Korrekte Software: Grundlagen und Methoden

Vorlesung 1 vom 21.04.20

Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Organisatorisches

- Veranstalter:

Christoph Lüth	Serge Autexier
christoph.lueth@dfki.de	serge.autexier@dfki.de
MZH 4186[1], Tel. 59830[2]	Cartesium 1.49[1], Tel. 59834[2]

- Termine:
  - Dienstag, 12 – 14
  - Donnerstag, 8 – 10 ⟵ **Verlegen?**
- Webseite:

  `http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss20`

---

[1] Zur Zeit im Home-Office
[2] Wird weitergeleitet.

Online-Konzept in Corona-Zeiten

- Keine lange Vorlesung, lieber integrierte Veranstaltung
- Kürzere Vortragseinheiten (Folie/Lifestream), dazwischen *Arbeitsfragen* (Kurzübungen)
  - Kein asynchrones Angebot (Aufzeichung der Meetings?)
- Wöchentliche Übungsaufgaben zur Vertiefung
- Technisch:
  - Nutzung von GotoMeeting: `https://www.gotomeet.me/DFKI-BAALL/ksgmss20`
  - Fragen/Kurzübungen in CodiMD: `http://hackmd.informatik.uni-bremen.de/`
  - Übungsblätter als ausfüllbare PDFs.

Prüfungsform und Übungsbetrieb

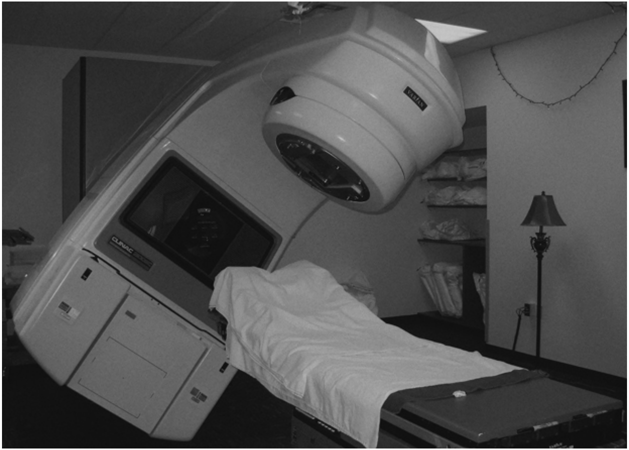
- 10 Übungsblätter (geplant)
- Bewertung:
  - A (sehr gut, 1.3) — nichts zu meckern, keine/kaum Fehler
  - B (gut, 2.3) — kleine Fehler, sonst gut
  - C (befriedigend, 3.3) — größere Fehler oder Mängel
  - Nicht bearbeitet — oder zu viele Fehler
- Prüfungsleistung:
  - Mündliche Prüfung
    - Einzelprüfung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

Arbeitsblatt 1.1: Jetzt seid ihr dran!

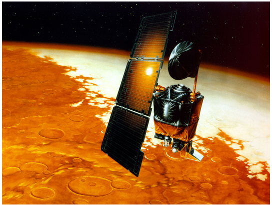
- Gruppiert euch in Gruppen zu drei Teilnehmenden! Nutzt dazu folgenden Doodle:
  `https://www.doodle.com/poll/utp4mg5yikbfta8d`
- Zu jeder Gruppe gibt es ein Arbeitsblatt:
  `https://hackmd.informatik.uni-bremen.de/s/SkVLK1Q_I`
- Auf diesem Arbeitsblatt bearbeitet ihr die Arbeitsfragen im Laufe des Kurses.
- Bitte nur in "eurem" Arbeitsblatt arbeiten
- Die Arbeitsblätter sind nicht notenrelevant.

Warum Korrekte Software?

Software-Disaster I: Therac-25



Software-Disasters II: Space



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

Software-Disaster III: AT&T (15.01.1990)

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

Software-Disaster IV: Airbus A400M

Sevilla, 09.05.2015

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/`

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
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Modellierung
- Spezifikation von Funktionen
- Referenzen und Speichermodelle
- Ausblick und Rückblick

Warum Semantik?

Idee

- Was wird hier berechnet?

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

Idee

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

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

Idee

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

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

Semantik von Programmiersprachen

Drei wesentliche Möglichkeiten:

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

Arbeitsblatt 1.3: Maschinen und Funktionen

Was genau kann man sich unter “abstrakten Maschine” vorstellen?

Betrachtet als Beispiel die Summe einer Liste von ganzen Zahlen:

- Wie könnte man eine abstrakte Maschine definieren, welche Listen von Zahlen summiert?
- Wie könnte man ein mathematisches Objekt definieren, welches Listen von Zahlen summiert?

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;  }
```

p	?
c	?
n	3

$\rightsquigarrow$

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; }
```

$$\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline \end{array} \rightsquigarrow \begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline \end{array} \rightsquigarrow$$

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; }
```

$$\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline \end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline \end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline \end{array} \leadsto$$

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; }
```

$$\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array} \leadsto$$

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;  }
```

p	?
c	?
n	3

$\leadsto$

p	1
c	?
n	3

$\leadsto$

p	1
c	1
n	3

$\leadsto$

p	1
c	1
n	3

$\leadsto$

p	1
c	2
n	3

$\leadsto$

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;  }
```

$$\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array} \leadsto$$

$$\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 2 \\ \hline n & 3 \\ \hline\end{array} \leadsto \begin{array}{|c|c|}\hline p & 2 \\ \hline c & 2 \\ \hline n & 3 \\ \hline\end{array} \leadsto$$

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; }
```

$$
\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
$$

$$
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 2 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 2 \\ \hline c & 2 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 2 \\ \hline c & 3 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
$$

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;  }
```

$$
\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
$$

$$
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 2 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 2 \\ \hline c & 2 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 2 \\ \hline c & 3 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
\begin{array}{|c|c|}\hline p & 6 \\ \hline c & 3 \\ \hline n & 3 \\ \hline\end{array}
\leadsto
$$

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;  }
```

$$
\begin{array}{|c|c|}\hline p & ? \\ \hline c & ? \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & ? \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 1 \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
$$

$$
\begin{array}{|c|c|}\hline p & 1 \\ \hline c & 2 \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 2 \\ \hline c & 2 \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 2 \\ \hline c & 3 \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 6 \\ \hline c & 3 \\ \hline n & 3 \\ \hline \end{array}
\rightsquigarrow
\begin{array}{|c|c|}\hline p & 6 \\ \hline c & 4 \\ \hline n & 3 \\ \hline \end{array}
$$

Arbeitsblatt 1.4: Operationale Semantik

Gegeben folgendes C0-Programm:

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

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

n	4
x	?

$\leadsto \ldots$

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
```

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

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
```

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

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

Denotationale Semantik

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

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

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

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

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

Denotationale Semantik

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

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

$$[\![p_1]\!](\sigma) = \sigma(p \mapsto 1)(c \mapsto 1)$$

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

$$[\![p_3]\!] = \mathit{fix}(\Gamma([\![c \le n]\!])([\![p_2]\!])) \circ [\![p_1]\!]$$

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

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

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)
```

(1) $n = 3$

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

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

(4) ???

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

Axiomatische Semantik

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

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

(1) $n = 3$

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

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

(4) $(p = 1 \land c = 1 \lor p = 1 \land c = 2 \lor p = 2 \land c = 3 \lor p = 6 \land c = 4) \land n = 3$

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

Axiomatische Semantik

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

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

(1) $n = 3$

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

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

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

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

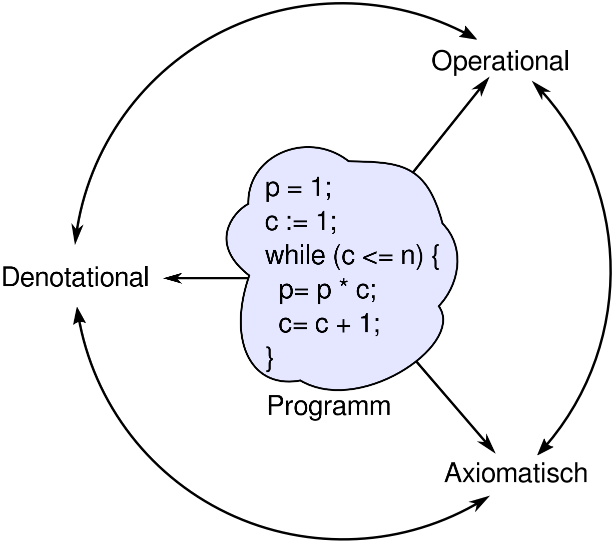
Arbeitsblatt 1.5: Zusicherungen

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?
- Was gilt stattdessen?

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 28.04.20

Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Zutaten

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

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

Unsere Programmiersprache

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

Ausbaustufe 1 kennt folgende Konstrukte:

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

C0: Ausdrücke und Anweisungen

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

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 **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*)
- $f[n/x]$ für den Update an der Stelle x mit dem Wert n:

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

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

$$\langle\rangle(x) = \bot$$

Arbeitsblatt 2.1: Jetzt seid ihr dran!

- In euren Gruppen-Arbeitsblättern unter `https://hackmd.informatik.uni-bremen.de/s/SkVLK1Q_I` 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:
  - A $\langle x \mapsto 1, a \mapsto 3\rangle$
  - B $\langle x \mapsto y, b \mapsto 6\rangle$
  - C $\langle x \mapsto y, b \mapsto 6, y \mapsto 2\rangle$
  - D $\langle x \mapsto 3, b \mapsto 6, y \mapsto 2\rangle$
- Update von Zuständen:
  - A $\langle x \mapsto 1, a \mapsto 3\rangle[1/y] := ??$
  - B $\langle x \mapsto 1, a \mapsto 3\rangle[3/x] := ??$
  - C $\langle x \mapsto 1, a \mapsto 3\rangle[3/x][y/1][4/x] := ??$

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:

  A $\langle x \mapsto 1, a \mapsto 3\rangle \;\; +$

  B $\langle x \mapsto y, b \mapsto 6\rangle \;\; -$

  C $\langle x \mapsto y, b \mapsto 6, y \mapsto 2\rangle \;\; -$

  D $\langle x \mapsto 3, b \mapsto 6, y \mapsto 2\rangle \;\; +$

- Update von Zuständen:

  A $\langle x \mapsto 1, a \mapsto 3\rangle[1/y] := \langle x \mapsto 1, a \mapsto 3, y \mapsto 1\rangle$

  B $\langle x \mapsto 1, a \mapsto 3\rangle[3/x] := \langle x \mapsto 3, a \mapsto 3\rangle$

  C $\langle x \mapsto 1, a \mapsto 3\rangle[3/x][y/1][4/x] := \langle x \mapsto 4, y \mapsto 1, a \mapsto 3\rangle$

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Regeln:

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Regeln:

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

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

Regelschreibweise vs. Funktionen

Sei $\mathsf{Int+} = \mathsf{Int} \cup \{\perp\}$

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

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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

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

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

Regelschreibweise vs. Funktionen

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

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

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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

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

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

Arbeitsblatt 2.2: Jetzt seid ihr dran!

- In euren Gruppen-Arbeitsblättern unter `https://hackmd.informatik.uni-bremen.de/s/SkVLK1Q_I` 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 $\perp$ könnt ihr einfach \bot schreiben.

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

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

Beispiel-Ableitungen

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

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

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

Beispiel-Ableitungen

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

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

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

Beispiel-Ableitungen

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

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

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

Operationale Semantik: Boolesche Ausdrücke

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

$$\langle b, \sigma \rangle \to_{Bexp} true \mid false \mid \perp$$

Regeln:

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

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

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

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

Operationale Semantik: Boolesche Ausdrücke

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

$$\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{true} \mid \mathit{false} \mid \perp$$

Regeln:

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{true}}{\langle !b, \sigma \rangle \rightarrow_{Bexp} \mathit{false}} \qquad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{false}}{\langle !b, \sigma \rangle \rightarrow_{Bexp} \mathit{true}} \qquad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle !b, \sigma \rangle \rightarrow_{Bexp} \perp}$$

$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \mathit{false}}{\langle b_1 \ \&\& \ b_2, \sigma \rangle \rightarrow_{Bexp} \mathit{false}} \qquad \frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle b_1 \ \&\& \ b_2, \sigma \rangle \rightarrow_{Bexp} \perp}$$

$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \mathit{true} \qquad \langle b_2, \sigma \rangle \rightarrow_{Bexp} t}{\langle b_1 \ \&\& \ b_2, \sigma \rangle \rightarrow_{Bexp} t}$$

Operationale Semantik: Boolesche Ausdrücke

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

$$\langle b, \sigma \rangle \rightarrow_{Bexp} true \mid false \mid \perp$$

Regeln:

$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} true}{\langle b_1 \ \&\& \ b_2, \sigma \rangle \rightarrow_{Bexp} true} \qquad \frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle b_1 \ \&\& \ b_2, \sigma \rangle \rightarrow_{Bexp} \perp}$$

$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} false \qquad \langle b_2, \sigma \rangle \rightarrow_{Bexp} t}{\langle b_1 \ || \ b_2, \sigma \rangle \rightarrow_{Bexp} t}$$

Operationale Semantik: Anweisungen

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

Beispiel:

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

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

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

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

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

Beispiel

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

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

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

(A)

$$\dfrac{\dfrac{\langle y - 1, \sigma_1\rangle \rightarrow_{Aexp} 1}{\langle y = y - 1, \sigma_1\rangle \rightarrow_{Stmt} \sigma_1[1/y] := \sigma_2} \quad \dfrac{\langle 2 * x, \sigma_2\rangle \rightarrow_{Aexp} 2}{\langle x = 2 * x, \sigma_2\rangle \rightarrow_{Stmt} \sigma_2[2/x] := \sigma_3}}{\langle y = y - 1; x = 2 * x, \sigma_1\rangle \rightarrow_{Stmt} \sigma_3}$$

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

(B)

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

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

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

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

$$
\begin{aligned}
\sigma_5 &= \sigma_4[4/x] = \sigma_3[0/y][4/x] = \sigma_2[2/x][0/y][4/x] \\
&= \sigma_1[1/y][2/x][0/y][4/x] = \langle y \mapsto 2\rangle[1/y][2/x][0/y][4/x] \\
&= \langle y \mapsto 0, x \mapsto 4\rangle
\end{aligned}
$$

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

Lineare, abgekürzte Schreibweise

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

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(w)  // ⟨y! = 0, ⟨y ↦ 2, x ↦ 1⟩⟩ →Bexp 1
|           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(w)  // ⟨y! = 0, ⟨y ↦ 2, x ↦ 1⟩⟩ →Bexp 1
|           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(w)  // ⟨y! = 0, ⟨y ↦ 1, x ↦ 2⟩⟩ →Bexp 1
|          y = y − 1;
|          // ⟨y ↦ 0, x ↦ 1⟩
|          x = 2 ∗ x;
|          // ⟨y ↦ 0, x ↦ 4⟩
while(w)  // ⟨y! = 0, ⟨y ↦ 0, x ↦ 2⟩⟩ →Bexp 0
// ⟨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;
}
```

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

Lineare, abgekürzte Schreibweise

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

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

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

- Wann sind sie gleich?

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

```
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?

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

```
A || (A && B)     und    A
```

Beweisen

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

Definition

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

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. $\langle b, \sigma\rangle \rightarrow_{Bexp} \perp$:

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

2. $\langle b, \sigma\rangle \rightarrow_{Bexp} \mathit{false}$:

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

Beweis II

3. $\langle b, \sigma \rangle \rightarrow_{Bexp} true$:

   1. $\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'$

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

   2. $\langle c, \sigma \rangle \rightarrow_{Stmt} \perp$

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

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 05.05.20
Denotationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

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

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

- **Denotationale Semantik:**

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

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

Denotationale Semantik — Motivation

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

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

oder

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

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

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** $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$

Denotierende Funktionen

- Arithmetische Ausdrücke:

  $a \in \textbf{Aexp}$ denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbb{Z}$

- Boolsche Ausdrücke:

  $b \in \textbf{Bexp}$ denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbb{B}$

- Anweisungen:

  $c \in \textbf{Stmt}$ denotiert eine partielle Funktion $\Sigma \rightharpoonup \Sigma$

Denotat von Aexp

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

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

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**:

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 $x \in \mathbf{Idt}$.
  Sei $a \equiv x$, dann $v_1 = \sigma(x) = v_2$.

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 $x \in \mathbf{Idt}$.
  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 $(\sigma, n_i) \in [\![a_i]\!]_{\mathcal{A}}, (\sigma, n'_i) \in [\![a_i]\!]_{\mathcal{A}}$ dann $n_i = n'_i$.
  Dann $v_1 = (\sigma, n_1 + n_2)$ mit $(\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, n_2) \in [\![a_2]\!]_{\mathcal{A}}$, und $v_2 = n'_1 + n'_2$ mit $(\sigma, n'_1) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, n'_2) \in [\![a_2]\!]_{\mathcal{A}}$. Aus der Annahme folgt $n_1 = n'_1$ und $n_2 = n'_2$, deshalb $v_1 = v_2$.

$\square$

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 3.1: 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?

Denotat von Bexp

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

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

Denotat von Bexp

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

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

Kompositionalität und Striktheit

Lemma (Partielle Funktion)

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

- Beweis analog zu $[\![-]\!]_{\mathcal{A}}$.
- Ist $[\![-]\!]_{\mathcal{B}}$ strikt?

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 $[\![b_1 \;\&\&\; b_2]\!]_{\mathcal{B}}(\sigma) = [\![b_1]\!]_{\mathcal{B}}(\sigma) = \mathit{false}$

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:

$$\begin{array}{ccc} \perp \wedge\, a = \perp & \mathit{false} \wedge a = \mathit{false} & \mathit{true} \wedge a = a \\ \perp \vee\, a = \perp & \mathit{true} \vee a = \mathit{true} & \mathit{false} \vee a = a \end{array}$$

Arbeitsblatt 3.2: Semantik II

Wir üben noch einmal die Nichtstrikheit. Gegeben $s = \langle x \mapsto 7 \rangle$ und
$b = (7 == x) \,||\, (x/0 == 1)$
Berechnenen Sie die Semantik als Relation in der Notation von oben:

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

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

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

Denotationale Semantik von Anweisungen

- Zuweisung: punktuelle Änderung des Zustands $\sigma \mapsto \sigma[n/x]$
- 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?

Denotationale Semantik von Anweisungen

- Zuweisung: punktuelle Änderung des Zustands $\sigma \mapsto \sigma[n/x]$
- 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 3.3: Komposition von Relationen

Zur Übung: betrachten Sie folgende Relationen:

$$R = \{(1,7),(2,3),(3,9),(4,3)\}$$
$$S = \{(1,0),(2,0),(3,1),(4,7),(5,9),(7,3),(8,15)\}$$

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

Denotat von Stmt

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

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

Denotat von Stmt

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

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

Aber was ist

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

Denotationale Semantik von while

- Sei $w \equiv \textbf{while}\ (b)\ c$ (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?

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?

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:

$$\begin{aligned} \Gamma^0(s) &\stackrel{def}{=} \emptyset \\ \Gamma^{i+1}(s) &\stackrel{def}{=} \Gamma(\Gamma^i(s)) \end{aligned}$$

- Dann ist

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

- 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 (s ist ein Denotat):

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

- b und c sind Parameter von Γ
- Dann ist

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

Denotation für Stmt

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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

s
-2
-1
0
1

Der Fixpunkt bei der Arbeit (I)

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

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

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

s	$\Gamma^0(s)$
-2	$\perp$
-1	$\perp$
0	$\perp$
1	$\perp$

Der Fixpunkt bei der Arbeit (I)

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

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

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

s	$\Gamma^0(s)$	$\Gamma^1(s)$
-2	$\perp$	$\Gamma^0(s[-1/x]) = \perp$
-1	$\perp$	$\Gamma^0(s[0/x]) = \perp$
0	$\perp$	0
1	$\perp$	1

Der Fixpunkt bei der Arbeit (I)

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

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

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

s	$\Gamma^0(s)$	$\Gamma^1(s)$	$\Gamma^2(s)$
-2	$\perp$	$\Gamma^0(s[-1/x]) = \perp$	$\Gamma^1(s[-1/x]) = \perp$
-1	$\perp$	$\Gamma^0(s[0/x]) = \perp$	$\Gamma^1(s[0/x]) = 0$
0	$\perp$	0	0
1	$\perp$	1	1

Der Fixpunkt bei der Arbeit (I)

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

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

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	$\perp$	$\Gamma^0(s[-1/x]) = \perp$	$\Gamma^1(s[-1/x]) = \perp$	$\Gamma^2(s[-1/x]) = 0$
-1	$\perp$	$\Gamma^0(s[0/x]) = \perp$	$\Gamma^1(s[0/x]) = 0$	$\Gamma^2(s[0/x]) = 0$
0	$\perp$	0	0	0
1	$\perp$	1	1	1

Der Fixpunkt bei der Arbeit (II)

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

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

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

s

n

-1

0

1

2

3

4

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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	$\perp$	$\perp$	0	-1	0	-1	0	-1	0	-1	0	-1
0	$\perp$	$\perp$	0	0	0	0	0	0	0	0	0	0
1	$\perp$	$\perp$	$\perp$	$\perp$	1	0	1	0	1	0	1	0
2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	3	0	3	0	3	0
3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	6	0	6	0
4	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	10	0

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

s
n
-2
-1
0
1
2
3

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

s	$\Gamma^0(s)$	
n	x	n
-2	$\perp$	$\perp$
-1	$\perp$	$\perp$
0	$\perp$	$\perp$
1	$\perp$	$\perp$
2	$\perp$	$\perp$
3	$\perp$	$\perp$

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

s	$\Gamma^0(s)$		$\Gamma^1(s)$	
n	x	n	x	n
-2	$\perp$	$\perp$	$\perp$	$\perp$
-1	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	$\perp$	0	0
1	$\perp$	$\perp$	$\perp$	$\perp$
2	$\perp$	$\perp$	$\perp$	$\perp$
3	$\perp$	$\perp$	$\perp$	$\perp$

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

s	$\Gamma^0(s)$		$\Gamma^1(s)$		$\Gamma^2(s)$	
n	x	n	x	n	x	n
-2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
-1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	$\perp$	0	0	0	0
1	$\perp$	$\perp$	$\perp$	$\perp$	1	0
2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

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	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
-1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	$\perp$	0	0	0	0	0	0	0	0
1	$\perp$	$\perp$	$\perp$	$\perp$	1	0	1	0	1	0
2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	3	0	3	0
3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	6	0

