Korrekte Software: Grundlagen und Methoden

Vorlesung 1 vom 03.04.24

Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Organisatorisches

- Veranstalter:

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

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

- Termine:
  - Mittwoch, 10 – 12, MZH 5600
  - Donnerstag, 10 – 12, MZH 5600
- Webseite: `https://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss24`

Veranstaltungskonzept

- Aus den letzten Jahren: **integrierte Veranstaltung** statt **langer Vorlesung**.
- Kürzere **Vortragseinheiten**, dazwischen **Arbeitsfragen** (Kurzübungen)
- Wöchentliche **Übungsaufgaben** zur Vertiefung
- Technisch:
  - Fragen/Kurzübungen in **HedgeDoc**: `http://hackmd.informatik.uni-bremen.de/`
  - Übungsblätter als **Markdown**, Abgabe über `gitlab`.

Prüfungsform

- 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 mehr Fehler als Bearbeitung
- **Prüfungsleistung**:
  - **Mündliche Prüfung**: Einzelprüfung ca. 20– 30 Minuten
  - **Übungsbetrieb** (bis zu 15% Bonuspunkte, keine Voraussetzung)

Übungsbetrieb

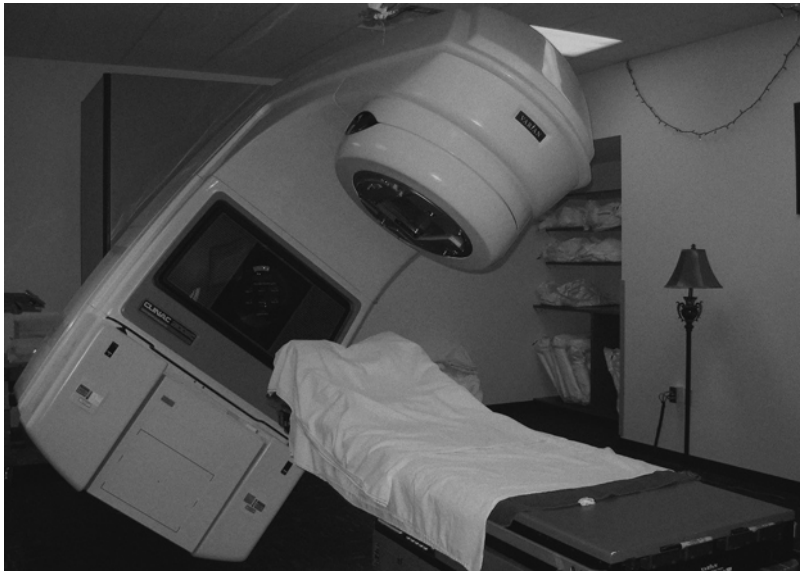
- Abgabe und Korrektur des Übungsbetriebs erfolgt über **gitlab**.
- Dazu legt **pro Gruppe** ein Repository an.
- Ladet uns (`clueth`, `autexier`) als Developer ein.
- Für jedes Übungsblatt:
  1. Das Übungsblatt ladet ihr von der Webseite herunter und bearbeitet es **elektronisch**.
  2. Die Lösung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

Arbeitsblatt 1.1: Jetzt seid ihr dran!

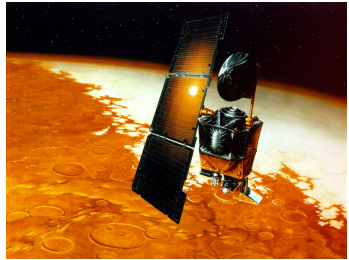
- Gruppiert euch in Gruppen zu drei Teilnehmenden!
- Tragt eure Namen in der Übersicht ein
  `https://hackmd.informatik.uni-bremen.de/s/iwDedtWRO#`
- Und kreiert eine eigene Hackmd Arbeitsblatt Seite pro Gruppe und verlinkt sie auf obiger Übersichtsseite.
- Auf diesem Arbeitsblatt bearbeitet ihr die Arbeitsfragen im Laufe des Kurses.
- Bitte nur in "eurem" Arbeitsblatt arbeiten
- Die Arbeitsblätter sind nicht notenrelevant.

I. Warum Korrekte Software?

Software-Disaster I: Therac-25



Software-Disasters II: Space



Software-Disaster III: AT&T (15.01.1990)

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

Software-Disaster IV: Ungeplantes Übergewicht

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

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

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

Quelle: *Guardian*, *09.04.2021*.

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

Software-Disaster V: Der Horizon-Skandal

- 1999 wurde für die lokalen Postämter in Großbritannien das System *Horizon* der Firma Fujitsu für Buchhaltung und Lagerhaltung eingeführt.
- Das System war fehlerhaft, so dass gelegentlich nicht-existente Fehlbestände angezeigt wurden.
- Das Post Office hat trotzdem die Fehlbeständen von den lokalen Postbeamten (subpostmaster) eingetrieben; einige wurden angeklagt und verurteilt, andere privatinsolvent oder schieden aus.
- Erste Berichte über die Fehler tauchten 2005 auf, und wurden 2009 in der Presse publik.
- Erst 2019 nach einer Sammelklage wurden die Fehler amtlich vom High Court festgestellt, und die bis dahin ergangenen Urteile für unrechtmäßig erklärt.

Software-Disaster V: Der Horizon-Skandal

- 1999 wurde für die lokalen Postämter in Großbritannien das System *Horizon* der Firma Fujitsu für Buchhaltung und Lagerhaltung eingeführt.
- Das System war fehlerhaft, so dass gelegentlich nicht-existente Fehlbestände angezeigt wurden.
- Das Post Office hat trotzdem die Fehlbeständen von den lokalen Postbeamten (subpostmaster) eingetrieben; einige wurden angeklagt und verurteilt, andere privatinsolvent oder schieden aus.
- Erste Berichte über die Fehler tauchten 2005 auf, und wurden 2009 in der Presse publik.
- Erst 2019 nach einer Sammelklage wurden die Fehler amtlich vom High Court festgestellt, und die bis dahin ergangenen Urteile für unrechtmäßig erklärt.
- *Horizon* läuft immer noch, Fujitsu hat einen Vertrag über 2.4 Mrd Pfund.

Quellen: `https://www.bbc.com/news/business-56718036`,

`https://www.theguardian.com/uk-news/2024/feb/02/post-office-scandal-key-takeaways-latest-court-hearings`

Arbeitsblatt 1.2: Jetzt seid ihr dran!

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

II. Inhalt der Vorlesung

Themen

Korrekte Software im Lehrbuch:

- Spielzeugsprache
- Wenig Konstrukte
- Kleine Beispiele

Korrekte Software im Einsatz:

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

Inhalt

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

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

III. 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 anhand der **Eigenschaften**

Arbeitsblatt 1.3: Maschinen und Funktionen

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

Betrachtet als Beispiele:

- Eine Taschenlampe
- Eine Waschmaschine
- Einen Taschenrechner

Was ist hier die Abstraktion?

Unsere Sprache C0

- C0 ist eine **Untermenge** der Sprache C
- C0-Programme sind **ausführbare** C-Programme
- Grundausbaustufe:
  - Zuweisungen, Fallunterscheidungen, Schleifen
  - Datentypen: ganze Zahlen mit Arithmetik
  - Relationen: Vergleich ($=$, $\leq$)
  - Boolsche Operatoren: Konjunktion, Disjunktion, Negation
- 1. Ausbaustufe: Felder und Strukturen
- 2. Ausbaustufe: Fehler und Ausnahmen
- 3. Ausbaustufe: Funktionen und Prozeduren (nur Ausblick)
- 4. 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; }
```

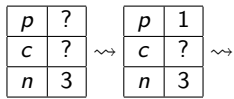
p	?
c	?
n	3

$\rightsquigarrow$

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

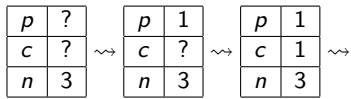
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



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

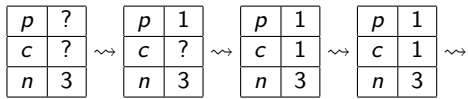
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



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

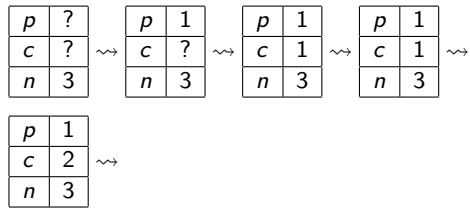
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



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

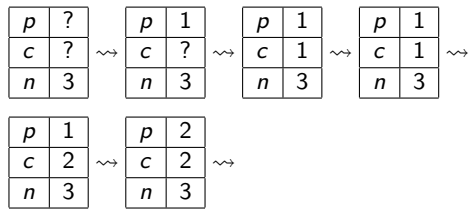
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



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

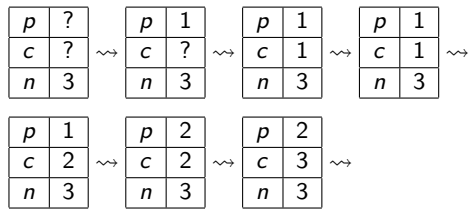
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



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

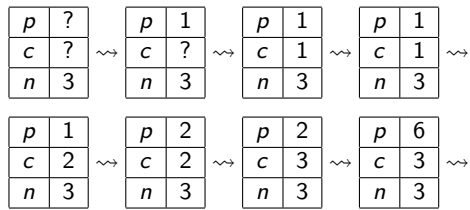
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



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

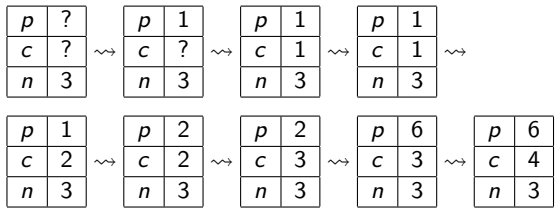
```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```



Operationale Semantik

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

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



Arbeitsblatt 1.4: Operationale Semantik

Gegeben folgendes C0-Programm:

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

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

n	4
x	?

$\rightsquigarrow \ldots$

Denotationale Semantik

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

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

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

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
```

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

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

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

Axiomatische Semantik

- Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- Zusicherungen sind zustandsabhängige Prädikate (Funktionen $\Sigma \to \mathbb{B}$)
- Beispiel (mit $n = 3$)

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
  // (4)
  p = p * c;
  c = c + 1;
  // (5)
  }
// (6)
```

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

(4) ???

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

Axiomatische Semantik

- Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- Zusicherungen sind zustandsabhängige Prädikate (Funktionen $\Sigma \to \mathbb{B}$)
- Beispiel (mit $n = 3$)

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
  // (4)
  p = p * c;
  c = c + 1;
  // (5)
  }
// (6)
```

(1) $n = 3$

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

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

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

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

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

Axiomatische Semantik

- Kernkonzept: Charakterisierung von Programmen durch **Zusicherungen**
- Zusicherungen sind zustandsabhängige Prädikate (Funktionen $\Sigma \to \mathbb{B}$)
- Beispiel (mit $n = 3$)

