vee$
$(\neg(a[i] \leq a[r]) \wedge (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r \leq i+1))$

3.1 $0 \leq n \longrightarrow \forall j.0 \leq j < 0 \longrightarrow a[j] \leq a[0]$

3.2 $0 \leq n \longrightarrow 0 \leq 0 \leq 0$

Vereinfacht:

- Sehr **lange** Verifikationsbedingungen (u.a. wegen Fallunterscheidung)
  - Insbesondere schwer zu **vereinfachen**
- Wie können wir das **beheben**?

Explizite Vorbedingungen

Lange Vorbedingung:

```
// {(P1 ∧ b) ∨ (P2 ∧ ¬b)}
if (b) {
  // {P1}
  ...
  // {Q}
  } else {
  // {P2}
  ...
  // {Q}
  }
```

Kurze Vorbedingung:

```
// {A}
if (b) {
  // {A ∧ b}
  ...
  // {Q}
  } else {
  // {A ∧ ¬b}
  ...
  // {Q}
  }
```

Dazu VCs:

$A \wedge b \longrightarrow P_1$
$A \wedge \neg b \longrightarrow P_2$

Spracherweiterung: Explizite Spezifikationen

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

  Assn $a ::= \ldots$ — Zusicherungen
  Stmt $c ::= l = e \mid c_1; c_2 \mid \{\} \mid$ **if** (b) c_1 **else** c_2
  $\mid$ **while** (b) /** **inv** a */ c
  $\mid$ /** $\{a\}$ */

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

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

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

$$\begin{aligned}
\text{awp}(\{\}, P) &\stackrel{def}{=} P \\
\text{awp}(l = e, P) &\stackrel{def}{=} P[e/l] \qquad \text{(Genauer: Folie 24 letzte VL)} \\
\text{awp}(c_1; c_2, P) &\stackrel{def}{=} \text{awp}(c_1, \text{awp}(c_2, P)) \\
\text{awp}(\textbf{if } (b) \ c_0 \ \textbf{else} \ c_1, P) &\stackrel{def}{=} Q \quad \text{wenn } \text{awp}(c_0, P) = b \wedge Q, \text{awp}(c_1, P) = \neg b \wedge Q \\
\text{awp}(\textbf{if } (b) \ c_0 \ \textbf{else} \ c_1, P) &\stackrel{def}{=} (b \wedge \text{awp}(c_0, P)) \vee (\neg b \wedge \text{awp}(c_1, P)) \\
\text{awp}(\text{/** } \{q\} \text{ */}, P) &\stackrel{def}{=} q \\
\text{awp}(\textbf{while } (b) \text{ /** } \textbf{inv } i \text{ */ } c, P) &\stackrel{def}{=} i \\
\text{wvc}(\{\}, P) &\stackrel{def}{=} \emptyset \\
\text{wvc}(l = e, P) &\stackrel{def}{=} \emptyset \\
\text{wvc}(c_1; c_2, P) &\stackrel{def}{=} \text{wvc}(c_1, \text{awp}(c_2, P)) \cup \text{wvc}(c_2, P) \\
\text{wvc}(\textbf{if } (b) \ c_0 \ \textbf{else} \ c_1, P) &\stackrel{def}{=} \text{wvc}(c_0, P) \cup \text{wvc}(c_1, P) \\
\text{wvc}(\text{/** } \{q\} \text{ */}, P) &\stackrel{def}{=} \{q \longrightarrow P\} \\
\text{wvc}(\textbf{while } (b) \text{ /** } \textbf{inv } i \text{ */ } c, P) &\stackrel{def}{=} \text{wvc}(c, i) \cup \{i \wedge b \longrightarrow \text{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}$$

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  // {(∀j. 0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 < 0}
3  i= 0;
4  r= 0;
5  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i}
6  while (i!= n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i} */
7  {
8    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i}
9    if (a[r] < a[i]) {
10     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ a[r] < a[i]}
11     // {(∀j. 0 ≤ j < i+1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < i+1}
12     r= i;
13     // {(∀j. 0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i+1}
14     }
15   else {
16     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i+1}
18     }
19   // {(∀j. 0 ≤ j < i+1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i+1}
20   i= i+1;
21   }
22 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```


Maximales Element mit Zusicherung: Beweisverpflichtungen

Unvereinfacht:

$$\begin{aligned}
&(1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge \neg(i \ne n) \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \\
&(2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge \neg(a[r] < a[i]) \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i+1 \\
&(3) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[r] < a[i] \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[i]) \wedge 0 \le r < i+1 \\
&(4) \quad 0 < n \longrightarrow (\forall j.\, 0 \le j < 0 \longrightarrow a[j] \le a[0]) \wedge 0 \le 0 < n
\end{aligned}$$

Maximales Element mit Zusicherung: Beweisverpflichtungen

Vereinfacht:

$$\begin{aligned}
&(1.1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge i = n \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \\
&(1.2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge i = n \longrightarrow 0 \le r < n \\
&(2.1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[i] \le a[r] \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[r]) \\
&(2.2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[i] \le a[r] \longrightarrow 0 \le r < i+1 \\
&(3.1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[r] < a[i] \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[i]) \\
&(3.2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[r] < a[i] \longrightarrow 0 \le r < i+1 \\
&(4.1) \quad 0 < n \longrightarrow (\forall j.\, 0 \le j < 0 \longrightarrow a[j] \le a[0]) \\
&(4.2) \quad 0 < n \longrightarrow 0 \le 0 < n
\end{aligned}$$

Beweismethoden

- Um $P_1 \wedge \ldots \wedge P_n \longrightarrow Q$ zu zeigen, nehmen wir $P_1, \ldots, P_n$ an und zeigen Q.
- Dabei nutzen wir **u.a.** folgende Regeln:

Wenn P, dann P (Trivial)
Wenn P und $x = t$, dann $P[t/x]$ (Subst)
$x \le x$ (Reflexität)
Wenn $x \le y$ und $y \le z$, dann $x \le z$ (Transitivität)
Wenn $x \le y$ und $y \le x$, dann $x = y$ (Antisymmetrie)
Wenn $x < y$, dann $x \le y + 1$ oder $x + 1 \le y$ (Inc)
Wenn $\forall x.\, P$, dann $P[t/x]$ (Instantiierung)
Wenn *false*, dann P (Ex falso)
Wenn $a \le b$ und $x \le y$, dann $a + x \le b + y$ und Variation mit $x = 0$ etc.
Umformungen mit $(0, +)$ und $(1, \cdot)$
Domänenspezifische Regeln

Arbeitsblatt 9.3: Beweisverpflichtungen Beweisen

Betrachtet die vereinfachten Verifikationsbedingungen:

$$\begin{aligned}
&(1.1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge i = n \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \\
&(1.2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge i = n \longrightarrow 0 \le r < n \\
&(2.1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[i] \le a[r] \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[r]) \\
&(2.2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[i] \le a[r] \longrightarrow 0 \le r < i+1 \\
&(3.1) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[r] < a[i] \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[i]) \\
&(3.2) \quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i \wedge a[r] < a[i] \longrightarrow 0 \le r < i+1 \\
&(4.1) \quad 0 < n \longrightarrow (\forall j.\, 0 \le j < 0 \longrightarrow a[j] \le a[0]) \\
&(4.2) \quad 0 < n \longrightarrow 0 \le 0 < n
\end{aligned}$$

Wie würdet ihr sie beweisen? Was für Methoden verwendet ihr?

Arbeitsblatt 9.4: Kopien

Dieses Programm kopiert ein Array:

```
i= 0;
while (i< m)
  /** inv ??? */ {
  b[m-1-i]= a[i];
  i= i+1;
  }
```

1. Spezifiziert die Funktionalität.
2. Findet die Invariante.
3. Berechnet die Verifikationsbedingungen (VCs) und schwächste Vorbedingung.
4. Beweist die VCs.

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.
- Warum eigentlich immer **rückwärts**?
  Jetzt gleich. . .

Korrekte Software: Grundlagen und Methoden
Vorlesung 10 vom 15.06.21
Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;
//{X = x ∧ Y = z}
y = x;
// {X = y ∧ Y = z}
x = z;
//{X = y ∧ 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)

$(a[3] == 7)[5/a[i]] \;=\; ((i == 3\ ?\ 5 : a[i]) == 7)$

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

I. Der Floyd-Hoare-Kalkül Vorwärts

Regelanwendung rückwärts

- Um Regel **rückwärts** anwenden zu können:
  1. **Nachbedingung** der Konklusion muss offene Variable sein
  2. Alle **Vorbedingungen** der Prämissen müssen disjunkte, offen Variablen sein.
  3. Gegenbeispiele: while-Regel, if-Regel
- Um Regeln **vorwärts** anwenden zu können:
  1. **Vorbedingung** der Konklusion muss offene Variable sein
  2. Alle **Nachbedingungen** der Prämissen müssen disjunkte, offene Variablen sein.
  3. Gegenbeispiele: ...

Vorwärtsanwendung der Regeln

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

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

- Andere Regeln passen bis auf if-Regel (keine **disjunkten** Variablen)

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

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

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

Arbeitsblatt 10.1: If-Regel Vorwärts

- Wie kann die If-Regel vorwärts aussehen?

Zuweisungsregel Vorwärts

- Alternative Zuweisungsregel (nach Floyd):

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

  - $FV(P)$ sind die **freien** Variablen in P.
- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden
- Ist keine abgeleitete Regel — muss als korrekt **bewiesen** werden

Arbeitsblatt 10.2: Das Leben mit dem Quantor

- Was bedeutet $\exists V.P$?
  - Die Formel ist wahr, wenn es **irgendeinen** Wert t für V gibt, so dass $P[t/V]$ wahr ist.
- Was bedeutet $\forall V.P$?
  - Die Formel ist wahr, wenn für **alle** Werte t für V $P[t/V]$ wahr ist.
- Sind folgende Formeln wahr (für $x, y \in \mathbb{Z}$)? (Finde Gegenbeispiele oder Zeugen)

$\exists x.\, x < 7$	$\exists x.\, x < 3 \wedge x > 7$	$\exists x.\, x < 7 \vee x < 3$
$\exists y \exists x.\, x + 3 = y$	$\forall x \exists y.\, x \cdot y = 3$	$\exists x \forall y.\, x \cdot y = y$

Vorwärtsverkettung

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

```
// {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:

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

Und jetzt...?

Regeln der Vorwärtsverkettung

Eigenschaften des Existenzquantors:

$$\begin{aligned}
P(V) \wedge V = t &\Longrightarrow P[t/V] \wedge V = t && V \notin FV(t) && (1)\\
\exists V.\, P(V) \wedge V = t &\Longrightarrow P[t/V] && V \notin FV(t) && (2)\\
(\exists V.\, P) \wedge Q &\Longleftrightarrow \exists V.\, P \wedge Q && V \notin FV(Q) && (3)\\
\exists V.\, P &\Longrightarrow P && V \notin FV(P) && (4)
\end{aligned}$$

Damit gelten folgende Regeln bei 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 (4)

Vorwärtsverkettung

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

```
// {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:

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

Vorwärtsverkettung bei der Arbeit

Vereinfachung erst am Ende:

```
// {i·i ≤ a ∧ t = 2·i+1 ∧ s = i·i+t ∧ s ≤ a}
t= t+2;
// {∃T.(i·i ≤ a ∧ t = 2·i+1 ∧ s = i·i+t ∧ s ≤ a)[T/t] ∧ t = (t+2)[T/t]}
// {∃T.i·i ≤ a ∧ T = 2·i+1 ∧ s = i·i+T ∧ s ≤ a ∧ t = T+2}
s= s+t;
// {∃S.(∃T.i·i ≤ a ∧ T = 2·i+1 ∧ s = i·i+T ∧ s ≤ a ∧ t = T+2)[S/s] ∧ s = (s+t)[S/s]}
// {∃S.∃T.i·i ≤ a ∧ T = 2·i+1 ∧ S = i·i+T ∧ S ≤ a ∧ t = T+2 ∧ s = S+t}
i= i+1;
// {∃I.(∃S.∃T.i·i ≤ a ∧ T = 2·i+1 ∧ S = i·i+T ∧ S ≤ a ∧ t = T+2 ∧ s = S+t)[I/i] ∧ i = (i+1)[I/i]}
// {∃I.∃S.∃T.I·I ≤ a ∧ T = 2·I+1 ∧ S = I·I+T ∧ S ≤ a ∧ t = T+2 ∧ s = S+t ∧ i = I+1}
// {∃I.∃S.∃T.I·I ≤ a ∧ S = I·I+T ∧ S ≤ a ∧ t = T+2 ∧ s = S+t ∧ i = I+1 ∧ T = 2·I+1}
// {∃I.∃S.I·I ≤ a ∧ S = I·I+2·I+1 ∧ S ≤ a ∧ t = 2·I+1+2 ∧ s = S+t ∧ i = I+1}
// {∃I.∃S.I·I ≤ a ∧ S ≤ a ∧ t = 2·I+1+2 ∧ s = S+t ∧ i = I+1 ∧ S = (I+1)·(I+1)}
// {∃I.I·I ≤ a ∧ (I+1)·(I+1) ≤ a ∧ t = 2·I+1+2 ∧ s = (I+1)·(I+1)+t ∧ i = I+1}
// {∃I.I·I ≤ a ∧ (I+1)·(I+1) ≤ a ∧ t = 2·I+3 ∧ s = (I+1)·(I+1)+t ∧ I = i−1}
// {(i−1)·(i−1) ≤ a ∧ ((i−1)+1)·((i−1)+1) ≤ a ∧ t = 2·(i−1)+3 ∧ s = ((i−1)+1)·((i−1)+1)+t}
// {i·i ≤ a ∧ t = 2·i+1 ∧ s = i·i+t}
```