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

s
-2
-1
0
1
2
3

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Arbeitsblatt 3.4: 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
```

Der Fixpunkt bei der Arbeit (V)

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

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

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

s	
n	i
0	0
0	1
1	0
1	1
1	2
2	0
2	1
2	2
2	3

Der Fixpunkt bei der Arbeit (V)

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

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

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

s		$\Gamma^0(s)$		
n	i	n	i	x
0	0	$\perp$	$\perp$	$\perp$
0	1	$\perp$	$\perp$	$\perp$
1	0	$\perp$	$\perp$	$\perp$
1	1	$\perp$	$\perp$	$\perp$
1	2	$\perp$	$\perp$	$\perp$
2	0	$\perp$	$\perp$	$\perp$
2	1	$\perp$	$\perp$	$\perp$
2	2	$\perp$	$\perp$	$\perp$
2	3	$\perp$	$\perp$	$\perp$

Der Fixpunkt bei der Arbeit (V)

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

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

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

s		$\Gamma^1(s)$		
n	i	n	i	x
0	0	$\perp$	$\perp$	$\perp$
0	1	0	1	x
1	0	$\perp$	$\perp$	$\perp$
1	1	$\perp$	$\perp$	$\perp$
1	2	1	2	x
2	0	$\perp$	$\perp$	$\perp$
2	1	$\perp$	$\perp$	$\perp$
2	2	$\perp$	$\perp$	$\perp$
2	3	2	3	x

Der Fixpunkt bei der Arbeit (V)

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

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

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

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

Der Fixpunkt bei der Arbeit (V)

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

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

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

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

Der Fixpunkt bei der Arbeit (V)

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

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

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

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
0	0	$\perp$	$\perp$	$\perp$	0	1	x	0	1	x	0	1	x
0	1	0	1	x	0	1	x	0	1	x	0	1	x
1	0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	1	2	$x+1$	1	2	$x+1$
1	1	$\perp$	$\perp$	$\perp$	1	2	$x+1$	1	2	$x+1$	1	2	$x+1$
1	2	1	2	x	1	2	x	1	2	x	1	2	x
2	0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	2	3	$x+3$
2	1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	2	3	$x+3$	2	3	$x+3$
2	2	$\perp$	$\perp$	$\perp$	2	3	$x+2$	2	3	$x+2$	2	3	$x+2$
2	3	2	3	x	2	3	x	2	3	x	2	3	x

Weitere Eigenschaften der denotationalen Semantik

Lemma (Partielle Funktion)

$[\![-]\!]_{\mathcal{C}}$ *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 $\mathit{fix}(\Gamma)$ rechtseindeutig.
- Eigenschaften der Iteration:
  - Sei $w \equiv \textbf{while}\ (b)\ c$
  - Dann

$$[\![w]\!]_{\mathcal{C}} = [\![\textbf{if}\ (b)\ \{c; w\}\ \textbf{else}\ \{\ \}]\!]_{\mathcal{C}} \tag{1}$$

$$(\sigma, \sigma') \in [\![w]\!]_{\mathcal{C}} \Longrightarrow (\sigma', \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \tag{2}$$

Beweis (1)

Zu zeigen: $\llbracket w \rrbracket_{\mathcal{C}} = \llbracket \textbf{if } (b)\ \{c; w\}\ \textbf{else}\ \{\,\} \rrbracket_{\mathcal{C}}$

$$
\begin{aligned}
\llbracket \text{if } (b)\ \{c;w\}\ \text{else}\ \{\} \rrbracket_{\mathcal{C}} &= \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma,\sigma') \in \llbracket c;w \rrbracket_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma,\sigma') \mid (\sigma,\mathit{false}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma,\sigma') \in \llbracket \{\} \rrbracket_{\mathcal{C}}\} \\
&= \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma,\sigma') \in \llbracket c \rrbracket_{\mathcal{C}} \circ \llbracket w \rrbracket_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma,\sigma') \mid (\sigma,\mathit{false}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma,\sigma') \in \mathit{Id}_{\Sigma}\} \\
&= \{(\sigma,\sigma') \mid (\sigma,\mathit{true}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma,\sigma') \in \llbracket c \rrbracket_{\mathcal{C}} \circ \llbracket w \rrbracket_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma,\sigma) \mid (\sigma,\mathit{false}) \in \llbracket b \rrbracket_{\mathcal{B}}\} \\
&= \Gamma(\llbracket w \rrbracket_{\mathcal{C}}) \\
&= \Gamma(\mathit{fix}(\Gamma)) = \mathit{fix}(\Gamma) = \llbracket w \rrbracket_{\mathcal{C}} \qquad \square
\end{aligned}
$$

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 12/14.05.20

Äquivalenz der Operationalen und Denotationalen Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma \rangle \rightarrow_{Aexp} n$	**Denotational** $[\![a]\!]_{\mathcal{A}}$
a_1 / a_2	$$\frac{\begin{array}{c}\langle a_1, \sigma \rangle \rightarrow_{Aexp} n \\ \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \\ m \neq 0 \qquad m, n \neq \bot\end{array}}{\langle a_1 \circ a_2, \sigma \rangle \rightarrow_{Aexp} n \circ^I m}$$ $$\frac{\begin{array}{c}\langle a_1, \sigma \rangle \rightarrow_{Aexp} n \\ \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \\ n = \bot, m = \bot \text{ oder } m = 0\end{array}}{\langle a_1 / a_2, \sigma \rangle \rightarrow_{Aexp} \bot}$$	$\{(\sigma, n/m) \mid \sigma \in \Sigma, (\sigma, n) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}, m \neq 0\}$

Äquivalenz operationale und denotationale Semantik

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

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

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

- 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

$$\begin{aligned} \forall x \in S. x \not\succ x & \qquad (\textit{irreflexiv}) \\ \forall x, y \in S. x \succ y \Rightarrow y \not\succ x & \qquad (\textit{assymetrisch}) \\ \forall x, y, z \in S. (x \succ y \wedge y \succ z) \Rightarrow x \succ z & \qquad (\textit{transitiv}) \end{aligned}$$

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

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

Induktionsprinzip

Noether'sche Induktion

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

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

	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

Arbeitsblatt 4.1: Übung zu struktureller Ordnung

Die strukturelle Ordnung auf arithmetischen Ausdrücken ist definiert als:

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

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. assymmetrisch und
  3. transitiv

  ist.

Besprechung

Argumentiert/beweist, die Relation "Teilausdruck-von" ist

1. irreflexiv Für Variablen und Zahlen gilt es nicht.

$$\begin{aligned}
&(a_1 \circ a_2) \text{ Teilausdruck-von} (a_1 \circ a_2) \\
\Leftrightarrow &(a_1 \circ a_2) = a_1 \vee (a_1 \circ a_2) \text{ Teilausdruck-von } a_1 \qquad \text{Widerspruch}
\end{aligned}$$

2. assymmetrisch

$$\begin{aligned}
&(a_1 \circ a_2) \text{ Teilausdruck-von} (a_1' \circ a_2') \\
\wedge &(a_1' \circ a_2') \text{ Teilausdruck-von} (a_1 \circ a_2) \\
\Leftrightarrow &[(a_1 \circ a_2) \text{ Teilausdruck-von } a_1' \\
&\ \vee (a_1 \circ a_2) \text{ Teilausdruck-von } a_2'] \\
&\ \wedge [(a_1' \circ a_2') \text{ Teilausdruck-von } a_1 \\
&\ \vee (a_1' \circ a_2') \text{ Teilausdruck-von } a_2]
\end{aligned}$$

Besprechung

Argumentiert/beweist, die Relation "Teilausdruck-von" ist

3. transitiv

$$a \text{ Teilausdruck-von}(a_1 \circ a_2) \wedge (a_1 \circ a_2) \text{ Teilausdruck-von}(a_1' \circ a_2')$$

$$\Leftrightarrow$$

1. Fall: $a = a_1 \vee a \text{ Teilausdruck-von } a_1 \Rightarrow a \text{ Teilausdruck-von}(a_1' \circ a_2')$
2. Fall: $a = a_2 \vee a \text{ Teilausdruck-von } a_2 \Rightarrow a \text{ Teilausdruck-von}(a_1' \circ a_2')$

Äquivalenz operationale und denotationale Semantik

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

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

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

- Beweis Prinzip?

Äquivalenz operationale und denotationale Semantik

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

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

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

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

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

Induktionsanfänge

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

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

- $a \equiv X \in \mathbf{Loc}$:

1. $X \in Dom(\sigma)$:

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

2. $X \notin Dom(\sigma)$:

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

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

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.

$$\begin{array}{ccc}
\langle a_1 + a_2, \sigma\rangle \to_{Aexp} m + n & & \\
\Updownarrow (\text{Def. } \langle .,. \rangle \to_{Aexp}.) & & \\
\langle a_1, \sigma\rangle \to_{Aexp} m & \xLeftrightarrow{\text{IA fuer } a_1} & (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}} \\
\& & & \& \\
\langle a_2, \sigma\rangle \to_{Aexp} n & \xLeftrightarrow{\text{IA fuer } a_2} & (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}} \\
& & \Updownarrow (Def.[\![.]\!]_{\mathcal{A}}) \\
& & (\sigma, m + n) \in [\![a_1 + a_2]\!]_{\mathcal{A}}
\end{array}$$

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

Induktionsschritte

- $a \equiv a_1 + a_2$: Induktionsannahme gilt für a_1 und a_2.

  2. Fall: $m = \perp$ oder $n = \perp$

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

  - Fall $n = \perp$.
    Aus Induktionsannahme folgt, dass $\langle a_1, \sigma \rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin Dom([\![a_1]\!]_{\mathcal{A}})$.
    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 $\sigma \notin Dom([\![a_1 + a_2]\!]_{\mathcal{A}})$.

  - Fall $n \neq \perp, m = \perp$: analog.

Beweis $\forall a \in \textbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle a, \sigma\rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$
$\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \quad \langle a, \sigma\rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin \textbf{\textit{Dom}}([\![a]\!]_{\mathcal{A}})$

Induktionsschritte

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

$$\begin{tikzcd}
\langle a_1/a_2, \sigma\rangle \rightarrow_{Aexp} m/n \arrow[d, Leftrightarrow, "(\text{Def. } \langle .,.\rangle \rightarrow_{Aexp} .)"] & \\
\langle a_1, \sigma\rangle \rightarrow_{Aexp} m \arrow[r, Leftrightarrow, "\text{IA fuer } a_1"] & (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}} \\
\& & \& \\
\langle a_2, \sigma\rangle \rightarrow_{Aexp} n \arrow[r, Leftrightarrow, "\text{IA fuer } a_2"] & (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}} \arrow[d, Leftrightarrow, "(Def.[\![.]\!]_{\mathcal{A}})"] \\
& (\sigma, m+n) \in [\![a_1/a_2]\!]_{\mathcal{A}}
\end{tikzcd}$$

Beweis $\forall a \in \textbf{Aexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle a, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$
$\wedge \quad \langle a, \sigma \rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin \textbf{\textit{Dom}}([\![a]\!]_{\mathcal{A}})$

Induktionsschritte

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

2. Fall:

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

- Fall $n = 0$.
  Aus Induktionsannahme folgt, dass $\langle a_2, \sigma \rangle \rightarrow_{Aexp} 0 \Leftrightarrow (\sigma, 0) \in [\![a_2]\!]_{\mathcal{A}}$.
  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 $\sigma \notin Dom([\![a_1/a_2]\!]_{\mathcal{A}})$.
- Fall $n = \perp, m = \perp$: analog wie bei $+$

q.e.d.

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

Äquivalenz operationale und denotationale Semantik

- Für alle $b \in \mathbf{Bexp}$, 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?

Äquivalenz operationale und denotationale Semantik

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

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

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

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

Beweis $\forall a \in \mathbf{Bexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}}$
$\wedge \quad \langle b, \sigma \rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin \textbf{\textit{Dom}}([\![b]\!]_{\mathcal{B}})$

Induktionsanfänge

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

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

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

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

Beweis $\forall a \in \mathbf{Bexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle b, \sigma\rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}}$
$\wedge \quad \langle b, \sigma\rangle \rightarrow_{Bexp} \bot \Leftrightarrow \sigma \notin \boldsymbol{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

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

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

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

$$
\begin{tikzcd}
\langle b_1 \&\& b_2, \sigma\rangle \rightarrow_{Bexp} \bot \arrow[d, Leftrightarrow, "(\text{Def. } \langle .,.\rangle \rightarrow_{Bexp} .)"] & \\
\langle b_1, \sigma\rangle \rightarrow_{Bexp} \bot \arrow[r, Leftrightarrow, "\text{IA fuer } b_1"] & \sigma \notin Dom([\![b_1]\!]_{\mathcal{B}}) \arrow[d, Rightarrow, "\text{Def. } [\![.]\!]_{\mathcal{B}}"'] \\
& \sigma \notin [\![b_1 \&\& b_2]\!]_{\mathcal{B}}
\end{tikzcd}
$$

Beweis $\forall a \in \textbf{Bexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle b, \sigma\rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}}$
$\wedge \quad \langle b, \sigma\rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin \textbf{\textit{Dom}}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

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

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

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

$$\begin{array}{lcl}
\langle b_1 \&\& b_2, \sigma\rangle \rightarrow_{Bexp} \mathit{false} & & \\
\Updownarrow \text{(Def. } \langle .,. \rangle \rightarrow_{Bexp} .) & & \\
\langle b_1, \sigma\rangle \rightarrow_{Bexp} \mathit{false} & \overset{\text{IA fuer } b_1}{\Longleftrightarrow} & (\sigma, \mathit{false}) \in [\![b_1]\!]_{\mathcal{B}} \\
& & \Downarrow \text{Def. } [\![.]\!]_{\mathcal{B}} \\
& & (\sigma, \mathit{false}) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}}
\end{array}$$

Beweis $\forall a \in \textbf{Bexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle b, \sigma\rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}}$
$\wedge \quad \langle b, \sigma\rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin \textbf{\textit{Dom}}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

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

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

Induktionsannahme gilt für b_1 und b_2.

- Fall $\langle b_1, \sigma\rangle \rightarrow_{Bexp} \mathit{true}, \langle b_2, \sigma\rangle \rightarrow_{Bexp} \mathit{false}$

$$\begin{array}{ccc}
\langle b_1\&\&b_2, \sigma\rangle \rightarrow_{Bexp} \mathit{false} & & \\
\Updownarrow \text{(Def. } \langle .,.\rangle \rightarrow_{Bexp}.) & & \\
\langle b_1, \sigma\rangle \rightarrow_{Bexp} \mathit{true} & \xLeftrightarrow{\text{IA fuer } b_1} & (\sigma, \mathit{true}) \in [\![b_1]\!]_{\mathcal{B}} \\
\& & & \& \\
\langle b_2, \sigma\rangle \rightarrow_{Bexp} \mathit{false} & \xLeftrightarrow{\text{IA fuer } b_2} & (\sigma, \mathit{false}) \in [\![b_2]\!]_{\mathcal{B}} \\
& & \Downarrow \text{Def. } [\![.]\!]_{\mathcal{B}} \\
& & (\sigma, \mathit{false}) \in [\![b_1\&\&b_2]\!]_{\mathcal{B}}
\end{array}$$

Beweis $\forall a \in \mathbf{Bexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}}$

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

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 $\langle b_1, \sigma \rangle \rightarrow_{Bexp} true, \langle b_2, \sigma \rangle \rightarrow_{Bexp} true$

$$\begin{array}{ccc}
\langle b_1 \&\& b_2, \sigma \rangle \rightarrow_{Bexp} true & & \\
\Updownarrow \text{(Def. } \langle .,. \rangle \rightarrow_{Bexp} .) & & \\
\langle b_1, \sigma \rangle \rightarrow_{Bexp} true & \xLeftrightarrow{\text{IA fuer } b_1} & (\sigma, true) \in [\![b_1]\!]_{\mathcal{B}} \\
\& & & \& \\
\langle b_2, \sigma \rangle \rightarrow_{Bexp} true & \xLeftrightarrow{\text{IA fuer } b_2} & (\sigma, true) \in [\![b_2]\!]_{\mathcal{B}} \\
& & \text{Def. } [\![.]\!]_{\mathcal{B}} \Updownarrow \\
& & (\sigma, true) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}}
\end{array}$$

Beweis $\forall a \in \mathbf{Bexp}.\forall n \in \mathbb{Z}.\forall \sigma. \quad \langle b, \sigma\rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in [\![b]\!]_{\mathcal{B}}$
$\wedge \quad \langle b, \sigma\rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin \mathbf{Dom}([\![b]\!]_{\mathcal{B}})$

Induktionsschritte

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

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

Induktionsannahme gilt für b_1 und b_2.

- Fall $\langle b_1, \sigma\rangle \rightarrow_{Bexp} \mathit{true}, \langle b_2, \sigma\rangle \rightarrow_{Bexp} \perp$

$$
\begin{array}{ccc}
\langle b_1 \&\& b_2, \sigma\rangle \rightarrow_{Bexp} \perp & & \\
\Updownarrow \text{(Def. } \langle .,. \rangle \rightarrow_{Bexp} .) & & \\
\langle b_1, \sigma\rangle \rightarrow_{Bexp} \mathit{true} & \overset{\text{IA fuer } b_1}{\Longleftrightarrow} & (\sigma, \mathit{true}) \in [\![b_1]\!]_{\mathcal{B}} \\
\& & & \& \\
\langle b_2, \sigma\rangle \rightarrow_{Bexp} \perp & \overset{\text{IA fuer } b_2}{\Longleftrightarrow} & \sigma \notin \mathit{Dom}([\![b_2]\!]_{\mathcal{B}}) \\
& & \Downarrow \text{Def. } [\![.]\!]_{\mathcal{B}} \\
& & \sigma \notin \mathit{Dom}([\![b_1 \&\& b_2]\!]_{\mathcal{B}})
\end{array}
$$

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

- $(\sigma, true) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} (\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$ und $(\sigma, true) \in [\![b_2]\!]_{\mathcal{B}}$
  - Siehe Folie 24
- $(\sigma, false) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}} \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow}$ $(\sigma, false) \in [\![b_1]\!]_{\mathcal{B}}$ oder
  $(\sigma, true) \in [\![b_1]\!]_{\mathcal{B}}$ und $(\sigma, false) \in [\![b_2]\!]_{\mathcal{B}}$
  - Siehe Folie 22 und 23
- $\sigma \notin Dom([\![b_1 \&\& b_2]\!]_{\mathcal{B}}) \overset{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} \sigma \notin Dom([\![b_1]\!]_{\mathcal{B}})$ oder $\sigma \notin Dom([\![b_2]\!]_{\mathcal{B}})$
  - Siehe Folie 21 und 25

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

Arbeitsblatt 4.2: Beweis Induktionsanfang

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

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

$$\begin{aligned} \forall a \in \textbf{Aexp}.\forall n \in \mathbb{Z}.\forall\sigma. \quad & \langle a, \sigma\rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in [\![a]\!]_{\mathcal{A}} \\ \wedge \quad & \langle b, \sigma\rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin Dom([\![a]\!]_{\mathcal{A}}) \end{aligned}$$

Beweis

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

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

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

$$\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp} true$$

$$\Updownarrow \text{(Def. } \langle .,.\rangle \rightarrow_{Bexp} .)$$

$$\langle a_1, \sigma\rangle \rightarrow_{Bexp} m \xLeftrightarrow{\text{IA fuer } a_1} (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}$$

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

$$\langle a_2, \sigma\rangle \rightarrow_{Bexp} m \xLeftrightarrow{\text{IA fuer } a_2} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$

$$\text{Def. } [\![.]\!]_{\mathcal{B}} \Updownarrow$$

$$(\sigma, true)[\![a_1 == a_2]\!]_{\mathcal{B}}$$

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

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

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

$$\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp} false$$

$$\Big\Updownarrow \text{(Def. } \langle .,. \rangle \rightarrow_{Bexp}\text{.)}$$

$$\langle a_1, \sigma\rangle \rightarrow_{Aexp} m \overset{\text{Lemma fuer } a_1}{\Longleftrightarrow} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$

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

$$\langle a_2, \sigma\rangle \rightarrow_{Aexp} n \overset{\text{Lemma fuer } a_2}{\Longleftrightarrow} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

$$\text{Def. } [\![.]\!]_{\mathcal{B}} \Big\Updownarrow$$

$$(\sigma, false) [\![a_1 == a_2]\!]_{\mathcal{B}}$$

Beweis
1. $\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp}$ ***false*** $\Leftrightarrow (\sigma, \textbf{\textit{false}}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
2. $\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp}$ ***true*** $\Leftrightarrow (\sigma, \textbf{\textit{true}}) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$
3. $\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin \textbf{\textit{Dom}}([\![a_1 == a_2]\!]_{\mathcal{B}})$

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

- Fall $\langle a_1, \sigma\rangle \rightarrow_{Bexp} \perp$:

$$
\begin{tikzcd}
\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp} \perp \arrow[d, Leftrightarrow, "(\text{Def. } \langle .,.\rangle \rightarrow_{Bexp}.)"] & \\
\langle a_1, \sigma\rangle \rightarrow_{Aexp} \perp \arrow[r, Leftrightarrow, "\text{Lemma fuer } a"] & \sigma \notin Dom([\![a_1]\!]_{\mathcal{A}}) \arrow[d, Rightarrow, "\text{Def. } [\![.]\!]_{\mathcal{B}}"'] \\
\& & \sigma \notin Dom([\![a_1 == a_2]\!]_{\mathcal{B}})
\end{tikzcd}
$$

Beweis

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

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

- Fall $\langle a_2, \sigma\rangle \rightarrow_{Bexp} \perp$:

$$
\begin{tikzcd}
\langle a_1 == a_2, \sigma\rangle \rightarrow_{Bexp} \perp \arrow[d, Leftrightarrow, "(\text{Def. } \langle .,. \rangle \rightarrow_{Bexp}.)"] & \\
\langle a_2, \sigma\rangle \rightarrow_{Aexp} \perp \arrow[r, Leftrightarrow, "\text{Lemma fuer } a"] & \sigma \notin Dom([\![a_2]\!]_{\mathcal{A}}) \arrow[d, Rightarrow, "\text{Def. } [\![.]\!]_{\mathcal{B}}"'] \\
\& & \sigma \notin Dom([\![a_1 == a_2]\!]_{\mathcal{B}})
\end{tikzcd}
$$

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

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

- $\sigma \notin Dom([\![a_1 == a_2]\!]_{\mathcal{B}}) \stackrel{\text{Def. } [\![.]\!]_{\mathcal{B}}}{\Longleftrightarrow} \sigma \notin Dom([\![a_1]\!]_{\mathcal{A}})$ oder $\sigma \notin Dom([\![a_2]\!]_{\mathcal{A}})$
  - Siehe die beiden Fälle auf den beiden vorangegangenen Folien.

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

Denotational $[\![c]\!]_{\mathcal{C}}$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \bot}{\langle c, \sigma \rangle \rightarrow_{Stmt} \bot}$$

if (b) c_0

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} true \quad \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma'}{\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'}$$

$$\{(\sigma, \sigma') | (\sigma, true) \in [\![b]\!]_{\mathcal{B}}, (\sigma, \sigma') \in [\![c_0]\!]_{\mathcal{C}}\}$$

else c_1

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} false \quad \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma'}{\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'}$$

$$\{(\sigma, \sigma') | (\sigma, false) \in [\![b]\!]_{\mathcal{B}}, (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}}\}$$

Operationale vs. denotationale Semantik

Operational **Denotational** $[\![c]\!]_{\mathcal{C}}$

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

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

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{false}}{\langle w, \sigma \rangle \rightarrow_{Stmt} \sigma} \qquad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \bot}{\langle w, \sigma \rangle \rightarrow_{Stmt} \bot} \qquad \mathit{fix}(\Gamma)$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{true} \quad \langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' \neq \bot \quad \langle w, \sigma' \rangle \rightarrow_{Stmt} \sigma''}{\langle w, \sigma \rangle \rightarrow_{Stmt} \sigma''}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{true} \quad \langle c, \sigma \rangle \rightarrow_{Stmt} \bot}{\langle w, \sigma \rangle \rightarrow_{Stmt} \bot}$$

mit

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

Äquivalenz operationale und denotationale Semantik

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

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

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

- $\Rightarrow$ 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:

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

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

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

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

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

Operationale Semantik: C0 Programme

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

Regeln:

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

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

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

Operationale Semantik: C0 Programme

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

Regeln:

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

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

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

Ableitungstiefe für Programme

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

$$\frac{\begin{array}{c}\vdots\\ \mathit{Prämisse}_1\end{array} \qquad \cdots \qquad \begin{array}{c}\vdots\\ \mathit{Prämisse}_n\end{array}}{\mathit{Conclusion}}$$

Operationale Semantik: C0 Programme

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

Regeln:

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

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

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

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

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

Programmstruktur

$\searrow$

$\searrow$

$\searrow$

$\rightarrow$

$\searrow$

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
$\dfrac{\langle c_1, \sigma\rangle \to_{Stmt} \sigma' \neq \bot \quad \langle c_2, \sigma'\rangle \to_{Stmt} \sigma'' \neq \bot}{\langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$	$\searrow$	$\searrow$
$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \qquad \langle c_1, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma'}$	$\searrow$	$\searrow$
$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{false} \qquad \langle c_2, \sigma\rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma'}$	$\searrow$	$\searrow$
$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \qquad \langle c, \sigma\rangle \to_{Stmt} \sigma' \quad \langle \textbf{while}\ (b)\ c, \sigma'\rangle \to_{Stmt} \sigma''}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \sigma''}$	$\rightarrow$	$\searrow$
$\dfrac{\langle b, \sigma\rangle \to_{Bexp} \mathit{true} \qquad \langle c, \sigma\rangle \to_{Stmt} \bot}{\langle \textbf{while}\ (b)\ c, \sigma\rangle \to_{Stmt} \bot}$	$\searrow$	$\searrow$

Äquivalenz operationale und denotationale Semantik

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

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

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

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

Äquivalenz operationale und denotationale Semantik

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

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

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

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

- $\Leftarrow$ Beweis Prinzip?

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

Induktionsanfang – Ableitungstiefe 1

- Fall $c \equiv x = a$:

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

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

$$\begin{tikzcd}
\langle x = a, \sigma\rangle \rightarrow_{Stmt} \sigma[m/x] \arrow[d, Leftrightarrow, "(\text{Def. } \langle .,.\rangle \rightarrow_{Stmt} .)"] & \\
\langle a, \sigma\rangle \rightarrow_{Aexp} m \in \mathbb{Z} \arrow[r, Leftrightarrow, "\text{Lemma fuer } a"] & (\sigma, m) \in [\![a]\!]_{\mathcal{A}} \arrow[d, Rightarrow, "\text{Def. } [\![.]\!]_{\mathcal{C}}"'] \\
& (\sigma, \sigma[m/x]) \in [\![x = a]\!]_{\mathcal{C}}
\end{tikzcd}$$

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

Induktionsanfang – Ableitungstiefe 1

- Fall $c \equiv x = a$:

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

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

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

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

Induktionsanfang – Ableitungstiefe 1

- Fall $c \equiv x = a$:

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

- Fall $c \equiv \{\}$: ...

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

Induktionsschritt:

- Fall $c \equiv \textbf{if}(b)\ c_1\ \textbf{else}\ c_2$:

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

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

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

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

Induktionsschritt:

- Fall $c \equiv \textbf{if}(b)\ c_1\ \textbf{else}\ c_2$:

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

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

$$\langle \textbf{if}(b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \rightarrow_{Stmt} \sigma'$$

$$\Updownarrow \text{(Def. } \langle .,.\rangle \rightarrow_{Stmt}.\text{)}$$

$$\langle b, \sigma\rangle \rightarrow_{Bexp} \mathit{false} \xLeftrightarrow{\text{Lemma fuer } b} (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}$$

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

$$\langle c_2, \sigma\rangle \rightarrow_{Stmt} \sigma' \xLeftrightarrow{\text{IH fuer } c_2} (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}}$$

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

$$(\sigma, \sigma') \in [\![\textbf{if}(b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}}$$

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

Induktionsschritt:

- Fall $c \equiv \textbf{if}(b)\ c_1\ \textbf{else}\ c_2$:

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

- Fall $\langle \sigma, b \rangle \rightarrow_{Bexp} \mathit{true}, \langle c_1, \sigma \rangle \rightarrow_{Stmt} \bot$:

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

$$\Big\Updownarrow (\text{Def. } \langle .,. \rangle \rightarrow_{Stmt} .)$$

$$\langle b, \sigma \rangle \rightarrow_{Bexp} \mathit{true} \xLeftrightarrow{\text{Lemma fuer } b} (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}$$

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

$$\langle c_1, \sigma \rangle \rightarrow_{Stmt} \bot \xLeftrightarrow{\text{IH fuer } c_1} \sigma \notin \mathit{Dom}([\![c_1]\!]_{\mathcal{C}})$$

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

$$\sigma \notin \mathit{Dom}([\![\textbf{if}(b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}})$$

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

Induktionsschritt:

- Fall $c \equiv \textbf{if}(b)\ c_1\ \textbf{else}\ c_2$:

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

  - Fall $\langle \sigma, b \rangle \rightarrow_{Bexp} \bot$:

$$\begin{tikzcd}
\langle \textbf{if}(b)\ c_1\ \textbf{else}\ c_2, \sigma \rangle \rightarrow_{Stmt} \bot \arrow[d, Leftrightarrow, "(\text{Def. } \langle .,. \rangle \rightarrow_{Stmt} .)"] & \\
\langle b, \sigma \rangle \rightarrow_{Bexp} \bot \arrow[r, Leftrightarrow, "\text{Lemma fuer } b"] & \sigma \notin \mathit{Dom}([\![b]\!]_{\mathcal{B}}) \arrow[d, Rightarrow, "\text{Def. } [\![.]\!]_{\mathcal{C}}"'] \\
& \sigma \notin \mathit{Dom}([\![\textbf{if}(b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}})
\end{tikzcd}$$

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

Induktionsschritt:

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

$$\langle \textbf{while}(b)\ c, \sigma\rangle \rightarrow_{Stmt} \sigma''$$

$$\Big\Updownarrow \text{(Def. } \langle .,.\rangle \rightarrow_{Stmt}.\text{)}$$

$$\langle b, \sigma\rangle \rightarrow_{Bexp} \mathit{true} \xLeftrightarrow{\text{Lemma fuer } b} (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}$$

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

$$\langle c, \sigma\rangle \rightarrow_{Stmt} \sigma' \xLeftrightarrow{\text{IH fuer } \langle c,\sigma\rangle \rightarrow_{Stmt} \sigma'} (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}}$$

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

$$\langle \textbf{while}(b)\ c, \sigma'\rangle \rightarrow_{Stmt} \sigma'' \xLeftrightarrow{\text{IH fuer } \langle \textbf{while}(b)\ c,\sigma'\rangle \rightarrow_{Stmt} \sigma''} (\sigma', \sigma'') \in [\![\textbf{while}(b)\ c]\!]_{\mathcal{C}}$$

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

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

Äquivalenz operationale und denotationale Semantik

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

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

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

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

- $\Leftarrow$ Beweis Prinzip?

Äquivalenz operationale und denotationale Semantik

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

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

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

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

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

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

Induktionsanfang:

- Fall $c \equiv x = a$:

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

$$\begin{aligned}
& (\sigma, \sigma') \in \{(\sigma'', \sigma''[t/x]) | (\sigma'', t) \in [\![a]\!]_{\mathcal{A}}\} \\
\overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} \quad & (\sigma, t) = [\![a]\!]_{\mathcal{A}} \land \sigma' = \sigma[t/x] \\
\overset{\text{Lemma } \textbf{AExp}}{\Longrightarrow} \quad & \langle a, \sigma \rangle \to_{Aexp} t \land \sigma' = \sigma[t/x] \\
\overset{\text{Def. } \langle .,. \rangle \to_{Stmt}.}{\Longrightarrow} \quad & \langle x = a, \sigma \rangle \to_{Stmt} \sigma[t/x] \land \sigma' = \sigma[t/x] \\
\Longrightarrow \quad & \langle x = a, \sigma \rangle \to_{Stmt} \sigma'
\end{aligned}$$

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

Induktionsanfang:

- Fall $c \equiv \{\}$

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

$$\begin{aligned}
& (\sigma, \sigma') \in \{(\sigma'', \sigma'') | \sigma'' \in \Sigma\} \\
\xRightarrow{\text{Def. } [\![.]\!]_{\mathcal{C}}.} \quad & \sigma = \sigma' \\
\xRightarrow{\text{Def. } \langle .,. \rangle \rightarrow_{Stmt}.} \quad & \langle \{\}, \sigma \rangle \rightarrow_{Stmt} \sigma \wedge \sigma = \sigma' \\
\Longrightarrow \quad & \langle \{\}, \sigma \rangle \rightarrow_{Stmt} \sigma'
\end{aligned}$$

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 **if** (b) c_1 **else** c_2:

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

Induktionsannahme gilt für c_1 und c_2

- Fall: $(\sigma, \sigma') \in \{(\sigma'', \sigma''')|(\sigma'', true) \in [\![b]\!]_{\mathcal{B}}, (\sigma'', \sigma''') \in [\![c_1]\!]_{\mathcal{C}}\}$

$$\begin{array}{rl} & (\sigma, \sigma') \in \{(\sigma'', \sigma''')|(\sigma'', true) \in [\![b]\!]_{\mathcal{B}}, (\sigma'', \sigma''') \in [\![c_1]\!]_{\mathcal{C}}\} \\ \overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} & (\sigma, true) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}} \\ \overset{\text{Lemma } \mathbf{BExp}}{\Longrightarrow} & \langle b, \sigma \rangle \rightarrow_{Bexp} true \wedge (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}} \\ \overset{\text{IA für } c_1}{\Longrightarrow} & \langle b, \sigma \rangle \rightarrow_{Bexp} true \wedge \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma' \\ \overset{\text{Def. } \langle .,. \rangle \rightarrow_{Stmt}.}{\Longrightarrow} & \langle \textbf{if } (b)\ c_1 \textbf{ else } c_2, \sigma \rangle \rightarrow_{Stmt} \sigma' \end{array}$$

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

Induktionsschritt:

- Fall **if** (b) c_1 **else** c_2:

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

Induktionsannahme gilt für c_1 und c_2

  - Fall: $(\sigma, \sigma') \in \{(\sigma'', \sigma''')|(\sigma'', false) \in [\![b]\!]_{\mathcal{B}}, (\sigma'', \sigma''') \in [\![c_2]\!]_{\mathcal{C}}\}$

$$
\begin{aligned}
& (\sigma, \sigma') \in \{(\sigma'', \sigma''')|(\sigma'', true) \in [\![b]\!]_{\mathcal{B}}, (\sigma'', \sigma''') \in [\![c_2]\!]_{\mathcal{C}}\} \\
\overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow}\quad & (\sigma, false) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}} \\
\overset{\text{Lemma } \textbf{BExp}}{\Longrightarrow}\quad & \langle b, \sigma \rangle \rightarrow_{Bexp} false \wedge (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}} \\
\overset{\text{IA für } c_1}{\Longrightarrow}\quad & \langle b, \sigma \rangle \rightarrow_{Bexp} false \wedge \langle c_2, \sigma \rangle \rightarrow_{Stmt} \sigma' \\
\overset{\text{Def. } \langle .,. \rangle \rightarrow_{Stmt}.}{\Longrightarrow}\quad & \langle \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, \sigma \rangle \rightarrow_{Stmt} \sigma'
\end{aligned}
$$

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

Induktionsschritt:

- Fall **while** (b) $c2$:

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

  Induktionshypothese gilt für c

$$
\begin{aligned}
& (\sigma, \sigma') \in [\![\textbf{while}\ (b)\ c]\!]_{\mathcal{C}} \\
\xrightarrow{\text{Def. } [\![.]\!]_{\mathcal{C}}..}\quad & (\sigma, \sigma') \in \mathit{fix}(\Gamma)
\end{aligned}
$$

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

Induktionsschritt:

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

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

Induktionshypothese gilt für c

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

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

Induktionsschritt:

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

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

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

Induktionshypothese gilt für c

$$(\sigma, \sigma') \in [\![\textbf{while}\ (b)\ c]\!]_{\mathcal{C}} \overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} (\sigma, \sigma') \in \mathit{fix}(\Gamma)$$

$$\overset{\text{Def. } \mathit{fix}(\Gamma)}{\Longrightarrow} (\sigma, \sigma') \in \bigcup_{i \in \mathbb{N}} \Gamma^i(\emptyset)$$

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

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

Induktionsschritt:

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

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

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

Induktionshypothese gilt für c

$$(\sigma, \sigma') \in [\![\textbf{while}\ (b)\ c]\!]_{\mathcal{C}} \overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} (\sigma, \sigma') \in \mathit{fix}(\Gamma)$$

$$\overset{\text{Def. } \mathit{fix}(\Gamma)}{\Longrightarrow} (\sigma, \sigma') \in \bigcup_{i \in \mathbb{N}} \Gamma^i(\emptyset)$$

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

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

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

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

$$(\sigma, \sigma') \in [\![\textbf{while}\ (b)\ c]\!]_{\mathcal{C}} \overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} (\sigma, \sigma') \in fix(\Gamma)$$
$$\overset{\text{Def. } fix(\Gamma)}{\Longrightarrow} (\sigma, \sigma') \in \bigcup_{i \in \mathbb{N}} \Gamma^i(\emptyset)$$
$$\overset{(1)}{\Longrightarrow} \langle \textbf{while}\ (b)\ c, \sigma \rangle \rightarrow_{Stmt} \sigma'$$

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

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

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

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

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

Beweis per Induktion über i:
Induktionsanfang

- $i = 0$:

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

Implikation trivialerweise erfüllt da $\mathit{false} \Rightarrow F$ immer wahr

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

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

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

Beweis per Induktion über i:

Induktionsschritt

- $i \rightarrow i+1$:
  Induktionsannahme (UB) gilt für i

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

Fallunterscheidung über Zugehörigkeit zu welcher Teilmenge

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

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

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

Beweis per Induktion über i:
Induktionsschritt

- $i \rightarrow i + 1$:

  Induktionsannahme (UB) gilt für i

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

$$(\sigma, \sigma') \in \Gamma(\Gamma^i(\emptyset))$$

$$\overset{\text{Def. }\Gamma}{\Rightarrow} \quad (\sigma, \sigma') \in \{(\sigma'', \sigma''') \mid (\sigma'', true) \in [\![b]\!]_{\mathcal{B}}, (\sigma'', \sigma'''') \in [\![c]\!]_{\mathcal{C}}, (\sigma'''', \sigma''') \in \Gamma^i(\emptyset)\} \cup \{(\sigma'', \sigma'') \mid (\sigma'', false) \in [\![b]\!]_{\mathcal{B}}\}$$

$$\overset{Fall}{\Longrightarrow} \quad \underbrace{(\sigma, true) \in [\![b]\!]_{\mathcal{B}}}_{\text{Lemma }\textbf{BExp}} \wedge \underbrace{(\sigma, \sigma'') \in [\![c]\!]_{\mathcal{C}}}_{\text{IH (IB)}} \wedge \underbrace{(\sigma'', \sigma') \wedge \Gamma^i(\emptyset)}_{\text{IH (UB) für } i}$$

$$\Longrightarrow \quad \langle b, \sigma \rangle \rightarrow_{Bexp} true \wedge \langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'' \wedge \langle \textbf{while}\ (b)\ c, \sigma'' \rangle \rightarrow_{Stmt} \sigma'$$

$$\overset{\langle \cdot, \cdot \rangle \rightarrow_{Stmt} \cdot}{\Longrightarrow} \quad \langle \textbf{while}\ (b)\ c, \sigma \rangle \rightarrow_{Stmt} \sigma'$$

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

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

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

Beweis per Induktion über i:
Induktionsschritt

- $i \rightarrow i+1$:

  Induktionsannahme (UB) gilt für i

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

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

q.e.d.

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

Induktionsschritt:

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

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

Induktionshypothese gilt für c

$$(\sigma, \sigma') \in [\![\textbf{while}\ (b)\ c]\!]_{\mathcal{C}} \overset{\text{Def. } [\![.]\!]_{\mathcal{C}}..}{\Longrightarrow} (\sigma, \sigma') \in \mathit{fix}(\Gamma)$$
$$\overset{\text{Def. } \mathit{fix}(\Gamma)}{\Longrightarrow} (\sigma, \sigma') \in \bigcup_{i \in \mathbb{N}} \Gamma^i(\emptyset)$$
$$\overset{(1)}{\Longrightarrow} \langle \textbf{while}\ (b)\ c, \sigma \rangle \rightarrow_{Stmt} \sigma'$$

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

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

Äquivalenz operationale und denotationale Semantik

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

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

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

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

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten und die 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 5 vom 19.05.20

Die Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

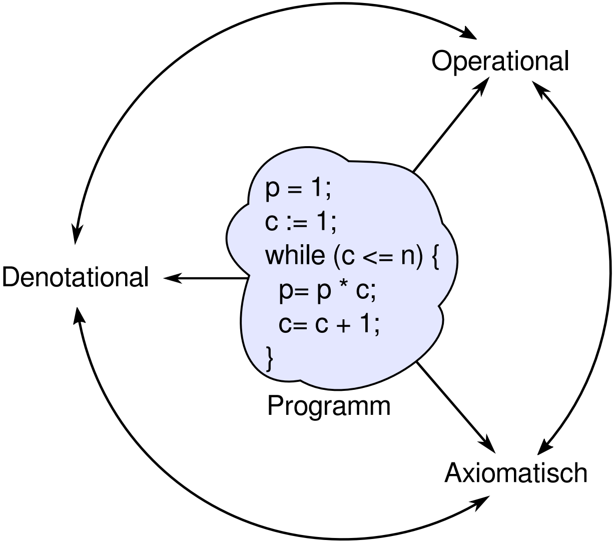
Universität Bremen

Sommersemester 2020

Fahrplan

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

Drei Semantiken — Eine Sicht



Floyd-Hoare-Logik: Idee

- Was wird hier berechnet?

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

Floyd-Hoare-Logik: Idee

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

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

Floyd-Hoare-Logik: Idee

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

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

Floyd-Hoare-Logik: Idee

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

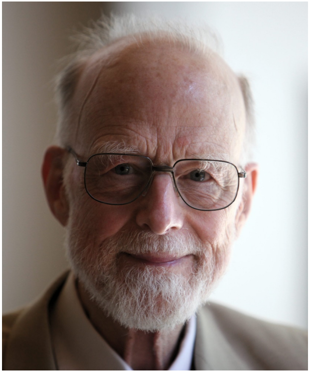
```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