```
// (1)
p = 1; // (2)
c = 1; // (3)
while(c <= n){
  // (4)
  p = p * c;
  c = c + 1;
  // (5)
  }
// (6)
```

(1) $n = 3$

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

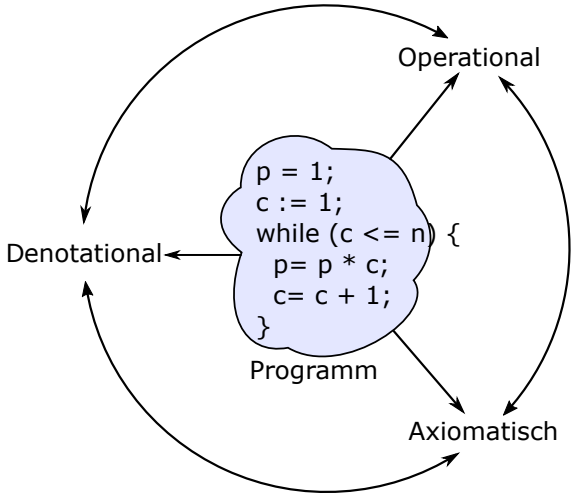
(3) $p = 1 \wedge c = 1 \wedge n = 3$

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

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

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

Drei Semantiken — Eine Sicht



IV. Mengen, Relationen, Regeln

Induktive Definitionen mit Regeln

- Wir nutzen **Regeln**, um induktiv definierte Mengen zu definieren.
  - Konkret: Relationen wie **Zustandsübergänge**.
- Regeln bestehen aus Voraussetzungen $R_1, \ldots, R_n$ und einer Konklusion S:

$$\frac{R_1 \qquad \ldots \qquad R_n}{S}$$

  - R_i und S sind beliebige Relationen.
  - Idee: (Genau dann) wenn $R_1, \ldots, R_n$ wahr sind, dann auch S.

Beispiel Fakultät

- fact(n) ist 1, wenn $n \leq 0$
- fact(n) ist $n \cdot \text{fact}(n-1)$, wenn $n > 0$
- Formalisierung als **Relation**

$$\text{fact}(n, m) \text{ mit } n, m \in N$$

- Können wir auch als rekursive Funktion auffassen, wenn rechtseindeutig

Beispiel Fakultät

- fact(n) ist 1, wenn $n \leq 0$
- fact(n) ist $n \cdot \text{fact}(n-1)$, wenn $n > 0$
- Formalisierung als **Relation**

$$\text{fact}(n, m) \text{ mit } n, m \in N$$

- Können wir auch als rekursive Funktion auffassen, wenn rechtseindeutig

$$\frac{n \leq 0}{\text{fact}(n, 1)}$$

Beispiel Fakultät

- $\text{fact}(n)$ ist 1, wenn $n \leq 0$
- $\text{fact}(n)$ ist $n \cdot \text{fact}(n-1)$, wenn $n > 0$
- Formalisierung als **Relation**

$$\text{fact}(n, m) \text{ mit } n, m \in N$$

$$\frac{n \leq 0}{\text{fact}(n, 1)}$$

$$\frac{n > 0 \qquad \text{fact}(n-1, m)}{\text{fact}(n, n \cdot m)}$$

- Können wir auch als rekursive Funktion auffassen, wenn rechtseindeutig

Beispiel Fakultät

- $\text{fact}(n)$ ist 1, wenn $n \leq 0$
- $\text{fact}(n)$ ist $n \cdot \text{fact}(n-1)$, wenn $n > 0$
- Formalisierung als **Relation**

$$\text{fact}(n, m) \text{ mit } n, m \in N$$

$$\frac{n \leq 0}{\text{fact}(n, 1)}$$

$$\frac{n > 0 \qquad \text{fact}(n-1, m)}{\text{fact}(n, n \cdot m)}$$

- Können wir auch als rekursive Funktion auffassen, wenn rechtseindeutig
- Berechnung von $\text{fact}(4, ?)$

Arbeitsblatt 1.5: Beispiel GGT

- Der ggT von n, m ist m wenn $n = 0$, oder n wenn $m = 0$.
- Ansonsten ist der ggT von n, m der ggT des kleineren von n und m und der Differenz der beiden.

1. Beschreibt den Algorithmus in Regelschreibweise
2. Berechnet damit $\text{ggT}(24, 18, ?)$

Arbeitsblatt 1.5: Beispiel GGT

- Der ggT von n, m ist m wenn $n = 0$, oder n wenn $m = 0$.
- Ansonsten ist der ggT von n, m der ggT des kleineren von n und m und der Differenz der beiden.

1. Beschreibt den Algorithmus in Regelschreibweise
2. Berechnet damit $\mathrm{ggT}(24, 18, ?)$

Formalisierung: $\mathrm{ggT}(n, m, p)$ mit $n, m, p \in \mathbb{N}$

$$\frac{}{\mathrm{ggT}(n, 0, n)}$$

Arbeitsblatt 1.5: Beispiel GGT

- Der ggT von n, m ist m wenn $n = 0$, oder n wenn $m = 0$.
- Ansonsten ist der ggT von n, m der ggT des kleineren von n und m und der Differenz der beiden.

1. Beschreibt den Algorithmus in Regelschreibweise
2. Berechnet damit $\mathrm{ggT}(24, 18, ?)$

Formalisierung: $\mathrm{ggT}(n, m, p)$ mit $n, m, p \in \mathbb{N}$

$$\frac{}{\mathrm{ggT}(n, 0, n)}$$

$$\frac{}{\mathrm{ggT}(0, m, m)}$$

Arbeitsblatt 1.5: Beispiel GGT

- Der ggT von n, m ist m wenn $n = 0$, oder n wenn $m = 0$.
- Ansonsten ist der ggT von n, m der ggT des kleineren von n und m und der Differenz der beiden.

1. Beschreibt den Algorithmus in Regelschreibweise
2. Berechnet damit $\mathrm{ggT}(24, 18, ?)$

Formalisierung: $\mathrm{ggT}(n, m, p)$ mit $n, m, p \in \mathbb{N}$

$$\frac{}{\mathrm{ggT}(n, 0, n)}$$

$$\frac{}{\mathrm{ggT}(0, m, m)}$$

$$\frac{n \leq m \qquad \mathrm{ggT}(m - n, n, p)}{\mathrm{ggT}(n, m, p)}$$

Arbeitsblatt 1.5: Beispiel GGT

- Der ggT von n, m ist m wenn $n = 0$, oder n wenn $m = 0$.
- Ansonsten ist der ggT von n, m der ggT des kleineren von n und m und der Differenz der beiden.

1. Beschreibt den Algorithmus in Regelschreibweise
2. Berechnet damit $\mathrm{ggT}(24, 18, ?)$

Formalisierung: $\mathrm{ggT}(n, m, p)$ mit $n, m, p \in \mathbb{N}$

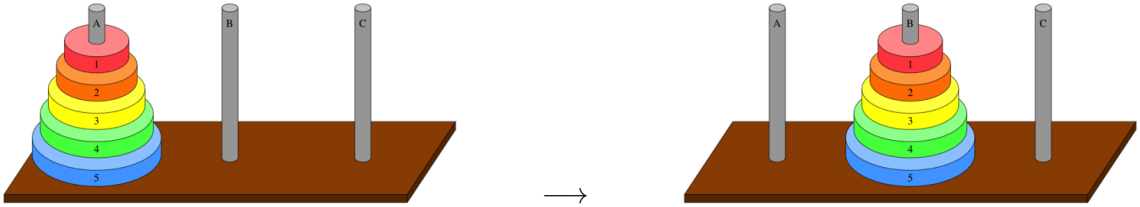
$$\frac{}{\mathrm{ggT}(n, 0, n)}$$

$$\frac{}{\mathrm{ggT}(0, m, m)}$$

$$\frac{n \leq m \qquad \mathrm{ggT}(m - n, n, p)}{\mathrm{ggT}(n, m, p)}$$

$$\frac{m < n \qquad \mathrm{ggT}(n - m, m, p)}{\mathrm{ggT}(n, m, p)}$$

Türme von Hanoi

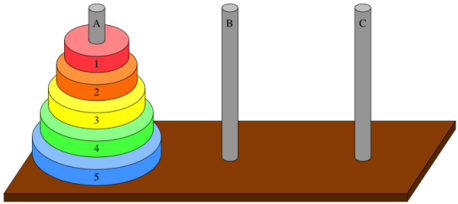


Quelle: https://www.khanacademy.org/computing/computer-science/algorithms/towers-of-hanoi/a/towers-of-hanoi

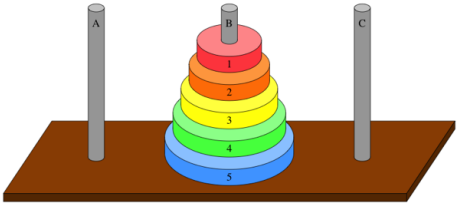
- Umstapeln zwischen den Stäben
- Jede Scheibe darf entweder auf einen leeren Stab oder eine größere Scheibe gelegt werden
  - Dies kann repräsentiert werden bei 9 Scheiben, dass
    1. "ein Stab ist leer" mit der Sequenz $\langle\,\rangle$ der Länge 0
    2. "Ein Stab enthält Scheiben $n_1, \ldots, n_k$" durch die Sequenz $\langle n_1, \ldots, n_k \rangle$ der Länge k, wobei gelten muss $n_i < n_{i+1}$.

    Damit lassen sich Spielzustände repräsentieren als $\langle f_A, f_B, f_C \rangle$ wobei f_A, f_B, f_C die Sequenzen der Nummern der Scheiben auf den entsprechenden Stäben.

Arbeitsblatt 1.6: Türme von Hanoi



$\longrightarrow$



- Seien die Züge beschrieben durch $\rightarrow_{AB}$ als Zug der obersten Scheibe auf A auf den Stapel B. Entsprechend $\rightarrow_{BA}, \rightarrow_{AC}, \rightarrow_{CA}, \rightarrow_{BC}$, und $\rightarrow_{CB}$.
- Beschreibt mittels Regeln die zulässigen Bewegungen zwischen den Stäben:

$$\frac{?}{\langle f_A, f_B, f_C\rangle \rightarrow_{AB} ?} \qquad \frac{?}{\langle f_A, f_B, f_C\rangle \rightarrow_{BA} ?} \qquad \frac{?}{\langle f_A, f_B, f_C\rangle \rightarrow_{AC} ?}$$

$$\frac{?}{\langle f_A, f_B, f_C\rangle \rightarrow_{CA} ?} \qquad \frac{?}{\langle f_A, f_B, f_C\rangle \rightarrow_{BC} ?} \qquad \frac{?}{\langle f_A, f_B, f_C\rangle \rightarrow_{CA} ?}$$

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 2 vom 10.04.24

Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Zutaten

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

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

Unsere Programmiersprache

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

Ausbaustufe 1 kennt folgende Konstrukte:

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

C0: Ausdrücke und Anweisungen

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

NB: Nicht die **konkrete** Syntax.

Was braucht die Semantik?

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

- Ein Programm besteht aus **Anweisungen** und **Ausdrücken**.
- Ausdrücke werden **zustandsabhängig** ausgewertet.
- Anweisungen **überführen** Zustände.

Was braucht die Semantik?

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

- Ein Programm besteht aus **Anweisungen** und **Ausdrücken**.
- Ausdrücke werden **zustandsabhängig** ausgewertet.
- Anweisungen **überführen** Zustände.

Woraus besteht die Semantik?

Was braucht die Semantik?

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

- Ein Programm besteht aus **Anweisungen** und **Ausdrücken**.
- Ausdrücke werden **zustandsabhängig** ausgewertet.
- Anweisungen **überführen** Zustände.

Woraus besteht die Semantik?

1. Mathematische Modellierung des **Zustands**

Was braucht die Semantik?

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

- Ein Programm besteht aus **Anweisungen** und **Ausdrücken**.
- Ausdrücke werden **zustandsabhängig** ausgewertet.
- Anweisungen **überführen** Zustände.

Woraus besteht die Semantik?

1. Mathematische Modellierung des **Zustands**
2. Auswertung von (arithmetischen und boolschen) Ausdrücken

Was braucht die Semantik?

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

- Ein Programm besteht aus **Anweisungen** und **Ausdrücken**.
- Ausdrücke werden **zustandsabhängig** ausgewertet.
- Anweisungen **überführen** Zustände.

Woraus besteht die Semantik?

1. Mathematische Modellierung des **Zustands**
2. Auswertung von (arithmetischen und boolschen) Ausdrücken
3. Auswertung von Anweisungen: Zustandsübergänge

Semantik von C0

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

Systemzustände

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

Partielle, endliche Abbildungen I

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

$$f : X \rightharpoonup A$$

Notation:

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

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

Partielle, endliche Abbildungen II

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

$$f : X \rightharpoonup A$$

Notation:

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

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

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

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

- Updates sind "linksassoziativ":

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

Arbeitsblatt 2.1: Zustände!

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

Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck a wertet unter Zustand σ zu einer ganzen Zahl n (Wert) aus.

$$\textbf{Aexp}\ a ::= \textbf{Z} \mid \textbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1\ /\ a_2 \qquad \langle a, \sigma \rangle \to_{Aexp} n$$

Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck a wertet unter Zustand σ zu einer ganzen Zahl n (Wert) aus.

$$\textbf{Aexp}\ a ::= \mathbf{Z} \mid \mathbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1\ /\ a_2 \qquad \langle a, \sigma \rangle \rightarrow_{Aexp} n$$

Regeln:

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

Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck a wertet unter Zustand σ zu einer ganzen Zahl n (Wert) aus.

$$\textbf{Aexp}\ a ::= \textbf{Z} \mid \textbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1\ /\ a_2 \qquad \langle a, \sigma \rangle \rightarrow_{Aexp} n$$

Regeln:

$$\frac{n \in \textbf{Z}}{\langle n, \sigma \rangle \rightarrow_{Aexp} [\![n]\!]} \qquad \frac{x \in \textbf{Idt}, x \in Dom(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \rightarrow_{Aexp} v}$$

Operationale Semantik: Arithmetische Ausdrücke

$$\textbf{Aexp}\ a ::= \textbf{Z} \mid \textbf{Idt} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2 \qquad \langle a, \sigma \rangle \to_{Aexp} n$$

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

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

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

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

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{\langle x, \sigma \rangle \rightarrow_{Aexp} ? \qquad \langle 3, \sigma \rangle \rightarrow_{Aexp} ?}{\langle x + 3, \sigma \rangle \rightarrow_{Aexp} ? + ?}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{}{\langle 3, \sigma \rangle \to_{Aexp} [\![3]\!]}$$

$$\frac{\langle x, \sigma \rangle \to_{Aexp} ? \qquad \langle 3, \sigma \rangle \to_{Aexp} ?}{\langle x + 3, \sigma \rangle \to_{Aexp} ? + ?}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{}{\langle 3, \sigma \rangle \to_{Aexp} 3}$$

$$\frac{\langle x, \sigma \rangle \to_{Aexp} ? \qquad \langle 3, \sigma \rangle \to_{Aexp} ?}{\langle x + 3, \sigma \rangle \to_{Aexp} ? + ?}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{x \in \mathrm{dom}(\sigma), \sigma(x) = ?}{\langle x, \sigma \rangle \to_{Aexp} ?} \qquad \frac{}{\langle 3, \sigma \rangle \to_{Aexp} 3}$$

$$\frac{\langle x, \sigma \rangle \to_{Aexp} ? \qquad \langle 3, \sigma \rangle \to_{Aexp} ?}{\langle x + 3, \sigma \rangle \to_{Aexp} ? + ?}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \qquad \frac{}{\langle 3, \sigma \rangle \to_{Aexp} 3}$$

$$\frac{\langle x, \sigma \rangle \to_{Aexp} ? \qquad \langle 3, \sigma \rangle \to_{Aexp} ?}{\langle x + 3, \sigma \rangle \to_{Aexp} ? + ?}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{}{\langle x, \sigma \rangle \rightarrow_{Aexp} 6} \qquad \frac{}{\langle 3, \sigma \rangle \rightarrow_{Aexp} 3}$$

$$\frac{\langle x, \sigma \rangle \rightarrow_{Aexp} ? \qquad \langle 3, \sigma \rangle \rightarrow_{Aexp} ?}{\langle x + 3, \sigma \rangle \rightarrow_{Aexp} ? + ?}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{}{\langle x, \sigma \rangle \rightarrow_{Aexp} 6} \qquad \frac{}{\langle 3, \sigma \rangle \rightarrow_{Aexp} 3}$$

$$\frac{\langle x, \sigma \rangle \rightarrow_{Aexp} 6 \qquad \langle 3, \sigma \rangle \rightarrow_{Aexp} 3}{\langle x + 3, \sigma \rangle \rightarrow_{Aexp} 6 + 3}$$

Ableitungen

- Regeln werden von **unten** nach **oben** gelesen
- Regeln werden **komponiert** — es entsteht ein **Ableitungsbaum**

Beispiel: Auswertung von x+3 mit $\sigma \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 5 \rangle$.

$$\frac{}{\langle x, \sigma \rangle \to_{Aexp} 6} \qquad \frac{}{\langle 3, \sigma \rangle \to_{Aexp} 3}$$

$$\frac{\langle x, \sigma \rangle \to_{Aexp} 6 \qquad \langle 3, \sigma \rangle \to_{Aexp} 3}{\langle x + 3, \sigma \rangle \to_{Aexp} 9}$$

Längere Beispiel-Ableitungen

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

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

Längere Beispiel-Ableitungen

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

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

Längere Beispiel-Ableitungen

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

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

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp}} \quad \dfrac{}{\langle x - y, \sigma \rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp}}$$

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{}{\langle x - y, \sigma \rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp}}$$

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp}}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp}}$$

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp}}$$

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp} 11}$$

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathsf{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathsf{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp} 11}$$

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

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathrm{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathrm{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp} 11}$$

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

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathrm{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathrm{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp} 11}$$

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

Längere Beispiel-Ableitungen

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

$$\dfrac{\dfrac{\dfrac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathrm{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x + y, \sigma \rangle \to_{Aexp} 11} \quad \dfrac{\dfrac{x \in \mathrm{dom}(\sigma), \sigma(x) = 6}{\langle x, \sigma \rangle \to_{Aexp} 6} \quad \dfrac{y \in \mathrm{dom}(\sigma), \sigma(y) = 5}{\langle y, \sigma \rangle \to_{Aexp} 5}}{\langle x - y, \sigma \rangle \to_{Aexp} 1}}{\langle (x + y) * (x - y), \sigma \rangle \to_{Aexp} 11}$$

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

Arbeitsblatt 2.2: Auswertung

Konstruiert wie oben die Ableitung für den Ausdruck (3*a)/b mit $\sigma \stackrel{def}{=} \langle a \mapsto 8, b \mapsto 7 \rangle$.

Hinweis: wahrscheinlich einfacher auf Papier...

Eigenschaften der Semantik

- **Frage:** Gegeben einen Ausdruck a, leitet **jeder** Zustand σ zu einem Wert n ab?

Eigenschaften der Semantik

- **Frage:** Gegeben einen Ausdruck a, leitet **jeder** Zustand σ zu einem Wert n ab?
- **Antwort:** Nein.
- Betrachte folgende Beispiele für $a \stackrel{def}{=} \mathrm{y+3/x}$

$$\langle a, \langle y \mapsto 5 \rangle \rangle \rightarrow_{Aexp} ??? \quad (1)$$

$$\langle a, \langle y \mapsto 5, x \mapsto 0 \rangle \rangle \rightarrow_{Aexp} ??? \quad (2)$$

- In diesen Beispielen läßt sich kein **vollständiger** Ableitungsbaum konstruieren.
- Die Auswertung ist **undefiniert** — die Semantik ist **partiell**.

Operationale Semantik: Boolesche Ausdrücke

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

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

Regeln:

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

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

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

Operationale Semantik: Boolesche Ausdrücke

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

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

Regeln:

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

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

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

Arbeitsblatt 2.3: Boolsche Ausdrücke

Konstruiert die Auswertung des Ausdrucks $b = \texttt{x} == 7 \;\&\&\; \texttt{y} == 3$ unter folgenden Zuständen:

1. $\sigma_1 \stackrel{def}{=} \langle x \mapsto 7, y \mapsto 3 \rangle$
2. $\sigma_2 \stackrel{def}{=} \langle x \mapsto 6, y \mapsto 3 \rangle$
3. $\sigma_3 \stackrel{def}{=} \langle y \mapsto 6 \rangle$
4. $\sigma_4 \stackrel{def}{=} \langle x \mapsto 7 \rangle$
5. $\sigma_5 \stackrel{def}{=} \langle x \mapsto 2 \rangle$

Striktheit

- Eine partielle Funktion f ist **strikt** wenn $f(x)$ undefiniert ist, sobald x undefiniert ist.
- In unserer Semantik sind alle Operatoren (arithmetisch und boolesch) strikt, **bis auf** && und || im **ersten** Argument.
  - Operational nennt man das auch abgekürzte Auswertung (*short-circuit evaluation*)
  - Das erlaubt Idiome wie **if** $(x \mathrel{!}= 0 \;\&\&\; 3/x > 1)\ \{\ \ldots\ \}$
- Wie erkennt man Striktheit an den **Regeln**?

Striktheit

- Eine partielle Funktion f ist **strikt** wenn $f(x)$ undefiniert ist, sobald x undefiniert ist.
- In unserer Semantik sind alle Operatoren (arithmetisch und boolesch) strikt, **bis auf** && und || im **ersten** Argument.
  - Operational nennt man das auch abgekürzte Auswertung (*short-circuit evaluation*)
  - Das erlaubt Idiome wie **if** $(x \mathrel{!}= 0 \;\&\&\; 3/x > 1)\ \{\ \ldots\ \}$
- Wie erkennt man Striktheit an den **Regeln**?
  Alle Variablen der Konklusion kommen in den Bedingungen vor.

Operationale Semantik: Anweisungen

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

Beispiel:

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

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

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

Operationale Semantik: Anweisungen

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

Beispiel:

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

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

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

Operationale Semantik: Anweisungen

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

Beispiel:

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

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

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

Beispiel

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

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

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

(A)

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

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

(B)

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

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

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

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

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

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

Lineare, abgekürzte Schreibweise

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

Was haben wir gezeigt?

```
// ⟨y ↦ 2⟩                    σ1
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
// ⟨y ↦ 0, x ↦ 4⟩             σE
```

- Für **einen festen Anfangszustand** $\sigma_1 = \langle y \mapsto 2 \rangle$ gilt am Ende $\sigma_E(x) = 4 = 2^2 = 2^{\sigma_1(y)}$.

Was haben wir gezeigt?

```
// ⟨y ↦ 2⟩                    σ1
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
// ⟨y ↦ 0, x ↦ 4⟩             σE
```

- Für **einen festen Anfangszustand** $\sigma_1 = \langle y \mapsto 2 \rangle$ gilt am Ende $\sigma_E(x) = 4 = 2^2 = 2^{\sigma_1(y)}$.
- Gilt das für alle?

Was haben wir gezeigt?

```
// ⟨y ↦ 2⟩                    σ1
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
// ⟨y ↦ 0, x ↦ 4⟩             σE
```

- Für **einen festen Anfangszustand** $\sigma_1 = \langle y \mapsto 2 \rangle$ gilt am Ende $\sigma_E(x) = 4 = 2^2 = 2^{\sigma_1(y)}$.
- Gilt das für alle?
- Für welche nicht?

Was haben wir gezeigt?

```
// ⟨y ↦ 2⟩                    σ1
x = 1;
// ⟨y ↦ 2, x ↦ 1⟩
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
// ⟨y ↦ 0, x ↦ 4⟩             σE
```

- Für **einen festen Anfangszustand** $\sigma_1 = \langle y \mapsto 2 \rangle$ gilt am Ende $\sigma_E(x) = 4 = 2^2 = 2^{\sigma_1(y)}$.
- 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 Auswertung der while-Schleife wächst unendlich)
- In linearer Schreibweise geht es immer wieder unten weiter.

Arbeitsblatt 2.4: Programme!

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

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

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**: sowohl w also auch **if** (b) $\{c; w\}$ **else** $\{\}$ werten zum gleichen Programmzustand aus (wenn sie auswerten).
- Der Beweis geht per Fallunterscheidung über die Auswertung von Teilausdrücken bzw. Teilprogrammen.

Beweis

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

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

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

Beweis

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

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

2. $\langle b, \sigma \rangle \to_{Bexp} \mathit{true}$: Sei $\langle c, \sigma \rangle \to_{Stmt} \sigma'$, dann:

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

Beweis

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

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

2. $\langle b, \sigma \rangle \to_{Bexp} \mathit{true}$: Sei $\langle c, \sigma \rangle \to_{Stmt} \sigma'$, dann:

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

3. $\langle b, \sigma \rangle$ wertet gar nicht aus — dann werten weder w noch $\textbf{if } (b)\ \{c; w\}\ \textbf{else}\ \{\}$ aus.

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 3 vom 17.04.24

Denotationale Semantik

Serge Autexier, Christoph Lüth

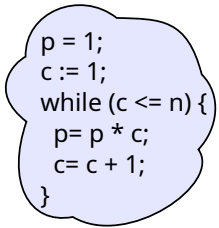
Universität Bremen

Sommersemester 2024

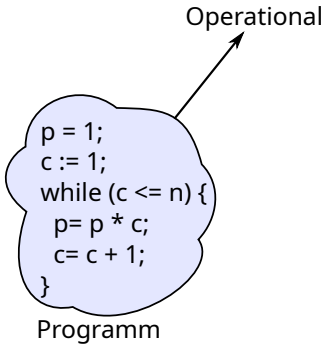
Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

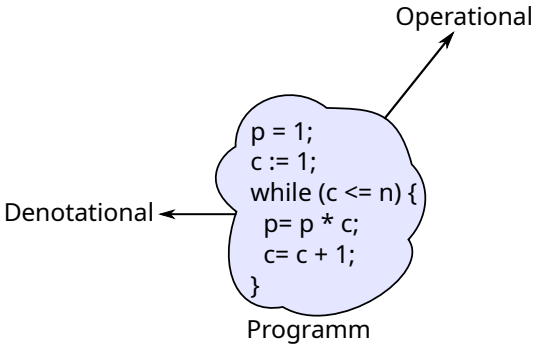
Überblick



Überblick



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 überführen:

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

- **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 — 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 Funktionen und ihre Graphen

- Der **Graph** einer partiellen Funktion $f : X \rightharpoonup Y$ ist eine Relation

$$grph(f) \subseteq X \times Y \stackrel{def}{=} \{(x, f(x)) \mid x \in \mathrm{dom}(f)\}$$

- Wir können eine partielle Funktion durch ihren Graph definieren:

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**)

- Wir benutzen beide Notationen, aber für die denotationale Semantik die Graph-Notation.
- **Systemzustände** sind partielle Abbildungen $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$ ($\longrightarrow$ letzte VL)

Beispiel

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

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

- Zuordnung:

0	$\mapsto$	0
1		
2		
3	$\mapsto$	1
4		
5		
6	$\mapsto$	2
7		
8		
9	$\mapsto$	3
10		

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

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

- Wir schreiben

$div3(3) = 1$	für $(3,1) \in div3$
$div3(5) = \bot$	für es gibt kein y mit $(5,y) \in div3$
$div3(5) = \bot$	für $\forall y.(5,y) \notin div3$

- Achtung, Partialität!

Achtung, Partialität!

- Beim Rechnen mit partiellen Funktionen muss die **Definiertheit** beachtet werden.
- Insbesondere darf nicht mit undefinierten Ausdrücken gerechnet werden.
- Bspw. gilt oben **nicht** im allgemeinen:

$$3 \cdot \mathit{div3}(x) = x \quad \times$$

oder

$$\mathit{div3}(1) = \bot = \mathit{div3}(2) \Longrightarrow \mathit{div3}(1) = \mathit{div3}(2) \quad \times$$

- Warum? Dann gälte

Achtung, Partialität!

- Beim Rechnen mit partiellen Funktionen muss die **Definiertheit** beachtet werden.
- Insbesondere darf nicht mit undefinierten Ausdrücken gerechnet werden.
- Bspw. gilt oben **nicht** im allgemeinen:

$$3 \cdot \mathit{div3}(x) = x \quad \times$$

oder

$$\mathit{div3}(1) = \bot = \mathit{div3}(2) \Longrightarrow \mathit{div3}(1) = \mathit{div3}(2) \quad \times$$

- Warum? Dann gälte

$$\mathit{div3}(1) = \mathit{div3}(2)$$

Achtung, Partialität!

- Beim Rechnen mit partiellen Funktionen muss die **Definiertheit** beachtet werden.
- Insbesondere darf nicht mit undefinierten Ausdrücken gerechnet werden.
- Bspw. gilt oben **nicht** im allgemeinen:

$$3 \cdot \mathit{div3}(x) = x \quad \times$$

oder

$$\mathit{div3}(1) = \bot = \mathit{div3}(2) \Longrightarrow \mathit{div3}(1) = \mathit{div3}(2) \quad \times$$

- Warum? Dann gälte

$$\mathit{div3}(1) = \mathit{div3}(2)$$
$$3 \cdot \mathit{div3}(1) = 3 \cdot \mathit{div3}(2)$$

Achtung, Partialität!

- Beim Rechnen mit partiellen Funktionen muss die **Definiertheit** beachtet werden.
- Insbesondere darf nicht mit undefinierten Ausdrücken gerechnet werden.
- Bspw. gilt oben **nicht** im allgemeinen:

$$3 \cdot \mathit{div3}(x) = x \quad \times$$

oder

$$\mathit{div3}(1) = \bot = \mathit{div3}(2) \Longrightarrow \mathit{div3}(1) = \mathit{div3}(2) \quad \times$$

- Warum? Dann gälte

$$\begin{aligned} \mathit{div3}(1) &= \mathit{div3}(2) \\ 3 \cdot \mathit{div3}(1) &= 3 \cdot \mathit{div3}(2) \\ 1 &= 2 \quad \lightning \end{aligned}$$

Achtung, Partialität!

- Beim Rechnen mit partiellen Funktionen muss die **Definiertheit** beachtet werden.
- Insbesondere darf nicht mit undefinierten Ausdrücken gerechnet werden.
- Bspw. gilt oben **nicht** im allgemeinen:

$$3 \cdot div3(x) = x \quad \times$$

oder

$$div3(1) = \bot = div3(2) \Longrightarrow div3(1) = div3(2) \quad \times$$

- Warum? Dann gälte

$$\begin{aligned} div3(1) &= div3(2) \\ 3 \cdot div3(1) &= 3 \cdot div3(2) \\ 1 &= 2 \quad \lightning \end{aligned}$$

- Vgl. `https://de.wikipedia.org/wiki/Trugschluss_(Mathematik)#Division_durch_0`

Arbeitsblatt 3.1: Relationen als Funktionen

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

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

Was ist der Wert folgender Ausdrücke:

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

Arbeitsblatt 3.1: Relationen als Funktionen

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

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

Was ist der Wert folgender Ausdrücke:

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

Denotierende Funktionen (Denotate)

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

Denotat von Aexp

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

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

Rechtseindeutigkeit

Lemma (Partielle Funktion)

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

Beweis.

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

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 $v_1 = \sigma(x) = v_2$.
- Induktionsschritt sind die anderen Klauseln.
  Sei $a \equiv a_1 + a_2$.
  Induktionsannahme ist: wenn $(\sigma, n_i) \in [\![a_i]\!]_{\mathcal{A}}, (\sigma, m_i) \in [\![a_i]\!]_{\mathcal{A}}$ dann $n_i = m_i$.
  Sei $v_1 = n_1 + n_2$ mit $(\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, n_2) \in [\![a_2]\!]_{\mathcal{A}}$, und $v_2 = m_1 + m_2$ mit $(\sigma, m_1) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, m_2) \in [\![a_2]\!]_{\mathcal{A}}$.

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

□

Kompositionalität und Striktheit

- Die Rechtseindeutigkeit erlaubt die Notation als partielle Funktion:

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

- Diese Notation versteckt die **Partialität**:

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

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

Arbeitsblatt 3.2: Semantik I

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

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

Berechnen Sie zum anderen die Semantik in der Funktionsnotation:

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

Ist das Ergebnis am Ende gleich?

Lösung

Denotat von Bexp

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

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

Denotat von Bexp

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

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

Kompositionalität und Striktheit

Lemma (Partielle Funktion)

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

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

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:

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

Arbeitsblatt 3.3: Semantik II

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

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

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

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

Lösung

Denotationale Semantik von Anweisungen

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

Definition (Komposition von Relationen)

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

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

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

- Leere Sequenz: Leere Funktion?

Denotationale Semantik von Anweisungen

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

Definition (Komposition von Relationen)

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

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

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

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

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

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

Arbeitsblatt 3.4: 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 $R \circ S = \{(1,?),\ldots\}$

Arbeitsblatt 3.4: 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 $R \circ S = \{(1,?),\ldots\}$

Arbeitsblatt 3.4: Komposition von Relationen

Zur Übung: betrachten Sie folgende Relationen:

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

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

Denotat von Stmt

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

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

Denotat von Stmt

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

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

Aber was ist

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

Denotationale Semantik von while

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

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

- Dann sollte auch gelten

$$\begin{aligned}[\![w]\!]_{\mathcal{C}} &\stackrel{?}{=} [\![\textbf{if } (b) \ \{c; w\} \ \textbf{ else } \ \{\,\}]\!]_{\mathcal{C}} \\ &= \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}} \circ [\![w]\!]_{\mathcal{C}}\} \\ &\quad \cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![\{\,\}]\!]_{\mathcal{C}}\}\end{aligned}$$

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

$$x = F(x)$$

- Das ist ein **Fixpunkt**:

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

- Was ist das?

Denotationale Semantik von while

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

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

- Dann sollte auch gelten

$$\begin{aligned}[\![w]\!]_{\mathcal{C}} &\stackrel{?}{=} [\![\textbf{if } (b) \ \{c; w\} \ \textbf{ else } \ \{\,\}]\!]_{\mathcal{C}} \\ &= \{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}} \circ [\![w]\!]_{\mathcal{C}}\} \\ &\quad \cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![\{\,\}]\!]_{\mathcal{C}}\}\end{aligned}$$

- 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.

Denotationale Semantik für die Iteration

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

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

- b und c sind Parameter von Γ
- Dann ist

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

Konstruktion des kleinsten Fixpunktes (Kurzversion)

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

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

- 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)

Denotation für Stmt

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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;
  }
```