Vorwärtsverkettung bei der Arbeit II

Mit Vereinfachung on-the-fly:

```
// {i·i ≤ a ∧ t = 2·i+1 ∧ s = i·i+t ∧ s ≤ a}
t= t+2;
// {∃T.(i·i ≤ a ∧ t = 2·i+1 ∧ s = i·i+t ∧ s ≤ a)[T/t] ∧ t = (t+2)[T/t]}
// {∃T.i·i ≤ a ∧ s = i·i+T ∧ s ≤ a ∧ t = T+2 ∧ T = 2·i+1}
// {i·i ≤ a ∧ s = i·i+2·i+1 ∧ s ≤ a ∧ t = (2·i+1)+2}
// {i·i ≤ a ∧ s = (i+1)·(i+1) ∧ s ≤ a ∧ t = 2·i+3}
s= s+t;
// {∃S.(i·i ≤ a ∧ s = (i+1)·(i+1) ∧ s ≤ a ∧ t = 2·i+3)[S/s] ∧ s = (s+t)[S/s]}
// {∃S.i·i ≤ a ∧ S = (i+1)·(i+1) ∧ S ≤ a ∧ t = 2·i+3 ∧ s = S+t}
// {i·i ≤ a ∧ (i+1)·(i+1) ≤ a ∧ t = 2·i+3 ∧ s = (i+1)·(i+1)+t}
i= i+1;
// {∃I.(i·i ≤ a ∧ (i+1)·(i+1) ≤ a ∧ t = 2·i+3 ∧ s = (i+1)·(i+1)+t)[I/i] ∧ i = (i+1)[I/i]}
// {∃I.I·I ≤ a ∧ (I+1)·(I+1) ≤ a ∧ t = 2·I+3 ∧ s = (I+1)·(I+1)+t ∧ i = I+1}
// {∃I.I·I ≤ a ∧ (I+1)·(I+1) ≤ a ∧ t = 2·I+3 ∧ s = (I+1)·(I+1)+t ∧ I = i−1}
// {(i−1)·(i−1) ≤ a ∧ ((i−1)+1)·((i−1)+1) ≤ a ∧ t = 2·(i−1)+3 ∧ s = ((i−1)+1)·((i−1)+1)+t}
// {i·i ≤ a ∧ t = 2·i+1 ∧ s = i·i+t}
```


Arbeitsblatt 10.3: Vorwärtsverkettung

Gegeben folgendes Programm. Berechnet die Vorwärtsverkettung der Vorbedingung mit Vereinfachung:

```
// {x = X ∧ y = Y}
x= x+y;
// {???}
y= x−y;
// {???}
x= x−y;
// {???}
```

Was bewirkt das Programm?

Beweis der Zuweisungsregel Vorwärts

Erinnert Euch an das **Substitutionslemma**:

$$\sigma \models^I B[e/x] \iff \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B$$

Zu zeigen:

$$\begin{aligned}
&\models \{P\}\, x = e\, \{\exists V.\, P[V/x] \wedge x = (e[V/x])\} \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I P \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![x = e]\!]_{\mathcal{C}} \implies \sigma' \models^I \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I P \implies \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}] \models^I \exists V.\, P[V/x] \wedge x = (e[V/x]) \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I P \implies \sigma \models^I (\exists V.\, P[V/x] \wedge x = (e[V/x]))[e/x] \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I P \implies \sigma \models^I (\exists V.\, P[V/x] \wedge e = (e[V/x])) \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I P \implies \sigma \models^I (P[x/x] \wedge e = (e[x/x])) \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I P \implies \sigma \models^I P \quad \square
\end{aligned}$$

Vorwärtsverkettung

- Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- Vorteil: Vorbedingung bleibt kleiner
- Nachteil: in der Anwendung **umständlicher**
- Die entstehende Nachbedingung beschreibt die **symbolische Auswärtung**
- Vereinfachung benötigt Rechnung mit Existenzquantor

Zwischenfazit: Der Floyd-Hoare-Kalkül ist symmetrisch

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

II. 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 $Q \implies Q'$.
- Semantische Charakterisierung:

Stärkste Nachbedingung

Gegeben Zusicherung $P \in$ **Assn** und Programm $c \in$ **Stmt**, dann

$$\models \{P\}\, c\, \{Q\} \iff \mathrm{sp}(P, c) \implies Q$$

- 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

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

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!}
```


Beispiel: Fakultät, stärkste Nachbedingung

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

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

$VC_1 = \{asp_3 \implies p = (c-1)! \wedge c - 1 \leq n\}$

Beispiel: Fakultät, stärkste Nachbedingung (Schleifenrumpf

```
1  // {0 ≤ n}
2  p= 1;
   // {0 ≤ n ∧ p = 1}
3  c= 1;
   // {0 ≤ n ∧ p = 1 ∧ c = 1}
4  while (c <= n) /** inv p = (c−1)! ∧ c−1 ≤ n; */ \{
5   p = p * c;
    // {∃V1.(p = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n)[V1/p] ∧ p = (p · c)[V1/p]}
    // {∃V1.(V1 = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n) ∧ p = (V1 · c)}
    // {c − 1 ≤ n ∧ c ≤ n ∧ p = (c − 1)! · c}
6   c = c + 1;
    // {∃V2.(c − 1 ≤ n ∧ c ≤ n ∧ p = (c − 1)! · c)[V2/c] ∧ c = (c + 1)[V2/c]}
    // {∃V2.(V2 − 1 ≤ n ∧ V2 ≤ n ∧ p = (V2 − 1)! · V2) ∧ c = (V2 + 1)}
    // {c − 2 ≤ n ∧ c − 1 ≤ n ∧ p = (c − 2)! · (c − 1)}
7   }
   // {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```


Beispiel: Fakultät, Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1  // {0 ≤ n}
2  p= 1;
   // svc2 = ∅
   c= 1;
   // svc3 = ∅
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5   p = p * c;
    // svc5 = ∅
6   c = c + 1;
    // svc6 = ∅
7   }
   // svc4 = {asp3 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n),
   //                asp6 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n)}
   // svc4 = {(0 ≤ n ∧ p = 1 ∧ c = 1) ⟹ (p = (c − 1)! ∧ c − 1 ≤ n),
   //                (c − 2 ≤ n ∧ c − 1 ≤ n ∧ p = (c − 2)! · (c − 1))
   //                ⟹ (p = (c − 1)! ∧ c − 1 ≤ n)}
8  // {p = n!}
```


Schließlich zu zeigen

$$
\begin{aligned}
svc_8 &= \{\{asp_8 \Longrightarrow p = n!\} \cup svc_4 \\
&= \{(p = (c-1)! \wedge c - 1 \leq n \,\&\&\, \neg(c <= n)) \Longrightarrow p = n!\}, \\
&\quad (0 \leq n \wedge p = 1 \wedge c = 1) \Longrightarrow (p = (c-1)! \wedge c - 1 \leq n), \\
&\quad (c - 2 \leq n \wedge c - 1 \leq n \wedge p = (c-2)! \cdot (c-1)) \\
&\qquad \Longrightarrow (p = (c-1)! \wedge c - 1 \leq n)\} \\
&\rightsquigarrow \{true\}
\end{aligned}
$$

Arbeitsblatt 10.4: Jetzt seid ihr dran!

Berechnet die stärkste Nachbedingung und Verifikationsbedingungen für die ganzzahlige Division:

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


Beispiel: Suche nach dem Maximalen Element

```
1  // {0 < n}
2  i= 0;
3  // {∃I0.(0 < n)[I0/i] ∧ i = 0[I0/i]}
4  // {0 < n ∧ i = 0}
5  r= 0;
6  // {0 < n ∧ i = 0 ∧ r = 0}
7  while (i!= n)
8    /** inv (∀j. 0≤ j < i ⟶a[j]≤ a[r])∧ 0≤ r< n */ {
9    if (a[r] < a[i]) {
10     r= i;
11   }
12   else {
13     }
14   i= i+1;
15   }
16 // {∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ ¬(i ≠ n)}
17 // {(∀j.0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```


Beispiel: Suche nach dem Maximalen Element (Schleifenrumpf)

```
1  while (i!= n)
2    /** inv (∀j. 0≤ j < i ⟶a[j]≤ a[r])∧ 0≤ r< n */ {
3    if (a[r] < a[i]) {
4      // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ a[r] < a[i]}
5      r= i;
6      // {∃R0.((∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ a[r] < a[i])[R0/r] ∧ r = i[R0/r]}
7      // {∃R0.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R0]) ∧ 0 ≤ R0 < n ∧ a[R0] < a[i] ∧ r = i}
8    }
9    else {
10     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
11     }
12   // {(∃R0.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R0]) ∧ 0 ≤ R0 < n ∧ a[R0] < a[i] ∧ r = i) }
         ∨((∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ a[i] ≤ a[r])
13   i= i+1;
14   // {∃I0.((∃R0.(∀j.0 ≤ j < I0 ⟶ a[j] ≤ a[R0]) ∧ 0 ≤ R0 < n ∧ a[R0] < a[I0] ∧ r = I0) }
           ∨((∀j.0 ≤ j < I0 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ a[I0] ≤ a[r])) ∧ i = I0 + 1
15   }
```


Verifikationsbedingungen

1 $0 < n \wedge i = 0 \wedge r = 0 \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$

2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge \neg(i \neq n) \longrightarrow (\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$

3 $(\exists I_0.((\exists R_0.(\forall j.0 \leq j < I_0 \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq R_0 < n \wedge a[R_0] < a[I_0] \wedge r = I_0) \vee ((\forall j.0 \leq j < I_0 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge a[I_0] \leq a[r])) \wedge i = I_0 + 1) \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$

Weitere Vereinfachungsregeln

Existenzquantoren und Disjunktionen können mit folgenden Regeln vereinfacht werden:

6. Der Gültigkeitsbereich des Existenzquantors kann verkleinert werden:
   - $(\exists x.P \vee Q) \rightsquigarrow (\exists x.P) \vee (\exists x.Q)$
7. Disjunktionen in der Prämisse ergeben eine Fallunterscheidung:
   - $A_1 \vee A_2 \longrightarrow B \rightsquigarrow A_1 \longrightarrow B, A_2 \longrightarrow B$
8. Konjunktion distributiert über Disjunktion:
   - $(A_1 \vee A_2) \wedge B \rightsquigarrow (A_1 \wedge B) \vee (A_2 \wedge B)$
9. . . . und andersherum:
   - $(A_1 \wedge A_2) \vee B \rightsquigarrow (A_1 \vee B) \wedge (A_2 \vee B)$

Vereinfachte Verifikationsbedingungen

1.1 $0 < n \wedge i = 0 \wedge r = 0 \longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r])$ ✓

1.2 $0 < n \wedge i = 0 \wedge r = 0 \longrightarrow 0 \leq r < n$ ✓

2.1 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge \neg(i \neq n) \longrightarrow (\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r])$ ✓

2.2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge \neg(i \neq n) \longrightarrow 0 \leq r < n$ ✓

3.1 $(\exists l_0.(\exists R_0.(\forall j.0 \leq j < l_0 \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq R_0 < n \wedge a[R_0] < a[l_0] \wedge r = l_0) \wedge i = l_0 + 1)$
$\longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r])$

3.1 $(\exists R_0.((\forall j.0 \leq j < r \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1)$
$\longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r])$ ✓

3.2 $(\exists l_0.(\forall j.0 \leq j < l_0 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1)$
$\longrightarrow (\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r])$ ✓

3.3 $(\exists l_0.(\exists R_0.(\forall j.0 \leq j < l_0 \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq R_0 < n \wedge a[R_0] < a[l_0] \wedge r = l_0) \wedge i = l_0 + 1)$
$\longrightarrow 0 \leq r < n$