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

Bob Floyd und Tony Hoare

Bildquelle: Stanford University

Robert Floyd
1936 – 2001



Bildquelle: Wikipedia

Sir Anthony Charles Richard Hoare
* 1934

Grundbausteine der Floyd-Hoare-Logik

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

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

Arbeitsblatt 5.1: Was berechnet dieses Programm?

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

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 auch **zustandsfreie** Aussagen, um Zustände **vergleichen** zu können.
- Die Logik **abstrahiert** den Effekt von Programmen durch **Vor**- und **Nachbedingung**.

Grundbausteine der Floyd-Hoare-Logik

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

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

$$\textbf{Assn} \quad b ::= \quad \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2 \mid\, !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \mid b_1 -- > b_2 \mid p(e_1, \ldots, e_n) \mid \backslash\textbf{forall}\ v.\, b \mid \backslash\textbf{exists}\ v.\, b$$

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:

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

$$\textbf{Assn} \quad b ::= \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b2 \mid b_1 \vee b_2 \mid b_1 \longrightarrow b_2 \mid p(e_1, \ldots, e_n) \mid \forall v.\, b \mid \exists v.\, b$$

Denotationale Semantik von Zusicherungen

- Erste Näherung: Funktion

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

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

- Aber: was ist mit den logischen Variablen?

Denotationale Semantik von Zusicherungen

- Erste Näherung: Funktion

$$
\begin{aligned}
[\![a]\!]_{\mathcal{A}} &: \textbf{Aexpv} \to (\Sigma \rightharpoonup \mathbb{Z}) \\
[\![b]\!]_{\mathcal{B}} &: \textbf{Assn} \to (\Sigma \rightharpoonup \mathcal{B})
\end{aligned}
$$

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

- Aber: was ist mit den logischen Variablen?
- Zusätzlicher Parameter **Belegung** der logischen Variablen $I : \textbf{Var} \to \mathbb{Z}$

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

Erfüllung von Zusicherungen

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

Erfülltheit von Zusicherungen

$b \in \textbf{Assn}$ 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:

$$a \quad \equiv \quad x = 2 \cdot X \longrightarrow x > X$$

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
s_1			
s_2			
s_3			

Fügen Sie eine zusätzliche Bedingung hinzu, so 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.

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

- 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.

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

Beispiele

- Folgendes **gilt**:

$$\models \{true\}\ \textbf{while}(1)\{\ \}\ \{true\}$$

Beispiele

- Folgendes **gilt**:

$$\models \{true\}\ \textbf{while}(1)\{\ \}\ \{true\}$$

- Folgendes gilt **nicht**:

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

Beispiele

- Folgendes **gilt**:

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

- Folgendes gilt **nicht**:

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

- Folgende **gelten**:

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

Wegen *ex falso quodlibet*: $false \Longrightarrow \phi$

Gültigkeit und Herleitbarkeit

- **Semantische Gültigkeit:** $\models \{P\}\, c\, \{Q\}$
  - Definiert durch denotationale Semantik:

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

  - Problem: müssten Semantik von c ausrechnen

Gültigkeit und Herleitbarkeit

- **Semantische Gültigkeit:** $\models \{P\}\, c\, \{Q\}$
  - Definiert durch denotationale Semantik:
    $$\models \{P\}\, c\, \{Q\} \Longleftrightarrow \forall I.\, \forall \sigma.\, \sigma \models^{I} P \wedge \exists \tau.\, (\sigma, \tau) \in [\![c]\!]_{\mathcal{C}} \Longrightarrow \tau \models^{I} Q$$
  - 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 = 5
//{x < 10}
```

Regeln des Floyd-Hoare-Kalküls: Zuweisung

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

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

```
//{(x < 10)[5/x]}
x = 5
//{x < 10}
```

Regeln des Floyd-Hoare-Kalküls: Zuweisung

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

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

```
//{(x < 10)[5/x] ⟺ 5 < 10}
x = 5
//{x < 10}
```

Regeln des Floyd-Hoare-Kalküls: Zuweisung

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

- 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 = x+ 1
//{x < 10}
```

Regeln des Floyd-Hoare-Kalküls: Zuweisung

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

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

```
//{(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?

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?

Ein allererstes Beispiel

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

Herleitung:

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

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?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Herleitung:

$$\frac{}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}$$

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?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Herleitung:

$$\dfrac{\dfrac{}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; \\ \quad \{?\}\end{array}} \qquad \dfrac{}{\begin{array}{l}\vdash \{?\} \\ \quad y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}$$

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?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Herleitung:

$$
\dfrac{\dfrac{}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; \\ \quad \{z = X \wedge x = Y\}\end{array}} \qquad \dfrac{}{\begin{array}{l}\vdash \{z = X \wedge x = Y\} \\ \quad y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}}{\begin{array}{l}\vdash \{x = X \wedge y = Y\} \\ \quad z = x; x = y; y = z; \\ \quad \{y = X \wedge x = Y\}\end{array}}
$$

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?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Herleitung:

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

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?
- $\vdash \{x = X \wedge y = Y\}\, p\, \{y = X \wedge x = Y\}$

Herleitung:

$$
\dfrac{
  \dfrac{
    \dfrac{}{\vdash \{x = X \wedge y = Y\}\; z = x;\; \{z = X \wedge y = Y\}}
    \quad
    \dfrac{}{\vdash \{z = X \wedge y = Y\}\; x = y;\; \{z = X \wedge x = Y\}}
  }{\vdash \{x = X \wedge y = Y\}\; z = x; x = y;\; \{z = X \wedge x = Y\}}
  \quad
  \dfrac{}{\vdash \{z = X \wedge x = Y\}\; y = z;\; \{y = X \wedge x = Y\}}
}{\vdash \{x = X \wedge y = Y\}\; z = x; x = y; y = z;\; \{y = X \wedge x = Y\}}
$$

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

Arbeitsblatt 5.3: Ein erster Beweis

Betrachte den Rumpf des Fakultätsprogramms:

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

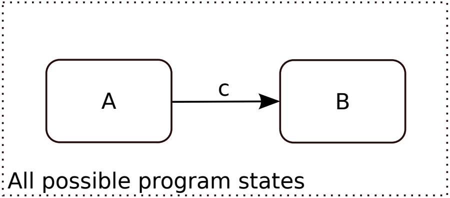
- Welche Zusicherungen gelten

  (i) an der Stelle (A)?

  (ii) an der Stelle (B)?

Regeln des Floyd-Hoare-Kalküls: Weakening

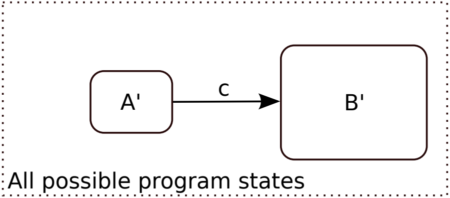
$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\,c\,\{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\,c\,\{B'\}}$$



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

Regeln des Floyd-Hoare-Kalküls: Weakening

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



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

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 müssem mit derselben Nachbedingung enden.

Arbeitsblatt 5.4: Ein zweiter Beweis

Betrachte folgendes Programm:

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

1. Was berechnet dieses Programm?
2. Wie spezifizieren wir das?
3. Wie beweisen wir die Gültigkeit?

Arbeitsblatt 5.5: Ein zweiter Beweis

Betrachte folgendes Programm:

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

1. Was berechnet dieses Programm?
2. Wie spezifizieren wir das?
3. Wie beweisen wir die Gültigkeit?

- Die Spezifikation wird zur Nachbedingung (A)
- Wir notieren Weakening durch aufeinanderfolgende Bedingungen:

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

- Welche Zusicherungen müssen an den Stellen (A) – (F) gelten?
- Wo müssen wir logische Umformungen nutzen?

Regeln des Floyd-Hoare-Kalküls: Iteration

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

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

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

Das Fakultätsbeispiel (I)

```
// {1 = 0!}
// {1 = (1 − 1)!}
p= 1;
// {p = (1 − 1)!}
c= 1;
// {p = (c − 1)!}
while (c<= n) {
   // {p = (c − 1)! ∧ c ≤ n}
   // {p ∗ c = (c − 1)! ∗ c}
   // {p ∗ c = c!}
   // {p ∗ c = ((c + 1) − 1)!}
   p= p∗c;
   // {p = ((c + 1) − 1)!}
   c= c+1;
   // {p = (c − 1)!}
   }
// {p = (c − 1)! ∧ ¬(c ≤ n)}
// {p = (c − 1)! ∧ c − 1 ≥ n}
// ??
// {p = n!}
```

Das Fakultätsbeispiel (II)

```
// {1 = 0! ∧ 0 ≤ n}
// {1 = (1 − 1)! ∧ 1 − 1 ≤ n}
p= 1;
// {p = (1 − 1)! ∧ 1 − 1 ≤ n}
c= 1;
// {p = (c − 1)! ∧ c − 1 ≤ n}
while (c<= n) {
  // {p = (c − 1)! ∧ c − 1 ≤ n ∧ c ≤ n}
  // {p ∗ c = (c − 1)! ∗ c ∧ c ≤ n} !!!
  // {p ∗ c = c! ∧ c ≤ n}
  // {p ∗ c = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  p= p∗c;
  // {p = ((c + 1) − 1)! ∧ (c + 1) − 1 ≤ n}
  c= c+1;
  // {p = (c − 1)! ∧ c − 1 ≤ n}
  }
// {p = (c − 1)! ∧ c − 1 ≤ n ∧ ¬(c ≤ n)}
// {p = (c − 1)! ∧ c − 1 ≤ n ∧ c > n}
// {p = (c − 1)! ∧ c − 1 ≤ n ∧ c − 1 ≥ n}
// {p = n!}
```

Das Fakultätsbeispiel (komplett)

```
// {1 = 0! ∧ 0 ≤ n}
// {1 = (1 − 1)! ∧ 1 ≤ 1 ∧ 1 − 1 ≤ n}
p= 1;
// {p = (1 − 1)! ∧ 1 ≤ 1 ∧ 1 − 1 ≤ n}
c= 1;
// {p = (c − 1)! ∧ 1 ≤ c ∧ c − 1 ≤ n}
while (c<= n) {
  // {p = (c − 1)! ∧ 1 ≤ c ∧ c − 1 ≤ n ∧ c ≤ n}
  // {p ∗ c = (c − 1)! ∗ c ∧ 1 ≤ c ∧ c ≤ n}
  // {p ∗ c = c! ∧ 1 ≤ c ∧ c ≤ n}
  // {p ∗ c = ((c + 1) − 1)! ∧ 1 ≤ c + 1 ∧ (c + 1) − 1 ≤ n}
  p= p∗c;
  // {p = ((c + 1) − 1)! ∧ 1 ≤ c + 1 ∧ (c + 1) − 1 ≤ n}
  c= c+1;
  // {p = (c − 1)! ∧ 1 ≤ c ∧ c − 1 ≤ n}
  }
// {p = (c − 1)! ∧ 1 ≤ c ∧ c − 1 ≤ n ∧ ¬(c ≤ n)}
// {p = (c − 1)! ∧ c − 1 ≤ n ∧ c > n}
// {p = (c − 1)! ∧ c − 1 ≤ n ∧ c − 1 ≥ n}
// {p = n!}
```

Arbeitsblatt 5.6: Exponents Revisited

Wir können jetzt das Programm vom Anfang korrekt beweisen:

```
/** ... */
x= 1;
c= 1;
/** x= 2^(c-1) && .. */
while (c<= y) {
  /** x= 2^(c-1) && ... && c<= y */
  /** ... */
  x= 2*x;
  /** ... */
  c= c+1;
  /** x= 2^(c-1) && ... */
  }
/** { x= 2^y && ... && ! (c<= y) */
/** ... */
/** { x= 2^y } */
```

- Findet den Rest der Invariante, und
- Füllt den restlichen Teil aus.

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Zusammenfassung Floyd-Hoare-Logik

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 6 vom 28.05.20

Invarianten und die Korrektheit des Floyd-Hoare-Kalküls

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Invarianten

Invarianten Finden: die Fakultät

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

Invariante:

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

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

Invarianten Finden: die Fakultät

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

Invariante:

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

- Kern der Invariante: Fakultät bis $c-1$ berechnet.
- Invariante impliziert Nachbedingung $p = n! = (c-1)!$

Invarianten Finden: die Fakultät

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

Invariante:

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

- Kern der Invariante: Fakultät bis $c - 1$ berechnet.
- Invariante impliziert Nachbedingung $p = n! = (c-1)!$
- 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

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

ist syntaktischer Zucker für

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

Beispiel 1: Zählende Schleife

```
// {0 ≤ n}
x= 0;
c= 1;
while (c <= n) {
 x= x+c;
 c= c+1;
}
// {x = ∑_0^n}
```

- Invariante:

Hierbei ist $\sum_a^b$ die Summe der Zahlen von a bis b, mit folgenden Eigenschaften:

$$\sum_0^0 = 0$$

$$a > 0 \Longrightarrow \sum_0^a = \sum_0^{a+1} + a$$

Beispiel 1: Zählende Schleife

```
// {0 ≤ n}
x= 0;
c= 1;
while (c <= n) {
  x= x+c;
  c= c+1;
}
// {x = ∑_0^n}
```

- Invariante:

$$x = \sum_{0}^{c-1}$$

Hierbei ist $\sum_a^b$ die Summe der Zahlen von a bis b, mit folgenden Eigenschaften:

$$\sum_{0}^{0} = 0$$

$$a > 0 \Longrightarrow \sum_{0}^{a} = \sum_{0}^{a+1} + a$$

Beispiel 1: Zählende Schleife

```
// {0 ≤ n}
x= 0;
c= 1;
while (c <= n) {
  x= x+c;
  c= c+1;
}
// {x = ∑_0^n}
```

- Invariante:

$$x = \sum_{0}^{c-1} \wedge c - 1 \leq n$$

Hierbei ist $\sum_a^b$ die Summe der Zahlen von a bis b, mit folgenden Eigenschaften:

$$\sum_{0}^{0} = 0$$

$$a > 0 \Longrightarrow \sum_{0}^{a} = \sum_{0}^{a+1} + a$$

Beispiel 2: Variante der zählenden Schleife

```
// {0 ≤ y}
x= 0;
c= 0;
while (c < y) {
  c= c+1;
  x= x+c;
}
// {x = ∑_0^n}
```

- Invariante:

Beispiel 2: Variante der zählenden Schleife

```
// {0 ≤ y}
x= 0;
c= 0;
while (c < y) {
  c= c+1;
  x= x+c;
}
// {x = ∑_0^n}
```

- Invariante:

$$x = \sum_{0}^{c}$$

Beispiel 2: Variante der zählenden Schleife

```
// {0 ≤ y}
x= 0;
c= 0;
while (c < y) {
  c= c+1;
  x= x+c;
}
// {x = ∑_0^n}
```

- Invariante:

$$x = \sum_{0}^{c} \wedge 0 \leq c$$

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Beispiel 3: Andere Variante der zählenden Schleife

```
// {n = N ∧ 0 ≤ n}
x= 0;
while (n != 0) {
  x= x+n;
  n= n-1;
}
// {x = ∑_0^N}
```

- Invariante:

Beispiel 3: Andere Variante der zählenden Schleife

```
// {n = N ∧ 0 ≤ n}
x= 0;
while (n != 0) {
 x= x+n;
 n= n-1;
}
// {x = ∑_0^N}
```

- Invariante:

$$x = \sum_{n}^{N}$$

Beispiel 3: Andere Variante der zählenden Schleife

```
// {n = N ∧ 0 ≤ n}
x= 0;
while (n != 0) {
  x= x+n;
  n= n-1;
}
// {x = ∑_0^N}
```

- Invariante:

$$x = \sum_{n}^{N} \;\wedge\; n \leq N$$

Arbeitsblatt 6.1: Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

```
// {0 ≤ n ∧ n = N}
p= 1;
while (0< n) {
  p= p*n;
  n= n-1;
}
// {p = N!}
```

- Finden Sie eine Invariante.
- Beweisen Sie die Korrektheit.

Für die Invariante benötigen sie ein indiziertes Produkt (analog zur Summenfunktion):

$$\prod_a^b = a \cdot (a+1) \cdot \ldots \cdot b$$

Für das Produkt gelten folgende Eigenschaften:

$$a! = \prod_1^a$$

$$a > b \Longrightarrow \prod_a^b = 1$$

$$a \leq b \Longrightarrow \prod_a^b = a \cdot \prod_{a+1}^b$$

Beispiel 4: Nicht-zählend (rekursiv)

```
// {0 ≤ a}
r= a;
q= 0;
while (b <= r) {
   r= r-b;
   q= q+1;
   }
// {a = b * q + r ∧ 0 ≤ r ∧ r < b}
```

Invariante:

Beispiel 4: Nicht-zählend (rekursiv)

```
// {0 ≤ a}
r= a;
q= 0;
while (b <= r) {
    r= r-b;
    q= q+1;
    }
// {a = b * q + r ∧ 0 ≤ r ∧ r < b}
```

Invariante:

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

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

Beispiel 5: Jetzt wird's kompliziert. . .

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

- Was berechnet das?

Beispiel 5: Jetzt wird's kompliziert. . .