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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

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

Der Fixpunkt bei der Arbeit (I)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

s
n
-1
0
1
2
3
4

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (II)

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

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

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

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

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

s
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;
  }
```

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (III)

Kleine Änderung im Beispielprogramm:

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

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

Jetzt ergibt sich:

s	$\Gamma^0(s)$		$\Gamma^1(s)$		$\Gamma^2(s)$		$\Gamma^3(s)$	
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;
  }
```

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

s
-2
-1
0
1
2
3

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Der Fixpunkt bei der Arbeit (IV)

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

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

Jetzt ergibt sich:

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

Arbeitsblatt 3.5: Semantik III

Wir betrachten das Beispielprogramm:

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

Berechnen Sie wie oben den Fixpunkt:

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

Arbeitsblatt 3.5: Semantik III

Wir betrachten das Beispielprogramm:

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

Der Fixpunkt bei der Arbeit (V)

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

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

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

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

Der Fixpunkt bei der Arbeit (V)

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

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

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

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

Der Fixpunkt bei der Arbeit (V)

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

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

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

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

Der Fixpunkt bei der Arbeit (V)

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

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

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

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

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

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

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

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

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

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

Weitere Eigenschaften der denotationalen Semantik

Lemma (Partielle Funktion)

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

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

Note

$$\mathit{fix}(\Gamma) = \Gamma(\mathit{fix}(\Gamma))$$

Beweis (1)

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

Note

$$\mathit{fix}(\Gamma) = \Gamma(\mathit{fix}(\Gamma))$$

Beweis (1)

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

Note

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

Beweis (1)

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

Note

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

Beweis (1)

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

Note

Beweis (1)

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

Note

Beweis (1)

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

Note

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

Beweis (1)

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

Note

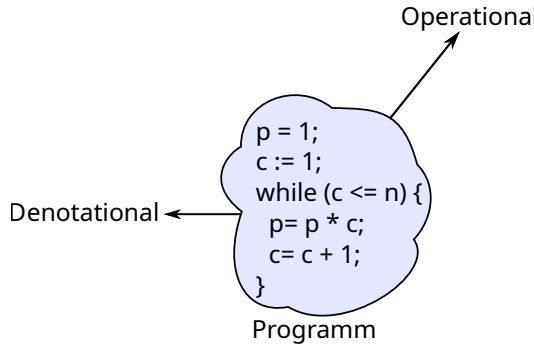
$$
\begin{aligned}
[\![\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1]\!]_{\mathcal{C}} = \ & \{(\sigma,\sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma,\sigma') \in [\![c_0]\!]_{\mathcal{C}}\} \\
& \cup \{(\sigma,\sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}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

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?**

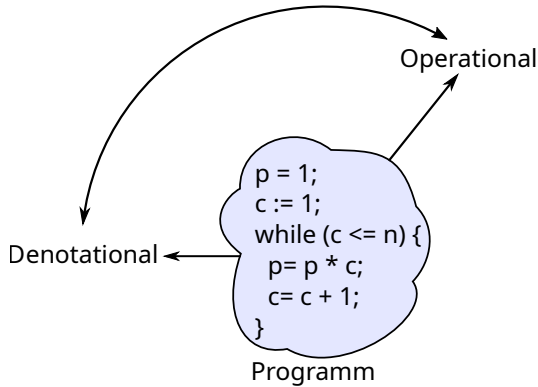
Zusammenfassung

- 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



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 24.04.24

Äquivalenz der Operationalen und Denotationalen Semantik

Serge Autexier, Christoph Lüth

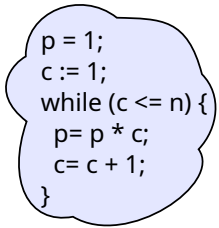
Universität Bremen

Sommersemester 2024

Fahrplan

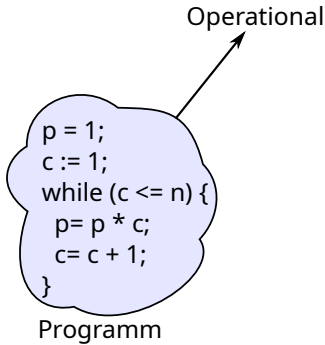
- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Operationale und Denotationale Semantik

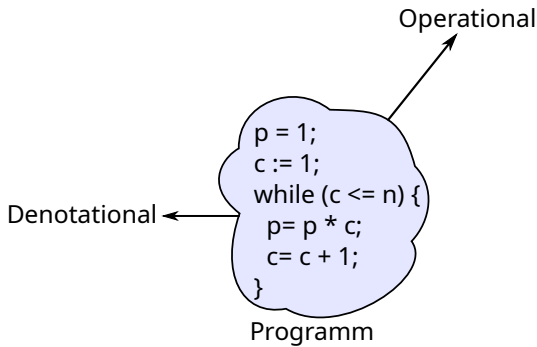


Programm

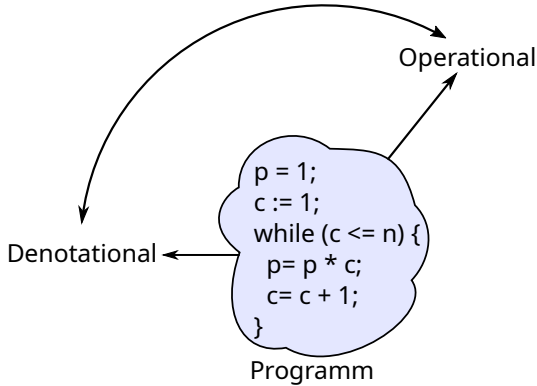
Operationale und Denotationale Semantik



Operationale und Denotationale Semantik



Operationale und Denotationale Semantik



Äquivalenz der Operationalen und Denotationalen Semantik

- Was müssen wir zeigen?

Äquivalenz der Operationalen und Denotationalen Semantik

- Was müssen wir zeigen?
- Auf oberster Ebene: für alle $c \in \mathbf{Stmt}, \sigma, \sigma' \in \Sigma$:

$$\langle c, \sigma \rangle \to_{Stmt} \sigma' \iff (\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}} \tag{1}$$

- Semantik von Anweisungen ist über Semantik von Ausdrücken definiert, deshalb benötigen wir Hilfsaussagen

$$\langle b, \sigma \rangle \to_{Bexp} t \iff (\sigma, t) \in [\![b]\!]_{\mathcal{B}} \tag{2}$$
$$\langle a, \sigma \rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a]\!]_{\mathcal{A}} \tag{3}$$

- Wie zeigen wir das?

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**	$\dfrac{x \in Dom(\sigma)}{\langle x, \sigma \rangle \rightarrow_{Aexp} \sigma(x)}$	$\{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in Dom(\sigma)\}$

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma\rangle \to_{Aexp} n$	**Denotational** $[\![a]\!]_{\mathcal{A}}$
$a_1 \otimes a_2$	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m}{\langle a_1 \otimes a_2, \sigma\rangle \to_{Aexp} n \otimes^I m}$ $\otimes \in \{+, *, -\}$	$\{(\sigma, n \otimes^I m) \mid \sigma \in \Sigma, (\sigma, n) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}\}$
a_1 / a_2	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad m \neq 0}{\langle a_1 / a_2, \sigma\rangle \to_{Aexp} n \div m}$	$\{(\sigma, n \div m) \mid \sigma \in \Sigma, (\sigma, n) \in [\![a_1]\!]_{\mathcal{A}}, (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}, m \neq 0\}$

Äquivalenz operationale und denotationale Semantik

- Zu zeigen Gleichung (3) von Folie 4:
- Für alle $a \in \mathbf{Aexp}$, für alle $n \in \mathbb{Z}$, für alle Zustände σ:

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

- Beweis Prinzip?

Exkurs: Beweisprinzipien

- Induktion über $\mathbb{N}$ ($\mathrm{nf}(n)$ ist der **Nachfolger** von n):

$$\frac{P(0) \wedge \forall n \in \mathbb{N}.\, P(n) \Longrightarrow P(\mathrm{nf}(n))}{\forall x \in \mathbb{N}.\, P(x)}$$

- Beispiel: Addition ist definiert durch

$$x + 0 = x$$
$$x + \mathrm{nf}(y) = \mathrm{nf}(x + y)$$

- Zeige $x + y = y + x$ durch Induktion über y:
  1. Basis: $x + 0 = 0 + x$
  2. Induktionsschritt: Annahme $x + y = y + x$, dann zeige $x + \mathrm{nf}(y) = \mathrm{nf}(y) + x$.
     - Benötigt Hilfsbeweise $0 + x = x$ und $\mathrm{nf}(x + y) = \mathrm{nf}(x) + y$

Arbeitsblatt 4.1: Natürliche Induktion

- Zeigt durch natürliche Induktion:

$$0 + x = x \qquad \mathsf{nf}(x + y) = \mathsf{nf}(x) + y$$

- Welche Variable benutzt ihr für die Induktion? Was ist der Unterschied?

Wohlfundiertheit

Wohlfundiertheit

Eine binäre Relation $\prec \subseteq S \times S$ ist **wohlfundiert**, wenn es keine unendlich **absteigenden** Ketten gibt

$$\cdots \prec a_3 \prec a_2 \prec a_1$$

Beispiele:

- $(N, \leq)$?

Wohlfundiertheit

Wohlfundiertheit

Eine binäre Relation $\prec \subseteq S \times S$ ist **wohlfundiert**, wenn es keine unendlich **absteigenden** Ketten gibt

$$\cdots \prec a_3 \prec a_2 \prec a_1$$

Beispiele:

- $(N, \leq)$? Nein: $\cdots \leq 1 \leq 1 \leq 1$
- $(\mathbb{N}, <)$?

Wohlfundiertheit

Wohlfundiertheit

Eine binäre Relation $\prec \subseteq S \times S$ ist **wohlfundiert**, wenn es keine unendlich **absteigenden** Ketten gibt

$$\cdots \prec a_3 \prec a_2 \prec a_1$$

Beispiele:

- $(N, \leq)$? Nein: $\cdots \leq 1 \leq 1 \leq 1$
- $(\mathbb{N}, <)$? Ja.
- $(\mathbb{Z}, <)$?

Wohlfundiertheit

Wohlfundiertheit

Eine binäre Relation $\prec \subseteq S \times S$ ist **wohlfundiert**, wenn es keine unendlich **absteigenden** Ketten gibt

$$\cdots \prec a_3 \prec a_2 \prec a_1$$

Beispiele:

- $(N, \leq)$? Nein: $\cdots \leq 1 \leq 1 \leq 1$
- $(\mathbb{N}, <)$? Ja.
- $(\mathbb{Z}, <)$? Nein: $\cdots < -3 < -2 < -1 < 0$
- $(\mathbb{Q}^+, <)$?

Wohlfundiertheit

Wohlfundiertheit

Eine binäre Relation $\prec \subseteq S \times S$ ist **wohlfundiert**, wenn es keine unendlich **absteigenden** Ketten gibt

$$\cdots \prec a_3 \prec a_2 \prec a_1$$

Beispiele:

- $(N, \leq)$? Nein: $\cdots \leq 1 \leq 1 \leq 1$
- $(\mathbb{N}, <)$? Ja.
- $(\mathbb{Z}, <)$? Nein: $\cdots < -3 < -2 < -1 < 0$
- $(\mathbb{Q}^+, <)$? Nein: $\ldots < \frac{1}{n} \ldots < \frac{1}{4} < \frac{1}{3} < \frac{1}{2} < 1$

Eigenschaften wohlfundierter Relationen

- Eine wohlfundierte Relation ist **irreflexiv:** $\forall x \in S.\, x \not\prec x$

Eigenschaften wohlfundierter Relationen

- Eine wohlfundierte Relation ist **irreflexiv:** $\forall x \in S.\, x \not\prec x$
  - Ansonsten gäbe es $\cdots \prec x \prec x \prec x$
- **Lemma**: $\prec$ ist wohlfundiert gdw. jede nicht-leere Untermenge $Q \subseteq S$ ein minimales Element $\min Q$ hat:

$$\min Q \in Q \wedge \forall b.\, b \prec \min Q \Longrightarrow b \notin Q$$

Wohlfundierte Induktion

Noethersche Induktion (Wohlfundierte Induktion)

Sei $\prec \subseteq R \times R$ **wohlfundiert** und P eine Aussage über Elemente von R. Dann gilt

$$\frac{\forall v \in R.\, (\forall u \in R.\, u \prec v \Longrightarrow P(u)) \Longrightarrow P(v)}{\forall x \in R.\, P(x)}$$

Beispiele:

- Mit $S = \mathbb{N}$, $a \prec a + 1$: natürliche Induktion.
- Warum?

Wohlfundierte Induktion

Noethersche Induktion (Wohlfundierte Induktion)

Sei $\prec \subseteq R \times R$ **wohlfundiert** und P eine Aussage über Elemente von R. Dann gilt

$$\frac{\forall v \in R.\, (\forall u \in R.\, u \prec v \Longrightarrow P(u)) \Longrightarrow P(v)}{\forall x \in R.\, P(x)}$$

Beispiele:

- Mit $S = \mathbb{N}$, $a \prec a + 1$: natürliche Induktion.
- Warum? Fallunterscheidung über v: entweder $v = 0$, dann gibt es kein u so dass $u \prec 0$ und die Voraussetzung ist $P(0)$; oder $v = w + 1$, dann $w \prec w + 1$, und die Voraussetzung ist $P(w) \Longrightarrow P(w + 1)$

Strukturelle Ordnung

Strukturelle Ordnung

Die strukturelle Ordnung auf arithmetischen Ausdrücken ist definiert als:

$$\forall a, a' \in \textbf{Aexp}., a' \prec a \iff a' \text{ ist Teilausdruck von } a$$

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

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

- Beispiel für strukturelle Induktion: Rechtseindeutigkeit von $[\![-]\!]_{\mathcal{A}}$ ($\longrightarrow$ Vorlesung 3)

Arbeitsblatt 4.2: Strukturelle Induktion

- **Beweist**, dass die Relation “Teilausdruck-von” wohlfundiert ist.

Äquivalenz operationale und denotationale Semantik

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

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

- Beweis Prinzip?

Äquivalenz operationale und denotationale Semantik

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

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

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

Beweis: $\forall a \in \mathbf{Aexp}. \forall n \in \mathbb{Z}. \forall \sigma. \langle a, \sigma \rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$

Induktionsanfänge

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

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

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

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

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

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

Beweis: $\forall a \in \mathbf{Aexp}.\, \forall n \in \mathbb{Z}.\, \forall \sigma.\, \langle a, \sigma\rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$

Induktionsschritte

- $a \equiv a_1 + a_2$ — Induktionsannahme: für alle m, n

$$\langle a_1, \sigma\rangle \to_{Aexp} m \iff (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\langle a_2, \sigma\rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

Dann;

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

$$\&$$

$$\langle a_2, \sigma\rangle \to_{Aexp} n \overset{\text{IA für } a_2}{\iff} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

$$\Updownarrow (Def. [\![.]\!]_{\mathcal{A}})$$

$$(\sigma, m + n) \in [\![a_1 + a_2]\!]_{\mathcal{A}}$$

Beweis: $\forall a \in \mathbf{Aexp}.\, \forall n \in \mathbb{Z}.\, \forall \sigma.\, \langle a, \sigma\rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$

Induktionsschritte

- $a \equiv a_1/a_2$ — Induktionsannahme:

$$\langle a_1, \sigma\rangle \to_{Aexp} m \iff (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\langle a_2, \sigma\rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

1. Fall: $n \neq 0$

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

$$\&$$

$$\langle a_2, \sigma\rangle \to_{Aexp} n \overset{\text{IA für } a_2}{\iff} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

$$\Updownarrow (Def.\, [\![.]\!]_{\mathcal{A}})$$

$$(\sigma, m/n) \in [\![a_1/a_2]\!]_{\mathcal{A}}$$

Beweis: $\forall a \in \mathbf{Aexp}.\, \forall n \in \mathbb{Z}.\, \forall \sigma.\, \langle a, \sigma \rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$

Induktionsschritte

- $a \equiv a_1 / a_2$ — Induktionsannahme:

$$\langle a_1, \sigma \rangle \to_{Aexp} m \iff (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$
$$\langle a_2, \sigma \rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

1. Fall: $n = 0$
   Dann gibt es kein v so dass $\langle a_1 / a_2, \sigma \rangle \to_{Aexp} v$, aber auch $\sigma \notin \operatorname{dom} [\![a_1 / a_2]\!]_{\mathcal{A}}$.

q.e.d.

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

Äquivalenz operationale und denotationale Semantik

- Zu zeigen Gleichung (2) von Folie 4:
- Für alle $b \in \mathbf{Bexp}$, für alle $t \in \mathbb{B}$, for alle Zustände σ:

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

- Beweis Prinzip?

Äquivalenz operationale und denotationale Semantik

- Zu zeigen Gleichung (2) von Folie 4:
- Für alle $b \in \mathbf{Bexp}$, für alle $t \in \mathbb{B}$, for alle Zustände σ:

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

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

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

Induktionsanfänge

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

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

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

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

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

Induktionsschritte

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

$$\langle b_1, \sigma \rangle \to_{Bexp} v \iff (\sigma, v) \in [\![b_1]\!]_{\mathcal{B}}$$
$$\langle b_2, \sigma \rangle \to_{Bexp} w \iff (\sigma, w) \in [\![b_2]\!]_{\mathcal{B}}$$

1. Fall $v = \mathit{false}$

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

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

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

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

Induktionsschritte

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

$$\langle b_1, \sigma\rangle \to_{Bexp} v \iff (\sigma, v) \in [\![b_1]\!]_{\mathcal{B}}$$
$$\langle b_2, \sigma\rangle \to_{Bexp} w \iff (\sigma, w) \in [\![b_2]\!]_{\mathcal{B}}$$

1. Fall $v = true$

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

$$\&$$

$$\langle b_2, \sigma\rangle \to_{Bexp} w \overset{\text{IA für } b_2}{\iff} (\sigma, w) \in [\![b_2]\!]_{\mathcal{B}}$$

$$\&$$

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

$$(\sigma, w) \in [\![b_1 \&\& b_2]\!]_{\mathcal{B}}$$

q.e.d.