3.3 $(\exists R_0.(\forall j.0 \leq j < r \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq R_0 < n \wedge a[R_0] < a[r]) \wedge i = r + 1$
$\longrightarrow 0 \leq r < n$ ×

3.4 $(\exists l_0.(\forall j.0 \leq j < l_0 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge a[l_0] \leq a[r] \wedge i = l_0 + 1)$
$\longrightarrow 0 \leq r < n$ ✓

Invariante muss verstärkt werden: $0 \leq i < n$

Beispiel: Suche nach dem Maximalen Element (Schleifenrumpf)

```
1  while (i!= n)
2    /** inv (∀j. 0≤ j < i ⟶a[j]≤ a[r])∧ 0≤ i< n ∧ 0≤ r< n */ {
3    if (a[r] < a[i]) {
4      // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
5      r= i;
6      // {∃R0.((∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[r] < a[i])[R0/r] ∧ r = i[R0/r]}
7      // {∃R0.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R0]) ∧ 0 ≤ i < n ∧ 0 ≤ R0 < n ∧ a[R0] < a[i] ∧ r = i}
8    }
9    else {
10     // {(∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
11     }
12   // {(∃R0.(∀j.0 ≤ j < i ⟶ a[j] ≤ a[R0]) ∧ 0 ≤ i < n ∧ 0 ≤ R0 < n ∧ a[R0] < a[i] ∧ r = i)  }
         ∨((∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n ∧ a[i] ≤ a[r])
13   i= i+1;
14   // {∃l0.((∃R0.(∀j.0 ≤ j < l0 ⟶ a[j] ≤ a[R0]) ∧ 0 ≤ l0 < n ∧ 0 ≤ R0 < n ∧ a[R0] < a[l0] ∧ r = l0)  }
              ∨((∀j.0 ≤ j < l0 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ l0 < n ∧ 0 ≤ r < n ∧ a[l0] ≤ a[r])) ∧ i = l0 + 1
15   }
```


Vereinfachte Verifikationsbedingungen

...

3.3 $(\exists l_0.(\exists R_0.(\forall j.0 \leq j < l_0 \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq l_0 < n \wedge 0 \leq R_0 < n$
$\wedge a[R_0] < a[l_0] \wedge r = l_0) \wedge i = l_0 + 1) \longrightarrow 0 \leq r < n$

3.3 $(\exists R_0.(\forall j.0 \leq j < r \longrightarrow a[j] \leq a[R_0]) \wedge 0 \leq r < n \wedge 0 \leq R_0 < n$
$\wedge a[R_0] < a[r]) \wedge i = r + 1 \longrightarrow 0 \leq r < n$ ✓

...

Läuft!

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.
- Bis auf die Invarianten an Schleifen können wir Korrektheit 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 22.06.21
Modellierung und Spezifikation

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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}
```


Beispiel: Sortierte Felder

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

```
int a[8];
// {∀j. 0 ≤ j ≤ n < 8. a[j] ≤ a[j + 1]}
```

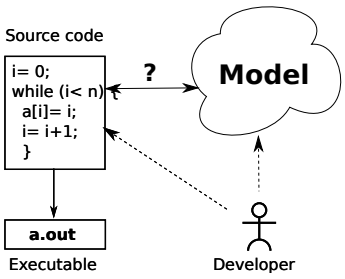
- Alternativ würden man auch gerne ein Prädikat definieren können

```
// {∀a. sorted(a, 0) ⟷ true}
// {∀a∀i. i ≥ 0 ⟶ (sorted(a, i + 1) ⟷ (a[i] ≤ a[i + 1] ∧ sorted(a, i)))}
```

- . . . und damit beweisen dass:

```
// {∀a∀n. sorted(a, n) ⟶ ∀i, j. 0 ≤ i ≤ j ≤ n ⟶ a[i] ≤ a[j]}
```


Generelles Problem: Modellbildung



Modell ist **abstrakte** Repräsentation:

- Verhalten des Programmes kann kürzer beschrieben werden
- Einfachere Beweise

Modell ist **treue** Repräsentation:

- Eigenschaften des Modelles gelten auch für das Programm

Was brauchen wir?

- Expressive **logische Sprache** (**Assn**)
- Konzeptbildung auf der Modellebene
  - Reichere Typen (bspw. Repräsentation von Feldern durch Listen)
  - Mehr Funktionen (bspw. auf Listen)
- Beispiele:
  - Separate Modellierungssprache, bspw. UML/OCL
  - Modellierungskonzepte in der Annotationssprache (JML, ACSL)

Modellierung von Typen: Integer

- Vereinfachung: **int** wird abgebildet auf $\mathbb{Z}$
- Das **kann** sehr falsch sein
- Manchmal **unerwartete** Effekte
- Behebung: statisch auf **Überlauf** prüfen
  - Nachteil: Plattformspezifisch

Binäre Suche

```
1  int binary_search(int val, int buf[], unsigned len)
2  {
3    // {0 ≤ len}
4    int low, high, mid, res;
5    low = 0; high = len;
6    while (low < high) {
7      mid= (low+ high)/2;
8      if (buf[mid] < val)
9        low = mid + 1;
10     else
11       high = mid;
12   }
13   if (low < len && buf[low] == val)
14     res= low;
15   else
16     res= −1;
17   // { res ≠ −1 ⟶ buf[res] = val ∧                    }
       //   res = −1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val
18 }
```

Binäre Suche, korrekt

```
1  int binary_search(int val, int buf[], unsigned len)
2  {
3    // {0 ≤ len}
4    int low, high, mid, res;
5    low = 0; high = len;
6    while (low < high) {
7      mid= low+ (high−low)/2;
8      if (buf[mid] < val)
9        low = mid + 1;
10     else
11       high = mid;
12   }
13   if (low < len && buf[low] == val)
14     res= low;
15   else
16     res= −1;
17   // { res ≠ −1 ⟶ buf[res] = val ∧                }
         res = −1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val
```


Typen: reelle Zahlen

- Vereinfachung: **double** wird abgebildet auf $\mathbb{R}$
- Auch hier **Fehler** und **unerwartete Effekte** möglich:
  - Kein Überlauf, aber **Rundungsfehler**
  - Fließkommazahlen: Standard IEEE 754-2008
- Mögliche Abhilfe:
  - Spezifikation der Abweichung von **exakter** (ideeller) Berechnung

Typen: labelled records

- Passen gut zu Klassen (Klassendiagramme in der UML)
- Bis auf Methoden: impliziter Parameter self
  - Werden nicht behandelt

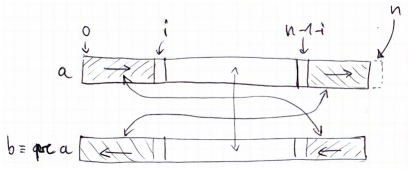
Typen: Felder

- Was repräsentiert **Felder**?
- **Sequenzen** (Listen)
- Modellierungssprache:
  - Annotation + **OCL**

Ein längeres Beispiel: reverse in-place

```
1  i= 0;
2  // {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
3  while (i< n/2) {
4    // ???
5    tmp= a[n−1−i];
6    a[n−1−i]= a[i];
7    a[i]= tmp;
8    i= i+1;
9  }
10 // {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```


reverse-in-place: die Invariante



Mathematisch:

$$\{\ \forall j.0 \le j < i \longrightarrow a[j] = b[n-1-j] \wedge$$
$$\forall j.n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge$$
$$\forall j.i \le j \le n-1-i \longrightarrow a[j] = b[j]\ \}$$

Ein längeres Beispiel: reverse in-place

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


Arbeitsblatt 11.1: Jetzt seid ihr dran

- Berechnet die Beweisverpflichtungen aus der While-Schleife bei reverse-in-place:

$$I \wedge b \longrightarrow \mathrm{awp}(c, I)$$

- Dazu berechnet ihr awp(c, I), mit
  $c =$

```
tmp= a[n−1−i];
a[n−1−i]= a[i];
a[i]= tmp;
i= i+1;
```

$I = \{\ \forall j.0 \le j < i \longrightarrow a[j] = b[n-1-j] \wedge$
$\forall j.n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge$
$\forall j.i \le j \le n-1-i \longrightarrow a[j] = b[j]\ \}$

- Ihr braucht noch nichts zu beweisen...

Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\text{awp}(\{\,\},P) &\stackrel{def}{=} P\\
\text{awp}(l=e,P) &\stackrel{def}{=} P[e/l] \quad (\textbf{Genauer}: \text{Folie 24 aus VL 8 bzw. nächste Folie})\\
\text{awp}(c_1;c_2,P) &\stackrel{def}{=} \text{awp}(c_1,\text{awp}(c_2,P))\\
\text{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1,P) &\stackrel{def}{=} Q \quad \text{wenn } \text{awp}(c_0,P)=b\wedge Q, \text{awp}(c_1,P)=\neg b\wedge Q\\
\text{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1,P) &\stackrel{def}{=} (b\wedge \text{awp}(c_0,P))\vee(\neg b\wedge \text{awp}(c_1,P))\\
\text{awp}(/{**}\ \{q\}\ {*}/,P) &\stackrel{def}{=} q\\
\text{awp}(\textbf{while } (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c,P) &\stackrel{def}{=} i\\
\text{wvc}(\{\,\},P) &\stackrel{def}{=} \emptyset\\
\text{wvc}(l=e,P) &\stackrel{def}{=} \emptyset\\
\text{wvc}(c_1;c_2,P) &\stackrel{def}{=} \text{wvc}(c_1,\text{awp}(c_2,P))\cup\text{wvc}(c_2,P)\\
\text{wvc}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1,P) &\stackrel{def}{=} \text{wvc}(c_0,P)\cup\text{wvc}(c_1,P)\\
\text{wvc}(/{**}\ \{q\}\ {*}/,P) &\stackrel{def}{=} \{q\longrightarrow P\}\\
\text{wvc}(\textbf{while } (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c,P) &\stackrel{def}{=} \text{wvc}(c,i)\cup\{i\wedge b\longrightarrow \text{awp}(c,i)\}\cup\{i\wedge\neg b\longrightarrow P\}
\end{aligned}
$$

Allgemeine Regel bei Ersetzungen (Nur Arrays)

Wie sieht nun die allgemeine Regel aus für

$$\overline{\vdash \{P[e/l]\}\ l=e\ \{P\}}$$

1. Wenn l Programmvariable ist, wie gewohnt substituieren
2. Wenn $l=a[s]$:
   - 2.1 Vorkommen der Form $a[t]$ in **Literalen** $L(a[t])$ und s und t beide in $\mathbb{Z}$ oder **Idt**,
     - dann ersetze $L(a[t])$ durch $L(e)$, falls $s=t$
   - 2.2 Vorkommen der Form $a[t]$ in **Literalen** $L(a[t])$ und s oder t sind nicht aus $\mathbb{Z}$,
     - dann ersetze $L(a[t])$ durch $(t=s\wedge L(e))\vee(t\neq s\wedge L(a[t]))$

   2.2 könnt ihr immer machen, 2.1 ist eine Optimierung
   - Das ist jetzt immer noch nicht die ganz allgemeine Form, aber für unsere Belange reicht das.

Vereinfacht mit Modellbildung

- $\text{seq}(a,n)$ ist ein Feld der Länge n repräsentiert als Liste (Sequenz)

$$
\begin{aligned}
n<0 &\longrightarrow \text{seq}(a,n)=[]\\
n\geq 0 &\longrightarrow \text{seq}(a,n)=\text{seq}(a,n-1)\text{++}(a[n]:[])
\end{aligned}
$$

- Aktionen auf Sequenzen:
  - $a:as$, $[]$ — Listenkonstruktoren
  - $\text{rev}(a)$ — Reverse
  - $a[i:j]$ — Slicing (à la Python)
  - ++ — Konkatenation
  - $[n]$ — Kurzform für $n:[]$

Interaktion mit der Substitution

- $\text{set}(a,i,v)$ ist der **funktionale Update** an Index i mit dem Wert v:

$$
\begin{aligned}
\text{set}([],i,v) &== []\\
\text{set}(a:as,0,v) &== v:as\\
i>0 &\longrightarrow \text{set}(a:as,i,v) == a:\text{set}(as,i-1,v)\\
i<0 &\longrightarrow \text{set}(as,i,v) == as
\end{aligned}
$$

- Damit ist

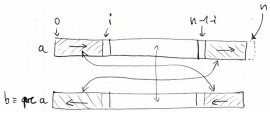
$$\text{seq}(a,n)[v/a[i]]=\text{set}(\text{seq}(a,n),i,v)$$

- und es gilt

$$as=\text{seq}(a,n)\wedge i\geq 0 \Longrightarrow \text{set}(as,i,x) == as[0:i]\text{++}[x]\text{++}as[i+1:n]$$

Reverse-in-Place mit Listen

```
1  i= 0;
2  // {bs = seq(a, n)}
3  while (i< n/2)
4    /** inv
5    */ {
6    tmp= a[n-1-i];
7    a[n-i-1]= a[i];
8    a[i]= tmp;
9    i= i+1;
10 }
11 // {as = seq(a, n) ⟹ rev(as) = bs}
```



- Damit vereinfachte VCs und vereinfachter Beweis.