```
// {0 ≤ a}
t= 1;
s= 1;
i= 0;
while (s <= a) {
   t= t+ 2;
   s= s+ t;
   i= i+ 1;
   }
// {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:
  - $s-t \leq a, s = i^2+t \Longrightarrow i^2 \leq a$.
- Nachbedingung 2:
  - $s = i^2 + t, t = 2 \cdot i + 1 \Longrightarrow s = (i+1)^2$
  - $a < s, s = (i+1)^2 \Longrightarrow a < (i+1)^2$

Korrektheit des Floyd-Hoare-Kalküls

Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

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

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

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

- **Frage**: $\vdash \{P\}\, c\, \{Q\} \quad \stackrel{?}{\leftrightsquigarrow} \quad \models \{P\}\, c\, \{Q\}$

Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

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

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

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

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

Korrektheit des Floyd-Hoare-Kalküls

Der Floyd-Hoare-Kalkül ist korrekt.

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

Beweis:

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

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

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

Arbeitsblatt 6.2: Korrektheit der Zuweisung

Beweisen Sie die Korrektheit der **Zuweisungsregel**:

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

1. Was genau ist zu zeigen?

2. Wir benötigen folgendes **Lemma**:

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

   Wie zeigen wir damit die Behauptung?

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:

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

- 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 $\mathrm{wp}(c_0; c_1, Q) := \mathrm{wp}(c_0, \mathrm{wp}(c_1, Q))$
  - $c \equiv \textbf{if}\ b\ c_0\ \textbf{else}\ c_1$: Wähle $\mathrm{wp}(c, Q) := (b \wedge \mathrm{wp}(c_0, Q)) \vee (\neg b \wedge \mathrm{wp}(c_1, Q))$
  - $c \equiv \textbf{while}\ (b)\ c_0$: ??

Vollständigkeitsbeweis: while

- $c \equiv$ **while** $(b)\ c_0$:
  Wie müssen eine Formel finden (wp(**while** $(b)\ c_0, Q$)) die alle σ charakterisiert, so dass

$$
\begin{aligned}
&\sigma \models^I \mathrm{wp}(\textbf{while}\ (b)\ c_0, Q)\\
&\longleftrightarrow \forall k \geq 0 \forall \sigma_0, \ldots, \sigma_k. \quad \sigma = \sigma_0\\
&\qquad\qquad\qquad\qquad\qquad\quad \forall 0 \leq i < k. (\sigma_i \models^I b \wedge \quad \underbrace{[\![c_0]\!]_{\mathcal{C}} \sigma_i = \sigma_{i+1}}_{c_0\ \underline{\text{terminiert}}\ \text{auf}\ \sigma_i\ \text{in}\ \sigma_{i+1}}\\
&\qquad\qquad\qquad\qquad\qquad\quad \sigma_k \models^I b \vee Q
\end{aligned}
$$

- 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

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 7 vom 4.6.20

Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Motivation

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

Arrays

- Beispiele:

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

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

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

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

  - Alle Felder haben **feste Größe**.

Zeichenketten

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

```
char hallo[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:

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

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

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

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

Werte und Zustände

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

Systemzustände

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

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

Beispiel

Programm

```c
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

$x[0].c[0] \mapsto 1$ $\quad x[1].c[0] \mapsto 3$

$x[0].c[1] \mapsto 2$ $\quad x[1].c[1] \mapsto 4$

$x[0].b.name[0] \mapsto `n'$ $\quad x[1].b.name[0] \mapsto `n'$

$x[0].b.name[1] \mapsto `a'$ $\quad x[1].b.name[1] \mapsto `a'$

$x[0].b.name[2] \mapsto `m'$ $\quad x[1].b.name[2] \mapsto `m'$

$x[0].b.name[3] \mapsto `e'$ $\quad x[1].b.name[3] \mapsto `e'$

$x[0].b.name[4] \mapsto `1'$ $\quad x[1].b.name[4] \mapsto `2'$

$x[0].b.name[5] \mapsto `\backslash 0'$ $\quad x[1].b.name[5] \mapsto `\backslash 0'$

Operationale Semantik: L-Werte

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

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

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

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

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

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

Operationale Semantik: Ausdrücke

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

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

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

- Auswertung für **C**:

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

wobei $\mathrm{Ord} : \mathbf{C} \rightarrow \mathbf{Z}$ eine bijektive Funktion ist, die jedem Character eine Ordinalzahl zuweist (zum Beispiel ASCII Wert).

Operationale Semantik: Zuweisungen

- Zuweisungen sind nur definiert für elementare Typen:

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

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

$$\langle m = e, \sigma \rangle \rightarrow_{Stmt} \perp$$

- Die restlichen Regeln bleiben

Denotationale Semantik

- Denotation für **Lexp**:

$$\llbracket - \rrbracket_{\mathcal{L}} : \textbf{Lexp} \to (\Sigma \rightharpoonup \textbf{Loc})$$

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

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

$$\llbracket c \rrbracket_{\mathcal{A}} = \{(\sigma, \mathsf{Ord}(c)) | \sigma \in \Sigma\}$$

- Denotation für **Zuweisungen**:

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

Floyd-Hoare-Kalkül

- Die Regeln des Floyd-Hoare-Kalküls berechnen geltende Zusicherungen

- Nötige Änderung: Substitution in Zusicherungen

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

  - Jetzt werden **Lexp** ersetzt, keine **Idt**
  - Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
  - Problem: Feldzugriffe

Beispiel

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

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

Beispiel

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

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

Beispiel

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

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

Beispiel

```
int a[3];
// {true}
//
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}
```

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

Beispiel

```
int a[3];
// {true}
//
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}
```

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

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}
```

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

Beispiel: Problem

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

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

Beispiel: Problem

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

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

Beispiel: Problem

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

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

Beispiel: Problem

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

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

Beispiel: Problem

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

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

Beispiel: Problem

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

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

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[i] = −1;
// {a[1] = 7}
```

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

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[i] = −1;
// {a[1] = 7}
```

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

Arbeitsblatt 7.1: 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] = m^2 - n^2}
```

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

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
//
while (i< n) {
  //
  //
  //
  //
  //
  //
  a[i]= i;
  //
  i= i+1;
  //
  }
//
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
//
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   //
   //
   //
   //
   //
   a[i]= i;
   //
   i= i+1;
   //
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   //
   //
   //
   //
   //
   a[i]= i;
   //
   i= i+1;
   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   //
   //
   //
   //
   //
   a[i]= i;
   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
   i= i+1;
   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   //
   //
   //
   // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (i ≠ i ∧ a[j] = j))
   //      ∧i + 1 ≤ n}
   a[i]= i;
   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
   i= i+1;
   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   //
   // {∀j.0 ≤ j < i ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
   //      ∧((i = i ∧ i = i) ∨ (i ≠ i ∧ a[i] = i)) ∧ i + 1 ≤ n}
   // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (i ≠ i ∧ a[j] = j))
   //      ∧i + 1 ≤ n}
   a[i]= i;
   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
   i= i+1;
   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

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

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
//
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i + 1 ≤ n}
   // {∀j.0 ≤ j < i ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
   //      ∧((i = i ∧ i = i) ∨ (i ≠ i ∧ a[i] = i)) ∧ i + 1 ≤ n}
   // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (i ≠ i ∧ a[j] = j))
   //      ∧i + 1 ≤ n}
   a[i]= i;
   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
   i= i+1;
   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

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

Erstes Beispiel: Ein Feld initialisieren

```
// {0 ≤ n}
// {∀j.0 ≤ j < 0 ⟶ a[j] = j ∧ 0 ≤ n}
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
   // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i + 1 ≤ n}
   // {∀j.0 ≤ j < i ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
   //       ∧((i = i ∧ i = i) ∨ (i ≠ i ∧ a[i] = i)) ∧ i + 1 ≤ n}
   // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (i ≠ i ∧ a[j] = j))
   //       ∧i + 1 ≤ n}
   a[i]= i;
   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
   i= i+1;
   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
   }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

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

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
//
while (i< n) {
  //
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
    }
  //
  i= i+1;
  //
  }
//
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  //
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
    }
  //
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
//
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  //
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
    }
  //
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
    }
  //
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    //
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  //
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    //
    //
    //
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    //
    //
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
    //
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
//
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ a[i] ≤ a[i] ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
//
i= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ∧ 0 ≤ 0 < n}
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ a[i] ≤ a[i] ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
// {(∀j. 0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 ∧ 0 ≤ 0 < n}
i= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ∧ 0 ≤ 0 < n}
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ a[i] ≤ a[i] ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] ≥ a[i]}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
//
while (i< n) {
  //
  //
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
  //
  i= i+1;
  //
  }
//
//
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  //
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
  //
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
//
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  //
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
  //
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  //
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    //
    }
  else {
    //
    //
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  //
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    //
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  //
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  if (a[i] == 0) {
    //
    //
    //
    //
    //
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  if (a[i] == 0) {
    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
    //
    //
    //
    //
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  if (a[i] == 0) {
    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
    //
    //
    //
```

$$// \ \{(\underbrace{i \neq -1}_{A(i)} \longrightarrow \underbrace{0 \leq i < i+1 \wedge a[i] = 0}_{B(i)}) \wedge \underbrace{0 \leq i+1 \leq n \wedge a[i] = 0}_{C}\}$$

```
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  if (a[i] == 0) {
    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
```

$$// \ \{\overbrace{0 \leq i < i+1 \land a[i] = 0 \land 0 \leq i+1 \leq n \land a[i] = 0}^{B(i) \land C}\}$$

$$//$$

$$//$$

$$// \ \{(\underbrace{i \neq -1}_{A(i)} \longrightarrow \underbrace{0 \leq i < i+1 \land a[i] = 0}_{B(i)}) \land \underbrace{0 \leq i+1 \leq n \land a[i] = 0}_{C}\}$$

```
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
// {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

// $\{0 \le n\}$

// $\{(-1 \ne -1 \longrightarrow 0 \le -1 < 0 \wedge a[-1] = 0) \wedge 0 \le 0 \le n\}$

`i= 0;`

// $\{(-1 \ne -1 \longrightarrow 0 \le -1 < i \wedge a[-1] = 0) \wedge 0 \le i \le n\}$

`r= -1;`

// $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge 0 \le i \le n\}$

`while (i< n) {`

  // $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge 0 \le i \le n \wedge i < n\}$

  // $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge 0 \le i + 1 \le n\}$

  `if (a[i] == 0) {`

    // $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge 0 \le i + 1 \le n \wedge a[i] = 0\}$

    // $\{\overbrace{0 \le i < i + 1 \wedge a[i] = 0 \wedge 0 \le i + 1 \le n \wedge a[i] = 0}^{B(i) \wedge C}\}$

    // $\{(\overbrace{i = -1}^{\neg A(i)} \wedge \overbrace{0 \le i + 1 \le n \wedge a[i] = 0}^{C}) \vee (\overbrace{0 \le i < i + 1 \wedge a[i] = 0}^{B(i)} \wedge \overbrace{0 \le i + 1 \le n \wedge a[i] = 0}^{C})\}$

    //

    // $\{(\underbrace{i \ne -1}_{A(i)} \longrightarrow \underbrace{0 \le i < i + 1 \wedge a[i] = 0}_{B(i)}) \wedge \underbrace{0 \le i + 1 \le n \wedge a[i] = 0}_{C}\}$

    `r= i;`

    // $\{(r \ne -1 \longrightarrow 0 \le r < i + 1 \wedge a[r] = 0) \wedge 0 \le i + 1 \le n\}$

    `}`

  `else {`

    // $\{(r \ne -1 \longrightarrow 0 \le r < i + 1 \wedge a[r] = 0) \wedge 0 \le i + 1 \le n \wedge a[i] \ne 0\}$

    // $\{(r \ne -1 \longrightarrow 0 \le r < i + 1 \wedge a[r] = 0) \wedge 0 \le i + 1 \le n\}$

  // $\{(r \ne -1 \longrightarrow 0 \le r < i + 1 \wedge a[r] = 0) \wedge 0 \le i + 1 \le n\}$

  `i= i+1;`

  // $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge 0 \le i \le n\}$

  `}`

// $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge 0 \le i \le n \wedge i \ge n\}$

// $\{(r \ne -1 \longrightarrow 0 \le r < i \wedge a[r] = 0) \wedge i = n\}$

Längeres Beispiel: Suche nach einem Null-Element

```
// {0 ≤ n}
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
i= 0;
// {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
r= −1;
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
while (i< n) {
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  if (a[i] == 0) {
    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
```

$$// \ \{\overbrace{0 \le i < i+1 \wedge a[i] = 0 \wedge 0 \le i+1 \le n \wedge a[i] = 0}^{B(i) \wedge C}\}$$

$$// \ \{(\overbrace{i = -1}^{\neg A(i)} \wedge \overbrace{0 \le i+1 \le n \wedge a[i] = 0}^{C}) \vee (\overbrace{0 \le i < i+1 \wedge a[i] = 0}^{B(i)} \wedge \overbrace{0 \le i+1 \le n \wedge a[i] = 0}^{C})\}$$

$$// \ \{(i = -1 \vee (0 \le i < i+1 \wedge a[i] = 0)) \wedge 0 \le i+1 \le n \wedge a[i] = 0\}$$

$$// \ \{(\underbrace{i \neq -1}_{A(i)} \longrightarrow \underbrace{0 \le i < i+1 \wedge a[i] = 0}_{B(i)}) \wedge \underbrace{0 \le i+1 \le n \wedge a[i] = 0}_{C}\}$$

```
    r= i;
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
    }
  else {
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
  i= i+1;
  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
  }
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n}
// {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
```

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

```
/** { 0 ≤ n } */
/** { 0 ≤ 0 ≤ n } */
i= 0;
/** { 0 ≤ i ≤ n } */
/** { (−1 ≠ −1 ⟶ 0 ≤ −1< i ∧ a[−1] == 0) ∧ 0 ≤ i ≤ n } */
r= −1;
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
while (i< n) {
  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i < n } */
  /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
  if (a[i] == 0) {
    /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
    /** { 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
    /** { (i ≠ −1 ⟶ 0 ≤ i< i+1 ∧ a[i] == 0) ∧ 0≤ i+1≤ n } */
    r= i;
    /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0≤ i+1≤ n } */
    }
  else {
     /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] ≠ 0 } */
     /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
    }
  /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
  i= i+1;
  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
  }
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n) } */
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n } */
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ i == n } */
/** { r ≠ −1 ⟶ 0 ≤ r< n ∧ a[r] == 0 } */
```

Allgemeine Regel bei Ersetzungen?

Wie sieht nun die allgemeine Regel aus für

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

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

```c
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

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

1. Wenn l Programmvariable ist, wie gewohnt substituieren
2. Wenn $l = a[s]$:
   - 2.1 Vorkommen der Form $m.a[t]$ in Literalen $L(m.a[t])$ und s und t beide in $\mathbb{Z}$,
     - 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 $(t = s \wedge L(e)) \vee (t \neq s \wedge L(a[t]))$

$\boxed{\text{2.2 könnt ihr immer machen, 2.1 ist eine Optimierung}}$

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

Arbeitsblatt 7.2: Längeres Beispiel: Suche nach dem ersten Null-Element

Ausgehend von dem vorherigem Beispiel, annotiert folgendes

```
// {0 ≤ n}
i= 0;
r= −1;
/* −− beforeloop −− */
while (i< n) {
  /* −− startloop −− */
  if (r== −1 && a[i] == 0) {
    r= i;
    }
  else {
  }
  /* −− afterif −− */
  i= i+1;
  /* −− endloop −− */
}
/* −− afterloop −− */
/** {(r ≠ −1 −→ (0 ≤ r< n ∧ a[r] == 0 ∧ (∀ int j . 0 ≤ j < r −→ a[j] ≠ 0)))
    ∧ (r==−1 −→ (∀ int j . 0 ≤ j < n −→ a[j] ≠ 0))} */
```

Längeres Beispiel: Suche nach dem ersten Null-Element

```
  /** {(r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i+1 ⟶ a[j] ≠ 0)) ∧ 0 ≤ i < n} */
  /** {(r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i+1 ⟶ a[j] ≠ 0)) ∧ 0 ≤ i+1 ≤ n } */
  i= i+1;
  /** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0)) ∧ 0 ≤ i ≤ n} */
  /* −− endloop −− */
  }
/** {(r ≠ −1 ⟶ (0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0)))
      ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
      ∧ 0 ≤ i ≤ n ∧ ¬(i < n)} */
/* −− afterloop −− */
/** {(r ≠ −1 ⟶ (0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0) ))
      ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
      ∧ 0 ≤ i ≤ n ∧ i ≥ n} */
/** {(r ≠ −1 ⟶ (0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0) ))
      ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
      ∧ i == n} */
/** {(r ≠ −1 ⟶ (0 ≤ r< n ∧ a[r] == 0 ∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0)))
      ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < n ⟶ a[j] ≠ 0))} */
/* −− end −− */
}
```

Längeres Beispiel: Suche nach dem ersten Null-Element

```
while (i< n) {
  /** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))  ∧ 0 ≤ i ≤ n ∧
i < n} */ /* — startloop — */
  /** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0)) ∧ 0 ≤ i < n} */
   if (r== −1 && a[i] == 0) {
     /** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
        ∧ 0 ≤ i < n ∧ r == −1 ∧ a[i] == 0} */
     /** {(∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0) ∧ a[i] == 0 ∧ 0 ≤ i < n} */
     /** {(i ≠ −1 ⟶ 0 ≤ i< i+1 ∧ a[i] == 0∧ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
          ∧ (i==−1 ⟶ (∀ int j . 0 ≤ j < i+1 ⟶ a[j] ≠ 0)) ∧ 0 ≤ i < n} */
     r= i;
     /** {(r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
          ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i+1 ⟶ a[j] ≠ 0)) ∧ 0 ≤ i < n} */
     }
   else {
     /** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
          ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
          ∧ 0 ≤ i < n ∧ ¬(r == −1 ∧ a[i] == 0)} */
     /** {(r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
          ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i+1 ⟶ a[j] ≠ 0))
          ∧ 0 ≤ i < n ∧ ¬(r == −1 ∧ a[i] == 0)} */
     /** {(r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
          ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i+1 ⟶ a[j] ≠ 0)) ∧ 0 ≤ i < n} */
     }
```

Längeres Beispiel: Suche nach dem ersten Null-Element

```
/** {0 ≤ n} */
/** {(∀ int j . 0 ≤ j < 0 ⟶ a[j] ≠ 0) ∧ 0 ≤ 0 ≤ n} */
i= 0;
/** {(∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0) ∧ 0 ≤ i ≤ n} */
/** {(−1 ≠ −1 ⟶ 0 ≤ −1< i ∧ a[−1] == 0∧ (∀ int j . 0 ≤ j < −1 ⟶ a[j] ≠ 0))
      ∧ (−1==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
      ∧ 0 ≤ i ≤ n}*/
r= −1;
/** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
      ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0))
      ∧ 0 ≤ i ≤ n} *//* −− beforeloop −− */
while (i< n) {
  /** {(r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0))
        ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < i ⟶ a[j] ≠ 0)) ∧ 0 ≤ i ≤ n ∧ i < n} */
```

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 8 vom 11.6.20

Verifikationsbedingungen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Idee

- Hier ist ein einfaches Programm:

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

Idee

- Hier ist ein einfaches Programm:

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

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}
```

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.

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?

Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — P passt auf jede beliebige Nachbedingung (siehe "Definition" Folie 24 der letzten Vorlesung)

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

Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — P passt auf jede beliebige Nachbedingung (siehe “Definition” Folie 24 der letzten Vorlesung)

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

- Was ist mit den anderen Regeln?

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

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

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

Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — P passt auf jede beliebige Nachbedingung (siehe "Definition" Folie 24 der letzten Vorlesung)

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

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

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

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

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

Berechnung von Vorbedingungen

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

Schwächste Vorbedingung

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

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

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

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

- Einfach für Programme ohne Schleifen:

$$
\begin{aligned}
\mathrm{wp}(\{\ \}, P) &\stackrel{def}{=} P \\
\mathrm{wp}(l = e, P) &\stackrel{def}{=} P[e/l] \quad \text{(Genauer: Folie 24 letzte VL)} \\
\mathrm{wp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wp}(c_1, \mathrm{wp}(c_2, P)) \\
\mathrm{wp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \land \mathrm{wp}(c_0, P)) \lor (\neg b \land \mathrm{wp}(c_1, P))
\end{aligned}
$$

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

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

  - Hilft auch nicht weiter…

Lösung: Annotierte Programme

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

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

Approximative schwächste Vorbedingung

- Für die **while**-Schleife:

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

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

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

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

Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(l = e, P) &\stackrel{def}{=} P[l/x] \quad \text{(Genauer: Folie 24 letzte VL)} \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0 \textbf{ else } c_1, P) &\stackrel{def}{=} (b \land \mathrm{awp}(c_0, P)) \lor (\neg b \land \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(\textbf{while } (b)\ //{**}\ \textbf{inv } i\ {*}/\ c, P) &\stackrel{def}{=} i
\end{aligned}
$$

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

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

Beispiel: das Fakultätsprogramm

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

```
// {0 ≤ n}
p= 1;
c= 1;
while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n} */
{ p = p * c;
  c = c + 1;
  }
// {p = n!}
```

- Berechnung der Verifikationsbedingungen zur Nachbedingung.

Notation für Verifikationsbedingungen

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

AWP 6 |

Notation für Verifikationsbedingungen

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

AWP

6	$p = ((c+1)-1)! \land ((c+1)-1) \leq n$
5	

Notation für Verifikationsbedingungen

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

AWP

6	$p = ((c+1)-1)! \wedge ((c+1)-1) \leq n$
5	$p \times c = ((c+1)-1)! \wedge ((c-1)+1) \leq n$
4	

Notation für Verifikationsbedingungen

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

AWP	6	$p = ((c+1)-1)! \wedge ((c+1)-1) \leq n$
	5	$p \times c = ((c+1)-1)! \wedge ((c-1)+1) \leq n$
	4	$p = (c-1)! \wedge c-1 \leq n$
	3	

Notation für Verifikationsbedingungen

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

$$
\textbf{AWP}\quad
\begin{array}{r|l}
6 & p = ((c+1)-1)! \land ((c+1)-1) \leq n \\
5 & p \times c = ((c+1)-1)! \land ((c-1)+1) \leq n \\
4 & p = (c-1)! \land c-1 \leq n \\
3 & p = (1-1)! \land (1-1) \leq n \\
2 &
\end{array}
$$

Notation für Verifikationsbedingungen

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

$$
\textbf{AWP}\quad
\begin{array}{r|l}
6 & p = ((c+1)-1)! \wedge ((c+1)-1) \leq n \\
5 & p \times c = ((c+1)-1)! \wedge ((c-1)+1) \leq n \\
4 & p = (c-1)! \wedge c-1 \leq n \\
3 & p = (1-1)! \wedge (1-1) \leq n \\
2 & 1 = (1-1)! \wedge (1-1) \leq n
\end{array}
$$

Notation für Verifikationsbedingungen

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

WVC $\quad 6, 5 \Big|$

Notation für Verifikationsbedingungen

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

WVC $\begin{array}{r|l} 6, 5 & \emptyset \\ 4 & \end{array}$

Notation für Verifikationsbedingungen

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

WVC

$$
\begin{array}{r|l}
6,5 & \emptyset \\
4 & (p = (c-1)! \land c - 1 \leq n \land c \leq n \longrightarrow \\
 & \qquad\qquad p \times n = (c-1)! \land c - 1 \leq n \land c \leq n) \\
 & \land (p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \longrightarrow \\
 & \qquad\qquad p = n! \\
3,2 &
\end{array}
$$

Notation für Verifikationsbedingungen

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

WVC

6, 5	$\emptyset$
4	$(p = (c-1)! \land c - 1 \leq n \land c \leq n \longrightarrow$ $p \times n = (c-1)! \land c - 1 \leq n \land c \leq n)$ $\land (p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \longrightarrow$ $p = n!$
3, 2	$\emptyset$
1	

Notation für Verifikationsbedingungen

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

WVC

6, 5	$\emptyset$
4	$(p = (c-1)! \land c - 1 \leq n \land c \leq n \longrightarrow$ $p \times n = (c-1)! \land c - 1 \leq n \land c \leq n)$ $\land (p = (c-1)! \land c - 1 \leq n \land \neg(c \leq n) \longrightarrow$ $p = n!$
3, 2	$\emptyset$
1	$0 \leq n \longrightarrow 1 = (1-1)! \land (1-1) \leq n$

Vereinfachung von Verifikationsbedingungen

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

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

Arbeitsblatt 8.1: Jetzt seid ihr dran!

```
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)}
```

- Wobei gilt: $\mathrm{sum}(i,j) = \begin{cases} 0 & \text{falls } i > j \\ i + \mathrm{sum}(i+1,j) & \text{sonst} \end{cases}$
- Berechnet die **AWP** für die Zeilen 5,4,3,2
- Berechnet die **WVC** für die Zeilen 5,4,3,2,1
- Sei c obiges Programm: Berechnet

$$WVC(\{0 \leq n \wedge n = N\}\, c\, \{p = \mathrm{sum}(1, N)\})$$

Jetzt seid ihr dran!

```
// {0 ≤ n ∧ n = N}
p= 0;
while (n>0) //** inv {p = sum(n+1, N);}*/
{ p = p + n;
  n = n - 1;
  }
// {p = sum(1, N)}
```

AWP 5 |

Jetzt seid ihr dran!

```
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)}
```

AWP $\begin{array}{r|l} 5 & p = \mathrm{sum}((n-1)+1, N) \\ 4 & \end{array}$

Jetzt seid ihr dran!

```
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)}
```

$$
\textbf{AWP}\quad \begin{array}{r|l}
5 & p = \mathrm{sum}((n-1)+1, N) \\
4 & p + n = \mathrm{sum}((n-1)+1, N) \\
3 &
\end{array}
$$

Jetzt seid ihr dran!

```
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)}
```

AWP	5	$p = \text{sum}((n-1)+1, N)$
	4	$p + n = \text{sum}((n-1)+1, N)$
	3	$p = \text{sum}(n+1, N)$
	2	

Jetzt seid ihr dran!

```
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)}
```

$$
\textbf{AWP}\;
\begin{array}{r|l}
5 & p = \mathrm{sum}((n-1)+1, N) \\
4 & p + n = \mathrm{sum}((n-1)+1, N) \\
3 & p = \mathrm{sum}(n+1, N) \\
2 & 0 = \mathrm{sum}(n+1, N)
\end{array}
$$

Jetzt seid ihr dran!

```
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)}
```

AWP	5	$p = \mathrm{sum}((n-1)+1, N)$
	4	$p + n = \mathrm{sum}((n-1)+1, N)$
	3	$p = \mathrm{sum}(n+1, N)$
	2	$0 = \mathrm{sum}(n+1, N)$
WVC	5	

Jetzt seid ihr dran!

```
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)}
```

AWP	5	$p = \mathrm{sum}((n-1)+1, N)$
	4	$p + n = \mathrm{sum}((n-1)+1, N)$
	3	$p = \mathrm{sum}(n+1, N)$
	2	$0 = \mathrm{sum}(n+1, N)$
WVC	5	$\emptyset$
	4	

Jetzt seid ihr dran!

```
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)}
```

AWP	5	$p = \mathrm{sum}((n-1)+1, N)$
	4	$p + n = \mathrm{sum}((n-1)+1, N)$
	3	$p = \mathrm{sum}(n+1, N)$
	2	$0 = \mathrm{sum}(n+1, N)$
WVC	5	$\emptyset$
	4	$\emptyset$
	3	

Jetzt seid ihr dran!