Arbeitsblatt 4.3: Beweis Induktionsanfang

$\langle a_1 == a_2, \sigma \rangle \to_{Bexp} v \iff (\sigma, v) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$

Beweist obige Aussage unter Verwendung des für arithmetische Ausdrücke geltenden Lemmas

$$\forall a \in \textbf{Aexp}. \forall n \in \mathbb{Z}. \forall \sigma. \langle a, \sigma \rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a]\!]_{\mathcal{A}}$$

1. Was sind die Annahmen?
2. Welche Fälle unterscheiden wir?

Beweis $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} v \iff (\sigma, v) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$

- Annahmen: für $n, m \in \mathbb{B}$:

$$\langle a_1, \sigma\rangle \to_{Aexp} m \iff (\sigma, m) \in [\![a_1]\!]_{\mathcal{B}}$$
$$\langle a_2, \sigma\rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a_2]\!]_{\mathcal{B}}$$

- 1. Fall: $v = true$ $(m = n)$

$$\langle a_1 == a_2, \sigma\rangle \to_{Bexp} true \overset{\text{(Def. } \langle .,. \rangle \to_{Bexp} .)}{\iff} \langle a_1, \sigma\rangle \to_{Bexp} m \overset{\text{Annahme für } a_1}{\iff} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$

$$\&$$

$$\langle a_2, \sigma\rangle \to_{Bexp} m \overset{\text{Annahme für } a_2}{\iff} (\sigma, m) \in [\![a_2]\!]_{\mathcal{A}}$$

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

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

Beweis $\langle a_1 == a_2, \sigma\rangle \to_{Bexp} v \iff (\sigma, v) \in [\![a_1 == a_2]\!]_{\mathcal{B}}$

- Annahmen: für $m, n \in \mathbb{B}$:

$$\langle a_1, \sigma\rangle \to_{Aexp} m \iff (\sigma, m) \in [\![a_1]\!]_{\mathcal{B}}$$
$$\langle a_2, \sigma\rangle \to_{Aexp} n \iff (\sigma, n) \in [\![a_2]\!]_{\mathcal{B}}$$

- 2. Fall: $v = \mathit{false}$ $(m \neq n)$

$$\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \mathit{false} \overset{\text{(Def. } \langle .,. \rangle \to_{Bexp} .)}{\iff} \langle a_1, \sigma\rangle \to_{Aexp} m \overset{\text{Annahme für } a_1}{\iff} (\sigma, m) \in [\![a_1]\!]_{\mathcal{A}}$$

$$\&$$

$$\langle a_2, \sigma\rangle \to_{Aexp} n \overset{\text{Annahme für } a_2}{\iff} (\sigma, n) \in [\![a_2]\!]_{\mathcal{A}}$$

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

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

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

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

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

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

$$\frac{\langle b, \sigma \rangle \to_{Bexp} \mathit{false}}{\langle w, \sigma \rangle \to_{Stmt} \sigma}$$

$fix(\Gamma)$

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

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

- Zu zeigen Gleichung (1) von Folie 4:
- Für alle $c \in \mathbf{Stmt}$, für alle Zustände σ, σ':

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

- $\Longrightarrow$ Beweis Prinzip?
- $\Longleftarrow$ Beweis Prinzip?

Operationale Semantik: C0 Programme

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

Regeln:

$$\frac{}{\langle \{\}, \sigma\rangle \to_{Stmt} \sigma} \qquad \frac{\langle a, \sigma\rangle \to_{Aexp} n \in \mathbb{Z}}{\langle x = a, \sigma\rangle \to_{Stmt} \sigma[x \mapsto n]} \qquad \frac{\langle c_1, \sigma\rangle \to_{Stmt} \sigma' \qquad \langle c_2, \sigma'\rangle \to_{Stmt} \sigma''}{\langle c_1; c_2, \sigma\rangle \to_{Stmt} \sigma''}$$

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

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

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

Operationale Semantik: C0 Programme

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

Regeln: **Programmstruktur**

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

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

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

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

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

Operationale Semantik: C0 Programme

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

Regeln: **Programmstruktur**

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

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

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

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

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

Strukturelle Induktion über c **nicht** möglich.

Ableitungstiefe für Programme

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

$$\frac{\vdots \quad \cdots \quad \vdots}{Prämisse_1 \qquad\qquad Prämisse_n}$$
$$Conclusion$$

Operationale Semantik: C0 Programme

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

Regeln: **Programmstruktur**

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

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

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

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

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

Operationale Semantik: C0 Programme

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

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

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

↘ ↘

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

↘ ↘

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

↘ ↘

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

→ ↘

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

↘ ↘

Äquivalenz operationale und denotationale Semantik

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

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

- $\Longrightarrow$ Beweis Prinzip?
- $\Longleftarrow$ Beweis Prinzip?

Äquivalenz operationale und denotationale Semantik

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

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' \Longleftrightarrow (\sigma, \sigma') \in [\![c]\!]_C$$

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

- $\Longleftarrow$ Beweis Prinzip?

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

Induktionsanfang — Ableitungstiefe 1

- Fall $c \equiv x = a$:

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

Sei $\langle a, \sigma\rangle \to_{Aexp} m \in \mathbb{Z}$:

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

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

Induktionsanfang — Ableitungstiefe 1

- Fall $c \equiv x = a$:

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

Sei $\langle a, \sigma\rangle \to_{Aexp} m \in \mathbb{Z}$:

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

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

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

Induktionsschritt:

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

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

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

$$\langle \textbf{if}(b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma' \overset{\text{(Def. } \langle \cdot, \cdot\rangle \to_{Stmt} \cdot)}{\Longleftrightarrow} \langle b, \sigma\rangle \to_{Bexp} \mathit{true} \overset{\text{Lemma für } b}{\Longleftrightarrow} (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}$$

$$\&$$

$$\langle c_1, \sigma\rangle \to_{Stmt} \sigma' \overset{\text{IH für } c_1}{\Longrightarrow} (\sigma, \sigma') \in [\![c_1]\!]_{\mathcal{C}}$$

$$\&$$

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

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

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

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

$$\langle \textbf{if}(b)\ c_1\ \textbf{else}\ c_2, \sigma\rangle \to_{Stmt} \sigma' \overset{\text{(Def. } \langle ,.\rangle \to_{Stmt} .)}{\Longleftrightarrow} \langle b, \sigma\rangle \to_{Bexp} \mathit{false} \overset{\text{Lemma für } b}{\Longleftrightarrow} (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}$$

$$\&$$

$$\langle c_2, \sigma\rangle \to_{Stmt} \sigma' \overset{\text{IH für } c_2}{\Longrightarrow} (\sigma, \sigma') \in [\![c_2]\!]_{\mathcal{C}} \quad \& \quad (\sigma, \sigma') \in [\![\textbf{if}(b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}}$$

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

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

Induktionsschritt:

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

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

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

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

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

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

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

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

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

Induktionsschritt:

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

$$\langle \textbf{while}(b)\ c, \sigma\rangle \to_{Stmt} \sigma \overset{\text{(Def. } \langle ..,\rangle \to_{Stmt_1})}{\Longleftrightarrow} \langle b, \sigma\rangle \to_{Bexp} \mathit{false} \overset{\text{Lemma für } b}{\Longleftrightarrow} (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}$$

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

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

Äquivalenz operationale und denotationale Semantik

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

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

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

- $\Longleftarrow$ Beweis Prinzip?

Äquivalenz operationale und denotationale Semantik

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

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

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

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

Induktionsanfang:

- Fall $c \equiv x = a$:

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

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

$$\overset{\text{Lemma } \textbf{Aexp}}{\Longrightarrow} \qquad \langle a, \sigma\rangle \rightarrow_{Aexp} t$$

$$\overset{\text{Def. } \langle .,. \rangle \rightarrow_{Stmt}}{\Longrightarrow} \langle x = a, \sigma\rangle \rightarrow_{Stmt} \sigma[x \mapsto t]$$

- Fall $c \equiv \{\,\}$

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

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

$$\overset{\text{Def. } \langle .,. \rangle \rightarrow_{Stmt}}{\Longrightarrow} \langle \{\,\}, \sigma\rangle \rightarrow_{Stmt} \sigma$$

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

Induktionsschritt:

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

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

Induktionsannahme gilt für c_1 und c_2

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

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

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

Induktionsschritt:

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

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

Induktionsannahme gilt für c_1 und c_2

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

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

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

Induktionsschritt:

- Fall **while** (b) c

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

Induktionsannahme gilt für c

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

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

Induktionsschritt:

- Fall **while** (b) c

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

Induktionsannahme gilt für c

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

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

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

Induktionsschritt:

- Fall **while** (b) c

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

Induktionsannahme gilt für c

$$
\begin{aligned}
(\sigma, \sigma') \in [\![\textbf{while } (b) \ c]\!]_{\mathcal{C}} &\Longrightarrow (\sigma, \sigma') \in \mathit{fix}(\Gamma) && \text{nach Def. } [\![.]\!]_{\mathcal{C}} \\
&\Longrightarrow (\sigma, \sigma') \in \bigcup_{i \in \mathbb{N}} \Gamma^i(\emptyset) && \text{nach Def. } \mathit{fix}(\Gamma) \\
&\Longrightarrow (\sigma, \sigma') \in \Gamma^i(\emptyset) \text{ für ein } i \in \mathbb{N} \\
&\Longrightarrow \langle \textbf{while } (b) \ c, \sigma\rangle \rightarrow_{Stmt} \sigma' && \text{nach (UB)}
\end{aligned}
$$

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

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

Es gilt die Induktionsannahme für c:

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

Beweis per Induktion über i:

- Induktionsanfang $i = 0$:

$$(\sigma, \sigma') \in \underbrace{\Gamma^0(\emptyset)}_{\emptyset} \Longrightarrow (\sigma, \sigma') \in \emptyset \Longrightarrow \mathit{false}$$

Implikation trivialerweise erfüllt da $\mathit{false} \Longrightarrow P$ immer wahr

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

Es gilt die Induktionsannahme für c:

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

Beweis per Induktion über i:

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

$$(\sigma, \sigma') \in \Gamma^{i+1}(\emptyset) \Longrightarrow (\sigma, \sigma') \in \Gamma(\Gamma^i(\emptyset))$$

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

- Fallunterscheidung über Zugehörigkeit zur Teilmenge

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

Es gilt die Induktionsannahme für c:

$$\forall \rho, \rho'.(\rho, \rho') \in [\![c]\!]_{\mathcal{C}} \Rightarrow \langle c, \rho\rangle \to_{Stmt} \rho' \qquad (*)$$

Beweis per Induktion über i:

- Induktionsschritt $i \to i+1$:
- Induktionsannahme (UB) gilt für i
- Fall $(\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}$ mit $(\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}}, (\sigma', \sigma'') \in \Gamma^i(\emptyset)$

$$\begin{aligned}(\sigma, \sigma'') \in \Gamma(\Gamma^i(\emptyset)) &\Longrightarrow \underbrace{(\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}}}_{\text{Lemma } \textbf{Bexp}} \wedge \underbrace{(\sigma, \sigma') \in [\![c]\!]_{\mathcal{C}}}_{\text{IA (*)}} \wedge \underbrace{(\sigma', \sigma'') \in \Gamma^i(\emptyset)}_{\text{IA (UB) für } i} \\ &\Longrightarrow \langle b, \sigma\rangle \to_{Bexp} \mathit{true} \wedge \langle c, \sigma\rangle \to_{Stmt} \sigma' \wedge \langle \textbf{while } (b)\ c, \sigma'\rangle \to_{Stmt} \sigma'' \\ &\Longrightarrow \langle \textbf{while } (b)\ c, \sigma\rangle \to_{Stmt} \sigma''\end{aligned}$$

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

Es gilt die Induktionsannahme für c:

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

Beweis per Induktion über i:

- Induktionsschritt $i \rightarrow i + 1$:
- Induktionsannahme (UB) gilt für i
- Fall $(\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}}$

$$\begin{aligned}(\sigma, \sigma') \in \Gamma(\Gamma^i(\emptyset)) &\Longrightarrow (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge \sigma' = \sigma \\ &\Longrightarrow \langle b, \sigma\rangle \rightarrow_{Bexp} \mathit{false} \wedge \sigma' = \sigma \qquad \text{Lemma für } \textbf{Bexp} \\ &\Longrightarrow \langle \textbf{while } (b)\ c, \sigma\rangle \rightarrow_{Stmt} \sigma (= \sigma') \qquad \square\end{aligned}$$

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

Induktionsschritt:

- Fall **while** (b) c

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

Induktionsannahme gilt für c

$$
\begin{aligned}
(\sigma, \sigma') \in [\![\textbf{while } (b) \ c]\!]_{\mathcal{C}} &\Longrightarrow (\sigma, \sigma') \in \mathit{fix}(\Gamma) && \text{nach Def. } [\![.]\!]_{\mathcal{C}} \\
&\Longrightarrow (\sigma, \sigma') \in \bigcup_{i \in \mathbb{N}} \Gamma^i(\emptyset) && \text{nach Def. } \mathit{fix}(\Gamma) \\
&\Longrightarrow (\sigma, \sigma') \in \Gamma^i(\emptyset) \text{ für ein } i \in \mathbb{N} \\
&\Longrightarrow \langle \textbf{while } (b) \ c, \sigma\rangle \rightarrow_{Stmt} \sigma' && \text{nach (UB)}
\end{aligned}
$$

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

Zusammenfassung: Äquivalenz der Semantiken

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

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

- Das ist äquivalent zu (für alle $c \in \textbf{Stmt}$, für alle Zustände σ, σ'):

$$[\![c]\!]_{\mathcal{C}} = \{(\sigma, \sigma') \mid \langle c, \sigma \rangle \to_{Stmt} \sigma'\}$$

- Insbesondere ist die Undefiniertheit gleich:
  wenn es keine Ableitung für c, σ gibt, dann ist auch $\sigma \notin \mathit{Dom}([\![c]\!]_{\mathcal{C}})$.

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Korrekte Software: Grundlagen und Methoden

Vorlesung 5 vom 02.05.24

Die Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

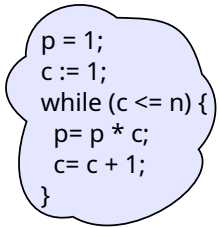
Universität Bremen

Sommersemester 2024

Fahrplan

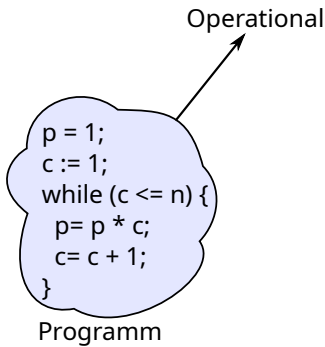
- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Drei Semantiken — Eine Sicht

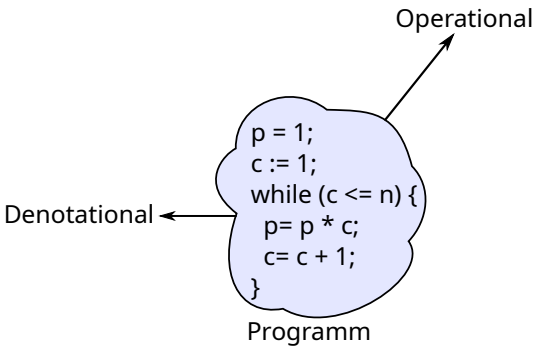


Programm

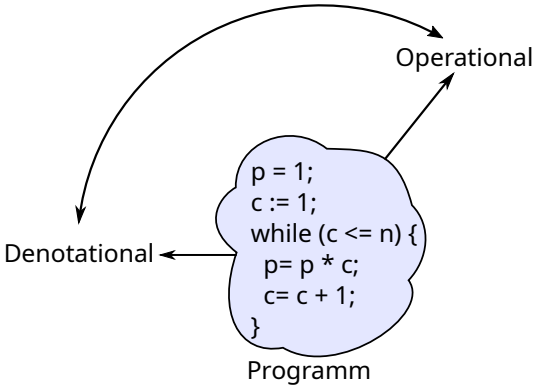
Drei Semantiken — Eine Sicht



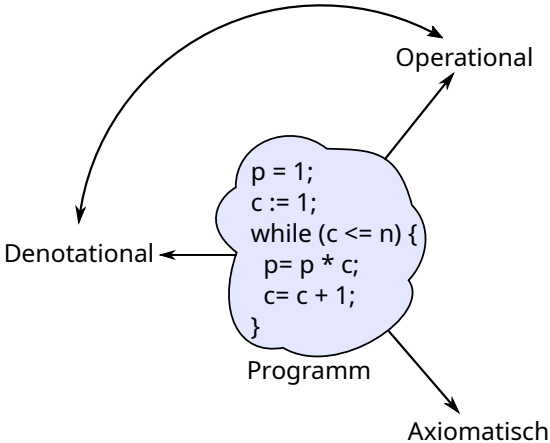
Drei Semantiken — Eine Sicht



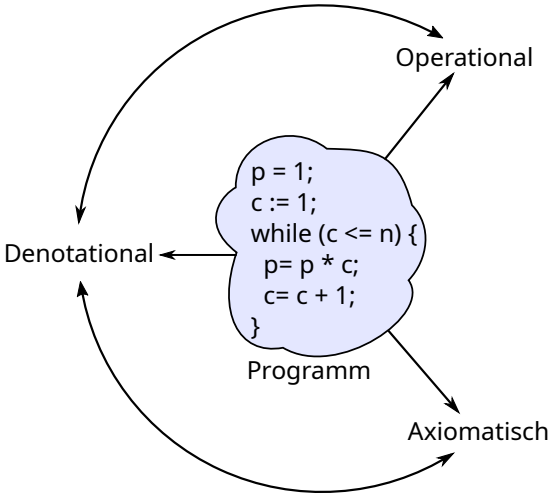
Drei Semantiken — Eine Sicht



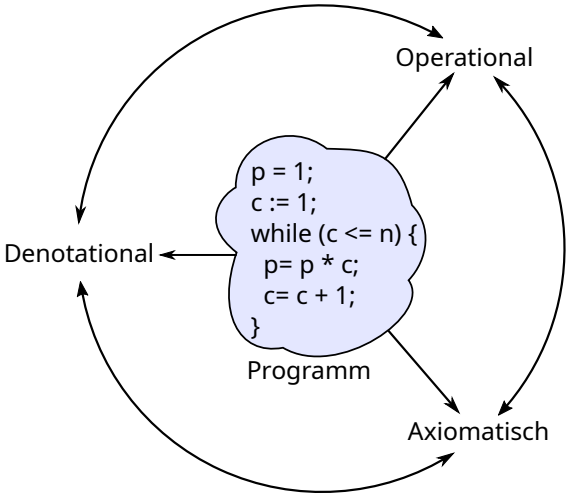
Drei Semantiken — Eine Sicht



Drei Semantiken — Eine Sicht



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**?
- Wir berechnen symbolisch, welche Werte Variablen über den Programmverlauf annehmen.

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

- Operationale/denotionale Semantik nicht für **Korrektheitsbeweise** geeignet: Ausdrücke werden zu groß, skaliert nicht — **Abstraktion** nötig.

Floyd-Hoare-Logik: Idee

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

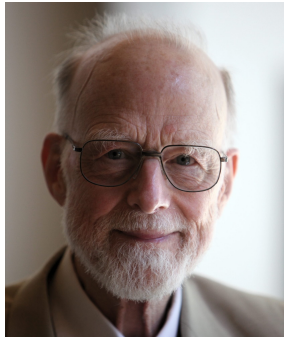
```
p= 1;
c= 1;
while (c <= n) {
  p = p * c;
  c = c + 1;
}
```

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

Bob Floyd und Tony Hoare

Bildquelle: Stanford University

Robert Floyd
1936 – 2001



Bildquelle: Wikipedia

Sir Anthony Charles Richard Hoare
* 1934

Grundbausteine der Floyd-Hoare-Logik

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

- **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 bei (A) der Wert von $n \geq 0$ ist, dann ist bei (E) $p = n!$

Grundbausteine der Floyd-Hoare-Logik

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

- **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 bei (A) der Wert von $n \geq 0$ ist, dann ist bei (E) $p = n!$
- Beobachtung:
  - n ist eine „Eingabevariable“, der Wert am Anfang des Programmes (A) ist relevant;

Grundbausteine der Floyd-Hoare-Logik

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

- **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 bei (A) der Wert von $n \geq 0$ ist, dann ist bei (E) $p = n!$
- Beobachtung:
  - n ist eine „Eingabevariable“, der Wert am Anfang des Programmes (A) ist relevant;
  - p ist eine „Ausgabevariable“, der Wert am Ende des Programmes (E) ist relevant;

Grundbausteine der Floyd-Hoare-Logik

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

- **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 bei (A) der Wert von $n \geq 0$ ist, dann ist bei (E) $p = n!$
- Beobachtung:
  - n ist eine „Eingabevariable“, der Wert am Anfang des Programmes (A) ist relevant;
  - p ist eine „Ausgabevariable“, der Wert am Ende des Programmes (E) ist relevant;
  - c ist eine „Arbeitsvariable“, der Wert am Anfang und Ende ist irrelevant

Arbeitsblatt 5.1: Was berechnet dieses Programm?