Reverse-in-Place mit Listen

```
1  i= 0;
2  // {bs = seq(a, n)}
3  while (i< n/2)
4    /** inv  as = seq(a, n) ⟹ rev(as[n − i : n])++as[i : n − i]++ rev(as[0 : i]) = bs
5    */ {
6    // {  as = seq(a, n)[0 : i]++[a[n − 1 − i]]++Seq(a, n)[i + 1 : n − 1 − i]++[a[i]]++ seq(a, n)[n − i : n]  }
         ⟹ rev(as[n − (i + 1) : n])++as[(i + 1) : n − (i + 1)]++ rev(as[0 : (i + 1)]) = bs
7    // {as = set(set(seq(a, n), i, a[n − 1 − i]), n − i − 1, a[i]) ⟹ rev(as[n − (i + 1) : n])++as[(i + 1) : n − (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
8    tmp= a[n-1-i];
9    // {as = set(set(seq(a, n), i, tmp), n − i − 1, a[i]) ⟹ rev(as[n − (i+1) : n])++as[(i+1) : n − (i+1)]++ rev(as[0 : (i + 1)]) = bs}
10   a[n-i-1]= a[i];
11   // {as = set(seq(a, n), i, tmp) ⟹ rev(as[n − (i + 1) : n])++as[(i + 1) : n − (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
12   a[i]= tmp;
13   // {as = seq(a, n) ⟹ rev(as[n − (i + 1) : n])++as[(i + 1) : n − (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
14   i= i+1;
15   // {as = seq(a, n) ⟹ rev(as[n − i : n])++as[i : n − i]++ rev(as[0 : i]) = bs}
16 }
17 // {as = seq(a, n) ⟹ rev(as) = bs}
```


Arbeitsblatt 11.2: Beweise mit Listen

- Beweist durch **strukturelle Induktion** auf Sequenzen:

$$\text{rev}(as\text{++}bs) == \text{rev}(bs)\text{++}\,\text{rev}(as)$$

- Strukturelle Induktion heißt:
  1. Induktionsbasis: zeige Aussage für $as\stackrel{def}{=}[]$.
  2. Induktionsschritt: Annahme der Aussage, zeige Aussage für $as\stackrel{def}{=}a:as$
- Beweis durch Umformung, Anwendung der Gleichungen für rev, ++

$$
\begin{aligned}
\text{rev}([]) &== [] & (x:xs)\text{++}ys &== x:(xs\text{++}ys)\\
\text{rev}(x:xs) &== \text{rev}(xs)\text{++}[x] & as\text{++}(bs\text{++}cs) &== (as\text{++}bs)\text{++}cs\\
ys\text{++}[] &== ys
\end{aligned}
$$

Fazit

- Die Abstraktion ermöglicht wesentlich **kürzere** Vorbedingungen und Verifikationsbedingungen.
- Die Beweise auf Ebene der Listen sind wesentlich **einfacher**.
- Die Theorie der Listen ist wesentlich **reicher**.

Formelsprache mit Quantoren

- Wir brauchen Programmausdrücken wie **Aexp**
- Wir müssen neue Funktionen verwenden können
  - Etwa eine Fakultätsfunkion
- Wir müssen neue Prädikate definieren können
  - rev, ++, *sorted*, ...
- Wir müssen Formeln bilden können
  - Analog zu **Bexp**
  - Zusätzlich mit Implikation $\longrightarrow$, Äquivalenz $\longleftrightarrow$
  - Zusätzlich Quantoren über logische Variablen wie in

$$(\forall j.\, 0 \leq j < n \longrightarrow P[j]) \wedge P[n] \longrightarrow \forall j.\, 0 \leq j < n+1 \longrightarrow P[j]$$
$$\forall i.\, i \geq 0 \longrightarrow (sorted(a, i+1) \longleftrightarrow (a[i] \leq a[i+1] \wedge sorted(a, i)))$$

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**~~ ~~$n!, \sum_{i=1}^{n} i, \ldots$~~
  - Funktionen und Prädikate selbst definieren
  - Implikation, **Äquivalenzen**, Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v.\, b, \exists v.\, b$
- Formal:

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

Die bisherigen Funktionen

Die bisherigen Funktionen selbst definiert:

$$n! == factorial(n)$$
$$i \leq 0 \longrightarrow factorial(i) == 1$$
$$i > 0 \longrightarrow factorial(i) == i \cdot factorial(i-1)$$
$$\sum_{i=a}^{b} i == sum(a, b)$$
$$a > b \longrightarrow sum(a, b) == 0$$
$$a <= b \longrightarrow sum(a, b) == a + sum(a+1, b)$$

Kombination aus eingebautem **syntaktische Zucker** und eigenen **Definitionen**.

Die bisherigen Funktionen

- $\sum_{i=a}^{b} e, \prod_{i=a}^{b} e$ benötigen Funktionen **höherer Ordnung** und **anonyme Funktionen**:
- Ganz allgemein:

$$a \leq b \longrightarrow [a\,..\,b] == a : [a+1\,..\,b]$$
$$a > b \longrightarrow [a\,..\,b] == []$$

$$\text{foldl}(f, c, a : as) == \text{foldl}(f, f(c, a), as)$$
$$\text{foldl}(f, c, []) == c$$

$$\sum_{i=a}^{b} e(i) == \text{foldl}(\lambda x\ i\ .x + e(i), 0, [a\,..\,b])$$
$$\prod_{i=a}^{b} e(i) == \text{foldl}(\lambda x\ i\ .x \cdot e(i), 0, [a\,..\,b])$$

Ein Zoo von Logiken

- Das grundlegende Dilemma:

Entscheidbarkeit ⟵⟶ Ausdrucksmächtigkeit

- Der Logik-Zoo:

	Entscheidbar	Vollständig	
Aussagenlogik (OPL)	✓	✓	$(A \wedge B) \vee C$
Pressburger Arithmetik	✓	✓	$n < x \longrightarrow n + a < x + a$
Prädikatenlogik (PL)	×	✓	$\forall x. \exists y. x = y$
Peano-Arithmetik	×	×	$n \cdot 0 = 0$
PL mit Ind. & Fkt.	×	×	*Z3*
Prädikatenlogik 2. Stufe	×	×	$\forall P.\, P(0) \longrightarrow \forall n.\, P(n)$
Logik höherer Stufe (HOL)	×	×	*Haskell*

- Auswahl der Logik: Kompromiss (*sweet spot*)

Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in$ **Assn** in einem Zustand σ?
  - Auswertung (denotationale Semantik) ergibt *true*
- Variablen denotieren:
  - Zahlen und Characters wie bisher
  - Arrays wie in seq(a, n)
  - Listen über Zahlen/Character wie in rev(as), $as[(i+1) : n-(i+1)]$
  - Es könnten auch Strukturen sein (Datentypen wie in Haskell)
    - Sei **T** die Menge aller anderen Werte wie Listen, Strukturen usw.
- **Belegung** der logischen Variablen: I : **Var** $\rightarrow$ (**Z** $\cup$ **C** $\cup$ **Array** $\cup$ **T**)
- Semantik von b unter der Belegung I: $[\![b]\!]^I_{\mathcal{B}v}, [\![a]\!]^I_{\mathcal{A}v}$

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

Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in$ **Assn** in einem Zustand σ?
  - Auswertung (denotationale Semantik) ergibt *true*
- **Belegung** der logischen Variablen: I : **Var** $\rightarrow$ (**Z** $\cup$ **C** $\cup$ **Array** $\cup$ **T**)
- Semantik von b unter der Belegung I:

$$[\![\forall v.b]\!]^I_{\mathcal{B}v} = \{(\sigma, true) \mid \text{für alle } i \in \textbf{Z} \text{ gilt } (\sigma, true) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\}$$
$$\cup \{(\sigma, false) \mid \text{für ein } i \in \textbf{Z} \text{ gilt } (\sigma, false) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\}$$
$$[\![\exists v.b]\!]^I_{\mathcal{B}v} = \{(\sigma, true) \mid \text{für ein } i \in \textbf{Z} \text{ gilt } (\sigma, true) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\}$$
$$\cup \{(\sigma, false) \mid \text{für alle } i \in \textbf{Z} \text{ gilt } (\sigma, false) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\}$$

Analog für andere Typen.

Erfülltheit von Zusicherungen

Erfülltheit von Zusicherungen

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

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

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 $\mathbf{Assn}^c \subseteq$ **Assn** die Menge der geschlossenen Formeln

Lemma

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

- Sei Γ eine endliche Menge von Formeln, dann definieren wir

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

Erfülltheit von Zusicherungen unter Kontext

Erfülltheit von Zusicherungen unter Kontext

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

$$[\![(\bigwedge \Gamma) \longrightarrow b]\!]^I_{\mathcal{B}v}(\sigma) = \mathit{true}$$

Floyd-Hoare-Tripel mit Kontext

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

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

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

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

Floyd-Hoare-Kalkül mit Kontext

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

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

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

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

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 σ und Belegungen I dass $\Gamma \longrightarrow (A' \longrightarrow A)$ wahr bzw. dass

$$[\![\Gamma \longrightarrow (A' \longrightarrow A)]\!]^I_{\mathcal{B}v}(\sigma) = \mathit{true}$$

- $[\![.]\!]^I_{\mathcal{B}v}(\sigma)$ im Allgemeinen nicht berechenbar wegen

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

- Unvollständigkeit der Prädikatenlogik

Zusammenfassung

- Spezifikation erfordert **Modellbildung**
- Herangehensweisen:
  - Modellbildung in der Annotation ("ghost-code")
  - Separate Modellierungssprache
- Erweiterung der Annotationssprache um logische Anteile
  - Quantoren, Typen, Kontexte
- Problem: Unvollständigkeit der Logik

Korrekte Software: Grundlagen und Methoden
Vorlesung 12 vom 29.06.21
Spezifikation von Funktionen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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?

Beispiel: Rekursion

```
int factorial(int n)
/** pre  0≤ n;
    post  \result = n!; */
{
  if (n == 0) return 1;
  else return n * factorial(n-1);
}
```

```
int factorial(int n)
/** pre  0≤ n;
    post  \result = n!; */
{
  return n == 0 ? 1 : n * factorial(n-1);
}
```


Beispiel: Reverse mittels Swap

```
int swap(int a[], int i, int j)
/** pre  i < a_len ∧ j < a_len;
    post  a[i]=a[j]@pre ∧ a[j]=a[i]@pre; */
    {
      int buf = a[j];
      a[j] = a[i];
      a[i] = buf;
    }
    return;
```

```
int rev(int a[], int a_len)
/** pre  0 < a_len;
    post  ...; */
{
  int i;

  i= 0;
  while (i< a_len/2)
    /** inv  ...; */
    {
    swap(a[], i, a_len-i);
    i= i+1;
  }
  return;
}
```


Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen
3. Spezifikation von Funktionsdefinitionen
4. Beweisregeln für Funktionsdefinitionen
5. Semantik des Funktionsaufrufs
6. Beweisregeln für Funktionsaufrufe

Von Anweisungen zu Funktionen

- Erweiterung unserer Kernsprache um Funktionsdefinition und Deklarationen:

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

- Abstrakte Syntax
- Größe von Feldern: **konstanter** Ausdruck
- **FunSpec** wird später erläutert

Rückgaben

Neue Anweisungen: Return-Anweisung

$$
\begin{aligned}
\textbf{Stmt} \quad & s ::= l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\\
& \mid \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ P\ {*}/\ c \mid /{*}{*}\ \{P\}\ {*}/\\
& \mid \textbf{return}\ a^{?}
\end{aligned}
$$

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

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

- Und als Mengen/partielle Funktionen formuliert:

$$g \circ_S f = \{(\sigma, \rho') \mid \exists \sigma'.\, (\sigma, \sigma') \in f \wedge (\sigma', \rho') \in g\} \cup \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in f\}$$