```
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)}
```

AWP	5	$p = \mathrm{sum}((n-1)+1, N)$
	4	$p + n = \mathrm{sum}((n-1)+1, N)$
	3	$p = \mathrm{sum}(n+1, N)$
	2	$0 = \mathrm{sum}(n+1, N)$
WVC	5	$\emptyset$
	4	$\emptyset$
	3	$\{(p = \mathrm{sum}(n+1, N) \land n > 0) \longrightarrow p + n = \mathrm{sum}((n-1)+1, N),$ $(p = \mathrm{sum}(n+1, N) \land \neg(n > 0)) \longrightarrow p = \mathrm{sum}(1, N)\}$
	2	

Jetzt seid ihr dran!

```
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)}
```

AWP	5	$p = \mathrm{sum}((n-1)+1, N)$
	4	$p + n = \mathrm{sum}((n-1)+1, N)$
	3	$p = \mathrm{sum}(n+1, N)$
	2	$0 = \mathrm{sum}(n+1, N)$
WVC	5	$\emptyset$
	4	$\emptyset$
	3	$\{(p = \mathrm{sum}(n+1, N) \land n > 0) \longrightarrow p + n = \mathrm{sum}((n-1)+1, N),$ $(p = \mathrm{sum}(n+1, N) \land \neg(n > 0)) \longrightarrow p = \mathrm{sum}(1, N)\}$
	2	$\emptyset \cup (3)$

Jetzt seid ihr dran!

```
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)}
```

$$WVC(\{0 \leq n \wedge n = N\}\; c\; \{p = \mathrm{sum}(1, N)\})$$

$$= \{(0 \leq n \wedge n = N) \longrightarrow 0 = \mathrm{sum}(n+1, N)\} \cup (3)$$

$$\begin{aligned} = \{&(0 \leq n \wedge n = N) \longrightarrow 0 = \mathrm{sum}(n+1, N), \\ &(p = \mathrm{sum}(n+1, N) \wedge n > 0) \longrightarrow p + n = \mathrm{sum}((n-1)+1, N), \\ &(p = \mathrm{sum}(n+1, N) \wedge \neg(n > 0)) \longrightarrow p = \mathrm{sum}(1, N)\} \end{aligned}$$

Jetzt seid ihr dran!

```
// {0 ≤ n ∧ n = N}
p= 0;
while (n>0) //** inv {0 ≤ n ∧ n ≤ N ∧ p = sum(n + 1, N); }*/ {
   p = p + n;
   n = n - 1;
   }
// {p = sum(1, N)}
```

AWP 5 |

Jetzt seid ihr dran!

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

AWP	5	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \mathrm{sum}((n-1)+1, N)$
	4	

Jetzt seid ihr dran!

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

AWP	5	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \mathrm{sum}((n-1)+1, N)$
	4	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \mathrm{sum}((n-1)+1, N)$
	3	

Jetzt seid ihr dran!

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

AWP	5	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \mathrm{sum}((n-1)+1, N)$
	4	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \mathrm{sum}((n-1)+1, N)$
	3	$0 \leq n \wedge n \leq N \wedge p = \mathrm{sum}(n+1, N)$
	2	

Jetzt seid ihr dran!

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

$$
\textbf{AWP}\quad
\begin{array}{r|l}
5 & 0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \mathrm{sum}((n-1)+1, N) \\
4 & 0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \mathrm{sum}((n-1)+1, N) \\
3 & 0 \leq n \wedge n \leq N \wedge p = \mathrm{sum}(n+1, N) \\
2 & 0 \leq n \wedge n \leq N \wedge 0 = \mathrm{sum}(n+1, N)
\end{array}
$$

Jetzt seid ihr dran!

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

AWP	5	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \mathrm{sum}((n-1)+1, N)$
	4	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \mathrm{sum}((n-1)+1, N)$
	3	$0 \leq n \wedge n \leq N \wedge p = \mathrm{sum}(n+1, N)$
	2	$0 \leq n \wedge n \leq N \wedge 0 = \mathrm{sum}(n+1, N)$
WVC	$5, 4$	

Jetzt seid ihr dran!

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

$$
\begin{array}{rr|l}
\textbf{AWP} & 5 & 0 \leq (n-1) \land (n-1) \leq N \land p = \mathrm{sum}((n-1)+1, N) \\
& 4 & 0 \leq (n-1) \land (n-1) \leq N \land p + n = \mathrm{sum}((n-1)+1, N) \\
& 3 & 0 \leq n \land n \leq N \land p = \mathrm{sum}(n+1, N) \\
& 2 & 0 \leq n \land n \leq N \land 0 = \mathrm{sum}(n+1, N) \\
\textbf{WVC} & 5,4 & \emptyset \\
& 3 &
\end{array}
$$

Jetzt seid ihr dran!

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

AWP	5	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \text{sum}((n-1)+1, N)$
	4	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \text{sum}((n-1)+1, N)$
	3	$0 \leq n \wedge n \leq N \wedge p = \text{sum}(n+1, N)$
	2	$0 \leq n \wedge n \leq N \wedge 0 = \text{sum}(n+1, N)$
WVC	5, 4	$\emptyset$
	3	$\{(0 \leq n \wedge n \leq N \wedge p = \text{sum}(n+1, N) \wedge n > 0) \longrightarrow (0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \text{sum}((n-1)+1, N)),$ $(n \geq 0 \wedge n \leq N \wedge p = \text{sum}(n+1, N) \wedge \neg(n > 0)) \longrightarrow p = \text{sum}(1, N)\}$
	2	

Jetzt seid ihr dran!

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

AWP	5	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p = \text{sum}((n-1)+1, N)$
	4	$0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \text{sum}((n-1)+1, N)$
	3	$0 \leq n \wedge n \leq N \wedge p = \text{sum}(n+1, N)$
	2	$0 \leq n \wedge n \leq N \wedge 0 = \text{sum}(n+1, N)$
WVC	5, 4	$\emptyset$
	3	$\{(0 \leq n \wedge n \leq N \wedge p = \text{sum}(n+1, N) \wedge n > 0) \longrightarrow (0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \text{sum}((n-1)+1, N)),$ $(n \geq 0 \wedge n \leq N \wedge p = \text{sum}(n+1, N) \wedge \neg(n > 0)) \longrightarrow p = \text{sum}(1, N)\}$
	2	$\emptyset \cup (3)$

Jetzt seid ihr dran!

```
// {0 ≤ n ∧ n = N}
p= 0;
while (n>0) //** inv {0 ≤ n ∧ n ≤ N ∧ p = sum(n + 1, N); }*/ {
   p = p + n;
   n = n - 1;
   }
// {p = sum(1, N)}
```

$$
\begin{aligned}
&WVC(\{0 \leq n \wedge n = N\}\, c\, \{p = \mathrm{sum}(1, N)\}) \\
&\quad = \{\{(0 \leq n \wedge n = N) \longrightarrow (0 \leq n \wedge n \leq N \wedge 0 = \mathrm{sum}(n+1, N))\} \cup (3) \\
&\quad = \{(0 \leq n \wedge n = N) \longrightarrow (0 \leq n \wedge n \leq N \wedge 0 = \mathrm{sum}(n+1, N)), \\
&\qquad (0 \leq n \wedge n \leq N \wedge p = \mathrm{sum}(n+1, N) \wedge n > 0) \\
&\qquad\qquad \longrightarrow (0 \leq (n-1) \wedge (n-1) \leq N \wedge p + n = \mathrm{sum}((n-1)+1, N)), \\
&\qquad (n \geq 0 \wedge n \leq N \wedge p = \mathrm{sum}(n+1, N) \wedge \neg(n > 0)) \longrightarrow p = \mathrm{sum}(1, N)\}
\end{aligned}
$$

Weiteres Beispiel: Maximales Element

1 // $\{0 < n\}$
2 `i= 0;`
3 `r= 0;`

4 `while (i != n)` //** inv $\{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}^{\varphi(i,r)}\}$ */
5 `{ if (a[r] < a[i]) {`
6 `      r= i; }`
7 `   else { }`
8 `   i= i+1; }`
9 // $\{\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n}_{\varphi(n,r)}\}$

AWP 8 |

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
```

$$\varphi(i,r)$$

```
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    else { }
8    i= i+1; }
9  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$\varphi(n, r)$$

AWP		
	8	$\varphi(i+1, r)$
	7	

Weiteres Beispiel: Maximales Element

1 `// ` $\{0 < n\}$
2 `i= 0;`
3 `r= 0;`

4 `while (i != n) //** inv ` $\{\overbrace{(\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \land 0 \le r < i}^{\varphi(i,r)}\}$ `*/`
5 `{ if (a[r] < a[i]) {`
6 `      r= i; }`
7 `   else { }`
8 `   i= i+1; }`
9 `// ` $\{\underbrace{(\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \land 0 \le r < n}_{\varphi(n,r)}\}$

AWP		
	8	$\varphi(i+1, r)$
	7	$\varphi(i+1, r)$
	6	

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;

                                               φ(i,r)
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    else { }
8    i= i+1; }
9  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
                     φ(n,r)
```

AWP

8	$\varphi(i+1, r)$
7	$\varphi(i+1, r)$
6	$\varphi(i+1, i)$
5	

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;

4  while (i != n) //** inv {(\overbrace{\forall j. 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i}^{\varphi(i,r)}} */
5  { if (a[r] < a[i]) {
6        r= i; }
7    else { }
8    i= i+1; }
9  // {(\underbrace{\forall j. 0 \le j < n \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n}_{\varphi(n,r)}}
```

AWP		
	8	$\varphi(i+1, r)$
	7	$\varphi(i+1, r)$
	6	$\varphi(i+1, i)$
	5	$(a[r] < a[i] \wedge \varphi(i+1, i)) \vee (\neg(a[r] < a[i]) \wedge \varphi(i+1, r))$
	4	

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;

4  while (i != n) //** inv {$\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r < i}^{\varphi(i,r)}$} */
5  { if (a[r] < a[i]) {
6        r= i; }
7    else { }
8    i= i+1; }
9  // {$\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \land 0 \leq r < n}_{\varphi(n,r)}$}
```

AWP

8	$\varphi(i+1, r)$
7	$\varphi(i+1, r)$
6	$\varphi(i+1, i)$
5	$(a[r] < a[i] \land \varphi(i+1, i)) \lor (\neg(a[r] < a[i]) \land \varphi(i+1, r))$
4	$\varphi(i, r)$
3	

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;

                                                  φ(i,r)
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    else { }
8    i= i+1; }
9  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
                    φ(n,r)
```

AWP

8	$\varphi(i+1, r)$
7	$\varphi(i+1, r)$
6	$\varphi(i+1, i)$
5	$(a[r] < a[i] \wedge \varphi(i+1, i)) \vee (\neg(a[r] < a[i]) \wedge \varphi(i+1, r))$
4	$\varphi(i, r)$
3	$\varphi(i, 0)$
2	

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;

                                       φ(i,r)
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    else { }
8    i= i+1; }
9  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
                     φ(n,r)
```

AWP		
	8	$\varphi(i+1, r)$
	7	$\varphi(i+1, r)$
	6	$\varphi(i+1, i)$
	5	$(a[r] < a[i] \wedge \varphi(i+1, i)) \vee (\neg(a[r] < a[i]) \wedge \varphi(i+1, r))$
	4	$\varphi(i, r)$
	3	$\varphi(i, 0)$
	2	$\varphi(0, 0)$

Weiteres Beispiel: Maximales Element

```
1  // $\{0 < n\}$
2  i= 0;
3  r= 0;

4  while (i != n) //** inv $\{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r < i}^{\varphi(i,r)}\}$ */
5  { if (a[r] < a[i]) {
6        r= i; }
7    else { }
8    i= i+1; }
9  // $\{\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \land 0 \leq r < n}_{\varphi(n,r)}\}$
```

WVC

$8, 7, 6, 5 \;\big|$

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
                                          φ(i,r)
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     else { }
8     i= i+1; }
9  // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
                    φ(n,r)
```

WVC

$$
\begin{array}{r|l}
8,7,6,5 & \emptyset \\
4 &
\end{array}
$$

Weiteres Beispiel: Maximales Element

1 `//` $\{0 < n\}$
2 `i= 0;`
3 `r= 0;`

4 `while (i!= n) //** inv` $\{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}^{\varphi(i,r)}\}$ `*/`
5 `{ if (a[r] < a[i]) {`
6 `      r= i; }`
7 `   else { }`
8 `   i= i+1; }`
9 `//` $\{\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n}_{\varphi(n,r)}\}$

WVC

$8, 7, 6, 5$	$\emptyset$
4	$(\varphi(i,r) \wedge i \neq n) \longrightarrow$ $((a[r] < a[i] \wedge \varphi(i+1, i)) \vee (\neg(a[r] < a[i]) \wedge \varphi(i+1, r)))$

Weiteres Beispiel: Maximales Element

1 `//` $\{0 < n\}$
2 `i= 0;`
3 `r= 0;`

4 `while (i != n)` `//**` inv $\{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < i}^{\varphi(i,r)}\}$ `*/`
5 `{ if (a[r] < a[i]) {`
6 `      r= i; }`
7 `   else { }`
8 `   i= i+1; }`
9 `//` $\{\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n}_{\varphi(n,r)}\}$

WVC

$$
\begin{array}{r|l}
8,7,6,5 & \emptyset \\
4 & (\varphi(i,r) \wedge i \neq n) \longrightarrow \\
 & \qquad ((a[r] < a[i] \wedge \varphi(i+1,i)) \vee (\neg(a[r] < a[i]) \wedge \varphi(i+1,r))) \\
 & (\varphi(i,r) \wedge \neg(i \neq n)) \longrightarrow \varphi(n,r) \\
3,2 &
\end{array}
$$

Weiteres Beispiel: Maximales Element

1 // $\{0 < n\}$

2 `i= 0;`

3 `r= 0;`

4 `while (i != n)` //** inv $\{\overbrace{(\forall j.\, 0 \leq j < i \longrightarrow a[j] \leq a[r]) \land 0 \leq r < i}^{\varphi(i,r)}\}$ */

5 `{ if (a[r] < a[i]) {`

6 `      r= i; }`

7 `   else { }`

8 `   i= i+1; }`

9 // $\{\underbrace{(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r]) \land 0 \leq r < n}_{\varphi(n,r)}\}$

WVC

$$
\begin{array}{r|l}
8,7,6,5 & \emptyset \\
4 & (\varphi(i,r) \land i \neq n) \longrightarrow \\
 & \qquad ((a[r] < a[i] \land \varphi(i+1,i)) \lor (\neg(a[r] < a[i]) \land \varphi(i+1,r))) \\
 & (\varphi(i,r) \land \neg(i \neq n)) \longrightarrow \varphi(n,r) \\
3,2 & \emptyset
\end{array}
$$

Weiteres Beispiel: Maximales Element

```
// {0 < n}
i= 0;
r= 0;
```

$$\texttt{while (i!= n) //** inv }\overbrace{\{(\forall j.\,0 \le j < i \longrightarrow a[j] \le a[r]) \land 0 \le r < i\}}^{\varphi(i,r)}\texttt{ */}$$

```
{ if (a[r] < a[i]) {
      r= i; }
  else { }
  i= i+1; }
```

$$\texttt{// }\{\underbrace{(\forall j.\,0 \le j < n \longrightarrow a[j] \le a[r]) \land 0 \le r < n}_{\varphi(n,r)}\}$$

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

Spracherweiterung: Explizite Spezifikationen

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

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

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

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

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

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

Überblick: Approximative schwächste Vorbedingung

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

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

Maximales Element mit Zusicherung

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

AWP 11 |

Maximales Element mit Zusicherung

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

$$\begin{array}{lr|l} \textbf{AWP} & 11 & \varphi(i+1, r) \\ & 9 & \end{array}$$

Maximales Element mit Zusicherung

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

AWP		
	11	$\varphi(i+1, r)$
	9	$\varphi(i, r) \land \neg(a[r] < a[i])$
	7	

Maximales Element mit Zusicherung

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

AWP		
	11	$\varphi(i+1, r)$
	9	$\varphi(i, r) \wedge \neg(a[r] < a[i])$
	7	$\varphi(i+1, i)$
	6	

Maximales Element mit Zusicherung

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

AWP

11	$\varphi(i+1, r)$	5	
9	$\varphi(i, r) \wedge \neg(a[r] < a[i])$		
7	$\varphi(i+1, i)$		
6	$\varphi(i, r) \wedge a[r] < a[i]$		

Maximales Element mit Zusicherung

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

AWP

$$\begin{array}{r|l}
11 & \varphi(i+1, r) \\
9 & \varphi(i, r) \land \neg(a[r] < a[i]) \\
7 & \varphi(i+1, i) \\
6 & \varphi(i, r) \land a[r] < a[i]
\end{array}
\qquad
\begin{array}{r|l}
5 & \varphi(i, r) \\
4 &
\end{array}$$

Maximales Element mit Zusicherung

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

AWP

11	$\varphi(i+1, r)$	5	$\varphi(i, r)$
9	$\varphi(i, r) \wedge \neg(a[r] < a[i])$	4	$\varphi(i, r)$
7	$\varphi(i+1, i)$	3	
6	$\varphi(i, r) \wedge a[r] < a[i]$		

Maximales Element mit Zusicherung

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

AWP

11	$\varphi(i+1, r)$	5	$\varphi(i, r)$
9	$\varphi(i, r) \wedge \neg(a[r] < a[i])$	4	$\varphi(i, r)$
7	$\varphi(i+1, i)$	3	$\varphi(i, 0)$
6	$\varphi(i, r) \wedge a[r] < a[i]$	2	

Maximales Element mit Zusicherung

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

AWP	11	$\varphi(i+1, r)$	5	$\varphi(i, r)$
	9	$\varphi(i, r) \wedge \neg(a[r] < a[i])$	4	$\varphi(i, r)$
	7	$\varphi(i+1, i)$	3	$\varphi(i, 0)$
	6	$\varphi(i, r) \wedge a[r] < a[i]$	2	$\varphi(0, 0)$

Maximales Element mit Zusicherung

```
// {0 < n}
i= 0;
r= 0;
while (i!= n) //** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n} */
{ if (a[r] < a[i]) {
      //{∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ a[r] < a[i]} */
      r= i; }
    else {
      //{∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])} */
      }
    i= i+1; }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

WVC 11

Maximales Element mit Zusicherung

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

WVC		
	11	$\emptyset$
	9	

Maximales Element mit Zusicherung

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

WVC	11	$\emptyset$
	9	$(\varphi(i,r) \land \neg(a[r] < a[i]))$
		$\longrightarrow \varphi(i+1,r)$
	7	

Maximales Element mit Zusicherung

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

WVC	11	$\emptyset$
	9	$(\varphi(i, r) \wedge \neg(a[r] < a[i])) \longrightarrow \varphi(i+1, r)$
	7	$\emptyset$
	6	

Maximales Element mit Zusicherung

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

WVC 11	$\emptyset$		5	
9	$(\varphi(i,r) \wedge \neg(a[r] < a[i])) \longrightarrow \varphi(i+1,r)$			
7	$\emptyset$			
6	$(\varphi(i,r) \wedge a[r] < a[i]) \longrightarrow \varphi(i+1,i)$			

Maximales Element mit Zusicherung

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

WVC

11	$\emptyset$
9	$(\varphi(i,r) \land \neg(a[r] < a[i])) \longrightarrow \varphi(i+1,r)$
7	$\emptyset$
6	$(\varphi(i,r) \land a[r] < a[i]) \longrightarrow \varphi(i+1,i)$
5	$(\varphi(i,r) \land \neg(a[r] < a[i])) \longrightarrow \varphi(i+1,r)$ $(\varphi(i,r) \land a[r] < a[i]) \longrightarrow \varphi(i+1,i)$

Maximales Element mit Zusicherung

```
// {0 < n}
i= 0;
r= 0;
while (i!= n) //** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n} */
{ if (a[r] < a[i]) {
      //{∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ a[r] < a[i]} */
      r= i; }
  else {
      //{∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])} */
      }
  i= i+1; }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

WVC 4 |

Maximales Element mit Zusicherung

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

WVC	4	(5)
		$(\varphi(i, r) \wedge i \neq n) \longrightarrow \varphi(i+1, r)$
		$(\varphi(i, r) \wedge \neg(i \neq n)) \longrightarrow \varphi(n, r)$
	$3, 2$	

Maximales Element mit Zusicherung

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

WVC

$$
\begin{array}{r|l}
4 & (5) \\
 & (\varphi(i, r) \land i \neq n) \longrightarrow \varphi(i + 1, r) \\
 & (\varphi(i, r) \land \neg(i \neq n)) \longrightarrow \varphi(n, r) \\
3, 2 & \emptyset
\end{array}
$$

Maximales Element mit Zusicherung

```
// {0 < n}
i= 0;
r= 0;
while (i != n) //** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n} */

{ if (a[r] < a[i]) {
    //{∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ a[r] < a[i]} */
    r= i; }
  else {
    //{∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])} */
    }
  i= i+1; }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

- Explizite Zusicherungen verkleinern Verifikationsbedingung

Zusammenfassung

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

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**?

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 9 vom 16.06.20

Vorwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Feedback Online-Lehre

- Was kann besser werden?
  - Aufgezeichnete Vorlesungen?
  - Lesematerial/“Flipped Classroom”?
  - Andere Formen der Gruppenarbeit?
- Was ist gut/schlecht an Zoom?
  - Technische Probleme?
  - Funktionalität?
  - Break-Out Rooms?
- Was wollen wir ändern?

Fahrplan

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

Idee

- Hier ist ein einfaches Programm:

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

Idee

- Hier ist ein einfaches Programm:

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

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}
```

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.

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)

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

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

Der Floyd-Hoare-Kalkül Vorwärts

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 seinM
  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:

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

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)

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

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

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

If-Regel Vorwärts

- Abgeleitete If-Regel:

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B_1\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B_2\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B_1 \vee B_2\}}$$

- Durch Verkettung der If-Regel mit Weakening: $B_1 \Longrightarrow B_1 \vee B_2$

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 9.1: Das Leben mit 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)

$$\exists x.\, x < 7 \qquad \exists x.\, x < 3 \wedge x > 7 \qquad \exists x.\, x < 7 \vee x < 3$$

$$\exists y \exists x.\, x + 3 = y \qquad \forall x \exists y.\, x * y = 3 \qquad \exists x \forall y.\, x * y > 1$$

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:

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

Vorwärtsverkettung

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

```
// {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:

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

Vorwärtsverkettung

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

```
// {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:

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

Vorwärtsverkettung

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

```
// {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:

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

Regeln der Vorwärtsverkettung

Eigenschaften des Existenzquantors:

$$P[V] \land V = t \Longrightarrow P[t/V] \land V = t \qquad (1)$$

$$\exists V.\, P[V] \land V = t \Longrightarrow P[t/V] \qquad (2)$$

$$\text{wenn } V \notin FV(Q) \text{ dann } (\exists V.\, P) \land Q \Longleftrightarrow \exists V.\, P \land Q \qquad (3)$$

$$\text{wenn } V \notin FV(P) \text{ dann } \exists V.\, P \Longrightarrow P \qquad (4)$$

Regeln der Vorwärtsverkettung

Eigenschaften des Existenzquantors:

$$P[V] \wedge V = t \Longrightarrow P[t/V] \wedge V = t \tag{1}$$

$$\exists V. P[V] \wedge V = t \Longrightarrow P[t/V] \tag{2}$$

$$\text{wenn } V \notin FV(Q) \text{ dann } (\exists V. P) \wedge Q \Longleftrightarrow \exists V. P \wedge Q \tag{3}$$

$$\text{wenn } V \notin FV(P) \text{ dann } \exists V. P \Longrightarrow P \tag{4}$$

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)

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a ;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a ;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b- 2*a;
// {∃a2. (∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b - 2a)[a2/a]}
// {∃a2∃b1∃a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b - 2a2}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b- 2*a;
// {∃a2. (∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b − 2a)[a2/a]}
// {∃a2∃b1∃a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b − 2a2}
// {∃a2∃b1∃a1. a1 < b1 ∧ b = 3a2 + b1 ∧ a = b − 2a2 ∧ a2 = b1 + a1}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a ;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b ;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b- 2*a ;
// {∃a2. (∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b − 2a)[a2/a]}
// {∃a2∃b1∃a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b − 2a2}
// {∃a2∃b1∃a1. a1 < b1 ∧ b = 3a2 + b1 ∧ a = b − 2a2 ∧ a2 = b1 + a1}
// {∃b1∃a1. a1 < b1 ∧ b = 3(b1 + a1) + b1 ∧ a = b − 2(b1 + a1)}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b− 2*a;
// {∃a2. (∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b − 2a)[a2/a]}
// {∃a2∃b1∃a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b − 2a2}
// {∃a2∃b1∃a1. a1 < b1 ∧ b = 3a2 + b1 ∧ a = b − 2a2 ∧ a2 = b1 + a1}
// {∃b1∃a1. a1 < b1 ∧ b = 3(b1 + a1) + b1 ∧ a = b − 2(b1 + a1)}
// {∃b1∃a1. a1 < b1 ∧ b = 3b1 + 3a1 + b1 ∧ a = b − 2b1 − 2a1)}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b- 2*a;
// {∃a2. (∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b − 2a)[a2/a]}
// {∃a2∃b1∃a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b − 2a2}
// {∃a2∃b1∃a1. a1 < b1 ∧ b = 3a2 + b1 ∧ a = b − 2a2 ∧ a2 = b1 + a1}
// {∃b1∃a1. a1 < b1 ∧ b = 3(b1 + a1) + b1 ∧ a = b − 2(b1 + a1)}
// {∃b1∃a1. a1 < b1 ∧ b = 3b1 + 3a1 + b1 ∧ a = b − 2b1 − 2a1)}
// {∃b1∃a1. a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = b − 2b1 − 2a1)}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1. (a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1. a1 < b ∧ a = b + a1}
b= 3*a+b;
// {∃b1. (∃a1. a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b- 2*a;
// {∃a2. (∃b1∃a1. a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b − 2a)[a2/a]}
// {∃a2∃b1∃a1. a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b − 2a2}
// {∃a2∃b1∃a1. a1 < b1 ∧ b = 3a2 + b1 ∧ a = b − 2a2 ∧ a2 = b1 + a1}
// {∃b1∃a1. a1 < b1 ∧ b = 3(b1 + a1) + b1 ∧ a = b − 2(b1 + a1)}
// {∃b1∃a1. a1 < b1 ∧ b = 3b1 + 3a1 + b1 ∧ a = b − 2b1 − 2a1)}
// {∃b1∃a1. a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = b − 2b1 − 2a1)}
// {∃b1∃a1. a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = (4b1 + 3a1) − 2b1 − 2a1)}
```

Beispiel Vorwärtsverkettung