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

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

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

Auf dem Weg zur Floyd-Hoare-Logik

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

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

- Der Wert von x wird um 1 erhöht
- Der Wert von x ist hinterher größer als vorher

Auf dem Weg zur Floyd-Hoare-Logik

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

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

  - Der Wert von x wird um 1 erhöht
  - Der Wert von x ist hinterher größer als vorher

- Wir benötigen **zustandsfreie** Aussagen, um von Zuständen unabhängig **vergleichen** zu können.
- Die Logik **abstrahiert** den Effekt von Programmen.

Grundbausteine der Floyd-Hoare-Logik

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

Zusicherungen (Assertions)

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

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

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

Zusicherungen (Assertions)

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

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

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

Denotationale Semantik von Zusicherungen

- Erste Näherung: Funktion

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

  - **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

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

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

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

- Bemerkung: $I : \mathbf{Var} \to \mathbb{Z}$ ist immer eine **totale Funktion** im Gegensatz zu einem Zustand.

Denotat von Aexp

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

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

Denotat von Aexpv

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

Sei $I : \mathbf{Var} \to \mathbb{Z}$ eine beliebige Belegung

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

Erfüllung von Zusicherungen

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

Erfülltheit von Zusicherungen

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

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

Arbeitsblatt 5.2: Zusicherungen

Betrachte folgende Zusicherung:

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

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

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

Unter welchen Belegungen und Zuständen ist a wahr?

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

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

Floyd-Hoare-Tripel

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

$\{P\}\, c\, \{Q\}$ ist **partiell korrekt**, wenn für all Belegungen I und alle Zustände σ, die P erfüllen, gilt: **wenn** die Ausführung von c mit σ in einem Zustand τ terminiert, **dann** erfüllt τ mit Belegung I Q.

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

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

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

$[P]\, c\, [Q]$ ist **total korrekt**, wenn für all Belegungen I und alle Zustande σ, die P erfüllen, die Ausführung von c mit σ in einem Zustand τ terminiert, und τ mit der Belegung I erfüllt Q.

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

Arbeitsblatt 5.3: Gültigkeit

Welche dieser Hoare-Tripel ist semantisch gültig?

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

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

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

Arbeitsblatt 5.3: Gültigkeit

Welche dieser Hoare-Tripel ist semantisch gültig?

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

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

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

Arbeitsblatt 5.3: Gültigkeit

Welche dieser Hoare-Tripel ist semantisch gültig?

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

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

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

Arbeitsblatt 5.3: Gültigkeit

Welche dieser Hoare-Tripel ist semantisch gültig?

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

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

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

Weitere Beispiele

- Folgendes **gilt**:

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

Weitere Beispiele

- Folgendes **gilt**:

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

- Folgendes gilt **nicht**:

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

Weitere Beispiele

- Folgendes **gilt**:

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

- Folgendes gilt **nicht**:

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

- Folgende **gelten**:

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

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

Gültigkeit und Herleitbarkeit

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

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

  - Problem: müssten Semantik von c ausrechnen

Gültigkeit und Herleitbarkeit

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

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

  - Problem: müssten Semantik von c ausrechnen
- **Syntaktische Herleitbarkeit:** $\vdash \{P\}\, c\, \{Q\}$
  - Durch **Regeln** definiert
  - Kann **hergeleitet** werden
  - Muss **korrekt** bezüglich semantischer Gültigkeit gezeigt werden
- Generelles Vorgehen in der Logik

Regeln des Floyd-Hoare-Kalküls

- Der Floyd-Hoare-Kalkül erlaubt es, Zusicherungen der Form $\vdash \{P\}\, c\, \{Q\}$ syntaktisch **herzuleiten**.
- Der **Kalkül** der Logik besteht aus sechs Regeln der Form

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

- Für jedes Konstrukt der Programmiersprache gibt es eine Regel.

Regeln des Floyd-Hoare-Kalküls: Zuweisung

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

- 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 = 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]}
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}
```

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 = x+ 1
//{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 < 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;
```

- 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:

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:

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

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:

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

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:

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

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:

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

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:

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

Vereinfachte Notation für Sequenzen

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

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

Vereinfachte Notation für Sequenzen

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

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

Vereinfachte Notation für Sequenzen

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

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

Arbeitsblatt 5.4: Ein erster Beweis

Betrachte den Rumpf des Fakultätsprogramms:

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

- Welche Zusicherungen gelten

  1. an der Stelle (A)?
  2. an der Stelle (B)?

Arbeitsblatt 5.4: Ein erster Beweis

Betrachte den Rumpf des Fakultätsprogramms:

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

- Welche Zusicherungen gelten

  1. an der Stelle (A)?
  2. an der Stelle (B)?

Arbeitsblatt 5.4: Ein erster Beweis

Betrachte den Rumpf des Fakultätsprogramms:

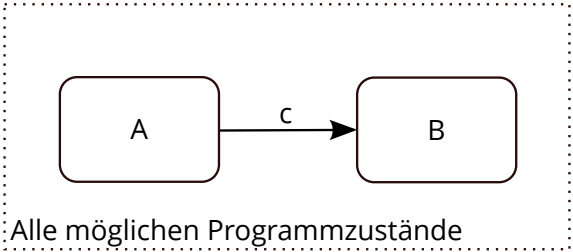
```
// (B)
p= p* c;
// (A)
c= c+ 1;
// {p = (c − 1)!}
```

- Welche Zusicherungen gelten

  1. an der Stelle (A)?
  2. an der Stelle (B)?

Regeln des Floyd-Hoare-Kalküls: Weakening

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

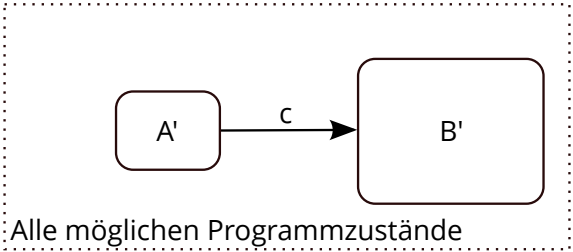


- $\models \{A\}\, c\, \{B\}$: Ausführung von c startet in Zustand, in dem A gilt, und endet (ggf) in Zustand, in dem B gilt.
- Zustandsprädikate beschreiben Mengen von Zuständen:

$$\{\sigma \in \Sigma | \sigma \models^I P\} \subseteq \{\sigma \in \Sigma | \sigma \models^I Q\} \text{ gdw. } P \Longrightarrow Q$$

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:

$$\{\sigma \in \Sigma | \sigma \models^I P\} \subseteq \{\sigma \in \Sigma | \sigma \models^I Q\} \text{ gdw. } P \Longrightarrow Q$$

Arbeitsblatt 5.5: Ein zweiter Beweis

Wir betrachten noch einmal das Vertauschen, diesmal ohne Hilfsvariable:

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

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

Arbeitsblatt 5.5: Ein zweiter Beweis

Wir betrachten noch einmal das Vertauschen, diesmal ohne Hilfsvariable:

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

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

Arbeitsblatt 5.5: Ein zweiter Beweis

Wir betrachten noch einmal das Vertauschen, diesmal ohne Hilfsvariable:

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

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

Arbeitsblatt 5.5: Ein zweiter Beweis

Wir betrachten noch einmal das Vertauschen, diesmal ohne Hilfsvariable:

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

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

Regeln des Floyd-Hoare-Kalküls: Fallunterscheidung

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

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

- Beide Zweige müssen mit derselben Nachbedingung enden.

Arbeitsblatt 5.6: Dreimal ist Bremer Recht

Betrachte folgendes Programm:

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

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

Regeln des Floyd-Hoare-Kalküls: Iteration

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

- Iteration korrespondiert zu **Induktion**.
- Bei wohlfundierter Induktion zeigen wir, dass die **gleiche** Eigenschaft für alle x gilt, $P(x)$, wenn sie für alle kleineren y gilt — d.h. wenn y größer wird muss die Eigenschaft weiterhin gelten.
- 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_2[e/x]}
x= e;
// {P_3}
while (x< n) {
    // {P_3 ∧ x < n}
    // {P_3[a/z]}
    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.
  - Muss genau auf Anweisung passen.
- Implizite Anwendung der Sequenzenregel.
- Weakening wird notiert durch mehrere Zusicherungen, und muss **bewiesen** werden.
  - Im Beispiel:$P \Longrightarrow P_2[e/x]$, $P_2 \Longrightarrow P_3$, $P_3 \wedge x < n \Longrightarrow P_4$, $P_3 \wedge \neg(x < n) \Longrightarrow Q$.

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Zusammenfassung Floyd-Hoare-Logik

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 6 vom 08.05.24

Invarianten im Floyd-Hoare-Logik (und wie wir sie finden)

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Die Floyd-Hoare-Logik bis hierher

- **Hoare-Tripel** $\{P\}\, c\, \{Q\}$ spezifizieren was c berechnet (**Korrektheit**)
  - Semantische **Gültigkeit** von Hoare-Tripeln: $\models \{P\}\, c\, \{Q\}$.
  - Syntaktische **Herleitbarkeit** von Hoare-Tripeln: $\vdash \{P\}\, c\, \{Q\}$
- **Zuweisungen** werden durch **Substitution** modelliert, d.h. die Menge der gültigen Aussagen ändert sich.
- Für Iterationen wird eine **Invariante** benötigt (die **nicht** hergeleitet werden kann).

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Invarianten finden: die Fakultät

Invariante:

```
p= 1;
c= 1;
// {I}
while (c <= n) {
  // {I ∧ c ≤ n}
  p = p * c;
  c = c + 1;
  // {I}
}
// {I ∧ ¬(c ≤ n)}
// {p = n!}
```

Invarianten finden: die Fakultät

```
p= 1;
c= 1;
// {I}
while (c <= n) {
  // {I ∧ c ≤ n}
  p = p * c;
  c = c + 1;
  // {I}
}
// {I ∧ ¬(c ≤ n)}
// {p = n!}
```

Invariante:

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

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

Invarianten finden: die Fakultät

```
p= 1;
c= 1;
// {I}
while (c <= n) {
  // {I ∧ c ≤ n}
  p = p * c;
  c = c + 1;
  // {I}
}
// {I ∧ ¬(c ≤ n)}
// {p = n!}
```

Invariante:

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

- Kern der Invariante: Fakultät bis $c-1$ berechnet.
- Invariante impliziert Nachbedingung $p = n! = (c-1)!$
  - $\neg(c \leq n) \Leftrightarrow c - 1 \geq n$ — was fehlt?

Invarianten finden: die Fakultät

```
p= 1;
c= 1;
// {I}
while (c <= n) {
  // {I ∧ c ≤ n}
  p = p * c;
  c = c + 1;
  // {I}
}
// {I ∧ ¬(c ≤ n)}
// {p = n!}
```

Invariante:

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

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

Invarianten finden

1. Initiale Invariante: momentaner Zustand der Berechnung

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

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

Zählende Schleifen

- Fakultät ist Beispiel für zählende Schleife (**for**).
- Für Nachbedingung $\psi[n]$ ist Invariante:

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

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

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

ist syntaktischer Zucker für

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

Arbeitsblatt 6.1: Summe I

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

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

Annotiert das Programm mit den Korrektheitszusicherungen!

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

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

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

$$x = sum(0, c)$$

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Variante der zählenden Schleife

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

- Was ist hier die Invariante?

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

- Kein C-Idiom
  - Startwert 0 wird ausgelassen

Arbeitsblatt 6.2: Summe II

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

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

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$n! \cdot p = N!$$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

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

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

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

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

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

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

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

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Fakultät Revisited

Dieses Programm berechnet die Fakultat von n:

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

$$
\begin{aligned}
& n! \cdot p = N! \\
\wedge\ & 0 \leq n \\
\wedge\ & n \leq n
\end{aligned}
$$

Arbeitsblatt 6.3: Nicht-zählende Schleife

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

Was ist hier die Invariante?

- Hinweis: es ist ganz einfach.

Beispiel 5: Jetzt wird's kompliziert...

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

- Was berechnet das?

Beispiel 5: Jetzt wird's kompliziert...

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

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

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

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

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;
  //
  }
//
// {?}
```

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;
  //
  }
//
// {i^2 ≤ a ∧ a < (i + 1)^2}
```

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;
  //
  }
// {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i^2 + t ∧ ¬(s ≤ a)}
// {i^2 ≤ a ∧ a < (i + 1)^2}
```

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;
  // {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i^2 + t}
  }
// {s − t ≤ a ∧ t = 2 · i + 1 ∧ s = i^2 + t ∧ ¬(s ≤ a)}
// {i^2 ≤ a ∧ a < (i + 1)^2}
```

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Beispiel 5: Jetzt wird's kompliziert...

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

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 7 vom 22.05.24

Korrektheit des Floyd-Hoare-Kalküls

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

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

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

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

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

Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

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

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

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

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

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

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

Korrektheit des Floyd-Hoare-Kalküls

Der Floyd-Hoare-Kalkül ist korrekt.

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

Beweis:

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

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

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

Korrektheit der Zuweisung

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

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

Korrektheit der Zuweisung

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

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

$\iff \quad \forall I.\, \forall \sigma.\, \sigma \models^I P[e/x] \land \exists \sigma'.\, (\sigma, \sigma') \in [\![x = e]\!]^I_{\mathcal{C}} \implies \sigma' \models^I P$

Korrektheit der Zuweisung

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

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

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

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

with $(\sigma, \sigma([x \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)])) \in [\![x = e]\!]_{\mathcal{C}}$

Korrektheit der Zuweisung

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

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

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

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

$$\text{with } (\sigma, \sigma([x \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)])) \in [\![x = e]\!]_{\mathcal{C}}$$

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

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B \tag{1}$$

$$[\![a[e/x]]\!]^I_{\mathcal{A}}(\sigma) = [\![a]\!]^I_{\mathcal{A}}(\sigma[x \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)]) \tag{2}$$

Arbeitsblatt 7.1: Substitution und Zustands-Update

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B \tag{3}$$

Beweis per strukureller Induktion über B. Zeigt die folgenden Fälle des Beweises:

1. Induktionsanfang: B ist $a_0 = a_1$
2. Induktionsschritt: B ist der Form $B_1 \wedge B_2$

Anmerkung:

- $\sigma \models^I B \Longleftrightarrow [\![B]\!]^I_{\mathcal{B}}(\sigma) = \mathit{true}$

Arbeitsblatt 7.1: Substitution und Zustands-Update

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B \tag{3}$$

Beweis per strukureller Induktion über B. Zeigt die folgenden Fälle des Beweises:

1. Induktionsanfang: B ist $a_0 = a_1$
2. Induktionsschritt: B ist der Form $B_1 \wedge B_2$

Anmerkung:

- $\sigma \models^I B \Longleftrightarrow [\![B]\!]^I_{\mathcal{B}}(\sigma) = \mathit{true}$
- $[\![a[e/y]]\!]^I_{\mathcal{A}}(\sigma) = [\![a]\!]^I_{\mathcal{A}}(\sigma[y \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)])$

Arbeitsblatt 7.1: Substitution und Zustands-Update

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B \tag{3}$$

Beweis per strukureller Induktion über B. Zeigt die folgenden Fälle des Beweises:

1. Induktionsanfang: B ist $a_0 = a_1$
2. Induktionsschritt: B ist der Form $B_1 \wedge B_2$

Anmerkung:

- $\sigma \models^I B \Longleftrightarrow [\![B]\!]^I_{\mathcal{B}}(\sigma) = \mathit{true}$
- $[\![a[e/y]]\!]^I_{\mathcal{A}}(\sigma) = [\![a]\!]^I_{\mathcal{A}}(\sigma[y \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)])$
- $[\![.]\!]_{\mathcal{A}}$ ist strikt

Arbeitsblatt 7.1: Substitution und Zustands-Update

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B \tag{3}$$

Beweis per strukureller Induktion über B. Zeigt die folgenden Fälle des Beweises:

1. Induktionsanfang: B ist $a_0 = a_1$
2. Induktionsschritt: B ist der Form $B_1 \wedge B_2$

Anmerkung:

- $\sigma \models^I B \Longleftrightarrow [\![B]\!]^I_{\mathcal{B}}(\sigma) = \mathit{true}$
- $[\![a[e/y]]\!]^I_{\mathcal{A}}(\sigma) = [\![a]\!]^I_{\mathcal{A}}(\sigma[y \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)])$
- $[\![.]\!]_{\mathcal{A}}$ ist strikt
- Falls für einen Ausdruck a $[\![a]\!]^I_{\mathcal{A}}$ undefiniert ist ($[\![a]\!]^I_{\mathcal{A}} = \bot$), dann ist $\sigma[x \mapsto [\![a]\!]^I_{\mathcal{A}}]$ auch undefiniert.

Arbeitsblatt 7.1: Substitution und Zustands-Update

$$\sigma \models^I B[e/x] \Longleftrightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)] \models^I B \tag{3}$$

Beweis per strukureller Induktion über B. Zeigt die folgenden Fälle des Beweises:

1. Induktionsanfang: B ist $a_0 = a_1$
2. Induktionsschritt: B ist der Form $B_1 \wedge B_2$

Anmerkung:

- $\sigma \models^I B \Longleftrightarrow [\![B]\!]^I_{\mathcal{B}}(\sigma) = \mathit{true}$
- $[\![a[e/y]]\!]^I_{\mathcal{A}}(\sigma) = [\![a]\!]^I_{\mathcal{A}}(\sigma[y \mapsto [\![e]\!]^I_{\mathcal{A}}(\sigma)])$
- $[\![.]\!]_{\mathcal{A}}$ ist strikt
- Falls für einen Ausdruck a $[\![a]\!]^I_{\mathcal{A}}$ undefiniert ist ($[\![a]\!]^I_{\mathcal{A}} = \bot$), dann ist $\sigma[x \mapsto [\![a]\!]^I_{\mathcal{A}}]$ auch undefiniert.
- $[\![.]\!]_{\mathcal{B}}$ ist nicht strikt.

Korrektheit der Sequenzenregel

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

Induktions-Annahmen:

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

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

Zu zeigen:

$$\models \{A\}\, c_1; c_2\, \{C\}$$

Korrektheit der Sequenzenregel

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

Induktions-Annahmen:

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

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

Zu zeigen:

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

Korrektheit der Sequenzenregel

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

Induktions-Annahmen:

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

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

Zu zeigen:

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

$$(\sigma, \sigma') \in [\![c_1; c_2]\!]^I_{\mathcal{C}} \iff (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \circ [\![c_2]\!]^I_{\mathcal{C}}$$

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

Korrektheit der Sequenzenregel

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

Induktions-Annahmen:

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

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

Zu zeigen:

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

$$(\sigma, \sigma') \in [\![c_1; c_2]\!]^I_{\mathcal{C}} \iff (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \circ [\![c_2]\!]^I_{\mathcal{C}}$$

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

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

Korrektheit der Sequenzenregel

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

Induktions-Annahmen:

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

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

Zu zeigen:

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

$$(\sigma, \sigma') \in [\![c_1; c_2]\!]^I_{\mathcal{C}} \iff (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \circ [\![c_2]\!]^I_{\mathcal{C}}$$

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

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

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

Korrektheit der Sequenzenregel

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

Induktions-Annahmen:

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

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

Zu zeigen:

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

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

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

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

Korrektheit der If-Then-Else-Regel

$$\frac{\vdash \{A \wedge b\}\, c_1\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_2\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\, \{B\}}$$

Induktions-Annahmen:

$$\begin{aligned}
(A1)\ \models \{A \wedge b\}\, c_1\, \{B\} &\iff \forall I.\, \forall \sigma.\, \sigma \models^I (A \wedge b) \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
&\iff \forall I.\, \forall \sigma.(\sigma, \mathit{true}) \in [\![A \wedge b]\!]^I_{\mathcal{B}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
(A2)\ \models \{A \wedge \neg b\}\, c_2\, \{B\} &\iff \forall I.\, \forall \sigma.\, \sigma \models^I (A \wedge \neg b) \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_2]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
&\iff \forall I.\, \forall \sigma.\, (\sigma, \mathit{true}) \in [\![(A \wedge \neg b)]\!]^I_{\mathcal{B}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_2]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B
\end{aligned}$$

Zu zeigen:

$$\models \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\, \{B\}$$

Korrektheit der If-Then-Else-Regel

$$\frac{\vdash \{A \wedge b\}\, c_1\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_2\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\, \{B\}}$$

Induktions-Annahmen:

$$\begin{aligned}
(A1)\ \models \{A \wedge b\}\, c_1\, \{B\} &\iff \forall I.\, \forall \sigma.\, \sigma \models^I (A \wedge b) \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
&\iff \forall I.\, \forall \sigma.(\sigma, \mathit{true}) \in [\![A \wedge b]\!]^I_{\mathcal{B}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
(A2)\ \models \{A \wedge \neg b\}\, c_2\, \{B\} &\iff \forall I.\, \forall \sigma.\, \sigma \models^I (A \wedge \neg b) \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_2]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
&\iff \forall I.\, \forall \sigma.\, (\sigma, \mathit{true}) \in [\![(A \wedge \neg b)]\!]^I_{\mathcal{B}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_2]\!]^I_{\mathcal{C}} \Longrightarrow \sigma' \models^I B
\end{aligned}$$

Zu zeigen:

$$\begin{aligned}
&\models \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\, \{B\} \\
\Longleftrightarrow &\forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I B
\end{aligned}$$

Korrektheit der If-Then-Else-Regel

Zu zeigen:

$$\models \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\ \{B\}$$

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

Korrektheit der If-Then-Else-Regel

Zu zeigen:

$$
\begin{aligned}
&\models \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\ \{B\} \\
\Longleftrightarrow &\forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I B \\
\Longleftrightarrow &\forall I.\, \forall \sigma.\, (\sigma, \mathit{true}) \in [\![A]\!]^I_{\mathcal{A}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}} \Longrightarrow \sigma' \models^I B
\end{aligned}
$$

Korrektheit der If-Then-Else-Regel

Zu zeigen:

$$
\begin{aligned}
&\models \{A\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\ \{B\} \\
\iff &\forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}} \implies \sigma' \models^I B \\
\iff &\forall I.\, \forall \sigma.\, (\sigma, \mathit{true}) \in [\![A]\!]^I_{\mathcal{A}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2]\!]_{\mathcal{C}} \implies \sigma' \models^I B
\end{aligned}
$$

Folgt aus Definition

$$
\begin{aligned}
[\![\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2]\!]^I_{\mathcal{C}} = &\{(\sigma, \sigma') \mid (\sigma, \mathit{true}) \in [\![b]\!]^I_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}}\} \\
&\cup \{(\sigma, \sigma') \mid (\sigma, \mathit{false}) \in [\![b]\!]^I_{\mathcal{B}} \wedge (\sigma, \sigma') \in [\![c_2]\!]^I_{\mathcal{C}}\}
\end{aligned}
$$

mit (A1) und (A2)

$$
\begin{aligned}
(A1)\ &\models \{A \wedge b\}\, c_1\, \{B\} \iff \forall I.\, \forall \sigma. (\sigma, \mathit{true}) \in [\![A \wedge b]\!]^I_{\mathcal{B}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_1]\!]^I_{\mathcal{C}} \implies \sigma' \models^I B \\
(A2)\ &\models \{A \wedge \neg b\}\, c_2\, \{B\} \iff \forall I.\, \forall \sigma.\, (\sigma, \mathit{true}) \in [\![(A \wedge \neg b)]\!]^I_{\mathcal{B}} \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![c_2]\!]^I_{\mathcal{C}} \implies \sigma' \models^I B
\end{aligned}
$$

Korrektheit der While-Regel

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

Induktionsannahme:

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

Zu zeigen:

$$\begin{aligned} \models \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\} \iff \quad & \forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in [\![\textbf{while}(b) c]\!]^I_{\mathcal{C}} \\ & \Longrightarrow \sigma' \models^I A \wedge \neg b \\ \iff \quad & \forall I.\, \forall \sigma.\, \sigma \models^I A \wedge \exists \sigma'.\, (\sigma, \sigma') \in \bigcup_{n \geq 0} \Gamma^n(\emptyset) \\ & \Longrightarrow \sigma' \models^I A \wedge \neg b \end{aligned}$$

Beweis per Induktion über n.