Semantik von Anweisungen

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

$$\begin{aligned}
[\![x = e]\!]_{\mathcal{C}} &= \{(\sigma, \sigma[l \mapsto a]) \mid (\sigma, l) \in [\![x]\!]_{\mathcal{L}}, (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\} \\
[\![c_1; c_2]\!]_{\mathcal{C}} &= [\![c_2]\!]_{\mathcal{C}} \circ_S [\![c_1]\!]_{\mathcal{C}} \quad \text{Komposition wie oben} \\
[\![\{\,\}]\!]_{\mathcal{C}} &= \mathbf{Id}_\Sigma \qquad \mathbf{Id}_\Sigma := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
[\![\textbf{if } (b) \ c_0 \ \textbf{else} \ c_1]\!]_{\mathcal{C}} &= \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \rho') \in [\![c_0]\!]_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma, \rho') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \rho') \in [\![c_1]\!]_{\mathcal{C}}\} \\
&\quad \text{mit } \rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U \\
[\![\textbf{return } e]\!]_{\mathcal{C}} &= \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\} \\
[\![\textbf{return}]\!]_{\mathcal{C}} &= \{(\sigma, (\sigma, *))\}
\end{aligned}$$

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

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

❶

$$\begin{aligned}
[\![x = 3; x = 4]\!]_{\mathcal{C}} &= [\![x = 4]\!]_{\mathcal{C}} \circ_S [\![x = 3]\!]_{\mathcal{C}} \\
&= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\
&= \{(\sigma, \sigma[x \mapsto 4])\}
\end{aligned}$$

❷

$$\begin{aligned}
[\![x = 3; \textbf{return } x; x = 4]\!]_{\mathcal{C}} &= [\![x = 4]\!]_{\mathcal{C}} \circ_S ([\![\textbf{return } x]\!]_{\mathcal{C}} \circ_S [\![x = 3]\!]_{\mathcal{C}}) \\
&= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \\
&\quad (\{((\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![x]\!]_{\mathcal{A}}\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\
&= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S (\{(\sigma, (\sigma, \sigma(x)))\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\
&= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, (\sigma[x \mapsto 3], \underbrace{\sigma[x \mapsto 3](x)}_{3}))\} \\
&= \{(\sigma, (\sigma[x \mapsto 3], 3))\}
\end{aligned}$$

Semantik von Funktionsdefinitionen

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

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

$$\begin{aligned}
&[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!]_{fd} v_1, \ldots, v_n = \\
&\quad \{(\sigma[p_1 \mapsto v_1] \ldots, [p_n \mapsto v_n], (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![blk]\!]_{blk}\}
\end{aligned}$$

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.

Semantik von Blöcken und Deklarationen

Blöcke bestehen aus Deklarationen und einer Anweisung.

$$[\![.]\!]_{blk} : \mathbf{Blk} \to \Sigma \rightharpoonup (\Sigma \times V_U)$$

$$[\![\mathit{decls}\ \mathit{stmts}]\!]_{blk} \stackrel{def}{=} \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![\mathit{stmts}]\!]_{\mathcal{C}}\}$$

- Von $[\![\mathit{stmts}]\!]_{\mathcal{C}}$ sind nur **Rückgabezustände** interessant.
  - Kein „fall-through“
  - Was passiert ohne **return** am Ende?
- Keine Initialisierungen, Deklarationen haben (noch) keine Semantik.

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:

FunSpec ::= /** **pre** **Assn** **post** **Assn** */

Vorbedingung **pre** sp; $\overbrace{\Sigma}^{\text{Vorzustand}} \to \mathbb{B}$

Nachbedingung **post** sp; $\underbrace{\Sigma}_{\text{Vorzustand}} \times \underbrace{(\Sigma \times \mathbf{V}_U)}_{\text{Nachzustand und Return-Wert}} \to \mathbb{B}$

e**@pre** 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)! ∧ 0≤ c ∧ c-1≤ n; */ {
    p= p*c;
    c= c+1;
    }
 return p;
}
```

Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre  \array (a, a_len) ∧ 0 < a_len;
    post ∀i. 0≤ i< a_len ⟶ a[i]≤ a[ \result ] ∧ 0≤  \result < a_len; */
{
  int r; int j;

  j= 0;
  r= 0;
  while (j< a_len)
    /** inv (∀j. 0≤ j < i⟶a[j]≤ a[r])∧ 0≤ i≤ n ∧ 0≤ r< n; */
    {
    if (a[j]> x) { r= j; }
    j= j+1;
    }
  return r;
}
```


Ziel: Gültigkeit von Spezifikationen

- Ziel ist eine **Semantik von Spezifikationen** $[\![.]\!]_{Bsp}$ zu definieren, um damit **semantische Gültigkeit** zu definieren:

$$\textbf{pre } p \textbf{ post } q \models fd \iff \forall v_1, \ldots, v_n.\ [\![fd]\!]_{fd}\ \Gamma\ v_1 \ldots v_n \in [\![\textbf{pre } p \textbf{ post } q]\!]_{Bsp}\ \Gamma$$

- Γ enthält globale Definitionen, insbesondere andere Funktionen.
  - Warum?

Beispiel: Reverse mittels Swap

```
int rev(int a[], int a_len)
/** pre  0 < a_len;
    post  ...; */
{
  int i;

  i= 0;
  while (i< a_len/2)
    /** inv  ...; */
    {
    swap(a[], i, a_len−i);
    i= i+1;
  }
  return;
}
```

```
void swap(int a[], int i, int j)
/** pre  ∃l. \array (a, l) ∧ i< l ∧ j< l;
    post a[i]=a[j]@pre ∧ a[j]=a[i]@pre; */
{
  int buf;

  buf = a[j];
  a[j] = a[i];
  a[i] = buf;
  return;
}
```


Beispiel: Rekursion

```
int factorial(int n)
/** pre  0≤ n;
    post  \result == n!; */
{
  int x;

  if (n == 0) {
    return 1;
    }
  else {
    x = factorial(n−1);
    return n * x;
    }
}
```


Semantik von Spezifikationen

- Vorbedingung: Auswertung als $[\![sp]\!]_{\mathcal{B}}\ \Gamma$ über dem Vorzustand
- Nachbedingung: Erweiterung von $[\![.]\!]_{\mathcal{B}}$ und $[\![.]\!]_{\mathcal{A}}$
  - Ausdrücke können in Vor- **oder** Nachzustand ausgewertet werden.
  - \result darf nicht in Funktionen vom Typ **void** auftreten.

Semantik von Spezifikationen

$$\begin{aligned}
[\![.]\!]_{Bsp} &: \textbf{Env} \to \textbf{Assn} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B} \\
[\![.]\!]_{Asp} &: \textbf{Env} \to \textbf{Aexpv} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V} \\
[\![!b]\!]_{Bsp}\ \Gamma &= \{((\sigma,(\sigma',v)), true) \mid ((\sigma,(\sigma',v)), false) \in [\![b]\!]_{Bsp}\Gamma\} \\
&\quad \cup \{((\sigma,(\sigma',v)), false) \mid ((\sigma,(\sigma',v)), true) \in [\![b]\!]_{Bsp}\Gamma\} \\
[\![x]\!]_{Asp}\ \Gamma &= \{((\sigma,(\sigma',v)), \sigma'(x))\} \\
&\ldots \\
[\![e@\textbf{pre}]\!]_{Bsp}\ \Gamma &= \{((\sigma,(\sigma',v)), b) \mid (\sigma, b) \in [\![e]\!]_{\mathcal{B}}\ \Gamma\} \\
[\![e@\textbf{pre}]\!]_{Asp}\ \Gamma &= \{((\sigma,(\sigma',v)), a) \mid (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![\backslash\textbf{result}]\!]_{Asp}\ \Gamma &= \{((\sigma,(\sigma',v)), v)\}
\end{aligned}$$

$$[\![\textbf{pre } p \textbf{ post } q]\!]_{Bsp}\ \Gamma = \{(\sigma,(\sigma',v)) \mid (\sigma, true) \in [\![p]\!]_{\mathcal{B}}\ \Gamma \wedge ((\sigma,(\sigma',v)), true) \in [\![q]\!]_{Bsp}\ \Gamma\}$$

Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre } p \textbf{ post } q \models fd \iff \forall v_1, \ldots, v_n.\ [\![fd]\!]_{fd}\ \Gamma\ v_1 \ldots v_n \in [\![\textbf{pre } p \textbf{ post } q]\!]_{Bsp}\ \Gamma$$

  - Γ enthält globale Definitionen, insbesondere andere Funktionen.
- Wie passt das zu den Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?
- Wie **beweisen** wir das? **Erweiterung** des Hoare-Kalküls

Erweiterung des Floyd-Hoare-Kalküls

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

Hoare-Tripel: zusätzliche Spezifikation für **Rückgabewert**.

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

c ist **partiell korrekt**, wenn für alle Zustände σ, die P erfüllen:

- die Ausführung von c mit σ in σ' regulär terminiert, so dass σ' die Spezifikation Q erfüllt,
- oder die Ausführung von c in σ' mit dem Rückgabewert v terminiert, so dass (σ', v) die Rückgabespezifikation Q_R erfüllt.

$$\begin{aligned}
&\Gamma \models \{P\}\, c\, \{Q \mid Q_R\} \iff \\
&\quad \forall \sigma.\ (\sigma, true) \in [\![P]\!]_{\mathcal{B}}\ \Gamma \wedge (\exists \sigma'.\ (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \implies ((\sigma,(\sigma',*)), true) \in [\![Q]\!]_{Bsp}\ \Gamma) \\
&\qquad\qquad \vee \\
&\qquad\qquad (\exists \sigma', v.\ (\sigma,(\sigma',v)) \in [\![c]\!]_{\mathcal{C}} \implies ((\sigma,(\sigma',v)), true) \in [\![Q_R]\!]_{Bsp}\ \Gamma)
\end{aligned}$$

Erweiterung des Floyd-Hoare-Kalküls: return

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q\, \{\textbf{return} \mid P\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/\backslash\textbf{result}]\, \{\textbf{return}\ e \mid P\}}$$

- 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

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\,\textbf{@pre}] \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid \mathit{false}\}}{\Gamma \vdash f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ c\}}$$

- Die Parameter x_i werden in **post** Q per Konvention nur als x_i referenziert, aber es ist immer der Wert im **Vorzustand** gemeint (eigentlich x_i **@pre**).
  - Deswegen wird in Q im Hoare-Tripel ersetzt
- Variablen unterhalb von (.) **@pre** werden bei der Substitution (Zuweisungsregel) **nicht ersetzt!**
- (.) **@pre** wird beim Weakening von der Vorbedingung P ersetzt
- Sequentielle Nachbedingung von c ist *false*

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

$$\frac{}{\Gamma \vdash \{\Gamma\}\, P\, \{\{\} \mid P\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P\, \{c_1 \mid R\} \quad \Gamma \vdash \{\Gamma\}\, R\, \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{c_1; c_2 \mid Q\}}$$

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/x]\, \{l = e \mid Q\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P \wedge b\, \{c \mid P\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\textbf{while}\ (b)\ c \mid P \wedge \neg b\}}$$

$$\frac{\Gamma \vdash \{\Gamma\}\, P \wedge b\, \{c_1 \mid Q\} \quad \Gamma \vdash \{\Gamma\}\, P \wedge \neg b\, \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid Q\}}$$

$$\frac{(\Gamma \wedge P) \longrightarrow P' \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid Q'\} \quad (\Gamma \wedge Q') \longrightarrow Q \quad (\Gamma \wedge R') \longrightarrow R}{\Gamma \vdash \{\Gamma\}\, P\, \{c \mid Q\}}$$

Erweiterter Floyd-Hoare-Kalkül II

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q\, \{\textbf{return} \mid P\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/\backslash\textbf{result}]\, \{\textbf{return}\ e \mid P\}}$$

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\,\textbf{@pre}] \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid \mathit{false}\}}{\Gamma \vdash f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ c\}}$$

Arbeitsblatt 12.2: Kurzbeispiel

Verifiziert folgendes Kurzbeispiel:

```
int f(int x)
/** post \result = x+1; */
{
  // ???
  x= x+1;
  // ???
  return x;
  // ???
}
```


Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  // {x + 1 = x @pre +1}
  x= x+1;
  // {x = x @pre +1}
  return x;
  // {false | \result = x @pre +1}
}
```

Weakening der Spezifikationsregel:

$$\mathit{true} \Longrightarrow (x + 1 = x\,\textbf{@pre} + 1)[x/x\,\textbf{@pre}]$$
$$x + 1 = x + 1 \quad \checkmark$$

Approximative schwächste Vorbedingung

- Erweiterung zu $\text{awp}(\Gamma, c, Q, Q_R)$ und $\text{wvc}(\Gamma, c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
  - Es werden der **Kontext** Γ und eine **Rückgabespezifikation** Q_R benötigt.
- Es gilt:

$$\bigwedge \text{wvc}(\Gamma, c, Q, Q_R) \Longrightarrow \Gamma \models \{\text{awp}(c, Q, Q_R)\}\, c\, \{Q \mid Q_R\}$$

- Berechnung von awp und wvc:

$$\text{awp}(\Gamma, f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \text{awp}(\Gamma', blk, \mathit{false}, Q[x_i\,\textbf{@pre}\,/x_i])$$

$$\text{wvc}(\Gamma, f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\,\textbf{@pre}]\} \cup \text{wvc}(\Gamma', blk, \mathit{false}, Q[x_i\,\textbf{@pre}\,/x_i])$$

$$\Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1, \ldots, x_n.\, (P, Q)]$$

$$P' \stackrel{def}{=} \text{awp}(\Gamma', blk, \mathit{false}, Q[x_i\,\textbf{@pre}\,/x_i])$$

Approximative schwächste Vorbedingung (Revisited)

$$\begin{aligned}
\text{awp}(\Gamma, \{\,\}, Q, Q_R) &\stackrel{def}{=} Q \\
\text{awp}(\Gamma, l = e, Q, Q_R) &\stackrel{def}{=} Q[e/l] \\
\text{awp}(\Gamma, c_1; c_2, Q, Q_R) &\stackrel{def}{=} \text{awp}(\Gamma, c_1, \text{awp}(c_2, Q, Q_R), Q_R) \\
\text{awp}(\Gamma, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, Q, Q_R) &\stackrel{def}{=} (b \wedge \text{awp}(\Gamma, c_0, Q, Q_R)) \vee (\neg b \wedge \text{awp}(\Gamma, c_1, Q, Q_R)) \\
\text{awp}(\Gamma, /{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} q \\
\text{awp}(\Gamma, \textbf{while}\ (b)\ /{*}\ \textbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} i \\
\text{awp}(\Gamma, \textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} Q_R[e/\backslash\textbf{result}] \\
\text{awp}(\Gamma, \textbf{return}, Q, Q_R) &\stackrel{def}{=} Q_R
\end{aligned}$$

Approximative Verifikationsbedingungen (Revisited)

$$
\begin{aligned}
\mathrm{wvc}(\Gamma, \{\,\}, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Gamma, l = e, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Gamma, c_1; c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c_1, \mathrm{awp}(c_2, Q, Q_R), Q_R) \cup \mathrm{wvc}(\Gamma, c_2, Q, Q_R) \\
\mathrm{wvc}(\Gamma, \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c_1, Q, Q_R) \cup \mathrm{wvc}(\Gamma, c_2, Q, Q_R) \\
\mathrm{wvc}(\Gamma, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} \{\Gamma \wedge q \Longrightarrow Q\} \\
\mathrm{wvc}(\Gamma, \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c, i, Q_R) \\
&\quad \cup \{\Gamma \wedge i \wedge b \Longrightarrow \mathrm{awp}(\Gamma, c, i, Q_R)\} \\
&\quad \cup \{\Gamma \wedge i \wedge \neg b \Longrightarrow Q\} \\
\mathrm{wvc}(\Gamma, \textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}
$$

Beispiel: Fakultät

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   // {1 = (1 − 1)! ∧ 0 < 1}
7   p= 1;
8   // {p = (1 − 1)! ∧ 0 < 1}
9   c= 1;
10  // {p = (c − 1)! ∧ 0 < c}
11  while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
12     p= p*c;
13     if (c == n) {
14       return p;
15       }
16     c= c+1;
17     }
18     // {false}
19 }
```


Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

(1) $0 \le n \longrightarrow 1 = (1-1)! \wedge 0 < 1$

(3) $p = (c-1)! \wedge 0 < c \wedge \neg \mathit{true} \longrightarrow \mathit{false}$

Vereinfacht:

(1.1) $0 \le n \longrightarrow 1 = 0!$ ✓

(1.2) $0 \le n \longrightarrow 0 < 1$ ✓

(3) $\mathit{false} \longrightarrow \mathit{false}$ ✓

Beispiel: Fakultät (Schleifenrumpf)

```
1
2    while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3      // {(c = n ∧ p · c = n@pre!) ∨ (c ≠ n ∧ p · c = ((c + 1) − 1)! ∧ 0 < c + 1)}
4      p= p*c;
5      // {(c = n ∧ p = n@pre!) ∨ (c ≠ n ∧ p = ((c + 1) − 1)! ∧ 0 < c + 1)}
6      if (c == n) {
7        // {p = n@pre!}
8        return p;
9        }
10     else {
11       // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
12       }
13     // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
14     c= c+1;
15     // {p = (c − 1)! ∧ 0 < c}
16     }
```


Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

(2) $p = (c-1)! \wedge 0 < c \wedge \mathit{true}$
$\longrightarrow (c = n \wedge p \cdot c = n!)$
$\vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1)$

Neue **Vereinfachungsregel**:

9. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

(2.1) $p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n\,\texttt{@pre}!$ × Benötigt $n = n\,\texttt{@pre}$

(2.2) $p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c!$ ✓ $((c-1)! \cdot c = c!)$

(2.3) $p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c+1$ ✓ $(c < c+1)$

Was fällt uns auf?

- Die Invariante ist $p = (c-1)! \wedge 0 < c \wedge n = n\,\texttt{@pre}$
- Da fehlt $c - 1 \le n$ — wie können wir $c - 1 = n$ am Ende beweisen?
- Mit der Schleifenbedingung 1 gilt **jede** Nachbedingung.
- Austritt aus der Schleife mit $c == n$ — vereinfacht den Beweis.
- Aber: müssen in der Invariante **explizit** spezifizieren, dass n sich nicht ändert.

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Deklarationen und Parameter ✓
2. Semantik von Funktionsdefinitionen ✓
3. Spezifikation von Funktionsdefinitionen ✓
4. Beweisregeln für Funktionsdefinitionen ✓
5. Semantik des Funktionsaufrufs
6. Beweisregeln für Funktionsaufrufe

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe
- Prozeduraufrufe (mit Zuweisung eines Rückgabewertes)

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

Zur Erinnerung: Semantik von Funktionsdefinitionen

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

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

$$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!]_{fd} = \{((x_1, \ldots, x_n), \sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![blk]\!]_{blk} \circ_S \{(\sigma, \sigma[p_i \mapsto x_i]_{i=1,\ldots,n})\}\}$$

- Die Funktionsargumente sind lokale Deklarationen, die beim Aufruf initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.

Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $[\![f]\!]_{fd}$
- 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**?
    - In C: Durch Übergabe von **Referenzen** als **Werte**
      $\Longrightarrow$ Erfordert Modellierung des Speichermodells (nächste Vorlesung)
    - Wir betrachten das hier/heute nicht, somit nur **reine Funktionen**!

Arbeitsblatt 12.3: Funktionsaufrufe

Wie werden Parameter in folgenden Programmiersprachen übergeben?

- **C:** Call-by-value für skalare Typen (arithmetische Typen und Referenzen), damit call-by-reference für aggregierte Typen (**struct**, Felder);
- **Java**:
- **Haskell**:
- **Python**:
- **Other:** (specify)

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von f dem Bezeichner f zuordnen.
- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- 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

$$[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{A}}\Gamma = \{(\sigma, v) \mid ((a_1, \ldots, a_n), \sigma, (\sigma', v)) \in \Gamma(f) \wedge (\sigma, a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$
$$[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, \sigma') \mid ((a_1, \ldots, a_n), \sigma, (\sigma', v)) \in \Gamma(f) \wedge (\sigma, a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$
$$[\![x = f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, \sigma'[x \mapsto v]) \mid ((a_1, \ldots, a_n), \sigma, (\sigma', v)) \in \Gamma(f) \wedge (\sigma, a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$

- Aufruf von Funktion $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{A}}$ ignoriert Endzustand
- Aufruf einer Prozedur $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}$ ignoriert Rückgabewert
- Somit: Kombination mit Zuweisung

Erweiterung des Kontext

- Der **Kontext** Γ muss Funktionsbezeichnern ihre **Spezifikation** (Vor/Nachbedingung) zuordnen.
- $\Gamma(f) = \forall x_1, \ldots, x_n.\,(P, Q)$, für Funktion $f(x_1, \ldots, x_n)$ mit Vorbedingung P und Nachbedingung Q.
- Korrektheit gilt immer nur im **Kontext**, dadurch kann jede Funktion separat verifiziert werden (**Modularität**).

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\,(P, Q)}{\Gamma \vdash \{\Gamma\}\ P[t_i/x_i] \wedge y_i\,\texttt{@pre} = y_i\,\{l = f(t_1, \ldots, t_n) \mid Q[t_i/x_i][l/\backslash\texttt{result}]\}}$$

- Γ muss f mit der Vor-/Nachbedingung P, Q enthalten
- In P und Q werden Parameter x_i durch Argumente t_i ersetzt.
- **\result** in Q wird durch l ersetzt
- Für alle Variablen y in Q, die mit y **@pre** referenziert werden, wird eine Gleichung $y = y$ **@pre** in die Vorbedingung eingefügt.
  - z.Zt. nur für global Variablen sinnvoll

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x@pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x@pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x@pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x@pre!}
19  return r* x;
20  // {false | \result = x@pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \le x \wedge x = x\,\texttt{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\texttt{@pre!}) \vee (x \neq 0 \wedge 0 \le x - 1)$

(1.1) $0 \le x \wedge x = x\,\texttt{@pre} \wedge x = 0 \longrightarrow 1 = x\,\texttt{@pre!}$ ✓

(1.2) $0 \le x \wedge x = x\,\texttt{@pre} \wedge x \neq 0 \longrightarrow 0 \le x - 1$ ✓

(2) $r = (x-1)! \longrightarrow r \cdot x = x\,\texttt{@pre!}$

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@pre**!

Beobachtung

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Der Aufruf einer Funktion **ersetzt** die momentane Nachbedingung — das ist ein Problem!
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben
- Rekursion benötigt keine Extrabehandlung
  - Termination von rekursiven Funktionen wird extra gezeigt

Frame Rule

- Konstanzregel (Rule of Constancy):

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P \wedge R\}\, c\, \{Q \wedge R\}}$$

- Nebenbedingung:
  - c verändert keine Variablen in R, **oder**
  - für alle Programm-Variablen x, die in R vorkommen, gibt es **keine** Zuweisung $x = \ldots$ in c
- Das ist eine **neue Regel**, die **bewiesen** werden muss
- Schwierig zu handhaben bei Rückwärts/Vorwärtsrechung
  - R muss **annotiert** weerden

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe mit Zuweisung eines Rückgabewertes

$$\begin{aligned}\textbf{Stmt}\ c ::= {} & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\ \mid {} & \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ a\ {*}/\ c \mid /{*}{*}\ \{a\}\ {*}/ \\ \mid {} & \textbf{Idt}(a^*) \\ \mid {} & /{*}{*}\,\textbf{const}\ R\ {*}/\ l = \textbf{Idt}(a^*) \\ \mid {} & \textbf{return}\ a^?\end{aligned}$$

Approximative schwächste Vorbedingung & Verifikationsbedingung

Sei $\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)$

$$\text{awp}(\Gamma, /{*}{*}\,\textbf{const}\ R\ {*}/ l = f(t_1, \ldots, t_n), U, U_R) \stackrel{def}{=} R \wedge P[t_i/x_i] \quad \text{wenn } l \notin FV(R)$$

$$\text{wvc}(\Gamma, /{*}{*}\,\textbf{const}\ R\ {*}/ l = f(t_1, \ldots, t_n), U, U_R) \stackrel{def}{=} \{R \wedge Q[t_i/x_i][l/\,\backslash\textbf{result}] \longrightarrow U\} \quad \text{wenn } l \notin FV(R)$$

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post \result = x!; */
4  {
5    int r = 0;
6
7    // {(x = 0 ∧ 1 = x@pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x@pre)}
8    if (x == 0) {
9      // {1 = x@pre!}
10     return 1;
11     // {0 ≤ x − 1 ∧ x = x@pre | \result = x@pre!}
12   } else {
13     // {0 ≤ x − 1 ∧ x = x@pre}
14     }
15   // {0 ≤ x − 1 ∧ x = x@pre}
16   /** const x= x@pre */ r = fac(x− 1);
17   // {r · x = x@pre!}
18   return r* x;
19   // {false | \result = x@pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \le x \wedge x = x\,\textbf{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\textbf{@pre!}) \vee (x \neq 0 \wedge 0 \le x - 1 \wedge x = x\,\textbf{@pre})$

(1.1) $0 \le x \wedge x = x\,\textbf{@pre} \wedge x = 0 \longrightarrow 1 = x!$ ✓

(1.2) $0 \le x \wedge x = x\,\textbf{@pre} \wedge x \neq 0 \longrightarrow 0 \le x - 1$ ✓

(1.3) $0 \le x \wedge x = x\,\textbf{@pre} \wedge x \neq 0 \longrightarrow x = x\,\textbf{@pre}$ ✓

(2) $x = x\,\textbf{@pre} \wedge r = (x-1)! \longrightarrow r \cdot x = x\,\textbf{@pre!}$ ✓