```
// {a < b}
a= b+a;
// {∃a1.(a < b)[a1/a] ∧ a = (b + a)[a1/a]}
// {∃a1.a1 < b ∧ a = b + a1}
b= 3∗a+b;
// {∃b1.(∃a1.a1 < b ∧ a = b + a1)[b1/b] ∧ b = (3a + b)[b1/b]}
// {∃b1∃a1.a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1}
a= b− 2∗a;
// {∃a2.(∃b1∃a1.a1 < b1 ∧ a = b1 + a1 ∧ b = 3a + b1)[a2/a] ∧ a = (b − 2a)[a2/a]}
// {∃a2∃b1∃a1.a1 < b1 ∧ a2 = b1 + a1 ∧ b = 3a2 + b1 ∧ a = b − 2a2}
// {∃a2∃b1∃a1.a1 < b1 ∧ b = 3a2 + b1 ∧ a = b − 2a2 ∧ a2 = b1 + a1}
// {∃b1∃a1.a1 < b1 ∧ b = 3(b1 + a1) + b1 ∧ a = b − 2(b1 + a1)}
// {∃b1∃a1.a1 < b1 ∧ b = 3b1 + 3a1 + b1 ∧ a = b − 2b1 − 2a1)}
// {∃b1∃a1.a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = b − 2b1 − 2a1)}
// {∃b1∃a1.a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = (4b1 + 3a1) − 2b1 − 2a1)}
// {∃b1∃a1.a1 < b1 ∧ b = 4b1 + 3a1 ∧ a = 2b1 + a1}
```

Beispiel Vorwärtsverkettung

// $\{a < b\}$

a= b+a;

// $\{\exists a_1.\,(a < b)[a_1/a] \wedge a = (b + a)[a_1/a]\}$

// $\{\exists a_1.\,a_1 < b \wedge a = b + a_1\}$

b= 3*a+b;

// $\{\exists b_1.\,(\exists a_1.\,a_1 < b \wedge a = b + a_1)[b_1/b] \wedge b = (3a + b)[b_1/b]\}$

// $\{\exists b_1 \exists a_1.\,a_1 < b_1 \wedge a = b_1 + a_1 \wedge b = 3a + b_1\}$

a= b− 2*a;

// $\{\exists a_2.\,(\exists b_1 \exists a_1.\,a_1 < b_1 \wedge a = b_1 + a_1 \wedge b = 3a + b_1)[a_2/a] \wedge a = (b - 2a)[a_2/a]\}$

// $\{\exists a_2 \exists b_1 \exists a_1.\,a_1 < b_1 \wedge a_2 = b_1 + a_1 \wedge b = 3a_2 + b_1 \wedge a = b - 2a_2\}$

// $\{\exists a_2 \exists b_1 \exists a_1.\,a_1 < b_1 \wedge b = 3a_2 + b_1 \wedge a = b - 2a_2 \wedge a_2 = b_1 + a_1\}$

// $\{\exists b_1 \exists a_1.\,a_1 < b_1 \wedge b = 3(b_1 + a_1) + b_1 \wedge a = b - 2(b_1 + a_1)\}$

// $\{\exists b_1 \exists a_1.\,a_1 < b_1 \wedge b = 3b_1 + 3a_1 + b_1 \wedge a = b - 2b_1 - 2a_1))\}$

// $\{\exists b_1 \exists a_1.\,a_1 < b_1 \wedge b = 4b_1 + 3a_1 \wedge a = b - 2b_1 - 2a_1))\}$

// $\{\exists b_1 \exists a_1.\,a_1 < b_1 \wedge b = 4b_1 + 3a_1 \wedge a = (4b_1 + 3a_1) - 2b_1 - 2a_1))\}$

// $\{\exists b_1 \exists a_1.\,a_1 < b_1 \wedge b = 4b_1 + 3a_1 \wedge a = 2b_1 + a_1\}$

Arbeitsblatt 9.2: Vorwärtsverkettung

Gegeben folgendes Programm. Berechnet die Vorwärtsverkettung der Vorbedingung

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

Was bewirkt das Programm?

Vorwärtsverkettung

- Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- Vorteil: Vorbedingung bleibt kleiner
- Nachteil: in der Anwendung **umständlicher**
- Vereinfachung benötigt Rechnungmit Existenzquantor

Zwischenfazit: Der Floyd-Hoare-Kalkül ist symmetrisch

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

Vorwärtsberechnung von Verifikationsbedingungen

Stärkste Nachbedingung

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

Stärkste Nachbedingung

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

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

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

Berechnung von Nachbedingungen

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

Überblick: Approximative stärkste Nachbedingung

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

$$
\begin{aligned}
\mathrm{svc}(P, \{\,\}) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, x = e) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, c_1; c_2) &\stackrel{def}{=} \mathrm{svc}(P, c_1) \cup \mathrm{svc}(\mathrm{asp}(P, c_1), c_2) \\
\mathrm{svc}(P, \textbf{if } (b)\ c_0\ \textbf{ else }\ c_1) &\stackrel{def}{=} \mathrm{svc}(P \land b, c_0) \cup \mathrm{svc}(P \land \neg b, c_1)
\end{aligned}
$$

Überblick: Approximative stärkste Nachbedingung

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

$$
\begin{aligned}
\mathrm{svc}(P, \{\,\}) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, x = e) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, c_1; c_2) &\stackrel{def}{=} \mathrm{svc}(P, c_1) \cup \mathrm{svc}(\mathrm{asp}(P, c_1), c_2) \\
\mathrm{svc}(P, \textbf{if } (b)\ c_0\ \textbf{ else }\ c_1) &\stackrel{def}{=} \mathrm{svc}(P \wedge b, c_0) \cup \mathrm{svc}(P \wedge \neg b, c_1) \\
\mathrm{svc}(P, //{*}{*}\ \{q\}\ {*}/) &\stackrel{def}{=} \{P \longrightarrow q\}
\end{aligned}
$$

Überblick: Approximative stärkste Nachbedingung

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

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

Überblick: Approximative stärkste Nachbedingung

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

Beispiel: Fakultät

```
// {0 ≤ n}
p= 1;
c= 1;
while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */
   p = p * c;
   c = c + 1;
   }
// {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp_2 =
   //
3  c= 1;
   // asp_3 =
   //
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     //
6    c = c + 1;
     //
7    }
   // asp_4 =
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp_2 = {∃V. 0 ≤ n[V/p] ∧ p = (1[V/p])}
   //
3  c= 1;
   // asp_3 =
   //
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     //
6    c = c + 1;
     //
7    }
   // asp_4 =
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp_2 = {∃V. 0 ≤ n[V/p] ∧ p = (1[V/p])}
   // asp_2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp_3 =
   //
4  while (c <= n)  //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     //
6    c = c + 1;
     //
7    }
   // asp_4 =
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp_2 = {∃V. 0 ≤ n[V/p] ∧ p = (1[V/p])}
   // asp_2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp_3 = {∃V. (0 ≤ n ∧ p = 1)[V/c] ∧ c = (1[V/c])}
   //
4  while (c <= n)  //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     //
6    c = c + 1;
     //
7    }
   // asp_4 =
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

1 // $\{0 \leq n\}$
2 `p= 1;`
// $asp_2 = \{\exists V.\, 0 \leq n[V/p] \land p = (1[V/p])\}$
// $asp_2 = \{0 \leq n \land p = 1\}$
3 `c= 1;`
// $asp_3 = \{\exists V.\, (0 \leq n \land p = 1)[V/c] \land c = (1[V/c])\}$
// $asp_3 = \{0 \leq n \land p = 1 \land c = 1\}$
4 **while** `(c <= n)` //** inv $\{p = (c-1)! \land c - 1 \leq n\}$; */ {
5 `p = p * c;`
//
6 `c = c + 1;`
//
7 }
// $asp_4 =$
8 // $\{p = n!\}$

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp₂ = {∃V. 0 ≤ n[V/p] ∧ p = (1[V/p])}
   // asp₂ = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp₃ = {∃V. (0 ≤ n ∧ p = 1)[V/c] ∧ c = (1[V/c])}
   // asp₃ = {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    }
   // asp₄ = {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1   // {0 ≤ n}
2   p= 1;
    // svc_2 =
3   c= 1;
    // svc_3 =
4   while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
      // svc_5 =
6     c = c + 1;
      // svc_6 =
7     }
      // svc_4 =
8   // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1  // {0 ≤ n}
2  p= 1;
   // svc_2 = ∅
3  c= 1;
   // svc_3 =
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     // svc_5 =
6    c = c + 1;
     // svc_6 =
7    }
     // svc_4 =
8  // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1  // {0 ≤ n}
2  p= 1;
   // svc_2 = ∅
3  c= 1;
   // svc_3 = ∅
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     // svc_5 =
6    c = c + 1;
     // svc_6 =
7    }
     // svc_4 =
8  // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1  // {0 ≤ n}
2  p= 1;
   // svc_2 = ∅
3  c= 1;
   // svc_3 = ∅
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     // svc_5 = ∅
6    c = c + 1;
     // svc_6 =
7    }
     // svc_4 =
8  // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1   // {0 ≤ n}
2   p= 1;
    // svc_2 = ∅
3   c= 1;
    // svc_3 = ∅
4   while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
      // svc_5 = ∅
6     c = c + 1;
      // svc_6 = ∅
7     }
      // svc_4 =
8   // {p = n!}
```

Fakultät: Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1   // {0 ≤ n}
2   p= 1;
    // svc_2 = ∅
3   c= 1;
    // svc_3 = ∅
4   while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
      // svc_5 = ∅
6     c = c + 1;
      // svc_6 = ∅
7     }
      // svc_4 = {asp_3 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n), asp_6 ⟹ (p = (c − 1)! ∧
    c − 1 ≤ n)}
8   // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp3 = {0 ≤ n ∧ p = 1 ∧ c = 1}
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5   p = p * c;
    // asp5 =
    //
    //
    c = c + 1;
    // asp6 =
    //
    //
    }
   // asp4 = {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp_2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp_3 = {0 ≤ n ∧ p = 1 ∧ c = 1}
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     // asp_5 = {∃V_1.(p = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n)[V_1/p] ∧ p = (p · c)[V_1/p]}
     //
     //
     c = c + 1;
     // asp_6 =
     //
     //
     }
   // asp_4 = {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

```
1  // {0 ≤ n}
2  p= 1;
   // asp2 = {0 ≤ n ∧ p = 1}
3  c= 1;
   // asp3 = {0 ≤ n ∧ p = 1 ∧ c = 1}
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5    p = p * c;
     // asp5 = {∃V1.(p = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n)[V1/p] ∧ p = (p · c)[V1/p]}
     // asp5 = {∃V1.(V1 = (c − 1)! ∧ (c − 1) ≤ n ∧ c ≤ n) ∧ p = (V1 · c)}
     //
     c = c + 1;
     // asp6 =
     //
     //
   }
   // asp4 = {¬(c ≤ n) ∧ p = (c − 1)! ∧ c − 1 ≤ n}
8  // {p = n!}
```

Beispiel: Fakultät, stärkste Nachbedingung

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

1 // $\{0 \leq n\}$
2 p= 1;
// $asp_2 = \{0 \leq n \wedge p = 1\}$
3 c= 1;
// $asp_3 = \{0 \leq n \wedge p = 1 \wedge c = 1\}$
4 **while** (c <= n) //** inv $\{p = (c-1)! \wedge c - 1 \leq n\}$; */ {
5 p = p * c;
// $asp_5 = \{\exists V_1. (p = (c-1)! \wedge (c-1) \leq n \wedge c \leq n)[V_1/p] \wedge p = (p \cdot c)[V_1/p]\}$
// $asp_5 = \{\exists V_1. (V_1 = (c-1)! \wedge (c-1) \leq n \wedge c \leq n) \wedge p = (V_1 \cdot c)\}$
// $asp_5 = \{c - 1 \leq n \wedge c \leq n \wedge p = (c-1)! \cdot c\}$
6 c = c + 1;
// $asp_6 =$
//
//
}
// $asp_4 = \{\neg(c \leq n) \wedge p = (c-1)! \wedge c - 1 \leq n\}$
8 // $\{p = n!\}$

Beispiel: Fakultät, stärkste Nachbedingung

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

1 // $\{0 \leq n\}$
2 `p= 1;`
// $asp_2 = \{0 \leq n \land p = 1\}$
3 `c= 1;`
// $asp_3 = \{0 \leq n \land p = 1 \land c = 1\}$
4 **while** `(c <= n)` //** inv $\{p = (c-1)! \land c - 1 \leq n\}$; */ {
5 `p = p * c;`
// $asp_5 = \{\exists V_1. (p = (c-1)! \land (c-1) \leq n \land c \leq n)[V_1/p] \land p = (p \cdot c)[V_1/p]\}$
// $asp_5 = \{\exists V_1. (V_1 = (c-1)! \land (c-1) \leq n \land c \leq n) \land p = (V_1 \cdot c)\}$
// $asp_5 = \{c - 1 \leq n \land c \leq n \land p = (c-1)! \cdot c\}$
6 `c = c + 1;`
// $asp_6 = \{\exists V_2. (c - 1 \leq n \land c \leq n \land p = (c-1)! \cdot c)[V_2/c] \land c = (c+1)[V_2/c]\}$
//
//
}
// $asp_4 = \{\neg(c \leq n) \land p = (c-1)! \land c - 1 \leq n\}$
8 // $\{p = n!\}$

Beispiel: Fakultät, stärkste Nachbedingung

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

1 // $\{0 \leq n\}$

2 `p= 1;`

// $asp_2 = \{0 \leq n \land p = 1\}$

3 `c= 1;`

// $asp_3 = \{0 \leq n \land p = 1 \land c = 1\}$

4 **while** `(c <= n)` //** inv $\{p = (c-1)! \land c - 1 \leq n\}$; */ {

5 `p = p * c;`

// $asp_5 = \{\exists V_1. (p = (c-1)! \land (c-1) \leq n \land c \leq n)[V_1/p] \land p = (p \cdot c)[V_1/p]\}$

// $asp_5 = \{\exists V_1. (V_1 = (c-1)! \land (c-1) \leq n \land c \leq n) \land p = (V_1 \cdot c)\}$

// $asp_5 = \{c - 1 \leq n \land c \leq n \land p = (c-1)! \cdot c\}$

6 `c = c + 1;`

// $asp_6 = \{\exists V_2. (c - 1 \leq n \land c \leq n \land p = (c-1)! \cdot c)[V_2/c] \land c = (c+1)[V_2/c]\}$

// $asp_6 = \{\exists V_2. (V_2 - 1 \leq n \land V_2 \leq n \land p = (V_2 - 1)! \cdot V_2) \land c = (V_2 + 1)\}$

//

}

// $asp_4 = \{\neg(c \leq n) \land p = (c-1)! \land c - 1 \leq n\}$

8 // $\{p = n!\}$

Beispiel: Fakultät, stärkste Nachbedingung

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

1 `//` $\{0 \leq n\}$
2 `p= 1;`
`//` $asp_2 = \{0 \leq n \land p = 1\}$
3 `c= 1;`
`//` $asp_3 = \{0 \leq n \land p = 1 \land c = 1\}$
4 **`while`** `(c <= n) //**` inv $\{p = (c-1)! \land c - 1 \leq n\}$; `*/ {`
5 `p = p * c;`
`//` $asp_5 = \{\exists V_1. (p = (c-1)! \land (c-1) \leq n \land c \leq n)[V_1/p] \land p = (p \cdot c)[V_1/p]\}$
`//` $asp_5 = \{\exists V_1. (V_1 = (c-1)! \land (c-1) \leq n \land c \leq n) \land p = (V_1 \cdot c)\}$
`//` $asp_5 = \{c - 1 \leq n \land c \leq n \land p = (c-1)! \cdot c\}$
6 `c = c + 1;`
`//` $asp_6 = \{\exists V_2. (c - 1 \leq n \land c \leq n \land p = (c-1)! \cdot c)[V_2/c] \land c = (c+1)[V_2/c]\}$
`//` $asp_6 = \{\exists V_2. (V_2 - 1 \leq n \land V_2 \leq n \land p = (V_2 - 1)! \cdot V_2) \land c = (V_2 + 1)\}$
`//` $asp_6 = \{c - 2 \leq n \land c - 1 \leq n \land p = (c-2)! \cdot (c-1)\}$
7 `}`
`//` $asp_4 = \{\neg(c \leq n) \land p = (c-1)! \land c - 1 \leq n\}$
8 `//` $\{p = n!\}$

Beispiel: Fakultät, Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

```
1   // {0 ≤ n}
2   p= 1;
    // svc_2 = ∅
    c= 1;
    // svc_3 = ∅
4   while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n}; */ {
5     p = p * c;
      // svc_5 = ∅
6     c = c + 1;
      // svc_6 = ∅
7    }
    // svc_4 = {asp_3 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n),
    //                  asp_6 ⟹ (p = (c − 1)! ∧ c − 1 ≤ n)}
    //
    //
    //
8   // {p = n!}
```

Beispiel: Fakultät, Verifikationsbedingungen

Notation: svc_x = in Zeile x generierte Verifikationsbedingung

1 // $\{0 \leq n\}$
2 `p= 1;`
// $svc_2 = \emptyset$
`c= 1;`
// $svc_3 = \emptyset$
4 `while (c <= n)` //** inv $\{p = (c-1)! \wedge c-1 \leq n\}$; */ {
5 `p = p * c;`
// $svc_5 = \emptyset$
6 `c = c + 1;`
// $svc_6 = \emptyset$
7 }
// $svc_4 = \{asp_3 \Longrightarrow (p = (c-1)! \wedge c-1 \leq n),$
// $asp_6 \Longrightarrow (p = (c-1)! \wedge c-1 \leq n)\}$
// $svc_4 = \{(0 \leq n \wedge p = 1 \wedge c = 1) \Longrightarrow (p = (c-1)! \wedge c-1 \leq n),$
// $(c-2 \leq n \wedge c-1 \leq n \wedge p = (c-2)! \cdot (c-1))$
// $\Longrightarrow (p = (c-1)! \wedge c-1 \leq n)\}$
8 // $\{p = n!\}$

Schließlich zu zeigen

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

Arbeitsblatt 9.3: Jetzt seid ihr dran!

Berechnet die stärkste Nachbedingung und Verifikationsbedingungen für die ganzzahlige Division:

```
/** {0 ≤ a} */
r= a;
q= 0;
while (b <= r) /** inv { a == b*q+r ∧ 0 <= r } */ {
  r= r-b;
  q= q+1;
  }
/** { a == b*q+ r ∧ 0 ≤ r ∧ r < b } */
```

Beispiel: Suche nach dem Maximalen Element

```
// {0 < n}
i= 0;
r= 0;
while (i != n) //** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n} */
  if (a[r] < a[i]) {
    r= i;
    }
  else {
    }
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

- Problem: wir müssen u.a. zeigen

$$\begin{aligned}(\exists V_1.\,(\forall j.\, 0 \leq j < i-1 \longrightarrow a[j] \leq a[V_1]) \wedge \\ i-1 \neq n \wedge a[V_1] < a[i-1] \wedge r = i-1) \longrightarrow 0 \leq r < n\end{aligned}$$

  Deshalb: Invariante **verstärken**!

Beispiel: Suche nach dem Maximalen Element

Verstärkte Invariante (und Schleifenbedingung):

```
// {0 < n}
i= 0;
r= 0;
while (i< n) //** inv (∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r])
                      ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n */ {
  if (a[r] < a[i]) {
    r= i;
    }
  else {
    }
  i= i+1;
  }
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$$
\begin{aligned}
(\exists V_1.\, (\forall j.\, 0 \leq j < i - 1 \longrightarrow a[j] \leq a[V_1]) \wedge \\
0 \leq i - 1 < n \wedge a[V_1] < a[i-1] \wedge r = i - 1) \longrightarrow 0 \leq r < n
\end{aligned}
$$

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 10 vom 23.06.20

Modellierung und Spezifikation

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
i= 0;
r= 0;
while (i< n) {
  if (a[r] < a[i]) {
    r= i;
    }
  else {
    }
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀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]}
```

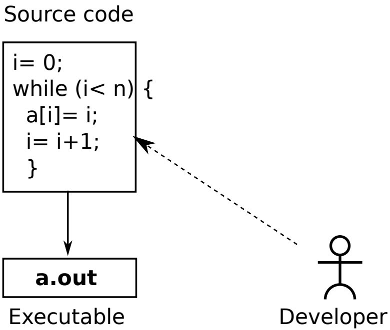
- Alternativ würden man auch gerne ein Prädikat definieren können

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

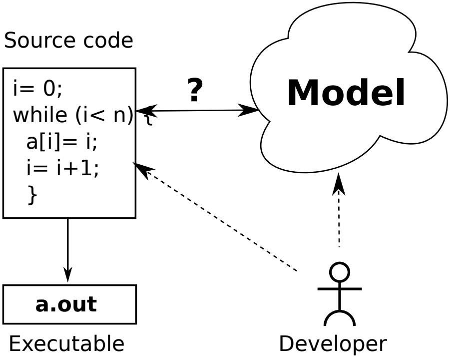
- . . . und damit beweisen dass:

$$//\{\forall a \forall n.\, sorted(a, n) \longrightarrow \forall i, j.\, 0 \leq i \leq j \leq n \longrightarrow a[i] \leq a[j]\}$$

Generelles Problem: Modellbildung



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 (ACSL, JML)

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

```
int binary_search(int val, int buf[], unsigned len)
{
  // {0 ≤ len}
  int low, high, mid, res;
  low = 0; high = len;
  while (low < high) {
    mid= (low+ high)/2;
    if (buf[mid] < val)
      low = mid + 1;
    else
      high = mid;
  }
  if (low < len && buf[low] == val)
    res= low;
  else
    res= −1;
  // { res ≠ −1 ⟶ buf[res] = val ∧                 }
        res = −1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val
}
```

Binäre Suche, korrekt

```
int binary_search(int val, int buf[], unsigned len)
{
  // {0 ≤ len}
  int low, high, mid, res;
  low = 0; high = len;
  while (low < high) {
    mid= low+ (high-low)/2;
    if (buf[mid] < val)
      low = mid + 1;
    else
      high = mid;
  }
  if (low < len && buf[low] == val)
    res= low;
  else
    res= -1;
  // { 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

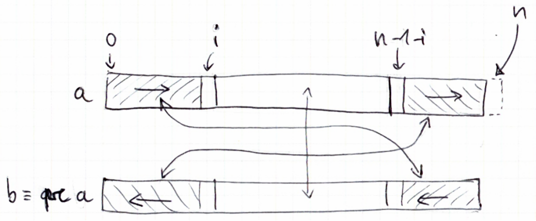
Typen: Felder

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

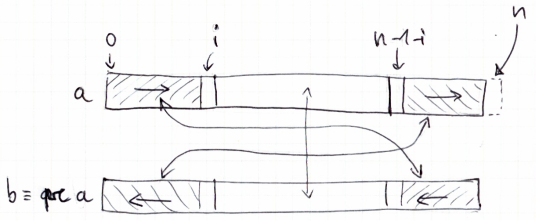
Ein längeres Beispiel: reverse in-place

```
i= 0;
// {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
while (i< n/2) {
  // ???
  tmp= a[n−1−i];
  a[n−1−i]= a[i];
  a[i]= tmp;
  i= i+1;
}
// {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```

reverse-in-place: die Invariante



reverse-in-place: die Invariante



Mathematisch:

$$\begin{aligned}
\{\ & \forall j. 0 \leq j < i \longrightarrow a[j] = b[n-1-j] \wedge \\
& \forall j. n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge \\
& \forall j. i \leq j \leq n-1-i \longrightarrow a[j] = b[j] \quad \}
\end{aligned}$$

Ein längeres Beispiel: reverse in-place

```
i= 0;
// {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
while (i< n/2) {
  // { ∀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]  }
  tmp= a[n−1−i];
  a[n−1−i]= a[i];
  a[i]= tmp;
  i= i+1;
}
// {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```

Arbeitsblatt 10.1: Jetzt seit ihr dran

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

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

- Dazu berechnet ihr $\mathrm{awp}(c, I)$, mit

  $c =$

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

$$\begin{aligned} I = \{ & \forall j. 0 \leq j < i \longrightarrow a[j] = b[n-1-j] \wedge \\ & \forall j. n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge \\ & \forall j. i \leq j \leq n-1-i \longrightarrow a[j] = b[j] \ \} \end{aligned}$$

- Ihr braucht noch nichts zu beweisen...

Vereinfacht mit Modellbildung

- $\text{seq}(a, n)$ ist ein Feld der Länge n repräsentiert als Liste (Sequenz)
- Aktionen auf Sequenzen:
  - $:$, $[]$ — Listenkonstruktoren
  - $\text{rev}(a)$ — Reverse
  - $a[i : j]$ — Slicing (à la Python)
  - $++$ — Konkatenation

Interaktion mit der Substitution

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

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

- Damit ist

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

Reverse-in-Place mit Listen

```
i= 0;
// {bs = seq(a, n)}
while (i< n/2)
  /** inv
  */ {
  tmp= a[n-1-i];
  a[n-i-1]= a[i];
  a[i]= tmp;
  i= i+1;
}
// {as = seq(a, n) ⟹ rev(as) = bs}
```

- Damit vereinfachte VCs und vereinfachter Beweis.

Reverse-in-Place mit Listen

```
i= 0;
// {bs = seq(a, n)}
while (i< n/2)
   /** inv   as = seq(a, n) ⟹
              rev(as[n − i : n])++as[i : n − i]++ rev(as[0 : i]) = bs
   */ {
   tmp= a[n−1−i];
   a[n−i−1]= a[i];
   a[i]= tmp;
   i= i+1;
}
// {as = seq(a, n) ⟹ rev(as) = bs}
```

- Damit vereinfachte VCs und vereinfachter Beweis.

Arbeitsblatt 10.2: Beweise mit Listen

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

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

- 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, $++$

$$\begin{aligned}
\mathrm{rev}([]) &== [] \\
\mathrm{rev}(x : xs) &== \mathrm{rev}(xs){+}{+}[x] \\
[]{+}{+}ys &== ys \\
(x : xs){+}{+}ys &== x : (xs{+}{+}ys)
\end{aligned}$$

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

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

Zusicherungen (Assertions)

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

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

Zusicherungen (Assertions)

- Erweiterung von **Aexp** and **Bexp** durch
  - **Logische** Variablen **Var** $v := N, M, L, U, V, X, Y, Z$
  - 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:

$$
\begin{aligned}
n! &== \mathit{factorial}(n) \\
i \leq 0 \longrightarrow \mathit{factorial}(i) &== 1 \\
i > 0 \longrightarrow \mathit{factorial}(i) &== i \cdot \mathit{factorial}(i-1) \\
\sum_{i=a}^{b} i &== \mathit{sum}(a, b) \\
a > b \longrightarrow \mathit{sum}(a, b) &== 0 \\
a <= b \longrightarrow \mathit{sum}(a, b) &== a + \mathit{sum}(a+1, b)
\end{aligned}
$$

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:

$$
\begin{aligned}
a \leq b \longrightarrow [a \mathinner{..} b] &== a : [a+1 \mathinner{..} b] \\
a > b \longrightarrow [a \mathinner{..} b] &== []
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{foldl}(f, c, a : as) &== \mathrm{foldl}(f, f(c, a), as) \\
\mathrm{foldl}(f, c, []) &== c
\end{aligned}
$$

$$
\begin{aligned}
\sum_{i=a}^{b} e(i) &== \mathrm{foldl}(\lambda xi.x + e(i), 0, [a \mathinner{..} b]) \\
\prod_{i=a}^{b} e(i) &== \mathrm{foldl}(\lambda xi.x \cdot e(i), 0, [a \mathinner{..} b])
\end{aligned}
$$

Ein Zoo von Logiken

- Das grundlegende Dilemma:

  Entscheidbarkeit $\longleftrightarrow$ Ausdrucksmächtigkeit

- Der Logik-Zoo:

	Entscheidbar	Vollständig	
Aussagenlogik (OPL)	✓	✓	$(A \wedge B) \vee C$
Pressburger Arithmetik	✓	✓	$n < x \longrightarrow n + a < x + a$
Prädikatenlogik (PL)	X	✓	$\forall x.\, \exists y.\, x = y$
Peano-Arithmetik	X	X	$n \cdot 0 = 0$
PL mit Ind. & Fkt.	X	X	*Z3*
Prädikatenlogik 2. Stufe	X	X	$\forall P.\, P(0) \longrightarrow \forall n.\, P(n)$
Logik höherer Stufe (HOL)	X	X	*Haskell*

- Auswahl der Logik: Kompromiss (*sweet spot*)

Erfüllung von Zusicherungen

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

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

Erfüllung von Zusicherungen

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

$$
\begin{aligned}
[\![\forall v. b]\!]^I_{\mathcal{B}v} = \{(\sigma, \mathit{true}) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma, \mathit{true}) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\} \\
\cup \{(\sigma, \mathit{false}) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma, \mathit{false}) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\} \\
[\![\exists v. b]\!]^I_{\mathcal{B}v} = \{(\sigma, \mathit{true}) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma, \mathit{true}) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\} \\
\cup \{(\sigma, \mathit{false}) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma, \mathit{false}) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\}
\end{aligned}
$$

Analog für andere Typen.

Erfülltheit von Zusicherungen

Erfülltheit von Zusicherungen

$b \in \mathbf{Assn}$ 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 $b \in \textbf{Assn}$ 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 $\textbf{Assn}^c \subseteq \textbf{Assn}$ 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

$$\bigwedge \Gamma := \begin{cases} b_1 \wedge \cdots \wedge b_n & \text{für alle } b_i \in \Gamma, \Gamma \neq \emptyset \\ \mathit{true} & \text{falls } \Gamma = \emptyset \end{cases}$$

Erfülltheit von Zusicherungen unter Kontext

Erfülltheit von Zusicherungen unter Kontext

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

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

Floyd-Hoare-Tripel mit Kontext

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

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.

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

Floyd-Hoare-Kalkül mit Kontext

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

Floyd-Hoare-Kalkül mit Kontext

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

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

Floyd-Hoare-Kalkül mit Kontext

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

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

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

Floyd-Hoare-Kalkül mit Kontext

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

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

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

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

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

$$[\![\Gamma \longrightarrow (A' \longrightarrow A)]\!]^{I}_{\mathcal{B}v}(\sigma) = \mathit{true}$$

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

$$[\![\Gamma \longrightarrow (A' \longrightarrow A)]\!]^{I}_{\mathcal{B}v}(\sigma) = true$$

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

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

- Unvollständigkeit der Prädiktenlogik

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 11 vom 02.07.20

Spezifikation von Funktionen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

Funktionen & Prozeduren

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

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;
}
```

```
int swap(int a[], int i, int j)
/** pre {i < a_len ∧ j < a_len};
    post {a[i] = \old(a[j]) ∧ a[j] = \old(a[i])}
; */
    {
      int buf = a[j];
      a[j] = a[i];
      a[i] = buf;
    }
    return;
```