Arbeitsblatt 7.2: Unterbeweis Korrektheit der While-Schleife

$$\begin{aligned}\Gamma^0(\emptyset) &:= \emptyset \\ \Gamma^{n+1}(\emptyset) &:= \{(\sigma,\sigma') | [\![b]\!]_{\mathcal{B}}\sigma = \mathit{true} \wedge (\sigma,\sigma') \in [\![c]\!]_{\mathcal{C}} \circ \Gamma^n(\emptyset)\} \\ &\quad \cup \{\sigma,\sigma) | [\![b]\!]_{\mathcal{B}}\sigma = \mathit{false}\}\end{aligned}$$

Gegeben Induktionsannahme aus Regelinduktionsbeweis:

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

Zu zeigen: für alle $n \geq 0$:

$$\forall \sigma, \sigma'.\, \sigma \models^I A \wedge (\sigma,\sigma') \in \Gamma^n(\emptyset) \Longrightarrow \sigma' \models^I A \wedge \neg b$$

Sequenzenregel

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

Induktionsannahme:

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

$$\forall I.\, \forall \sigma.,\, \sigma \models^I (A' \implies A) \iff \forall I.\, \forall \sigma.,\, \sigma \models^I A' \implies \sigma \models^I A$$

$$\forall I.\, \forall \sigma.,\, \sigma \models^I (B \implies B') \iff \forall I.\, \forall \sigma.,\, \sigma \models^I B \implies \sigma \models^I B'$$

Zu zeigen:

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

Vollständigkeit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist vollständig modulo weakening.

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

- Beweis durch Konstruktion einer schwächsten Vorbedingung $\mathrm{wp}(c, Q)$.
  - Problemfall: while-Schleife.

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 $\mathrm{wp}(c_0; c_1, Q) := \mathrm{wp}(c_0, \mathrm{wp}(c_1, Q))$
  - $c \equiv$ **if** b c_0 **else** c_1: Wähle $\mathrm{wp}(c, Q) := (b \wedge \mathrm{wp}(c_0, Q)) \vee (\neg b \wedge \mathrm{wp}(c_1, Q))$
  - $c \equiv$ **while** (b) c_0: ??

Vollständigkeitsbeweis: while

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

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

- Wenn wir eine gültige Zusicherung nicht herleiten können, liegt das nur daran, dass wir eine Beweisverpflichtung nicht beweisen können.
- Logik erster Stufe ist unvollständig, also **können** wir gar nicht besser werden.

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden

Vorlesung 8 vom 29.05.24

Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Organisatorisches

Die Vorlesung am 05.06.2024 muss wegen Abwesenheit der Veranstalter **entfallen**!

Am 06.06. geht es weiter.

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Motivation

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

Arrays

- Beispiele:

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

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

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

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

  - Alle Felder haben **feste Größe**.

Zeichenketten

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

```
char hallo[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-Ausdrücke (l-values), abstrakte Speicheradressen
- Neuer Basisdatentyp **C** für Zeichen
- Erweiterte Grammatik:

$$\begin{aligned}
\textbf{Lexp}\quad & l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \\
\textbf{Aexp}\quad & a ::= \textbf{Z} \mid \textbf{C} \mid \textbf{Lexp} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2 \\
\textbf{Bexp}\quad & b ::= \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,||\, b_2 \\
\textbf{Exp}\quad & e := \textbf{Aexp} \mid \textbf{Bexp}
\end{aligned}$$

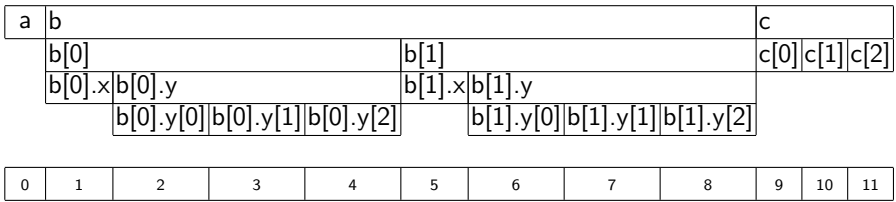
- Keine strukturierten **Werte** (Arrays, Strukturen)
- Semantik: strukturiertes Speichermodell
  - **Idt** $\rightharpoonup$ **V** nicht mehr ausreichend

Speichermodelle I: Compiler

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in konkretes **Speicherlayout**:



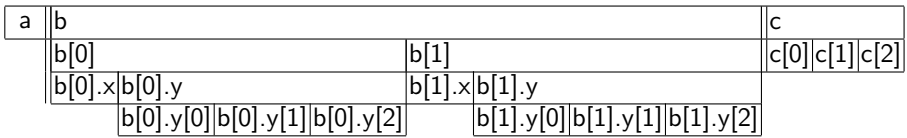
Denotation von $b[0].y[1]$ ist 3

Speichermodelle II: C-Standard

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in abstraktes **Speicherlayout**:



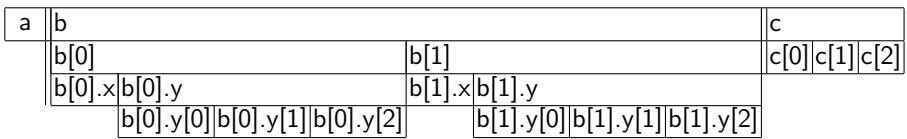
Denotation von $b[0].y[1]$ ist $m + 3$, mit m **unbestimmter** Adresse

Speichermodelle III: Symbolisch

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in **symbolische Adressen**:



Denotation von $b[0].y[1]$ ist $b[0].y[1]$

Speichermodelle im Überblick

- Speichermodell I:
  - Ausführbar, aber **spezifisch** für Compiler und **Architektur**
  - Uneingeschränkte **Zeigerarithmetik**: $\mathbf{Loc} = \mathbb{N}$
- Speichermodell II:
  - Nahe dem **C-Standard**
  - **Eingeschränkte** Form der **Zeigerarithmetik** $p + n$ mit p Zeiger, $n \in \mathbb{N}$
- Speichermodell III:
  - **Symbolische** Adressierung

Werte und Zustände

- Systemzustand bildet **strukturierte** Adressen auf Werte ab.

Systemzustände

- **Basisadressen:** $\mathbf{Addr} \stackrel{def}{=} \mathbb{N}$
- **Locations:** $\mathbf{Loc} \stackrel{def}{=} \mathbf{Addr} \times \mathbb{N}$ (Basisadresse mit Offset)
- Werte: $\mathbf{V} = \mathbb{Z}$ (Einbettung von $\mathbf{C}$ in $\mathbb{Z}$)
- Zustände: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

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

Beispiel

Programm

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

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

Zustand

1. Die Basisadresse von x sei m.
2. Dann:

$$\langle m, 0\rangle \mapsto 1 \qquad \langle m, 23\rangle \mapsto 3$$
$$\langle m, 1\rangle \mapsto 2 \qquad \langle m, 24\rangle \mapsto 4$$
$$\langle m, 2\rangle \mapsto 5 \qquad \langle m, 25\rangle \mapsto 5$$
$$\langle m, 3\rangle \mapsto \text{`}n\text{'} \qquad \langle m, 26\rangle \mapsto \text{`}n\text{'}$$
$$\langle m, 4\rangle \mapsto \text{`}a\text{'} \qquad \langle m, 27\rangle \mapsto \text{`}a\text{'}$$
$$\langle m, 5\rangle \mapsto \text{`}m\text{'} \qquad \langle m, 28\rangle \mapsto \text{`}m\text{'}$$
$$\langle m, 6\rangle \mapsto \text{`}e\text{'} \qquad \langle m, 29\rangle \mapsto \text{`}e\text{'}$$
$$\langle m, 7\rangle \mapsto \text{`}1\text{'} \qquad \langle m, 30\rangle \mapsto \text{`}2\text{'}$$
$$\langle m, 8\rangle \mapsto \text{`}\backslash 0\text{'} \qquad \langle m, 31\rangle \mapsto \text{`}\backslash 0\text{'}$$

Umgebung und Typen

- Unsere Sprache kennt folgende **Typen**:

$$\begin{aligned}\textbf{Type} &::= \textbf{char} \mid \textbf{int} \mid \textbf{Struct} \mid \textbf{Array}\\ \textbf{Struct} &::= \textbf{struct}\ \ \textbf{Idt}^{?}\ \{(\textbf{Type}\ \textbf{Idt})^{+}\}\\ \textbf{Array} &::= \textbf{Type}\ \textbf{Idt}[\textbf{Aexp}]\end{aligned}$$

- Die Umgebung Γ ordnet **Bezeichnern** $x \in \textbf{Idt}$ einen **Typ** und eine **Basisadresse** zu.
- $\Gamma \vdash e : t$ Ausdruck e hat Typ t im Kontext Γ
- Semantik benötigt Γ als **zusätzlichen** (statischen!) Parameter

Größen und Offsets

- **Größe** eines Typen (vgl. **sizeof** in C, aber vereinfacht):

$$sizeof(\textbf{char}) \stackrel{def}{=} 1$$

$$sizeof(\textbf{int}) \stackrel{def}{=} 1$$

$$sizeof(\textbf{struct}\ \ s\ \{t_1 f_1; \ldots t_n f_n\}) \stackrel{def}{=} \sum_{i=1}^{n} sizeof(t_i)$$

$$sizeof(t\ \ i[n]) \stackrel{def}{=} sizeof(t) \cdot n$$

- Offset $off_t(a)$ eines Feldes $a \in$ **Idt** für einen Strukturtyp t:

$$off_{\textbf{struct}\ \ i\ \{t_1\ \ f_1;\ldots;t_n\ \ f_n\}}(f_k) \stackrel{def}{=} \sum_{i=1}^{k-1} sizeof(t_i) \text{für } 1 \leq k \leq n$$

Operationale Semantik: L-Ausdrücke

- $m \in \textbf{Lexp}$ wertet zu $l \in \textbf{Loc}$ aus: $\langle m, \sigma \rangle \to^{\Gamma}_{Lexp} l$

$$\frac{x \in \textbf{Idt}, x \in \Gamma}{\langle x, \sigma \rangle \to^{\Gamma}_{Lexp} \langle \Gamma(x), 0 \rangle}$$

$$\frac{\Gamma \vdash m : t[\,] \qquad \langle m, \sigma \rangle \to^{\Gamma}_{Lexp} l \qquad \langle e, \sigma \rangle \to^{\Gamma}_{Aexp} n}{\langle m[e], \sigma \rangle \to^{\Gamma}_{Lexp} l + n \cdot sizeof(t)}$$

$$\frac{\Gamma \vdash m : t \qquad \langle \sigma, m \rangle \to^{\Gamma}_{Lexp} l}{\langle m.a, \sigma \rangle \to^{\Gamma}_{Lexp} l + off_t(a)}$$

- Notation: für $a = \langle b, m \rangle \in \textbf{Loc}$, $n \in \mathbb{N}$, $a + n \stackrel{def}{=} \langle b, m + n \rangle$

Operationale Semantik: Ausdrücke und Zuweisungen

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

$$\frac{\langle m, \sigma\rangle \to^{\Gamma}_{Lexp} l \qquad m \in Dom(\sigma)}{\langle m, \sigma\rangle \to^{\Gamma}_{Aexp} \sigma(l)}$$

- Zuweisung:

$$\frac{\langle m, \sigma\rangle \to^{\Gamma}_{Lexp} l \qquad \langle e, \sigma\rangle \to^{\Gamma}_{Aexp} v}{\langle m = e, \sigma\rangle \to^{\Gamma}_{Stmt} \sigma[l \mapsto v]}$$

- Die restlichen Regeln bleiben (mit Γ garnieren)

Denotationale Semantik

- Denotation für **Lexp**:

$$[\![-]\!]_{\mathcal{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$[\![x]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, \langle \Gamma(x), 0 \rangle) \mid x \in \Gamma\}$$
$$[\![m[e]]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + n \cdot \mathit{sizeof}(t)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, (\sigma, n) \in [\![e]\!]^{\Gamma}_{\mathcal{A}}, \Gamma \vdash m : t[x]\}$$
$$[\![m.a]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + \mathit{off}_t(a)) \mid \Gamma \vdash m : t, (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}\}$$

- Denotation für **Ausdrücke** ($m \in \mathbf{Lexp}$):

$$[\![m]\!]^{\Gamma}_{\mathcal{A}} = \{(\sigma, \sigma(l)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, l \in \mathit{Dom}(\sigma)\}$$

- Denotation für **Zuweisungen**:

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

Arbeitsblatt 8.1: Semantik

Gegeben folgende Deklaration:

```
struct p {
  int a[5];
  struct q {
    int a;
    int b;
    } b[5];
  int c[2];
  } x;
```

Für $\Gamma \stackrel{def}{=} \langle x \mapsto l \rangle, \sigma = \emptyset$
berechne die Semantik von folgendem Programmfragment:

```
x.a[2]= 7;
x.c[1]= 9;
x.b[3].b= 17;
```

Berechne dazu die Größen von p und q und die Offsets von a, b, c.

Links oder Rechts?

- In beiden Semantiken hat ein **Lexp** unterschiedliche Semantik auf der **linken** oder **rechten** Seite einer Zuweisung.
  - x = x+1 — Links: Lokation, rechts: Wert (im Zustand an dieser Stelle)
- Lösung in C: "Except when it is (...) the operand of the unary & operator, 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

Floyd-Hoare-Kalkül

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

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

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

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

    Anmerkung: l und e enthalten **keine** logischen Variablen.
  - Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
  - Problem: Feldzugriffe

Von der Substitution zur Ersetzung

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

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

$$n[e/l] \stackrel{def}{=} n \qquad (n \in \mathbf{Z} \uplus \mathbf{C})$$
$$m[e/l] \stackrel{def}{=} \begin{cases} e & \text{falls } l \cong m \\ m & \text{sonst} \end{cases} \qquad (m \in \mathbf{Lexp})$$
$$(a_1 + a_2)[e/l] \stackrel{def}{=} a_1[e/l] + a_2[e/l]$$
$$\ldots$$

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

Von der Substitution zur Ersetzung

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

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

$$n[e/l] \stackrel{def}{=} n \qquad (n \in \mathbf{Z} \uplus \mathbf{C})$$
$$m[e/l] \stackrel{def}{=} \begin{cases} e & \text{falls } l \cong m \\ \mathrm{sub}(m, l, e) & \text{sonst} \end{cases} \qquad (m \in \mathbf{Lexp})$$
$$(a_1 + a_2)[e/l] \stackrel{def}{=} a_1[e/l] + a_2[e/l]$$
$$\ldots$$
$$\mathrm{sub}(x, m, e) \stackrel{def}{=} x$$
$$\mathrm{sub}(l[i], m, e) \stackrel{def}{=} l[i[e/m]]$$
$$\mathrm{sub}(l.a, m, e) \stackrel{def}{=} (\mathrm{sub}(l, m, e)).a$$

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

Gleichheit von L-Ausdrücken

- Das Substitutionslemma muss gelten:

$$[\![P[e/m]]\!]^{\Gamma}_{\mathcal{B}v}(\sigma) = [\![P]\!]^{\Gamma}_{\mathcal{B}v}(\sigma[[\![m]\!]^{\Gamma}_{\mathcal{L}} \mapsto [\![e]\!]^{\Gamma}_{\mathcal{A}v}(\sigma)])$$
$$[\![a[e/m]]\!]^{\Gamma}_{\mathcal{A}v}(\sigma) = [\![a]\!]^{\Gamma}_{\mathcal{A}v}(\sigma[[\![m]\!]^{\Gamma}_{\mathcal{L}} \mapsto [\![e]\!]^{\Gamma}_{\mathcal{A}v}(\sigma)])$$

- $l \cong m$ gdw. $[\![l]\!]^{\Gamma}_{\mathcal{L}} = [\![m]\!]^{\Gamma}_{\mathcal{L}}$

$$\frac{x \in \mathbf{Idt}}{x \cong x} \qquad \frac{l \cong m}{l.a \cong m.a} \qquad \frac{l \cong m, [\![i]\!]^{\Gamma}_{\mathcal{A}} = [\![j]\!]^{\Gamma}_{\mathcal{A}}}{l[i] \cong m[j]}$$

- Gleichheit von L-Ausdrücken reduziert zu Gleichheit von Feldindizes

Ersetzung leicht gemacht

Wie ersetzen wir in $P[e/l]$?

- Wenn l ein Identifier $x \in$ **Idt** ist, wie gewohnt substituieren
- Wenn $l = a[s]$ und P in der Form $a[t]$ in $L(a[t])$ (wobei L keine Quantoren enthält)
  1. Wenn s und t **beide** in $\mathbb{Z}$ oder **Idt**, ersetze $L(a[t])$ durch $L(e)$, falls $s = t$.
  2. Wenn s oder t nicht aus $\mathbb{Z}$ oder **Idt**, ersetze $L(a[t])$ durch $(t = s \wedge L(e)) \vee (t \neq s \wedge L(a[t]))$

  (2) ist immer möglich, (1) eine Optimierung
- Ansonsten $\forall i.\, P[e/i] = \forall i.\, (P[e/l])$ etc.
- Das ist jetzt immer noch nicht die ganz allgemeine Form (was ist mit geschachtelten Indizes $a[i][j]$?), aber für unsere Belange reicht das.

Arbeitsblatt 8.2: Ersetzungen

Berechnet folgende Substitutionen (mit $P \stackrel{def}{=} \forall i.\, 0 \leq i \longrightarrow b.x[i].y < 15$):

1. $P[5/a.x[j].y] = ?$
2. $P[3/x[7].y] = ?$
3. $P[9/b.x[3].y] = ?$
4. $(c[7 + b.x[i].i].y \leq i)[99/i] = ?$
5. $(\forall i.\, 0 \leq i \longrightarrow c[7 + b.x[i].y].a = i)[17/b.x[j].y] = ?$

Beispiel

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

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

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}
```

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

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}
```

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

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}
```

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

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}
```

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

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}
```

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

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}
```

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

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}
```

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

Beispiel: Problem

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

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

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 ∧ −1 = 7) ∨ (i ≠ 1 ∧ a[1] = 7)}
a[i] = −1;
// {a[1] = 7}
```

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

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 ∧ −1 = 7) ∨ (i ≠ 1 ∧ a[1] = 7)}
a[i] = −1;
// {a[1] = 7}
```