Zusammenfassung

- Funktionen sind **zentrales Modularisierungskonzept**
- Behandlung von Funktionen erfordert **vielfältige Erweiterungen**
- Erweiterung der **Semantik:**
  - Erweiterung der Semantik um **Rückgabezustand** $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$
  - Die Semantik einer Funktion ist **parametrisiert** $\mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$
- Erweiterung der **Spezifikationen:**
  - Spezifikation von Funktionen: **Vor-/Nachzustand** statt logischer Variablen
- Erweiterung des **Hoare-Kalküls**:
  - **Gesonderte Nachbedingung** für Rückgabewert/Endzustand
  - Aufruf einer Funktion **ersetzt** Vor/Nachbedingung, daher **Framing**
- **Einschränkungen**: nur call-by-value
- Fazit: **ohne Referenzen** sind Funktionen wenig brauchbar

Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 06.07.21
Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Prüfungstermine

- Fr, 16.07.2021: Onlineprüfungen (9:00- 16:00, alle 30 Minuten)
- Di, 27.07.2021: Onlineprüfungen (9:00- 12:00, alle 30 Minuten)
- Mi, 28.07.2021: Onlineprüfungen (10:00- 17:00, alle 30 Minuten)
- Do, 02.09.2021: Onlineprüfungen (9:00- 17:00, alle 30 Minuten)
- Fr, 03.09.2021: Onlineprüfungen (9:00- 17:00, alle 30 Minuten)

Prüfungsmodalitäten

- Anmeldung über stud.ip.
- Onlineprüfung:
  1. Für die Prüfung müsst ihr euch **zwingend** mit eurem **Uni-Bremen-Zoom-Account** anmelden, sonst kommt ihr nicht in den Zoom-Raum. Bitte ruhig zehn Minuten vor der Prüfung den Zoom-Raum betreten, ihr könnt dann im Warteraum warten.
  2. Haltet einen Lichtbildausweis (Studentenausweis, Perso, Führerschein o.ä.) bereit, um Eure Identität nachzuweisen.
  3. Die Kamera ist Pflicht, und bitte während der der gesamten Prüfung eingeschaltet zu lassen. (Ihr könnt gerne einen virtuellen Hintergrund wählen, wenn ihr wollt, aber ohne Kamera können wir keine Prüfung abnehmen.)
  4. Bitte sucht euch einen ruhigen Ort, ohne Hintergrundgeräusche (insbesondere Stimmen) im Hintergrund. Wir wollen hören können, ob ihr mit jemand anders redet.

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül I
- Der Floyd-Hoare-Kalkül II: Invarianten
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Modellierung
- Spezifikation von Funktionen
- Referenzen und Speichermodelle
- Ausblick und Rückblick

Motivation

- Warum Referenzen?
  - Nötig für *call by reference*
  - Funktionen 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:
  - (Fast) alles ist eine Referenz
  - Schwach getypt (Subtyping und Typumwandlung)
- Haskell, SML, OCaml:
  - Stark getypt (typsicher)
- Scriptsprachen (Python, Ruby):
  - Ähnlich Java

Ausdrücke

- Neue Operatoren: Addressoperator (&) und Derefenzierung ($*$)

$$
\begin{array}{rl}
\textbf{Lexp} & l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \mid *a \\
\textbf{Aexp} & a ::= \textbf{Z} \mid \textbf{C} \mid \textbf{Lexp} \mid \&l \\
 & \quad \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid \textbf{Idt}(\textbf{Exp}^*) \\
\textbf{Bexp} & b ::= \ldots \\
\textbf{Exp} & e := \textbf{Aexp} \mid \textbf{Bexp} \\
\textbf{Stmt} & c ::= \ldots \\
\textbf{Type} & t ::= \textbf{char} \mid \textbf{int} \mid *t \mid \textbf{struct}\ \textbf{Idt}^?\ \{\textbf{Decl}^+\} \mid t\ \textbf{Idt}[a]
\end{array}
$$

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.
- An der Stelle (*) zwei Bezeichner für die gleiche Loc: a und *p
- Großes Problem für Semantik und Hoare-Kalkül.
- Modellierung der Zuweisung durch Substitution nicht mehr möglich

Erweiterung des Zustandmodells

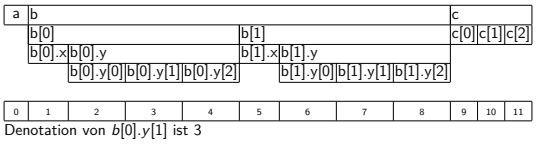
- Bisheriger Zustand $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$ mit
  - **Locations:** **Loc** ::= **Idt** | **Loc**[$\mathbb{Z}$] | **Loc**.**Idt**
  - Werte: $\mathbf{V} = \mathbb{Z}$
- Ansatz reicht nicht mehr:
  - Werte müssen auch Locations sein: $\mathbf{V} \stackrel{def}{=} \mathbb{Z} + \mathbf{Loc}$
  - **Idt** als Location nicht ausreichend für Referenzen und Funktionen
- Man kann den Zustand **modellbasiert** oder **axiomatisch** beschreiben.

Speichermodelle I: Konkret (Compiler)

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in konkretes **Speicherlayout**:



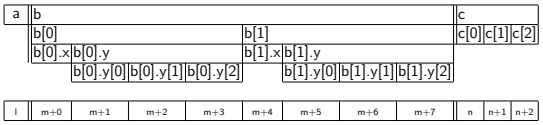
Denotation von $b[0].y[1]$ ist 3

Speichermodelle II: Abstrakt (C-Standard)

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in abstraktes **Speicherlayout**:



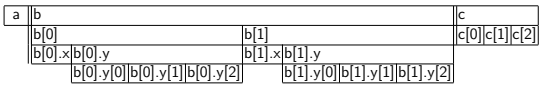
Denotation von $b[0].y[1]$ ist $m + 3$, mit m **unbestimmte** Adresse

Speichermodelle III: Symbolisch

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in symbolische Adressen:



Denotation von $b[0].y[1]$ ist $m[0].y[1]$, mit m unbestimmte Adresse

Abstrakte Zeigerarithmetik

- Adressen sind ein abstrakter Datentyp **Loc** so dass:
  - Es gibt **unbestimmte** Adressen
  - Operation *off* addiert Offset (Feldzugriff)
  - Operation *fld* selektiert Feld (**struct**)
  - Problem: Gleichheit und Ungleichheit

$$off : \mathbf{Loc} \to \mathbf{Z} \to \mathbf{Loc} \qquad fld : \mathbf{Loc} \to \mathbf{Idt} \to \mathbf{Loc}$$

$$off(l, 0) = l \qquad fld(l, f) \neq l$$
$$off(off(l, a), b) = off(l, a + b) \qquad fld(l, f) = fld(l, g) \Longrightarrow f = g$$
$$off(l, a) = l \Longrightarrow a = 0 \qquad fld(l, f) = fld(m, f) \Longrightarrow l = m$$
$$off(l, a) = off(l, b) \Longrightarrow a = b \qquad f \neq g \Longrightarrow fld(l, f) \neq fld(m, g)$$

Arbeitsblatt 13.1: Jetzt mit Zeigern!

Hier eine weitere Folge von Deklarationen:

```
int *a[1];
struct {
  int p[2];
  struct {
    int x;
    int y; } q[2];
  } b;
```

- Skizziert hier das Speichermodell — konkret, abstrakt, symbolisch.
- Welches sind die jeweiligen Adressen (**Loc**)?
- Was sind die Denotationen für a[1], b.p[1], b.q[0], b.q[1].y?
- Welche davon sind definiert/undefiniert?

Arbeitsblatt 13.2: Jetzt mit noch mehr Zeigern!

Hier eine weitere Folge von Deklarationen:

```
int *a[1];
struct {
  int p[2];
  struct {
    int x;
    int y; } *q[2];
  } b;
```

- Skizziert hier das Speichermodell — konkret, abstrakt, symbolisch.
- Welches sind die jeweiligen Adressen (**Loc**)?
- Was sind die Denotationen für a[1], b.p[1], b.q[0], (*b.q[1]).y?
- Welche davon sind definiert/undefiniert?

Axiomatisches Zustandsmodell

- Der Zustand ist ein abstrakter Datentyp Σ mit zwei Operationen und folgenden Gleichungen:

$$\begin{aligned} \mathit{read} &: \Sigma \to \mathbf{Loc} \rightharpoonup \mathbf{V} \\ \mathit{upd} &: \Sigma \to \mathbf{Loc} \to \mathbf{V} \rightharpoonup \Sigma \\ \mathbf{V} &\stackrel{\mathit{def}}{=} \mathbb{Z} + \mathbf{Loc} \end{aligned}$$

$$\begin{aligned} \mathit{read}(\mathit{upd}(\sigma, l, v), l) &= v \\ l \neq m \Longrightarrow \mathit{read}(\mathit{upd}(\sigma, l, v), m) &= \mathit{read}(\sigma, m) \\ \mathit{upd}(\mathit{upd}(\sigma, l, v), l, w) &= \mathit{upd}(\sigma, l, w) \\ l \neq m \Longrightarrow \mathit{upd}(\mathit{upd}(\sigma, l, v), m, w) &= \mathit{upd}(\mathit{upd}(\sigma, m, w), l, v) \end{aligned}$$

- 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*:

$$\begin{aligned} \mathit{fresh}(\sigma) &\notin \mathit{dom}(\sigma) \\ \mathit{dom}(\mathit{rem}(\sigma, l)) &= \mathit{dom}(\sigma) \setminus \{l\} \\ l \neq m \Longrightarrow \mathit{read}(\mathit{rem}(\sigma, l), m) &= \mathit{read}(\sigma, m) \end{aligned}$$

Erweiterung der Semantik: Umgebung

- Für Funktionen brauchten wir eine **Umgebung** (Environment):

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

- Diese muss erweitert werden für Variablen:

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup ([\![\mathbf{FunDef}]\!] \uplus \mathbf{Loc})$$

  - Insbesondere: gleicher Namensraum für Funktionen und Variablen (*C99 Standard*, §6.2.3)

Kurze Frage

- Wieso modellieren wir **Loc** nicht als Datentyp (so wie bisher):

$$l ::= \mathbf{Idt} \mid l[\mathbf{Z}] \mid l.\mathbf{Idt}$$

  Dann wäre $\mathit{off}(l, n) \stackrel{\mathit{def}}{=} l[n]$, $\mathit{fld}(l, i) \stackrel{\mathit{def}}{=} l.i$.
- $[\![a]\!]$ wäre immer a. Damit funktionieren drei Dinge nicht:
  1. Wir können globale nicht von lokalen Variablen unterscheiden.
  2. Beim rekursiven Aufruf wird keine neue Instanz erzeugt.
  3. Generell funktioniert call-by-reference nicht, z.B.

```
void f(int *x)
{
 int a;
 a= *x;
}
```

```
void g()
{
 int a;
 f(&a);
}
```


Erweiterung der Semantik: Problem

- 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

Erweiterung der Semantik: Lexp

$$[\![-]\!]_{\mathcal{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$\begin{aligned} [\![x]\!]_{\mathcal{L}}\ \Gamma &= \{(\sigma, \Gamma(x)) \mid \sigma \in \Sigma\} \\ [\![\mathit{lexp}[a]]\!]_{\mathcal{L}}\ \Gamma &= \{(\sigma, \mathit{off}(l, i)) \mid (\sigma, l) \in [\![\mathit{lexp}]\!]_{\mathcal{L}}\ \Gamma, (\sigma, i) \in [\![a]\!]_{\mathcal{A}}\ \Gamma\} \\ [\![\mathit{lexp}.f]\!]_{\mathcal{L}}\ \Gamma &= \{(\sigma, \mathit{fld}(l, f)) \mid (\sigma, l) \in [\![\mathit{lexp}]\!]_{\mathcal{L}}\ \Gamma\} \\ [\![*e]\!]_{\mathcal{L}}\ \Gamma &= [\![e]\!]_{\mathcal{A}}\ \Gamma \end{aligned}$$

Erweiterung der Semantik: Aexp(1)

$$[\![-]\!]_{\mathcal{A}} : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned} [\![n]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbb{N} \\ [\![e]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, \mathit{read}(\sigma, l)) \mid (\sigma, l) \in [\![e]\!]_{\mathcal{L}}\ \Gamma\} \\ &\qquad e \in \mathbf{Lexp} \text{ und } e \text{ kein Array-Typ} \\ [\![e]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in [\![e]\!]_{\mathcal{L}}\ \Gamma\} \\ &\qquad e \in \mathbf{Lexp} \text{ und } e \text{ ist Array-Typ} \\ [\![\&e]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in [\![e]\!]_{\mathcal{L}}\ \Gamma\} \end{aligned}$$

Erweiterung der Semantik: Aexp(2)

$$[\![-]\!]_{\mathcal{A}} : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned} [\![a_0 + a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\ [\![a_0 - a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\ [\![a_0 * a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\ [\![a_0 / a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma \\ &\qquad \wedge\, n_1 \neq 0\} \end{aligned}$$

Erweiterung der Semantik: Stmt

$$[\![x = e]\!]_{\mathcal{C}}\ \Gamma = \{(\sigma, upd(\sigma, l, a)) \mid (\sigma, l) \in [\![x]\!]_{\mathcal{L}}\ \Gamma \\ \wedge (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\ \Gamma\}$$

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

Arbeitsblatt 13.3: Pop-Quiz

Gegeben folgende Funktionen:

```
int f(int *x)
{
  int a;
  a= *x;
  *x= a+1;
  return a;
}
```

```
int a[3] = {0, 0, 0};
void g()
{
  int x= 1;
  a[x]= f(&x);
}
```

Was ist der Wert des Feldes `a` am Ende von `g`?