Beispiel: Rekursion

```
int factorial(int n)
/** pre    {n ≥ 0}
    post {\result = n!}  */
{
  if (n==0) return 1;
  else return n * factorial(n-1);
}
```

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen
3. Spezifikation von Funktionsdefinitionen

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

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

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:

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

- Abstrakte Syntax
- Größe von Feldern: **konstanter** Ausdruck
- **FunSpec** wird später erläutert

Rückgaben

Neue Anweisungen: Return-Anweisung

$$
\begin{array}{lrl}
\textbf{Stmt} & s ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
 & & \mid\ \textbf{while}\ (b)\ //{*}{*}\ \textbf{inv}\ P\ {*}/\ c \mid //{*}{*}\ \{P\}\ {*}/ \\
 & & \mid \textbf{return}\ a^{?}
\end{array}
$$

Rückgabewerte

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

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

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

Rule 14.7 (required)

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

  - Nicht immer möglich, unübersichtlicher Code …

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

Erweiterte Semantik

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

Erweiterte Semantik

- Denotat einer Anweisung: $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- 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:

$$\begin{aligned} g \circ_S f = & \{(\sigma, \rho') \mid (\sigma, \sigma') \in f \wedge (\sigma', \rho') \in g\} \\ & \cup \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in f\} \end{aligned}$$

Semantik von Anweisungen

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

$$\begin{aligned}
[\![x = e]\!]_{\mathcal{C}} &= \{(\sigma, \sigma[a/l]) \mid (\sigma, l) \in [\![x]\!]_{\mathcal{L}}, (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\} \\
[\![c_1 ; c_2]\!]_{\mathcal{C}} &= [\![c_2]\!]_{\mathcal{C}} \circ_S [\![c_1]\!]_{\mathcal{C}} \qquad \text{Komposition wie oben} \\
[\![\{\,\}]\!]_{\mathcal{C}} &= \mathbf{Id}_{\Sigma} \qquad\qquad\qquad \mathbf{Id}_{\Sigma} := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
[\![\textbf{if } (b)\ c_0 \ \textbf{ else } \ c_1]\!]_{\mathcal{C}} &= \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \land (\sigma, \rho') \in [\![c_0]\!]_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma, \rho') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \land (\sigma, \rho') \in [\![c_1]\!]_{\mathcal{C}}\} \\
&\quad \text{mit } \rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U \\
[\![\textbf{return } \ e]\!]_{\mathcal{C}} &= \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\} \\
[\![\textbf{return}]\!]_{\mathcal{C}} &= \{(\sigma, (\sigma, *))\}
\end{aligned}$$

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

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

Arbeitsblatt 11.1: Jetzt seid ihr mal dran...

- Berechnet die Denotate der folgenden Programme:
  - $$\begin{aligned}\llbracket x = 3; x = 4 \rrbracket_{\mathcal{C}} &= \llbracket x = 4 \rrbracket_{\mathcal{C}} \circ_S \llbracket x = 3 \rrbracket_{\mathcal{C}} \\ &= \{(\sigma, \sigma[4/x])\} \circ_S \{(\sigma, \sigma[3/x])\} \\ &= \{(\sigma, \sigma[4/x])\}\end{aligned}$$
  - $$\begin{aligned}\llbracket x = 3; \textbf{return}\ \ x; x = 4 \rrbracket_{\mathcal{C}} &= \llbracket x = 4 \rrbracket_{\mathcal{C}} \circ_S (\llbracket \textbf{return}\ \ x \rrbracket_{\mathcal{C}} \circ_S \llbracket x = 3 \rrbracket_{\mathcal{C}}) \\ &= \{(\sigma, \sigma[4/x])\} \circ_S \\ &\quad\ (\{(\sigma, (\sigma, a)) \mid (\sigma, a) \in \llbracket x \rrbracket_{\mathcal{A}}\} \circ_S \{(\sigma, \sigma[3/x])\}) \\ &= \{(\sigma, \sigma[4/x])\} \circ_S (\{(\sigma, (\sigma, \sigma(x)))\} \circ_S \{(\sigma, \sigma[3/x])\}) \\ &= \{(\sigma, \sigma[4/x])\} \circ_S \{(\sigma, (\sigma[3/x], \underbrace{\sigma[3/x](x)}_{3}))\} \\ &= \{(\sigma, (\sigma[3/x], 3))\}\end{aligned}$$

Semantik von Funktionsdefinitionen

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

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

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

- 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.

$$\mathcal{D}_{blk}[\![.]\!] : \textbf{Blk} \to \Sigma \rightharpoonup (\Sigma \times V_U)$$

$$\mathcal{D}_{blk}[\![decls\ stmts]\!] \stackrel{def}{=} \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![stmts]\!]_{\mathcal{C}}\}$$

- Von $[\![stmts]\!]_{\mathcal{C}}$ 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:

$$\textbf{FunSpec} ::= /** \;\textbf{pre}\; \textbf{Assn}\; \textbf{post}\; \textbf{Assn}\; */$$

Vorbedingung $\quad \textbf{pre}\ sp; \quad \overbrace{\Sigma}^{\text{Vorzustand}} \to \mathbb{B}$

Nachbedingung $\quad \textbf{post}\ sp; \quad \underbrace{\Sigma}_{\text{Vorzustand}} \times \underbrace{(\Sigma \times \mathbf{V}_U)}_{\text{Nachzustand und Return-Wert}} \to \mathbb{B}$

\old(e)	Wert von *e* im **Vorzustand**
\result	**Rückgabewert** der Funktion

Beispiel: Fakultät

```
int fac(int n)
/** pre   {0 ≤ n};
    post  {\result == n!};
 */
{
 int p;
 int c;

 p= 1;
 c= 1;
 while (c<= n) /** inv {p == (c − 1)! ∧ c ≤ n + 1 ∧ 0 < c} */ {
     p= p*c;
     c= c+1;
     }
 return p;
}
```

Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre {\array(a, a_len) ∧ 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;
}
```

Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre {0 < a_len};
    post {\result = max(seq(a, a_len))}; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /** inv {j > 0 ⟶ x = max(seq(a, j)) ∧ j ≤ a_len}; */
    {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```

Ziel: Gültigkeit von Spezifikationen

- Ziel ist eine **Semantik von Spezifikationen** $\mathcal{B}_{sp}[\![.]\!]$ zu definieren, um damit **semantische Gültigkeit** zu definieren:

$$\textbf{pre}\ p\ \textbf{post}\ q \models fd$$
$$\Longleftrightarrow \forall v_1, \ldots, v_n.\ [\![fd]\!]_{\mathcal{D}_{fd}}\ \Gamma\ v_1 \ldots v_n \in \mathcal{B}_{sp}[\![\textbf{pre}\ p\ \textbf{post}\ q]\!]\ \Gamma$$

  - 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;
}
```

```
int swap(int a[], int i, int j)
/** pre {i < a_len ∧ j < a_len};
    post {a[i] = \old(a[j]) ∧ a[j] = \old(a[i])}
; */
    {
      int buf = a[j];
      a[j] = a[i];
      a[i] = buf;
    }
    return;
```

Beispiel: Rekursion

```
int factorial(int n)
/** pre    {n ≥ 0}
    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** kann nicht in Funktionen vom Typ **void** auftreten.

$$
\begin{aligned}
\mathcal{B}_{sp}[\![.]\!] &: \mathbf{Env} \to \mathbf{Assn} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B} \\
\mathcal{A}_{sp}[\![.]\!] &: \mathbf{Env} \to \mathbf{Aexpv} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V} \\
\mathcal{B}_{sp}[\![!b]\!]\;\Gamma &= \{((\sigma, (\sigma', v)), \mathit{true}) \mid ((\sigma, (\sigma', v)), \mathit{false}) \in \mathcal{B}_{sp}[\![b]\!]\Gamma\} \\
&\quad \cup \{((\sigma, (\sigma', v)), \mathit{false}) \mid ((\sigma, (\sigma', v)), \mathit{true}) \in \mathcal{B}_{sp}[\![b]\!]\Gamma\} \\
\mathcal{A}_{sp}[\![x]\!]\;\Gamma &= \{((\sigma, (\sigma', v)), \sigma'(x))\} \\
&\dots \\
\mathcal{B}_{sp}[\![\backslash\mathrm{old}(e)]\!]\;\Gamma &= \{((\sigma, (\sigma', v)), b) \mid (\sigma, b) \in [\![e]\!]_{\mathcal{B}}\;\Gamma\} \\
\mathcal{A}_{sp}[\![\backslash\mathrm{old}(e)]\!]\;\Gamma &= \{((\sigma, (\sigma', v)), a) \mid (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\;\Gamma\} \\
\mathcal{A}_{sp}[\![\backslash\mathbf{result}]\!]\;\Gamma &= \{((\sigma, (\sigma', v)), v)\}
\end{aligned}
$$

$$
\mathcal{B}_{sp}[\![\mathbf{pre}\ p\ \mathbf{post}\ q]\!]\;\Gamma = \{(\sigma, (\sigma', v)) \mid \begin{array}{l} (\sigma, \mathit{true}) \in [\![p]\!]_{\mathcal{B}}\;\Gamma \land \\ ((\sigma, (\sigma', v)), \mathit{true}) \in \mathcal{B}_{sp}[\![q]\!]\;\Gamma\} \end{array}
$$

Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre}\ p\ \textbf{post}\ q \models fd$$
$$\iff \forall v_1, \ldots, v_n.\, [\![fd]\!]_{\mathcal{D}_{fd}}\ \Gamma\ v_1 \ldots v_n \in \mathcal{B}_{sp}[\![\textbf{pre}\ p\ \textbf{post}\ q]\!]\ \Gamma$$

  - Γ enthält globale Definitionen, insbesondere andere Funktionen.

- Wie passt das zu den Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?

- Wie **beweisen** wir das?

Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre}\ p\ \textbf{post}\ q \models fd$$
$$\iff \forall v_1, \ldots, v_n.\, [\![fd]\!]_{\mathcal{D}_{fd}}\ \Gamma\ v_1 \ldots v_n \in \mathcal{B}_{sp}[\![\textbf{pre}\ p\ \textbf{post}\ q]\!]\ \Gamma$$

  - 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 ($\models \{P\}\, c\, \{Q|Q_R\}$)

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.

$$
\begin{aligned}
&\Gamma \models \{P\}\, c\, \{Q|Q_R\} \iff \\
&\forall \sigma.\, (\sigma, \mathit{true}) \in [\![P]\!]_{\mathcal{B}}\,\Gamma \Longrightarrow \exists \sigma'.\, (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}} \wedge ((\sigma, (\sigma', *)), \mathit{true}) \in \mathcal{B}_{sp}[\![Q]\!]\,\Gamma \\
&\qquad\qquad\qquad\qquad\qquad \vee \\
&\qquad\qquad\qquad \exists \sigma', v.\, (\sigma, (\sigma', v)) \in [\![c]\!]_{\mathcal{C}} \wedge ((\sigma, (\sigma', v)), \mathit{true}) \in \mathcal{B}_{sp}[\![Q_R]\!]\,\Gamma
\end{aligned}
$$

Erweiterung des Floyd-Hoare-Kalküls: **return**

$$\frac{}{\Gamma \vdash \{Q\}\ \textbf{return}\ \{P|Q\}} \qquad \frac{}{\Gamma \vdash \{Q[e/\backslash\textbf{result}]\}\ \textbf{return}\ \ e\ \{P|Q\}}$$

- Bei **return** wird die Rückgabespezifikation Q zur Vorbedingung, die reguläre Nachfolgespezifikation wird ignoriert, da die Ausführung von **return** kein Nachfolgezustand hat.
- **return** ohne Argument darf nur bei einer Nachbedingung Q auftreten, die kein $\backslash$**result** enthält.
- Bei **return** mit Argument ersetzt der Rückgabewert den $\backslash$**result** in der Rückgabespezifikation.

Erweiterung des Floyd-Hoare-Kalküls: Spezifikation

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/\backslash\mathbf{old}(x_i)] \quad \Gamma \vdash \{P'\}\, c\, \{false | Q[\backslash\mathbf{old}(x_i)/x_i]\}}{\Gamma \vdash f(x_1, \ldots, x_n)/\text{** pre } P \text{ post } Q \text{ */ } \{ds\ c\}}$$

- 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 **old**(x_i)).
  - Deswegen wird in Q im Hoare-Tripel ersetzt
- Variablen unterhalb von **old**(.) werden bei der Substitution (Zuweisungsregel) **nicht ersetzt!**
- **old**(.) wird beim Weakening von der Vorbedingung P ersetzt
- Sequentielle Nachbedingung von c ist *false*

Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre {0 < a_len};
    post {\result = max(seq(a, a_len))}; */
{ ...
```

$$\frac{\begin{array}{c}(\Gamma \wedge 0 < a\_len) \Longrightarrow P'[a/\backslash\mathbf{old}(a), a\_len/\backslash\mathbf{old}(a\_len)] \\ \Gamma \vdash \{P'\}\, c\, \{false \mid \backslash\mathbf{result} = \max(seq(\backslash\mathbf{old}(a), \backslash\mathbf{old}(a\_len)))\}\end{array}}{\begin{array}{ll}\Gamma \vdash & findmax(int\ a[], int\ a\_len) \\ & /{**}\ \text{pre}\ \{0 < a\_len\} \\ & \quad \text{post}\ \{\backslash\mathbf{result} = \max(seq(a, a\_len))\}{*}/\ \{\ldots\}\end{array}}$$

- Wobei P' noch Ausdrücke $\backslash\mathbf{old}(a\_len)$ enthalten kann,
- die dann ersetzt werden zu a_len in $P'[a/\backslash\mathbf{old}(a), a\_len/\backslash\mathbf{old}(a\_len)]$

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

$$\frac{}{\Gamma \vdash \{P\}\,\{\}\,\{P|Q_R\}} \qquad \frac{\Gamma \vdash \{P\}\,c_1\,\{R|Q_R\} \quad \Gamma \vdash \{R\}\,c_2\,\{Q|Q_R\}}{\Gamma \vdash \{P\}\,c_1; c_2\,\{Q|Q_R\}}$$

$$\frac{}{\Gamma \vdash \{Q[e/x]\}\,l = e\,\{Q|Q_R\}} \qquad \frac{\Gamma \vdash \{P \wedge b\}\,c\,\{P|Q_R\}}{\Gamma \vdash \{P\}\,\textbf{while}\ (b)\ c\,\{P \wedge \neg b|Q_R\}}$$

$$\frac{\Gamma \vdash \{P \wedge b\}\,c_1\,\{Q|Q_R\} \quad \Gamma \vdash \{P \wedge \neg b\}\,c_2\,\{Q|Q_R\}}{\Gamma \vdash \{P\}\,\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\,\{Q|Q_R\}}$$

$$\frac{(\Gamma \wedge P) \longrightarrow P' \quad \Gamma \vdash \{P'\}\,c\,\{Q'|R'\} \quad (\Gamma \wedge Q') \longrightarrow Q \quad (\Gamma \wedge R') \longrightarrow R}{\Gamma \vdash \{P\}\,c\,\{Q|R\}}$$

Erweiterter Floyd-Hoare-Kalkül II

$$\frac{}{\Gamma \vdash \{Q\}\ \textbf{return}\ \{P|Q\}} \qquad \frac{}{\Gamma \vdash \{Q[e/\backslash\textbf{result}]\}\ \textbf{return}\ \ e\,\{P|Q\}}$$

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/\backslash\textbf{old}(x_i)] \quad \Gamma \vdash \{P'\}\,c\,\{\mathit{false}|Q[\backslash\textbf{old}(x_i)/x_i]\}}{\Gamma \vdash f(x_1, \ldots, x_n)/\text{** pre } P \text{ post } Q \text{ */ } \{ds\ c\}}$$

Approximative schwächste Vorbedingung

- Erweiterung zu $\mathrm{awp}(\Gamma, c, Q, Q_R)$ und $\mathrm{wvc}(\Gamma, c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
  - Es werden der **Kontext** Γ und eine **Rückgabespezifikation** Q_R benötigt.
- Es gilt:

$$\bigwedge \mathrm{wvc}(\Gamma, c, Q, Q_R) \Longrightarrow \Gamma \models \{\mathrm{awp}(c, Q, Q_R)\}\, c\, \{Q | Q_R\}$$

- Berechnung von awp und wvc:

$$
\begin{aligned}
&\mathrm{awp}(\Gamma, f(x_1, \ldots, x_n)/\text{** pre } P \text{ post } Q \text{ */ } \{ds\ blk\}) \stackrel{def}{=} \\
&\qquad\qquad \mathrm{awp}(\Gamma', blk, false, Q[\backslash\mathbf{old}(x_i)/x_i]) \\
&\mathrm{wvc}(\Gamma, f(x_1, \ldots, x_n)/\text{** pre } P \text{ post } Q \text{ */ } \{ds\ blk\}) \stackrel{def}{=} \\
&\quad \{(\Gamma \wedge P) \Longrightarrow P'[x_i/\backslash\mathbf{old}(x_i)]\} \cup \mathrm{wvc}(\Gamma', blk, false, Q[\backslash\mathbf{old}(x_i)/x_i]) \\
&\qquad\qquad \Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1, \ldots, x_n.\,(P, Q)] \\
&\qquad\qquad P' \stackrel{def}{=} \mathrm{awp}(\Gamma', blk, Q[\backslash\mathbf{old}(x_i)/x_i], Q[\backslash\mathbf{old}(x_i)/x_i])
\end{aligned}
$$

Approximative schwächste Vorbedingung (Revisited)

$$
\begin{aligned}
\mathrm{awp}(\Gamma, \{\,\}, Q, Q_R) &\stackrel{def}{=} Q \\
\mathrm{awp}(\Gamma, l = e, Q, Q_R) &\stackrel{def}{=} Q[e/l] \\
\mathrm{awp}(\Gamma, c_1; c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{awp}(\Gamma, c_1, \mathrm{awp}(c_2, Q, Q_R), Q_R) \\
\mathrm{awp}(\Gamma, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, Q, Q_R) &\stackrel{def}{=} (b \land \mathrm{awp}(\Gamma, c_0, Q, Q_R)) \\
&\qquad \lor(\neg b \land \mathrm{awp}(\Gamma, c_1, Q, Q_R)) \\
\mathrm{awp}(\Gamma, //{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} q \\
\mathrm{awp}(\Gamma, \textbf{while}\ (b)\ //{*}{*}\ \textbf{inv}\ i\ {*}/\ c, Q_R) &\stackrel{def}{=} i \\
\mathrm{awp}(\Gamma, \textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} Q_R[e/\backslash\textbf{result}] \\
\mathrm{awp}(\Gamma, \textbf{return}, Q, Q_R) &\stackrel{def}{=} Q_R
\end{aligned}
$$

Approximative Verifikationsbedingungen (Revisited)

$$
\begin{aligned}
\mathrm{wvc}(\Gamma, \{\,\}, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Gamma, l = e, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Gamma, c_1; c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c_1, \mathrm{awp}(c_2, Q, Q_R), Q_R) \\
&\qquad \cup\ \mathrm{wvc}(\Gamma, c_2, Q, Q_R) \\
\mathrm{wvc}(\Gamma, \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c_1, Q, Q_R) \cup \mathrm{wvc}(\Gamma, c_2, Q, Q_R) \\
\mathrm{wvc}(\Gamma, //{**}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} \{\Gamma \wedge q \Longrightarrow Q\} \\
\mathrm{wvc}(\Gamma, \textbf{while}\ (b)\ //{**}\ \textbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c, i, Q_R) \\
&\qquad \cup \{\Gamma \wedge i \wedge b \Longrightarrow \mathrm{awp}(\Gamma, c, i, Q_R \\
&\qquad \cup \{\Gamma \wedge i \wedge \neg b \Longrightarrow Q\} \\
\mathrm{wvc}(\Gamma, \textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}
$$

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
     //
     //
     //
     if (c == n) { return p; } else {}
     //
     p= p*c;
     //
     //
     c= c+1;
     //
     }
}
```

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
     // {p == (c − 1)! ∧ true}
     //
     //
     if (c == n) { return p; } else {}
     //
     p= p*c;
     //
     //
     c= c+1;
     // {p == (c − 1)!}
     }
}
```

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
    // {p == (c − 1)! ∧ true}
    //
    //
    if (c == n) { return p; } else {}
    //
    p= p*c;
    //
    // {p == ((c + 1) − 1)!}
    c= c+1;
    // {p == (c − 1)!}
    }
}
```

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
    // {p == (c − 1)! ∧ true}
    //
    //
    if (c == n) { return p; } else {}
    //
    p= p*c;
    // {p == c!}
    // {p == ((c + 1) − 1)!}
    c= c+1;
    // {p == (c − 1)!}
    }
}
```

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
    // {p == (c − 1)! ∧ true}
    //
    //
    if (c == n) { return p; } else {}
    // {p ∗ c == c!}
    p= p∗c;
    // {p == c!}
    // {p == ((c + 1) − 1)!}
    c= c+1;
    // {p == (c − 1)!}
    }
}
```

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
    // {p == (c − 1)! ∧ true}
    //
    // {(c == n ∧ p == n!) ∨ (c ≠ n ∧ p ∗ c = c!)}
    if (c == n) { return p; } else {}
    // {p ∗ c == c!}
    p= p*c;
    // {p == c!}
    // {p == ((c + 1) − 1)!}
    c= c+1;
    // {p == (c − 1)!}
    }
}
```

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)!; */ {
     // {p == (c − 1)! ∧ true}
     // ↯
     // {(c == n ∧ p == n!) ∨ (c ≠ n ∧ p ∗ c = c!)}
     if (c == n) { return p; } else {}
     // {p ∗ c == c!}
     p= p*c;
     // {p == c!}
     // {p == ((c + 1) − 1)!}
     c= c+1;
     // {p == (c − 1)!}
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)! ∧ 0< c; */ {
     //
     //
     p= p*c;
     //
     //
     if (c == n) {
        //** {c == n ∧ p == n! ∧ 0 < c} */
        //
        return p;
     } else {}
     //
     //
     c= c+1;
     //
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c− 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     //
     p= p∗c;
     //
     //
     if (c == n) {
        //** {c == n ∧ p == n! ∧ 0 < c} */
        //
        return p;
     } else {}
     //
     //
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     //
     p= p*c;
     //
     //
     if (c == n) {
        //** {c == n ∧ p == n! ∧ 0 < c} */
        //
        return p;
     } else {}
     //
     // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     //
     p= p*c;
     //
     //
     if (c == n) {
       //** {c == n ∧ p == n! ∧ 0 < c} */
       //
       return p;
     } else {}
     // {p == c! ∧ 0 < c}
     // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c− 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     //
     p= p*c;
     //
     //
     if (c == n) {
        //** {c == n ∧ p == n! ∧ 0 < c} */
        // {c == n ∧ p == n!}
        return p;
     } else {}
     // {p == c! ∧ 0 < c}
     // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)! ∧ 0< c; */ {
    // {p == (c − 1)! ∧ 0 < c ∧ true}
    //
    p= p*c;
    //
    // {(c == n ∧ p == n! ∧ 0 < c) ∨ (c ≠ n ∧ p == c! ∧ 0 < c)}
    if (c == n) {
      //** {c == n ∧ p == n! ∧ 0 < c} */
      // {c == n ∧ p == n!}
      return p;
    } else {}
    // {p == c! ∧ 0 < c}
    // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
    c= c+1;
    // {p == (c − 1)! ∧ 0 < c}
    }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     //
     p= p*c;
     // {p == c! ∧ 0 < c}
     // {(c == n ∧ p == n! ∧ 0 < c) ∨ (c ≠ n ∧ p == c! ∧ 0 < c)}
     if (c == n) {
       //** {c == n ∧ p == n! ∧ 0 < c} */
       // {c == n ∧ p == n!}
       return p;
     } else {}
     // {p == c! ∧ 0 < c}
     // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```

Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c- 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     // {p ∗ c == c! ∧ 0 < c}
     p= p∗c;
     //  {p == c! ∧ 0 < c}
     //  {(c == n ∧ p == n! ∧ 0 < c) ∨ (c ≠ n ∧ p == c! ∧ 0 < c)}
     if (c == n) {
       //** {c == n ∧ p == n! ∧ 0 < c} */
       // {c == n ∧ p == n!}
       return p;
     } else {}
     // {p == c! ∧ 0 < c}
     // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```

Zusammenfassung

- Funktionen sind **zentrales Modularisierungskonzept**
- Wir müssen Funktionen **modular** verifizieren können
- Erweiterung der **Semantik:**
  - Semantik von Deklarationen und Parameter — straightforward
  - Semantik von **Rückgabewerten** — Erweiterung der Semantik
- Erweiterung der **Spezifikationen:**
  - Spezifikation von Funktionen: **Vor-/Nachzustand** statt logischer Variablen
- Erweiterung des Hoare-Kalküls:
  - Environment, um andere Funktionen zu nutzen
  - Gesonderte Nachbedingung für Rückgabewert/Endzustand
- Es fehlt: **Funktionsaufruf** und **Parameterübergabe**

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Deklarationen und Parameter ✓
2. Semantik von Funktionsdefinitionen ✓
3. Spezifikation von Funktionsdefinitionen ✓
4. Beweisregeln für Funktionsdefinitionen ✓

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

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)

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

Zur Erinnierung: Semantik von Funktionsdefinitionen

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

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

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

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

Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $[\![f]\!]_{\mathcal{D}_{fd}}$
- 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**!

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von f dem Bezeichner f zuordnen.
- Deshalb brauchen wir eine **Umgebung** (Environment):

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

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

Nebenbedingungen von Funktionsaufrufen

- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- **Reine Funktion** (pure function):
  - keine (sichtbaren) Seiteneffekte und Spezifikation der Form

$$Q[\backslash\textbf{result}]$$

…und Q enthält nur formale Parameter **innerhalb von** **old**(.)

Semantik von Funktionsaufrufen

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

- Aufruf von Funktion $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{A}}$ ignoriert Endzustand
- Aufruf einer rein funktionalen Prozedur $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}$ ohne Rückgabewert hat keinen Effekt

Semantik von Funktionsaufrufen

$$
\begin{aligned}
[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}\Gamma &= \{(\sigma,\sigma') \mid (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \\
&\qquad \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\} \\
[\![x = f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}\Gamma &= \{(\sigma,\sigma'[v/x]) \mid (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \\
&\qquad \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}
\end{aligned}
$$

- Aufruf von Funktion $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}$ ignoriert Endzustand
- Aufruf einer rein funktionalen Prozedur $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}$ ohne Rückgabewert hat keinen Effekt
- Somit: Kombination mit Zuweisung
- Zuweisungen gehen nur anm Programmvariablen, Feldeinträge oder Struktur-Einträge vom Typ **Z** oder **C**.

Beispiel: Reverse mittels Swap geht nicht...

```
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;
}
```

```
int swap(int a[], int i, int j)
/** pre {i < a_len ∧ j < a_len};
    post {a[i] = \old(a[j]) ∧ a[j] = \old(a[i])}
; */
    {
      int buf = a[j];
      a[j] = a[i];
      a[i] = buf;
    }
    return;
```

Kontext

- Wir benötigen ferner einen **Kontext** Γ, der Funktionsbezeichnern ihre **Spezifikation** (Vor/Nachbedingung) zuordnet.
- $\Gamma(f) = \forall x_1, \ldots, x_n. (P, Q)$, für Funktion $f(x_1, \ldots, x_n)$ mit Vorbedingung P und Nachbedingung Q.
- Korrektheit gilt immer nur im **Kontext**, dadurch kann jede Funktion separat verifiziert werden (**Modularität**)

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)}{\begin{array}{l}\Gamma \vdash \{P[t_i/x_i]\} \\ \qquad l = f(t_1, \ldots, t_n) \\ \qquad \{Q[t_i/x_i][l/\backslash\mathbf{result}]_{\backslash\mathbf{old}(Y) \to Y} | Q_R\}\end{array}}$$

- Γ muss f mit der Vor-/Nachbedingung P, Q enthalten
- In P und Q werden Parameter x_i durch Argumente t_i ersetzt.
- In Q werden die x_i unterhalb von **\old**(.) durch t_i ersetzt,
- Alle Ausdrücke der Form **\old**(e) werden durch e ersetzt,
- **\result** in Q wird durch l ersetzt

Beispiel: die Fakultätsfunktion, rekursiv

```
int fac(int x)
/** pre   0 ≤ x;
    post \result = \old(x!)  */
{
 int r = 0;
 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r* x;
}
```

$$\frac{\Gamma(fac) = \forall x_1, \ldots, x_n.\,(0 \leq x, \backslash\mathbf{result} = \backslash\mathbf{old}(x!))}{\Gamma \vdash \{\qquad\qquad\}\ l = fac(2 * y)\ \{\qquad\qquad | Q_R\}}$$

Beispiel: die Fakultätsfunktion, rekursiv

```
int fac(int x)
/** pre   0 ≤ x;
    post \result = \old(x!)  */
{
 int r = 0;
 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r* x;
}
```

$$\frac{\Gamma(fac) = \forall x_1, \ldots, x_n. (0 \leq x, \backslash\textbf{result} = \backslash\textbf{old}(x!))}{\Gamma \vdash \{0 \leq 2 * y\}\, l = fac(2 * y)\, \{l = (2 * y)! | Q_R\}}$$

Beobachtung

- Der Aufruf einer Funktion **ersetzt** die momentane Nachbedingung — das ist ein Problem bei Schleifen!
- Wir brauchen keine Invariante mehr — 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
- Ist aber schwierig zu handhaben als Teil von wvc()
  - Hier braucht man eine Behandlung ähnlich zum Einfügen von Zwischenbedingungen

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe mit Zuweisung eines Rückgabewertes

$$
\begin{array}{rrl}
\textbf{Stmt}\ c & ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
& \mid & \textbf{while}\ (b)\ //{*}{*}\ \textbf{inv}\ a\ {*}/\ c \mid //{*}{*}\ \{a\}\ {*}/ \\
& \mid & \textbf{Idt}(a^*) \\
& \mid & //{*}{*}\ \textbf{const}\ R\ {*}/\ l = \textbf{Idt}(a^*) \\
& \mid & \textbf{return}\ a^?
\end{array}
$$

Approximative schwächste Vorbedingung & Verifikationsbedingung

$$\Gamma(f) = \forall x_1, \ldots, x_n.\,(P, Q)$$

$$\text{awp}(\Gamma, //{*}{*}\ \textbf{const}\ \ R\ {*}/ l = f(t_1, \ldots, t_n), Q, Q_R) \stackrel{def}{=} R \wedge P[t_i/x_i]$$
$$\text{wenn } l \notin R$$

$$\text{wvc}(\Gamma, //{*}{*}\ \textbf{const}\ \ R\ {*}/ l = f(t_1, \ldots, t_n), Q, Q_R)$$
$$\stackrel{def}{=} \{R \wedge Q[t_i/x_i][l/\ \backslash\textbf{result}]_{\backslash\textbf{old}(Y)\to Y} \Longrightarrow Q\}$$
$$\text{wenn } l \notin R$$

Beispiel: die Fakultätsfunktion

```
// {y = 5 ∧ x = 2 ∗ y}
/** const y = 5 ∧ x = 2 ∗ y */
l = fac(x);
// {l = 10!}
```

```
int fac(int x)
/** pre 0 ≤ x;
    post \result = \old(x!) */{
 int r = 0;
 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r* x;
}
```

Beispiel: die Fakultätsfunktion

```
// {y = 5 ∧ x = 2 ∗ y}
/** const y = 5 ∧ x = 2 ∗ y */
l = fac(x);
// {l = 10!}
```

```
int fac(int x)
/** pre   0 ≤ x;
    post \result = \old(x!) */{
  int r = 0;
  if (x == 0) { return 1; }
  r = fac(x- 1);
  return r* x;
}
```

$$\mathrm{awp}(\Gamma, //** \;\textbf{const}\; y = 5 \land x = 2 * y */ \; l = fac(x), l = 10!, Q_R)$$
$$\stackrel{def}{=} \; y = 5 \land x = 2 * y \land 0 \le x$$

Beispiel: die Fakultätsfunktion

```
// {y = 5 ∧ x = 2 * y}
/** const y = 5 ∧ x = 2 * y  */
l = fac(x);
// {l = 10!}
```

```
int fac(int x)
/** pre   0 ≤ x;
    post \result = \old(x!)  */{
  int r = 0;
  if (x == 0) { return 1; }
  r = fac(x- 1);
  return r* x;
}
```

$$\mathrm{awp}(\Gamma, /\!/\!**\ \textbf{const}\ \ y = 5 \land x = 2 * y\ */\ l = \mathit{fac}(x), l = 10!, Q_R)$$
$$\stackrel{def}{=}\ y = 5 \land x = 2 * y \land 0 \leq x$$

$$\mathrm{wvc}(\Gamma, /\!/\!**\ \textbf{const}\ \ y = 5 \land x = 2 * y\ */\ l = \mathit{fac}(x), l = 10!, Q_R)$$
$$\stackrel{def}{=}\ \{y = 5 \land x = 2 * y \land l = x! \Longrightarrow l = 10!\}$$

Zusammenfassung

- Aufruf von Funktionen:
  - Funktionen ohne Seiteneffekt in Kombination mit Zuweisung
- Aufruf einer Funktion **ersetzt** Vor/Nachbedingung
- **Einschränkungen**
  - Keine Seiteneffekte
  - Keine Veränderungen von/Zuweisungen ganzen Strukturen oder Feldern
  - Prozeduren sind unbrauchbar/überflüssig
- Fazit: Funktionen sind nicht ganz so straightforward

Korrekte Software: Grundlagen und Methoden

Vorlesung 12 vom 09.07.20

Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Prüfungstermine

- Mo, 20.07.2020: Präsenzprüfungen (9:00- 16:30, je zur vollen Stunde)
- Di, 21.07.2020: Onlineprüfungen (9:00- 13:00, alle 30 Minuten)

- Mo, 24.08.2020: Präsenzprüfungen (9:00- 16:30, je zur vollen Stunde)
- Di, 25.08.2020: Onlineprüfungen (9:00- 13:00, alle 30 Minuten)

Prüfungsmodalitäten

- Anmeldung über stud.ip.
- Präsenzprüfungen:
  - Im Raum 4380
  - Bitte **nicht vor dem Prüfungsraum versammeln** (sondern kurz vorher hochkommen)
  - **Wichtig**: Ausweispapiere mitbringen, unten am MZH ausweisen. Eingang ins MZH nur über die Ostseite (zur Enrique-Schmidt-Straße).
- Onlineprüfung:
  - Über Zoom, gleiche Meeting-Id wie gewohnt.
  - Wir lassen euch zur Prüfung in das Meeting, alle anderen bleiben draussen.
  - Eine Kamera ist **zwingend** erforderlich.
  - Die Prüfung muss in einem ruhigen Raum stattfinden. Es darf sich keine weitere Person im Raum befinden. Hilfsmittel sind nicht zugelassen.

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten und die 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*
  - Funktion können sonst nur **globale** Seiteneffekte haben
  - Effizienz
- Kurze Begriffsklärung:
  - Referenzen: getypt, eingeschränkte Arithmetik
  - Zeiger: ungetypt, Zeigerarithmetik

Refenzen in C

- Pointer in C ("pointer type"):
  - Schwach getypt (**void** $*$ kompatibel mit allen Zeigertypen, Typumwandlung)
  - Eingeschränkte Zeigerarithmetik (Addition, Subtraktion)
  - Felder werden durch Zeigerarithmetik implementiert
- Pointer sind *first-class-values*
- C-Standard läßt das Speichermodell relativ offen
  - Repräsentation von Objekten

Referenzen in anderen Sprachen

- Java:
  - (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 ($\&a$) und Derefenzierung ($*l$)

$$
\begin{array}{rl}
\textbf{Lexp} & l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \mid *a \\
\textbf{Aexp} & a ::= \textbf{Z} \mid \textbf{C} \mid \textbf{Lexp} \mid \&l \\
 & \quad\quad \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid \textbf{Idt}(\textbf{Exp}^*) \\
\textbf{Bexp} & b ::= \ldots \\
\textbf{Exp} & e := \textbf{Aexp} \mid \textbf{Bexp} \\
\textbf{Stmt} & c ::= \ldots \\
\textbf{Type} & t ::= \textbf{char} \mid \textbf{int} \mid *t \mid \textbf{struct}\ \textbf{Idt}^? \ \{\textbf{Decl}^+\} \mid t\ \textbf{Idt}[a]
\end{array}
$$

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 $\Sigma \stackrel{\text{def}}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$ mit
  - **Locations:** $\mathbf{Loc} ::= \mathbf{Idt} \mid \mathbf{Loc}[\mathbb{Z}] \mid \mathbf{Loc}.\mathbf{Idt}$
  - Werte: $\mathbf{V} = \mathbb{Z}$
- Ansatz reicht nicht mehr:
  - Werte müssen auch Locations sein: $\mathbf{V} \stackrel{\text{def}}{=} \mathbb{Z} + \mathbf{Loc}$
  - **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**:

<table>
<tr><td>a</td><td colspan="8">b</td><td colspan="3">c</td></tr>
<tr><td></td><td colspan="4">b[0]</td><td colspan="4">b[1]</td><td>c[0]</td><td>c[1]</td><td>c[2]</td></tr>
<tr><td></td><td>b[0].x</td><td colspan="3">b[0].y</td><td>b[1].x</td><td colspan="3">b[1].y</td><td colspan="3"></td></tr>
<tr><td></td><td></td><td>b[0].y[0]</td><td>b[0].y[1]</td><td>b[0].y[2]</td><td></td><td>b[1].y[0]</td><td>b[1].y[1]</td><td>b[1].y[2]</td><td colspan="3"></td></tr>
</table>

0	1	2	3	4	5	6	7	8	9	10	11

Denotatation 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**:

<table>
<tr><td>a</td><td colspan="8">b</td><td colspan="3">c</td></tr>
<tr><td></td><td colspan="4">b[0]</td><td colspan="4">b[1]</td><td>c[0]</td><td>c[1]</td><td>c[2]</td></tr>
<tr><td></td><td>b[0].x</td><td colspan="3">b[0].y</td><td>b[1].x</td><td colspan="3">b[1].y</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>b[0].y[0]</td><td>b[0].y[1]</td><td>b[0].y[2]</td><td></td><td>b[1].y[0]</td><td>b[1].y[1]</td><td>b[1].y[2]</td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><td>l</td><td>m+0</td><td>m+1</td><td>m+2</td><td>m+3</td><td>m+4</td><td>m+5</td><td>m+6</td><td>m+7</td><td>n</td><td>n+1</td><td>n+2</td></tr>
</table>

Denotatation 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:

<table>
<tr><td>a</td><td colspan="8">b</td><td colspan="3">c</td></tr>
<tr><td></td><td colspan="4">b[0]</td><td colspan="4">b[1]</td><td>c[0]</td><td>c[1]</td><td>c[2]</td></tr>
<tr><td></td><td>b[0].x</td><td colspan="3">b[0].y</td><td>b[1].x</td><td colspan="3">b[1].y</td><td colspan="3"></td></tr>
<tr><td></td><td></td><td>b[0].y[0]</td><td>b[0].y[1]</td><td>b[0].y[2]</td><td></td><td>b[1].y[0]</td><td>b[1].y[1]</td><td>b[1].y[2]</td><td colspan="3"></td></tr>
</table>

Denotatation 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

$$off : \mathbf{Loc} \to \mathbf{Z} \to \mathbf{Loc}$$

$$\begin{aligned} off(l, 0) &= l \\ off(off(l, a), b) &= off(l, a + b) \\ off(l, a) = l &\Longrightarrow a = 0 \\ off(l, a) = off(l, b) &\Longrightarrow a = b \end{aligned}$$

$$fld : \mathbf{Loc} \to \mathbf{Idt} \to \mathbf{Loc}$$

$$\begin{aligned} fld(l, f) &\neq l \\ fld(l, f) = fld(l, g) &\Longrightarrow f = g \\ fld(l, f) = fld(m, f) &\Longrightarrow l = m \\ f \neq g &\Longrightarrow fld(l, f) \neq fld(m, g) \end{aligned}$$

Arbeitsblatt 12.1: 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]).x`, `(*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):

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

- 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 $\mathit{off}(l, n) \stackrel{def}{=} l[n]$, $\mathit{fld}(l, i) \stackrel{def}{=} l.i$.

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 $\mathit{off}(l, n) \stackrel{def}{=} l[n]$, $\mathit{fld}(l, i) \stackrel{def}{=} l.i$.

- $[\![a]\!]$ wäre immer a. Damit funktionieren drei Dinge nicht:
  1. Wir können globale nicht von lokale 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

$$[\![-]\!]_{\mathcal{L}} : \textbf{Env} \to \textbf{Lexp} \to \Sigma \rightharpoonup \textbf{Loc}$$

$$\begin{aligned}
[\![x]\!]_{\mathcal{L}}\ \Gamma =& \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\} \\
[\![lexp[a]]\!]_{\mathcal{L}}\ \Gamma =& \{(\sigma, \mathit{off}(l, i)) \mid (\sigma, l) \in [\![lexp]\!]_{\mathcal{L}}\ \Gamma, (\sigma, i) \in [\![a]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![lexp.f]\!]_{\mathcal{L}}\ \Gamma =& \{(\sigma, \mathit{fld}(l, f)) \mid (\sigma, l) \in [\![lexp]\!]_{\mathcal{L}}\ \Gamma\} \\
[\![*e]\!]_{\mathcal{L}}\ \Gamma =& [\![e]\!]_{\mathcal{A}}\ \Gamma
\end{aligned}$$

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)

$$[\![-]\!]_{\mathcal{A}} : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
[\![a_0 + a_1]\!]_{\mathcal{A}}\ \Gamma =& \{(\sigma, n_0 + n_1 \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![a_0 - a_1]\!]_{\mathcal{A}}\ \Gamma =& \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![a_0 * a_1]\!]_{\mathcal{A}}\ \Gamma =& \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![a_0 / a_1]\!]_{\mathcal{A}}\ \Gamma =& \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma\ \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma \\
& \wedge\ n_1 \neq 0\}
\end{aligned}$$

Erweiterung der Semantik: Stmt

$$
\begin{aligned}
[\![x = e]\!]_{\mathcal{C}}\ \Gamma = \{(\sigma, upd(\sigma, l, a)) \mid\ & (\sigma, l) \in [\![x]\!]_{\mathcal{L}}\ \Gamma \\
& \wedge (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\ \Gamma\}
\end{aligned}
$$

$$
\begin{aligned}
[\![x = f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, upd(\sigma', l, v)) \mid\ & (\sigma, (\sigma', v)) \in \Gamma(f)(v_1, \ldots, v_n) \\
& \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma \\
& \wedge (\sigma, l) \in [\![x]\!]_{\mathcal{L}}\Gamma\}
\end{aligned}
$$

Arbeitsblatt 12.2: 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 12.3: 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**?

Und jetzt?

- Zustand erweitert, so dass wir Zeiger modellieren können.
- Semantik entsprechend erweitert.
- Was machen wir mit dem Hoare-Kalkül, speziell der **Zuweisung**?
- Vorherige Modellierung — Zuweisung durch Substitution modelliert — nicht mehr ausreichend.
- Daher: **explizite Zustandsprädikate**

Explizite Zustandsprädikate

- Zusicherungenen (**Assn**) sind zustandsabhängige Prädikate
  - Mit anderen Worten, Prädikate über Programmvariablen.
- Axiomatische Beschreibung des Zustandes erforderte neue Modellierung auf der Ebene der Prädikate
- Explizite Zustandsprädikate modellieren die Zustandsoperationen *read* und *upd* **explizit**

Explizite Zustandsprädikate

- Erweiterung von **Aexpv** um *read*, neue Sorte **State** mit Operation *upd*:

$$
\begin{array}{rl}
\textbf{Lexp}_s & l ::= \ldots \mid \cancel{*a} \\
\textbf{Assn}_s & b ::= \ldots \\
\textbf{Aexp}_s & a ::= \mathit{read}(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid l \mid \cancel{\&l} \mid \ldots \mid \backslash\textbf{old}(e) \mid \ldots \\
\textbf{State} & S ::= \mathit{StateVar} \mid \mathit{upd}(S, l, e)
\end{array}
$$

- 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:

$$
\begin{aligned}
\mathcal{B}_{sp}[\![.]\!] &: \textbf{Env} \to \textbf{Assn}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B} \\
\mathcal{A}_{sp}[\![.]\!] &: \textbf{Env} \to \textbf{Aexp}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}
\end{aligned}
$$

Hoare-Triple

$$\Gamma \models \{P\}\, c\, \{Q|R\}$$

- $P, Q, R \in \mathbf{Assn}_s$ sind **explizite Zustandsprädikate**
- Deklarationen (**Decl**) allozieren für jede Variable eine Location (*fresh*), und ordnen diese in Γ dem Namen zu.
- Gültigkeit von Hoare-Tripeln (partielle, totale Korrektheit) wie vorher

Floyd-Hoare-Kalkül

Alte Regel

$$\overline{\Gamma \vdash \{Q[upd(\sigma, x, e)/\sigma]\}\, x = e\, \{Q|R\}}$$

- Ein **Lexp** l auf der rechten Seite e wird durch $read(\sigma, l)$ ersetzt.[1]
- & dient lediglich dazu, diese Konversion zu **verhindern**.
- $*$ **erzwingt** diese Konversion, auch auf der linken Seite x.
- Beispiel: $*$a = $*$&b;

[1] Außer l ist ein Array-Typ.

Formal: Konversion in Zustandsprädikate

$$
\begin{aligned}
(-)^{\dagger} &: \mathbf{Lexp} \to \mathbf{Lexp}_s \\
i^{\dagger} &= i \quad (i \in \mathbf{Idt}) \\
l.id^{\dagger} &= l^{\dagger}.id \\
l[e]^{\dagger} &= l^{\dagger}[e^{\#}] \\
*l^{\dagger} &= l^{\#}
\end{aligned}
$$

$$
\begin{aligned}
(-)^{\#} &: \mathbf{Aexp} \to \mathbf{Aexp}_s \\
e^{\#} &= read(\sigma, e^{\dagger}) \quad (e \in \mathbf{Lexp}) \\
n^{\#} &= n \\
v^{\#} &= v \quad (v \text{ logische Variable}) \\
\&e^{\#} &= e^{\dagger} \\
e_1 + e_2{}^{\#} &= e_1{}^{\#} + e_2{}^{\#} \\
\backslash\mathbf{result}^{\#} &= \backslash\mathbf{result} \\
\backslash\mathbf{old}(e)^{\#} &= \backslash\mathbf{old}(e)
\end{aligned}
$$

Angepasste Regeln des Hoare-Kalküls

$$\overline{\Gamma \vdash \{Q[upd(\sigma, x^{\dagger}, e^{\#})/\sigma]\}\, x = e\, \{Q|R\}}$$

$$\overline{\Gamma \vdash \{Q[e^{\#} / \backslash\textbf{result}]\}\ \textbf{return}\ \ e\, \{P|Q\}}$$

Arbeitsblatt 12.4: 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;

 //
 //
 x= 7;
 //
 *p= 99;
 //
 // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
 int x;

 //
 //
 x= 7;
 //
 *p= 99;
 // {read(s, x) = 7}
 // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
 int x;

 //
 //
 x= 7;
 // {read(upd(s, read(s, p), 99), x) = 7}
 *p= 99;
 // {read(s, x) = 7}
 // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
 int x;

 //
 // {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}
}
```

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}
}
```

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)

$\backslash\mathbf{valid}(x) \Longleftrightarrow \mathit{read}(\sigma, x^{\dagger})$ ist definiert

- Insbesondere: $\backslash\mathbf{valid}(*x) \Longleftrightarrow \mathit{read}(\sigma, \mathit{read}(\sigma, x))$ 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.

$$\backslash\mathbf{array}(a, n) \Longleftrightarrow (\forall i.\, 0 \leq i < n \Longrightarrow \backslash\mathbf{valid}(a[i]))$$

- Gültigkeit kann abgeleitet werden:

$$\frac{\texttt{x= \&e}}{\texttt{\textbackslash valid(*x)}} \qquad \frac{\texttt{\textbackslash array(a, n)} \quad 0 \leq i \quad i < n}{\texttt{\textbackslash valid(a[i])}}$$

Was noch fehlt. . .

- **Vorwärtsrechung** mit expliziten Zustandsprädikaten.
  - Statt Existenzquantoren über Variablenwerte **unbestimmte Zwischenzustände** $\rho_1, \rho_2, \ldots$:

$$\frac{\rho_i \notin FV(P)}{\Gamma \vdash \{P\}\, x = e\, \{P[\rho_i/\sigma] \wedge \sigma = upd(\rho_i, x^\dagger[\rho_i/\sigma], e^\#[\rho_i/\sigma]) | R\}}$$

  - 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

Korrekte Software: Grundlagen und Methoden

Vorlesung 13 vom 16.07.20

Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Vorwärts mit Floyd und Hoare
- Modellierung
- Spezifikation von Funktionen
- Referenzen und Speichermodelle
- Ausblick und Rückblick

Was gibt's heute?

- Rückblick
- Ausblick
- Feedback
- Prüfungsvorbereitung

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

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 $(-)^{\dagger}, (-)^{\#}$

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*

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?

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 $\not\Phi$ zu widerlegen

Beweis einer VC:

$$x \geq 0 \wedge y > 0 \Longrightarrow x = 0 * y + x$$

Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
   (not (=> (and (>= x 0) (> y 0))
            (= x (+ (* 0 y) x))))
)
(check-sat)
```

Antwort:

```
unsat
```

Unerfüllbare VC:

$$x \geq 0 \wedge y > 0 \Longrightarrow x \geq y$$

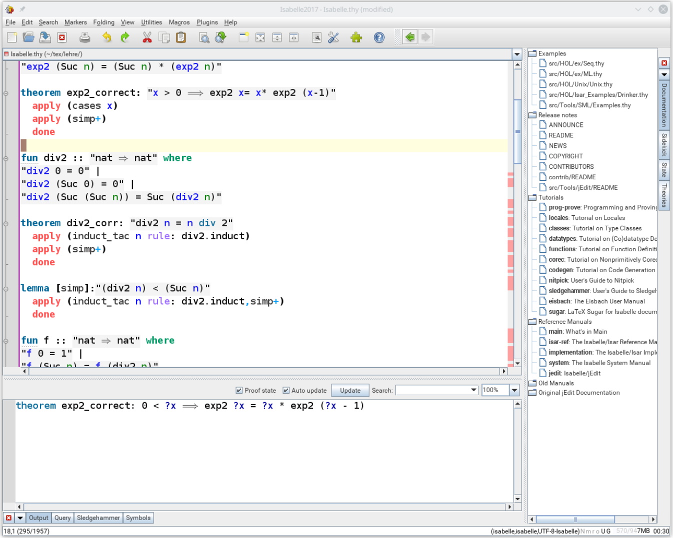
Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
   (not (=> (and (>= x 0) (> y 0))
            (>= x y)))
)
(check-sat)
```

Antwort:

```
sat
```

Beispiel: Isabelle



Korrekte Software in der Industrie

- Meist in speziellen Anwendungsgebieten: Luft-/Raumfahrt, Automotive, sicherheitskritische Systeme, Betriebssysteme
- Ansätze:
  1. Vollautomatisch: **statische Analyse** (Abstrakte Interpretation) für spezielle Aspekte: Freiheit von Ausnahmen und Unter/Überläufen, Programmsicherheit, Laufzeitverhalten (WCET) (nicht immer korrekt, meist vollständig)
     - Werkzeuge: `absint`
  2. Halbautomatisch: **Korrektheitsannotationen**, Überprüfung automatisch
     - Werkzeuge: Spark (ADA), Frama-C (C), JML (ESC/Java, Krakatao; Java), Boogie und Why (generisches VCG), VCC (C)
  3. Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
     - Beispiele: L4.verified, CompCert, SAMS

Feedback

Deine Meinung zählt

- Was war gut, was nicht?
- Arbeitsaufwand?
- Mehr **Theorie** oder mehr **Praxis**?
- Programmieraufgaben?
- Leichtgewichtiger Übungsbetrieb — mehr oder weniger?
- Bitte auch die **Evaluation** auf stud.ip beantworten!

Tschüß!