1. a == {0, 0, 1}
2. a == {0, 0, 2}
3. a == {0, 1, 0}
4. a == {0, 2, 0}

Arbeitsblatt 13.4: Kurze Semantik

Gegeben folgende Deklarationen:

```
struct {
  int x;
  int y; } p[5];
int a;
```

mit folgender Umgebung

$$\Gamma \stackrel{def}{=} \langle p \mapsto l_1, a \mapsto l_2 \rangle, l_1 \neq l_2$$

Berechnet die denotationale Semantik von

```
a= a+ p[3].x;
```


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*:

  $\mathbf{Lexp}_s$ $l ::= \ldots \mid \sout{*a}$
  $\mathbf{Assn}_s$ $b ::= \ldots$
  $\mathbf{Aexp}_s$ $a ::= read(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid l \mid \sout{\&l} \mid \ldots \mid e\ @\mathbf{pre} \mid \ldots$
  $\mathbf{State}$ $S ::= StateVar \mid upd(S, l, e)$
- Zustandsvariablen $StateVar$:
  - Aktueller Zustand σ, Vorzustand ρ_{old}, Zwischenzustände $\rho_0, \rho_1, \rho_2, \ldots$
- Explizite Zustandsprädikate enthalten kein $*$ oder &
- Im Gegensatz zur Semantik rechnen wir mit **symbolischen Namen**
- Damit Semantik:

$$[\![.]\!]_{\mathcal{B}sp} : \mathbf{Env} \to \mathbf{Assn}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B}$$
$$[\![.]\!]_{\mathcal{A}sp} : \mathbf{Env} \to \mathbf{Aexp}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}$$

Hoare-Triple

$$\Gamma \models \{P\}\, c\, \{Q \mid R\}$$

- $P, Q, R \in \mathbf{Assn}_s$ sind **explizite Zustandsprädikate**
- Deklarationen (**Decl**) allozieren für jede Variable eine Location (*fresh*), und ordnen diese in Γ dem Namen zu.
- Gültigkeit von Hoare-Tripeln (partielle, totale Korrektheit) wie vorher

Floyd-Hoare-Kalkül

Alte Regel

$$\overline{\Gamma \vdash \{\Gamma\}\, Q[upd(\sigma, x, e)/\sigma]\, \{x = e \mid Q\}}$$

- 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

$$(-)^\dagger : \mathbf{Lexp} \to \mathbf{Lexp}_s$$
$$i^\dagger = i \quad (i \in \mathbf{Idt})$$
$$I.id^\dagger = I^\dagger.id$$
$$I[e]^\dagger = I^\dagger[e^\#]$$
$$*I^\dagger = I^\#$$

$$(-)^\# : \mathbf{Aexp} \to \mathbf{Aexp}_s$$
$$e^\# = read(\sigma, e^\dagger) \quad (e \in \mathbf{Lexp})$$
$$n^\# = n$$
$$v^\# = v \quad (v \text{ logische Variable})$$
$$(\&e)^\# = e^\dagger$$
$$(e_1 + e_2)^\# = {e_1}^\# + {e_2}^\#$$
$$\backslash\mathbf{result}^\# = \backslash\mathbf{result}$$
$$e\,\mathbf{@pre}^\# = e\,\mathbf{@pre}$$

Angepasste Regeln des Hoare-Kalküls

$$\frac{}{\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, x^\dagger, e^\#)/\sigma]\ \{x = e \mid Q\}}$$

$$\frac{}{\Gamma \vdash \{\Gamma\}\ Q[e^\#/\backslash\mathbf{result}]\ \{\mathbf{return}\ e \mid P\}}$$

Insbesondere gilt (`int x, y, *z;`):

- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, x, read(\sigma, y))/\sigma]\ \{x = y \mid Q\}$
- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, y, x)/\sigma]\ \{y = \&x \mid Q\}$
- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, read(\sigma, z), read(\sigma, x))/\sigma]\ \{*z = x \mid Q\}$

Arbeitsblatt 13.5: Ein kurzes Beispiel

Betrachtet folgendes Beispiel:

```
void foo(){
 int x, y, z;
 x= 1;
 z= x;
 y= x;
 z= 5;
 // {0 < y}
}
```

1. Konvertiert das Prädikat $0 < y$ in ein explizites Zustandsprädikat.
2. Berechnet (rückwärts) die jeweils gültigen Zwischenzustände.
3. Vereinfacht nach jedem Schritt die Zwischenzustände.

Ein Beispiel mit Zeigern

```
void foo(){
 int x, y, *z;
 z= &x;
 x= 0;
 *z= 5;
 y= x;
 // {0 < y}
```


Ein Beispiel mit Zeigern

```
void foo(){
 int x, y, *z;

 /** { 0< 5 } */
 /** { 0< read(upd(...                    , x                    , 5), x) } */
 /** { 0< read(upd(upd(upd(s, z, x), x, 0), x                    , 5), x) } */
 /** { 0< read(upd(upd(upd(s, z, x), x, 0), read(upd(s, z, x), z), 5), x) } */
 z= &x;
 /** { 0< read(upd(upd(s, x, 0), read(s, z), 5), x) } */
 /** { 0< read(upd(upd(s, x, 0), read(    s,         z), 5), x) } */
 /** { 0< read(upd(upd(s, x, 0), read(upd(s, x, 0), z), 5), x) } */
 x= 0;
 /** { 0< read(upd(s, read(s, z), 5), x) } */
 *z= 5;
 /** { 0< read(s, x) } */
 /** { 0<              read(s, x)      } */
 /** { 0< read(upd(s, y, read(s, x)), y) } */
 y= x;
 /** { 0< read(s, y) } */
}
```


Ein problematisches Beispiel

```
void foo(int *p) {
 int x;

 // {read(upd(upd(s, x, 7), read(s, p), 99), x) = 7}
 // {read(upd(upd(s, x, 7), read(upd(s, x, 7), p), 99), x) = 7}
 x= 7;
 // {read(upd(s, read(s, p), 99), x) = 7}
 *p= 99;
 // {read(s, x) = 7}
 // {x = 7}
}
```

- Können **weder** beweisen, dass $read(s, p) = x$ **noch** $read(s, p) \neq x$
- 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

```
int findmax(int a[], int a_len)
  /** pre  \array(a, a_len) ∧ 0 < 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 zu Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger ein Feld ist.
- $\backslash\mathbf{array}(a, n)$ bedeutet: a ist ein Feld der Länge n, d.h.

$$\backslash\mathbf{array}(a, n) \iff (\forall i.\, 0 \leq i < n \implies \backslash\mathbf{valid}(a[i]))$$

- Gültigkeit kann abgeleitet werden:

$$\frac{\text{x= \&e}}{\backslash\text{valid(*x)}} \qquad \frac{\backslash\text{array(a, n)} \quad 0 \leq i \quad i < n}{\backslash\text{valid(a[i])}}$$

Was noch fehlt...

- **Vorwärtsrechung** mit expliziten Zustandsprädikaten.
  - Statt Existenzquantoren über Variablenwerte **unbestimmte Zwischenzustände** $\rho_1, \rho_2, \ldots$:

$$\frac{\rho_i \notin FV(P)}{\Gamma \vdash \{\Gamma\}\ P\,\{x = e \mid P[\rho_i/\sigma] \wedge \sigma = \mathit{upd}(\rho_i, x^\dagger[\rho_i/\sigma], e^\#[\rho_i/\sigma])\}}$$

  - Zwischenzustände sind **existenzquantifiziert**, d.h. das Prädikat gilt für **irgendeinen** Zustand ρ_i (aber für alle σ).
- Schwächste **Vorbedingung** und stärkste **Nachbedingung**:
  - Ergibt sich aus den Hoare-Regeln.
  - Erfordert durchgängige und agressive **Vereinfachung**.

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

Fahrplan

Korrekte Software: Grundlagen und Methoden
Vorlesung 14 vom 13.07.21
Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Was gibt's heute?

- Rückblick
- Ausblick
- Feedback
- Prüfungsvorbereitung

I. Rückblick

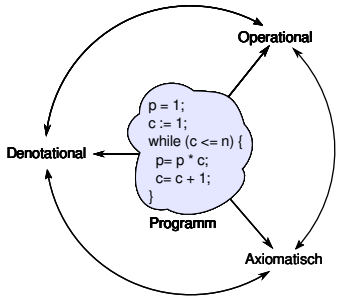
Semantik

- 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

Zusammenhang der Semantiken



Erweiterungen der Programmiersprache

- Für jede Erweiterung:
  - Wie modellieren wir semantisch?
  - Wie ändern sich die Regeln der Logik?

1. Erweiterung der Programmiersprache

- Strukturen und Felder
  - Lokationen: strukturierte Werte **Lexp**
  - Erweiterte Substitution in Zuweisungsregel
  - Sonstige Regeln bleiben

2. Erweiterung der Programmiersprache

- Prozeduren und Funktionen
  - Modellierung von **return**: Erweiterung zu $\Sigma \rightharpoonup \Sigma + \Sigma \times \mathbf{V}_U$
  - Spezifikation von Funktionen durch Vor-/Nachbedingungen
  - Spezifikation der Funktionen muss im Kontext stehen
  - Unterscheidung zwischen zwei Nachbedingungen
  - Regeln für den Funktionsaufruf

3. Erweiterung der Programmiersprache

- 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 $(-)^\dagger, (-)^\#$

II. 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 ϕ zu beweisen, versuchen wir $\neg\Phi$ 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:

$x \geq 0 \wedge y > 0 \Longrightarrow x \geq y$

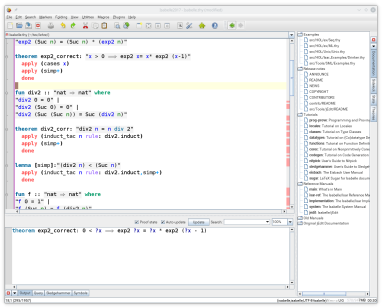
Input Z3:

```
(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

III. Prüfungsvorbereitung

Prüfungsvorbereitung

- Mündliche Modulprüfung, 20– 30 Minuten
- Schwerpunkte:
  - **Verständnis** des Stoffes, weniger Folien auswendig lernen
  - Stoff der Vorlesung und Übungsblätter, weniger eure Lösungen
- Bewertung
  - Sicherheit/Beherrschung des Stoffes
  - *covered ground*

Struktur des Stoffes

1. Basisstoff:
   - Operationale Semantik
   - Denotationale Semantik
   - Floyd-Hoare-Kalkül
2. Erweiterungen:
   - Datentypen
   - Funktionen
   - Referenzen

Mögliche Fragen I

- Was haben wir in KSGM gemacht?
- Wie funktioniert die operationale Semantik und wozu?
- Wie funktioniert die denotationale Semantik und wozu? Was ist ein Fixpunkt, und wozu?
- Was bedeutet die Äquivalenz der Semantiken? Wie haben wir das bewiesen? Was ist der Unterschied zwischen struktureller und Regelinduktion?
- Was ist der Floyd-Hoare-Kalkül? Was bedeutet $\vdash \{P\}\, c\, \{Q\}$ und $\models \{P\}\, c\, \{Q\}$?
- Wieviele Regeln hat der Floyd-Hoare-Kalkül und warum?
- Wie beweisen wir die Korrektheit dieses Programmes?

Mögliche Fragen II

- Welche Probleme tauchen bei folgenden Erweiterungen der Programmiersprache auf, und wie behandeln wir sie:
  - Felder und Strukturen,
  - Funktionen und Funktionsaufrufe,
  - Referenzen.
- Was ist der Unterschied zwischen dem Kalkül vorwärts und rückwärts? Wie sind die Regeln?
- Wie funktioniert die Generierung von Verifikationsbedingungen?

IV. 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!