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

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 ∧ −1 = 7) ∨ (i ≠ 1 ∧ a[1] = 7)}
a[i] = −1;
// {a[1] = 7}
```

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

Beispiel: Problem

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

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

Beispiel: Problem

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

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

Arbeitsblatt 8.3: 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[2] * a[0];
// {a[2] = 3 * n}
```

Erstes Beispiel: Ein Feld initialisieren

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

Erstes Beispiel: Ein Feld initialisieren

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

Erstes Beispiel: Ein Feld initialisieren

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

Erstes Beispiel: Ein Feld initialisieren

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

Erstes Beispiel: Ein Feld initialisieren

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

Erstes Beispiel: Ein Feld initialisieren

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

- Wichtiges Theorem:

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

Erstes Beispiel: Ein Feld initialisieren

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

- Wichtiges Theorem:

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

Erstes Beispiel: Ein Feld initialisieren

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

- Wichtiges Theorem:

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Beispiel: Suche nach dem maximalen Element

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

Vorgehensweise

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

Vorgehensweise

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

1. Finde/rate/formuliere Invariante I

Vorgehensweise

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

1. Finde/rate/formuliere Invariante I
2. Beweise $(I \wedge \neg b) \longrightarrow \Phi$

Vorgehensweise

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

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

Vorgehensweise

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

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

Längeres Beispiel: Suche nach einem Null-Element

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

Längeres Beispiel: Suche nach einem Null-Element

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

Merkt euch folgende korrekten logischen Umformungen:

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

Längeres Beispiel: Suche nach einem Null-Element

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

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    //
10   if (a[i] == 0) {
11     //
12     //
13     //
14     //
15     //
16     r= i;
17     //
18     }
19   else {
20     //
21     //
22   }
23   //
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 //
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    //
10   if (a[i] == 0) {
11     //
12     //
13     //
14     //
15     //
16     r= i;
17     //
18     }
19   else {
20     //
21     //
22   }
23   //
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    //
10   if (a[i] == 0) {
11     //
12     //
13     //
14     //
15     //
16     r= i;
17     //
18     }
19   else {
20     //
21     //
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    //
10   if (a[i] == 0) {
11     //
12     //
13     //
14     //
15     //
16     r= i;
17     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18     }
19   else {
20     //
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    //
10   if (a[i] == 0) {
11     //
12     //
13     //
14     //
15     //
16     r= i;
17     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18     }
19   else {
20     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10   if (a[i] == 0) {
11     //
12     //
13     //
14     //
15     //
16     r= i;
17     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18     }
19   else {
20     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10   if (a[i] == 0) {
11     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
12     //
13     //
14     //
15     //
16     r= i;
17     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18     }
19   else {
20     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  //
3  i= 0;
4  //
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10   if (a[i] == 0) {
11     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
12     //
13     //
14     //
15     // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                 A(i)              B(i)                C
16       r= i;
17       // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18       }
19   else {
20     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1   // {0 ≤ n}
2   //
3   i= 0;
4   //
5   r= −1;
6   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7   while (i< n) {
8     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10    if (a[i] == 0) {
11      // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                                          B(i)∧C
12      // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
13      //
14      //
15      // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                A(i)          B(i)                    C
16        r= i;
17        // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18        }
19    else {
20      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22    }
23    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24    i= i+1;
25    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26  }
27  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1   // {0 ≤ n}
2   //
3   i= 0;
4   //
5   r= −1;
6   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7   while (i< n) {
8     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10    if (a[i] == 0) {
11      // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                                  B(i)∧C
12      // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
            ¬A(i)          C                B(i)          C
13      // {(i = −1 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0) ∨ (0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0)}
14      //
15      // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
            A(i)          B(i)          C
16        r= i;
17        // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18        }
19    else {
20      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22    }
23    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24    i= i+1;
25    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1   // {0 ≤ n}
2   //
3   i= 0;
4   //
5   r= −1;
6   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7   while (i< n) {
8     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10    if (a[i] == 0) {
11      // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                                  B(i)∧C
12      // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
          ¬A(i)        C                B(i)              C
13      // {(i = −1 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0) ∨ (0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0)}
14      // {(i = −1 ∨ (0 ≤ i < i + 1 ∧ a[i] = 0)) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
15      // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
          A(i)              B(i)                C
16      r= i;
17      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18      }
19    else {
20      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22    }
23    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24    i= i+1;
25    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1   // {0 ≤ n}
2   //
3   i= 0;
4   // {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
5   r= −1;
6   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7   while (i< n) {
8     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10    if (a[i] == 0) {
11      // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
12      // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
13      // {(i = −1 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0) ∨ (0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0)}
14      //
15      // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
16      r= i;
17      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18      }
19    else {
20      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21      // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22    }
23    // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24    i= i+1;
25    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  // {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
3  i= 0;
4  // {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10   if (a[i] == 0) {
11     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                                  B(i)∧C
12     // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
          ¬A(i)          C                B(i)          C
13     // {(i = −1 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0) ∨ (0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0)}
14     //
15     // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
          A(i)          B(i)          C
16       r= i;
17       // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18       }
19     else {
20       // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21       // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22     }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
```

Benutzte Logische Umformungen

- Zeilen 11-12:
  - $[D \wedge C] \Rightarrow [C]$ und
  - Erweiterung von C auf $B(i) \wedge C$, weil $C \vdash B(i)$ gilt.
- $[\varphi] \Rightarrow [\psi \vee \varphi]$ in der Form

$$[(B(i) \wedge C)] \Rightarrow [(\neg A(i) \wedge C) \vee (B(i) \wedge C))]$$

- DeMorgan:

$$[(\neg A(i) \wedge C) \vee (B(i) \wedge C))] \Rightarrow [(\neg A(i) \vee B(i)) \wedge C]$$

- Klassische Implikation:

$$[\neg U \vee V] \Leftrightarrow [U \Rightarrow V]$$

Längeres Beispiel: Suche nach einem Null-Element

```
10  /** { 0 ≤ n } */
11  /** { 0 ≤ 0 ≤ n } */
12  i= 0;
13  /** { 0 ≤ i ≤ n } */
14  /** { (−1 ≠ −1 ⟶ 0 ≤ −1< i ∧ a[−1] == 0) ∧ 0 ≤ i ≤ n } */
15  r= −1;
16  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
17  while (i< n) {
18    /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i < n } */
19    /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
20    if (a[i] == 0) {
21      /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
22      /** { 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
23      /** { (i ≠ −1 ⟶ 0 ≤ i< i+1 ∧ a[i] == 0) ∧ 0≤ i+1≤ n } */
24      r= i;
25      /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0≤ i+1≤ n } */
26      }
27    else {
28       /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] ≠ 0 } */
29       /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
30      }
31    /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
32    i= i+1;
33    /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
34    }
35  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n) } */
36  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n } */
37  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ i == n } */
38  /** { r ≠ −1 ⟶ 0 ≤ r< n ∧ a[r] == 0 } */
```

Zusammenfassung

- Strukturierte Datentypen (Felder und Structs) erfordern strukturierte Adressen
- Abstraktion über „echtem“ Speichermodell
- Änderungen in der Semantik und im Floyd-Hoare-Kalkül überschaubar
- ... aber mit erheblichen Konsequenzen:
  - Substitution wird zur Ersetzung
  - Anwendung der Zuweisungsregel führt i.A. zu großen Formeln

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Korrekte Software: Grundlagen und Methoden

Vorlesung 9 vom 5/12.06.24

Verifikationsbedingungen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Idee

- Hier ist ein einfaches Programm:

```
//
z = y;
//
y = x;
//
x = z;
// {x = Y ∧ y = X}
```

Idee

- Hier ist ein einfaches Programm:

```
//
z = y;
//
y = x;
// {z = Y ∧ y = X}
x = z;
// {x = Y ∧ y = X}
```

Idee

- Hier ist ein einfaches Programm:

```
//
z = y;
// {z = Y ∧ x = X}
y = x;
// {z = Y ∧ y = X}
x = z;
// {x = Y ∧ y = X}
```

Idee

- Hier ist ein einfaches Programm:

```
// {y = Y ∧ x = X}
z = y;
// {z = Y ∧ x = X}
y = x;
// {z = Y ∧ y = X}
x = z;
// {x = Y ∧ y = X}
```

Idee

- Hier ist ein einfaches Programm:

```
// {y = Y ∧ x = X}
z = y;
// {z = Y ∧ x = X}
y = x;
// {z = Y ∧ y = X}
x = z;
// {x = Y ∧ y = X}
```

- Wir sehen:
  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Vorbedingung kann **berechnet** werden.

Idee

- Hier ist ein einfaches Programm:

```
// {y = Y ∧ x = X}
z = y;
// {z = Y ∧ x = X}
y = x;
// {z = Y ∧ y = X}
x = z;
// {x = Y ∧ y = X}
```

- Wir sehen:
  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Vorbedingung kann **berechnet** werden.
- Geht das immer?

Idee

- Hier ist ein einfaches Programm:

```
// {y = Y ∧ x = X}
z = y;
// {z = Y ∧ x = X}
y = x;
// {z = Y ∧ y = X}
x = z;
// {x = Y ∧ y = X}
```

- Wir sehen:
  1. Die Verifikation erfolgt **rückwärts** (von hinten nach vorne).
  2. Die Vorbedingung kann **berechnet** werden.
- Geht das immer?
- Was bringt uns das?

Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — P passt auf jede beliebige Nachbedingung

$$\frac{}{\vdash \{P[e/l]\}\ l = e\ \{P\}}$$

Rückwärtsanwendung der Regeln

- Zuweisungsregel kann **rückwärts** angewandt werden, weil die Nachbedingung eine offene Variable ist — P passt auf jede beliebige Nachbedingung

$$\frac{}{\vdash \{P[e/l]\}\, l = e\, \{P\}}$$

- Was ist mit den anderen Regeln?

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c_1\, \{B\} \qquad \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}\ (b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Arbeitsblatt 9.1: Eine Kleine Fallunterscheidung

Berechnet die Vorbedingungen an den Stellen (A), (B), (?) für folgendes Programm:

```
// (?)
if (y == 7) {
  // (A)
  x= 3;
  //
  }
else {
  // (B)
  y= 0;
  //
  x= 10;
  //
  }
// {x + y == 10}
```

Rückwärtsanwendung: if

```
//
if (b) {
  //
  ...
  // {Q}
  }
else {
  //
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Rückwärtsanwendung: if

```
//
if (b) {
  //
  ...
  // {Q}
  }
else {
  // {P2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Rückwärtsanwendung: if

```
//
if (b) {
  // {P1}
  ...
  // {Q}
  }
else {
  // {P2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Rückwärtsanwendung: if

```
// ?
if (b) {
  // {P1}
  ...
  // {Q}
  }
else {
  // {P2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Regel in der Form nicht geeignet. Besser:

$$A \stackrel{def}{=} (P_1 \wedge b) \vee (P_2 \wedge \neg b)$$

$$A \wedge b \Longrightarrow P_1 \tag{1}$$

$$A \wedge \neg b \Longrightarrow P_2 \tag{2}$$

Kombiniert mit Weakening ergibt neue Regel:

$$\frac{\dfrac{A \wedge b \Longrightarrow P_1 \quad \vdash \{P_1\}\, c_0\, \{B\}}{\vdash \{A \wedge b\}\, c_0\, \{B\}} \qquad \dfrac{A \wedge \neg b \Longrightarrow P_2 \quad \vdash \{P_2\}\, c_1\, \{B\}}{\vdash \{A \wedge \neg b\}\, c_1\, \{B\}}}{\vdash \{\underbrace{(P_1 \wedge b) \vee (P_2 \wedge \neg b)}_{A}\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Rückwärtsanwendung: if

```
//
if (b) {
  // {P1}
  ...
  // {Q}
  }
else {
  // {P2}
  ...
  // {Q}
  }
// {Q}
```

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Regel in der Form nicht geeignet. Besser:

$$A \stackrel{def}{=} (P_1 \wedge b) \vee (P_2 \wedge \neg b)$$

$$A \wedge b \Longrightarrow P_1 \tag{1}$$

$$A \wedge \neg b \Longrightarrow P_2 \tag{2}$$

Kombiniert mit Weakening ergibt neue Regel:

$$\frac{\vdash \{P_1\}\, c_0\, \{B\} \quad \vdash \{P_2\}\, c_1\, \{B\}}{\vdash \{(P_1 \wedge b) \vee (P_2 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

Arbeitsblatt 9.2: Etwas Logik

Beweist Lemma (1) und (2) von der vorherigen Folie:

$$A = (P_1 \wedge b) \vee (P_2 \wedge \neg b)$$

$$A \wedge b \Longrightarrow P_1 \tag{1}$$

$$A \wedge \neg b \Longrightarrow P_2 \tag{2}$$

Hinweis:

- Nutzt logische Umformungen: Distributivität, Idempotenz, ...

Neue Regeln

- Wir können aus dem Hoare-Kalkül **neue Regeln** ableiten, in dem wir
  1. Existierende Regeln **instantiieren**, oder
  2. existierende Regeln **verknüpfen**.
- Wir benötigen das hier, um die Regeln des Hoare-Kalkül in eine Form zu bringen, welche die Rückwärtsrechnung ermöglicht.

Das Hinzufügen abgeleiteter Regeln ist eine **konservative Erweiterung** — es lassen sich damit nicht mehr oder weniger Hoare-Tripel $\vdash \{P\}\, c\, \{Q\}$ herleiten.

Regeln für die Rückwärtsrechnung

1. **Nachbedingung** der **Konklusion** ist von der Form $\{Q\}$ (**offene** Meta-Variable)
2. Alle **Vorbedingungen** der **Prämissen** ist von der Form $\{P_i\}$ (**unterschiedliche** P_i)
3. Alle Variablen in den Vorbedingungen der Konklusion, den Weakenings und Nachbedingungen der Prämisse sind **determiniert**[1].

Welche Regeln passen noch nicht?

[1] **Entweder** in der Nachbedingung oder dem Programmausdruck der Konklusion, **oder** den Vorbedingungen der Prämisse enthalten.

Regeln für die Rückwärtsrechnung

1. **Nachbedingung** der **Konklusion** ist von der Form $\{Q\}$ (**offene** Meta-Variable)
2. Alle **Vorbedingungen** der **Prämissen** ist von der Form $\{P_i\}$ (**unterschiedliche** P_i)
3. Alle Variablen in den Vorbedingungen der Konklusion, den Weakenings und Nachbedingungen der Prämisse sind **determiniert**[1].

Welche Regeln passen noch nicht? **while**-Regel passt noch nicht ...

[1] **Entweder** in der Nachbedingung oder dem Programmausdruck der Konklusion, **oder** den Vorbedingungen der Prämisse enthalten.

Regeln für die Rückwärtsrechnung: while

- **while**-Regel (3) wird mit Weakening zu (4):

$$\frac{\vdash \{I \wedge b\}\, c\, \{I\}}{\vdash \{I\}\ \textbf{while}\ (b)\ c\, \{I \wedge \neg b\}} \tag{3}$$

$$\frac{I \wedge b \Longrightarrow R \qquad \vdash \{R\}\, c\, \{I\} \qquad I \wedge \neg b \Longrightarrow Q}{\vdash \{I\}\ \textbf{while}\ (b)\ c\, \{Q\}} \tag{4}$$

- Implikationen $I \wedge b \Longrightarrow R$, $I \wedge \neg b \Longrightarrow Q$ werden zu **Beweisverpflichtungen**
- Bedingung I (**Invariante**) muss **vorgegeben** werden.
  - Durch **Annotation**: **while** (b) /** **inv** I */ c

Übersicht: Regeln für den Hoare-Kalkül Rückwärts

$$\frac{}{\vdash \{P[e/l]\}\, l = e\, \{P\}} \qquad \frac{}{\vdash \{A\}\, \{\,\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c_1\, \{B\} \vdash \{B\}\, c_2\, \{C\}}{\vdash \{A\}\, c_1; c_2\, \{C\}}$$

$$\frac{\vdash \{A_0\}\, c_0\, \{B\} \quad \vdash \{A_1\}\, c_1\, \{B\}}{\vdash \{(A_0 \wedge b) \vee (A_1 \wedge \neg b)\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{I \wedge b \Longrightarrow B \quad \vdash \{B\}\, c\, \{I\} \quad I \wedge \neg b \Longrightarrow C}{\vdash \{I\}\ \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ I\ {*}/\ c\, \{C\}}$$

Der Hoare-Kalkül Rückwärts

- Mit diesen Regeln können wir für ein gegebenes Programm c und eine Nachbedingung Q eine Vorbedingung P_{pre} **berechnen**.
- Was bringt uns das?

Der Hoare-Kalkül Rückwärts

- Mit diesen Regeln können wir für ein gegebenes Programm c und eine Nachbedingung Q eine Vorbedingung P_{pre} **berechnen**.
- Was bringt uns das?
  - Um ein **Hoare-Tripel** $\vdash \{P\}\, c\, \{Q\}$ zu **beweisen**, berechnen wir wie oben die **Vorbedingung** P_{pre}.
  - Jetzt müssen wir nur noch $P \Longrightarrow P_{\text{pre}}$ zeigen.
  - Damit **reduzieren** wir die **Korrektheit** auf eine Menge von (zustandsfreien) **Beweisverpflichtungen**.

Der Hoare-Kalkül Rückwärts

- Mit diesen Regeln können wir für ein gegebenes Programm c und eine Nachbedingung Q eine Vorbedingung P_{pre} **berechnen**.
- Was bringt uns das?
  - Um ein **Hoare-Tripel** $\vdash \{P\}\, c\, \{Q\}$ zu **beweisen**, berechnen wir wie oben die **Vorbedingung** P_{pre}.
  - Jetzt müssen wir nur noch $P \Longrightarrow P_{\text{pre}}$ und alle Weakenings aus der Berechnung von P_{pre} zeigen.
  - Damit **reduzieren** wir die **Korrektheit** auf eine Menge von (zustandsfreien) **Beweisverpflichtungen**.

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?

Annotierte Programme

- Wir helfen dem Rechner weiter und **annotieren** die Schleifeninvariante I am Programm.
- Damit berechnen wir:
  - die **approximative** schwächste Vorbedingung $\mathsf{awp}(c, Q)$
  - zusammen mit einer Menge von **Verifikationsbedingungen** $\mathsf{wvc}(c, Q)$
- Die Verifikationsbedingungen treten dort auf, wo die Weakening-Regel angewandt wird.
- Es gilt:

$$\bigwedge \mathsf{wvc}(c, Q) \Longrightarrow \models \{\mathsf{awp}(c, Q)\}\, c\, \{Q\}$$

Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(l = e, P) &\stackrel{def}{=} P[e/l] \qquad \text{(\textbf{Genauer}: Folie 24 letzte VL)} \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(\textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} i
\end{aligned}
$$

Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\},P) &\stackrel{def}{=} P \\
\mathrm{awp}(l=e,P) &\stackrel{def}{=} P[e/l] \qquad \text{(Genauer: Folie 24 letzte VL)} \\
\mathrm{awp}(c_1;c_2,P) &\stackrel{def}{=} \mathrm{awp}(c_1,\mathrm{awp}(c_2,P)) \\
\mathrm{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1,P) &\stackrel{def}{=} (b\wedge\mathrm{awp}(c_0,P))\vee(\neg b\wedge\mathrm{awp}(c_1,P)) \\
\mathrm{awp}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c,P) &\stackrel{def}{=} i
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{wvc}(\{\,\},P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(l=e,P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1;c_2,P) &\stackrel{def}{=} \mathrm{wvc}(c_1,\mathrm{awp}(c_2,P))\cup\mathrm{wvc}(c_2,P) \\
\mathrm{wvc}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1,P) &\stackrel{def}{=} \mathrm{wvc}(c_0,P)\cup\mathrm{wvc}(c_1,P) \\
\mathrm{wvc}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c,P) &\stackrel{def}{=} \mathrm{wvc}(c,i)\cup\{i\wedge b\longrightarrow\mathrm{awp}(c,i)\}\cup\{i\wedge\neg b\longrightarrow P\} \\
\mathrm{wvc}(\{P\}\,c\,\{Q\}) &\stackrel{def}{=} \{P\longrightarrow\mathrm{awp}(c,Q)\}\cup\mathrm{wvc}(c,Q)
\end{aligned}
$$

Berechnung der Verifikationsbedingungen

Programmkorrektheit

- Gegeben: Annotiertes Programm c mit Vorbedingung P und Nachbedingung Q.
- Gesucht: $\mathrm{wvc}(\{P\}\, c\, \{Q\})$

1. Rekursiv von der Nachbedigung ausgehend berechnen wir für jede Zeile des Programmes die gültige approximative Vorbedingung $\mathrm{awp}(c, -)$.
2. Dabei notieren wir alle auftretenden Verifikationsbedingungen $\mathrm{wvc}(c, -)$
3. Dabei werden **keine** Vereinfachungen vorgenommen.

Beispiel: das Fakultätsprogramm

- Sei F das annotierte Fakultätsprogramm:

```
1  // {0 ≤ n}
2  p= 1;
3  c= 1;
4  while (c <= n) //** inv {p = (c − 1)! ∧ c − 1 ≤ n ∧ 0 < c} */
5  { p = p * c;
6    c = c + 1;
7    }
8  // {p = n!}
```

- Berechnung der Verifikationsbedingungen zur Nachbedingung $\mathrm{wvc}(F, p = n!)$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  //
7  while (c <= n)
8    /** inv  p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    //
10   p = p * c;
11   //
12   c = c + 1;
13   //
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  //
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    //
10   p = p * c;
11   //
12   c = c + 1;
13   //
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 | $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  //
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    //
10   p = p * c;
11   //
12   c = c + 1;
13   // {p = (c − 1)! ∧ c − 1 ≤ n ∧ 0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 | $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  //
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    //
10   p = p * c;
11   // {p = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
12   c = c + 1;
13   // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 | $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  //
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    // {p·c = ((c+1)−1)!∧(c+1)−1 ≤ n∧0 < c+1}
10   p = p * c;
11   // {p = ((c+1)−1)!∧(c+1)−1 ≤ n∧0 < c+1}
12   c = c + 1;
13   // {p = (c−1)!∧c−1 ≤ n∧0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 | $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  //
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    // {p·c = ((c+1)−1)!∧(c+1)−1 ≤ n∧0 < c+1}
10   p = p * c;
11   // {p = ((c+1)−1)!∧(c+1)−1 ≤ n∧0 < c+1}
12   c = c + 1;
13   // {p = (c−1)!∧c−1 ≤ n∧0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 $p = (c-1)! \wedge c-1 \leq n \wedge 0 < c \wedge \neg(c \leq n)$
$\longrightarrow p = n!$

2 $p = (c-1)! \wedge c-1 \leq n \wedge 0 < c \wedge c \leq n$
$\longrightarrow p \cdot c = ((c+1)-1)! \wedge$
$(c+1)-1 \leq n \wedge$
$0 < c+1$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  //
5  c= 1;
6  // {p = (c − 1)! ∧ c − 1 ≤ n ∧ 0 < c}
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    // {p·c = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
10   p = p * c;
11   // {p = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
12   c = c + 1;
13   // {p = (c − 1)! ∧ c − 1 ≤ n ∧ 0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n)$
$\longrightarrow p = n!$

2 $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n$
$\longrightarrow p \cdot c = ((c+1)-1)! \wedge$
$(c+1) - 1 \leq n \wedge$
$0 < c + 1$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  //
3  p= 1;
4  // {p = (1−1)! ∧ 1−1 ≤ n ∧ 0 < 1}
5  c= 1;
6  // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    // {p·c = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
10   p = p * c;
11   // {p = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
12   c = c + 1;
13   // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 $p = (c-1)! \land c-1 \leq n \land 0 < c \land \neg(c \leq n) \longrightarrow p = n!$

2 $p = (c-1)! \land c-1 \leq n \land 0 < c \land c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \land (c+1)-1 \leq n \land 0 < c+1$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  // {1 = (1−1)! ∧ 1−1 ≤ n ∧ 0 < 1}
3  p= 1;
4  // {p = (1−1)! ∧ 1−1 ≤ n ∧ 0 < 1}
5  c= 1;
6  // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    // {p·c = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
10   p = p * c;
11   // {p = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
12   c = c + 1;
13   // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 $p = (c-1)! \wedge c-1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$

2 $p = (c-1)! \wedge c-1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1)-1 \leq n \wedge 0 < c+1$

Notation für Verifikationsbedingungen

AWP wird am Programm annotiert:

```
1  // {0 ≤ n}
2  // {1 = (1−1)! ∧ 1−1 ≤ n ∧ 0 < 1}
3  p= 1;
4  // {p = (1−1)! ∧ 1−1 ≤ n ∧ 0 < 1}
5  c= 1;
6  // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
7  while (c <= n)
8    /** inv p= (c−1)! ∧ c−1 ≤ n ∧ 0<c*/ {
9    // {p·c = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
10   p = p * c;
11   // {p = ((c+1)−1)! ∧ (c+1)−1 ≤ n ∧ 0 < c+1}
12   c = c + 1;
13   // {p = (c−1)! ∧ c−1 ≤ n ∧ 0 < c}
14   }
15 // {p = n!}
```

WVC wird daneben notiert:

1 | $p = (c-1)! \land c-1 \leq n \land 0 < c \land \neg(c \leq n) \longrightarrow p = n!$

2 | $p = (c-1)! \land c-1 \leq n \land 0 < c \land c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \land (c+1)-1 \leq n \land 0 < c+1$

3 | $0 \leq n \longrightarrow 1 = (1-1)! \land (1-1) \leq n \land 0 < 1$

Vereinfachung von Verifikationsbedingungen

Wir nehmen folgende **strukturelle Vereinfachungen** vor:

1. Konjunktionen in der Konklusion werden zu einzelnen Verifikationsbedingungen
   - Bsp: $A_1 \wedge A_2 \wedge A_3 \longrightarrow P \wedge Q \rightsquigarrow A_1 \wedge A_2 \wedge A_3 \longrightarrow P, A_1 \wedge A_2 \wedge A_3 \longrightarrow Q$
2. Auswertung konstanter arithmetischer Ausdrücke, einfache arithmetische Gesetze
   - Bsp. $(x + 1) - 1 \rightsquigarrow x$, $1 - 1 \rightsquigarrow 0$
3. Normalisierung der Relationen (zu $<$, $\leq$, $=$, $\neq$) und Vereinfachung
   - Bsp: $\neg(x \leq y) \rightsquigarrow x > y \rightsquigarrow y < x$, $x \leq x \rightsquigarrow \mathit{true}$, $4 \leq 5 \rightsquigarrow \mathit{true}$
4. Alle Bedingungen mit einer Prämisse *false* oder einer Konklusion *true* sind trivial erfüllt.

Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$

Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$

- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1) - 1 \leq n \wedge 0 < c + 1$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = c!$
  $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$
  $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow 0 < c + 1$

Vereinfachung am Beispiel

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge \neg(c \leq n) \longrightarrow p = n!$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$

- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = ((c+1)-1)! \wedge (c+1) - 1 \leq n \wedge 0 < c + 1$
  $\rightsquigarrow p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = c!$
  $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow c \leq n \rightsquigarrow \mathit{true}$
  $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow 0 < c + 1$

- 3: $0 \leq n \longrightarrow 1 = (1-1)! \wedge (1-1) \leq n$
  $\rightsquigarrow 0 \leq n \longrightarrow 1 = 0!$
  $0 \leq n \longrightarrow 0 \leq n \rightsquigarrow \mathit{true}$

Es bleibt zu zeigen

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$
  Aus $n < c$ folgt $n \leq c - 1$, also $c - 1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.

Es bleibt zu zeigen

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$
  Aus $n < c$ folgt $n \leq c - 1$, also $c - 1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.

- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = c!$
  Aus $p = (c-1)!$ folgt $p \cdot c = c \cdot (c-1)!$, und mit $0 < c$ und $c \cdot (c-1)! = c!$ folgt die Behauptung.

Es bleibt zu zeigen

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$
  Aus $n < c$ folgt $n \leq c - 1$, also $c - 1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.

- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = c!$
  Aus $p = (c-1)!$ folgt $p \cdot c = c \cdot (c-1)!$, und mit $0 < c$ und $c \cdot (c-1)! = c!$ folgt die Behauptung.

- 3: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow 0 < c + 1$
  Aus $0 < c$ folgt $0 < c + 1$

Es bleibt zu zeigen

- 1: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge n < c \longrightarrow p = n!$
  Aus $n < c$ folgt $n \leq c - 1$, also $c - 1 = n$, und mit $p = (c-1)!$ folgt die Behauptung.

- 2: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow p \cdot c = c!$
  Aus $p = (c-1)!$ folgt $p \cdot c = c \cdot (c-1)!$, und mit $0 < c$ und $c \cdot (c-1)! = c!$ folgt die Behauptung.

- 3: $p = (c-1)! \wedge c - 1 \leq n \wedge 0 < c \wedge c \leq n \longrightarrow 0 < c + 1$
  Aus $0 < c$ folgt $0 < c + 1$

- 4: $1 = 0!$ folgt direkt aus der Definition der Fakultät.

Arbeitsblatt 9.3: Da summt was...

```
1 // {0 ≤ n ∧ n = N}
2 p= 0;
3 while (n>0) //** inv p= sum(n+1,N); */
4 { p = p + n;
5   n = n - 1;
6   }
7 // {p = sum(1, N)}
```

1. Berechnet zuerst die **unvereinfachten** VCs (dafür sind die AWPs nötig)
2. Danach vereinfacht die VCs **schematisch** wie oben beschrieben.
3. Welche VCs sind beweisbar?

Dabei gilt: $sum(i,j) = \begin{cases} 0 & i > j \\ i + sum(i+1,j) & i \leq j \end{cases}$

Weiteres Beispiel: Maximales Element

```
1  // {0 < n}
2  i= 0;
3  r= 0;
4  while (i< n) //** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
5  { if (a[r] < a[i]) {
6       r= i;
7    }
8    else {
9    }
10   i= i+1;
11   }
12 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$\varphi(i,r)$

Maximales Element (Schleifenrumpf)

VC:

```
while (i< n)
   /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
   //
   if (a[r] < a[i]) {
      //
      r= i;
      //
      }
   else {
      //
   }
   //
   i= i+1;
   //
   }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$\varphi(i,r)$

Maximales Element (Schleifenrumpf)

```
while (i< n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    //
  }
  //
  i= i+1;
  // {φ(i,r)}
  }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$\varphi(i,r)$

VC:

1 | $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

Maximales Element (Schleifenrumpf)

```
while (i< n)
                                  φ(i,r)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    //
  }
  // {φ(i+1,r)}
  i= i+1;
  // {φ(i,r)}
  }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

VC:

1 | $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

Maximales Element (Schleifenrumpf)

```
while (i< n)
                                  φ(i,r)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    //
    }
  else {
    // {φ(i+1, r)}
  }
  // {φ(i+1, r)}
  i= i+1;
  // {φ(i, r)}
  }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

VC:

1 | $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

Maximales Element (Schleifenrumpf)

```
while (i< n)
                                  φ(i,r)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
  //
  if (a[r] < a[i]) {
    //
    r= i;
    // {φ(i+1,r)}
    }
  else {
    // {φ(i+1,r)}
  }
  // {φ(i+1,r)}
  i= i+1;
  // {φ(i,r)}
  }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

VC:

1 | $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

Maximales Element (Schleifenrumpf)

```
while (i< n)
   /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
    */
{
   //
   if (a[r] < a[i]) {
      // {φ(i+1, i)}
      r= i;
      // {φ(i+1, r)}
      }
   else {
      // {φ(i+1, r)}
   }
   // {φ(i+1, r)}
   i= i+1;
   // {φ(i, r)}
   }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

$\varphi(i,r)$

VC:

1 | $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j < n \longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

Maximales Element (Schleifenrumpf)

```
while (i< n)
   /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
   // {a[r] < a[i] ∧ φ(i+1, i) ∨ (¬(a[r] < a[i]) ∧ φ(i+1, r))}
   if (a[r] < a[i]) {
      // {φ(i+1, i)}
      r= i;
      // {φ(i+1, r)}
      }
   else {
      // {φ(i+1, r)}
   }
   // {φ(i+1, r)}
   i= i+1;
   // {φ(i, r)}
   }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

VC:

1 | $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r])$
$\wedge$
$0 \le r < n$

Maximales Element (Schleifenrumpf)

```
while (i< n)
                                  φ(i,r)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
 */
{
  // {a[r] < a[i] ∧ φ(i+1,i) ∨ (¬(a[r] < a[i]) ∧ φ(i+1,r))}
  if (a[r] < a[i]) {
    // {φ(i+1,i)}
    r= i;
    // {φ(i+1,r)}
    }
  else {
    // {φ(i+1,r)}
  }
  // {φ(i+1,r)}
  i= i+1;
  // {φ(i,r)}
  }
// {(∀j. 0 ≤ j <⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

VC:

1 $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r])$
$\wedge$
$0 \le r < n$

2 $\varphi(i,r) \wedge i < n \longrightarrow$
$a[r] < a[i] \wedge \varphi(i+1,i)$
$\vee$
$\neg(a[r] < a[i]) \wedge \varphi(i+1,r)$

Maximales Element (Initialisierung)

```
// {0 < n}
//
i= 0;
//
r= 0;
// {φ(i, r)}
while (i< n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n}
 */
```

($\varphi(i,r)$ über dem invariant)

VC:

1 $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j <\!\!\longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

2 $\varphi(i,r) \wedge i < n \longrightarrow$
$a[r] < a[i] \wedge \varphi(i+1, i)$
$\vee$
$\neg(a[r] < a[i]) \wedge \varphi(i+1, r)$

Maximales Element (Initialisierung)

```
// {0 < n}
//
i= 0;
// {φ(i,0)}
r= 0;
// {φ(i,r)}
while (i< n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n}
 */
```

($\varphi(i,r)$ is the invariant)

VC:

1 $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j <\!\!\longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

2 $\varphi(i,r) \wedge i < n \longrightarrow$
$a[r] < a[i] \wedge \varphi(i+1, i)$
$\vee$
$\neg(a[r] < a[i]) \wedge \varphi(i+1, r)$

Maximales Element (Initialisierung)

```
// {0 < n}
// {φ(0,0)}
i= 0;
// {φ(i,0)}
r= 0;
// {φ(i,r)}
while (i< n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n}
 */
```

($\varphi(i,r)$ over the invariant)

VC:

1 $\varphi(i,r) \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j <\!\!\longrightarrow a[j] \leq a[r])$
$\wedge$
$0 \leq r < n$

2 $\varphi(i,r) \wedge i < n \longrightarrow$
$a[r] < a[i] \wedge \varphi(i+1, i)$
$\vee$
$\neg(a[r] < a[i]) \wedge \varphi(i+1, r)$

Maximales Element (Initialisierung)

```
// {0 < n}
// {φ(0,0)}
i= 0;
// {φ(i,0)}
r= 0;
// {φ(i,r)}
while (i< n)
  /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n}
 */
```

VC:

1	$\varphi(i,r) \wedge \neg(i < n) \longrightarrow$ $(\forall j.\, 0 \leq j \longrightarrow a[j] \leq a[r])$ $\wedge$ $0 \leq r < n$
2	$\varphi(i,r) \wedge i < n \longrightarrow$ $a[r] < a[i] \wedge \varphi(i+1, i)$ $\vee$ $\neg(a[r] < a[i]) \wedge \varphi(i+1, r)$
3	$0 \leq n \longrightarrow \varphi(0,0)$

Maximales Element (Verifikationsbedingungen)

Unvereinfacht:

1 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i \leq n \wedge \neg(i < n) \longrightarrow$
$(\forall j.\, 0 \leq j < \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n$

2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i \leq n \wedge i < n \longrightarrow$
$((a[r] < a[i] \wedge (\forall j.0 \leq j < i + 1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i < n \wedge 0 \leq i + 1 \leq n) \vee$
$(\neg(a[r] < a[i]) \wedge (\forall j.0 \leq j < i + 1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i + 1 \leq n))$

3 $0 \leq n \longrightarrow (\forall j.0 \leq j < 0 \longrightarrow a[j] \leq a[0]) \wedge 0 \leq 0 < n \wedge 0 \leq 0 \leq n$

- Sehr **lange** Verifikationsbedingungen (u.a. wegen Fallunterscheidung)
  - Insbesondere schwer zu **vereinfachen**
- Wie können wir das **beheben**?

Maximales Element (Verifikationsbedingungen)

Vereinfacht:

1.1 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i \leq n \wedge n \leq i \longrightarrow$
$\forall j.0 \leq j < n \longrightarrow a[j] \leq a[r]$

1.2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i \leq n \wedge n \leq i \longrightarrow 0 \leq r \leq n$

2 $(\forall j.0 \leq j < i \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i \leq n \wedge i < n \longrightarrow$
$((a[r] < a[i] \wedge (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[i]) \wedge 0 \leq i < n \wedge 0 \leq i+1 \leq n) \vee$
$(\neg(a[i] \leq a[r]) \wedge (\forall j.0 \leq j < i+1 \longrightarrow a[j] \leq a[r]) \wedge 0 \leq r < n \wedge 0 \leq i+1 \leq n))$

3.1 $0 \leq n \longrightarrow \forall j.0 \leq j < 0 \longrightarrow a[j] \leq a[0]$

3.2 $0 \leq n \longrightarrow 0 \leq 0 < 0$

3.2 $0 \leq n \longrightarrow 0 \leq 0 \leq 0$

- Sehr **lange** Verifikationsbedingungen (u.a. wegen Fallunterscheidung)
  - Insbesondere schwer zu **vereinfachen**
- Wie können wir das **beheben**?

Explizite Vorbedingungen

Lange Vorbedingung:

```
// {(P1 ∧ b) ∨ (P2 ∧ ¬b)}
if (b) {
  // {P1}
  ...
  // {Q}
  } else {
  // {P2}
  ...
  // {Q}
  }
```

Kurze Vorbedingung:

```
// {A}
if (b) {
  // {A ∧ b}
  ...
  // {Q}
  } else {
  // {A ∧ ¬b}
  ...
  // {Q}
  }
```

Dazu VCs:

$$A \wedge b \longrightarrow P_1$$
$$A \wedge \neg b \longrightarrow P_2$$

Spracherweiterung: Explizite Spezifikationen

- Erweiterung der Sprache C0 um Invarianten für Schleifen und **explizite Zusicherung**

$$
\begin{array}{lll}
\textbf{Assn} & a ::= \ldots & \text{— Zusicherungen} \\
\textbf{Stmt} & c ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
& \mid & \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ a\ {*}/\ c \\
& \mid & /{*}{*}\ \{a\}\ {*}/
\end{array}
$$

- Zusicherungen haben **keine Semantik** (Kommentar!), sondern erzwingen eine neue Vorbedingung.
- Dazu vereinfachte Regel für Fallunterscheidung:

$$\text{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) \stackrel{def}{=} (b \wedge \text{awp}(c_0, P)) \vee (\neg b \wedge \text{awp}(c_1, P))$$

Wenn $\text{awp}(c_0, P) = b \wedge P_0$, $\text{awp}(c_1, P) = \neg b \wedge P_0$, dann gilt

$$(b \wedge b \wedge P_0) \vee (\neg b \wedge \neg b \wedge P_0) = (b \wedge P_0) \vee (\neg b \wedge P_0) = (b \vee \neg b) \wedge P_0 = P_0$$

Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(l = e, P) &\stackrel{def}{=} P[e/l] \qquad \text{(Genauer: Folie 24 letzte VL)} \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} Q \quad \text{wenn } \mathrm{awp}(c_0, P) = b \wedge Q, \mathrm{awp}(c_1, P) = \neg b \wedge Q \\
\mathrm{awp}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \mathrm{q} \\
\mathrm{awp}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv } i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(l = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if } (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \{q \longrightarrow P\} \\
\mathrm{wvc}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv } i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}
$$

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    //
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     //
12     r= i;
13     //
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     //
18     }
19   //
20   i= i+1;
21   //
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    //
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     //
12     r= i;
13     //
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     //
18     }
19   //
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    //
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     //
12     r= i;
13     //
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     //
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    //
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     //
12     r= i;
13     //
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    //
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     //
12     r= i;
13     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    //
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < n ∧ 0 ≤ i + 1 ≤ n}
12     r= i;
13     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  //
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ i < n}
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < n ∧ 0 ≤ i + 1 ≤ n}
12     r= i;
13     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  //
3  i= 0;
4  r= 0;
5  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i ∧ i ≤ n}
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ i < n}
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < n ∧ 0 ≤ i + 1 ≤ n}
12     r= i;
13     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung

```
1  // {0 < n}
2  // {(∀j. 0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 < 0 ∧ 0 ≤ n}
3  i= 0;
4  r= 0;
5  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < i ∧ i ≤ n}
6  while (i< n) /** inv {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n} */
7  {
8    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r] ∧ 0 ≤ r < i ∧ i < n}
9    if (a[r] < a[i]) {
10     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ a[r] < a[i]}
11     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i < n ∧ 0 ≤ i + 1 ≤ n}
12     r= i;
13     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
14     }
15   else {
16     // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i < n ∧ ¬(a[r] < a[i])}
17     // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
18     }
19   // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i + 1 ≤ n}
20   i= i+1;
21   // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n ∧ 0 ≤ i ≤ n}
22   }
23 // {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Maximales Element mit Zusicherung: Beweisverpflichtungen

Unvereinfacht:

$$
\begin{array}{ll}
(1) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \wedge \neg(i < n) \\
& \longrightarrow (\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \\
(2) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge \neg(a[r] < a[i]) \\
& \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[r]) \wedge 0 \le r < i+1 \wedge 0 \le i+1 \le n \\
(3) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[r] < a[i] \\
& \longrightarrow (\forall j.\, 0 \le j < i+1 \longrightarrow a[j] \le a[i]) \wedge 0 \le i < n \wedge 0 \le i+1 \le n \\
(4) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \longrightarrow \\
& (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \\
(5) & 0 < n \longrightarrow (\forall j.\, 0 \le j < 0 \longrightarrow a[j] \le a[0]) \wedge 0 \le 0 < n \wedge 0 \le 0 \le n
\end{array}
$$

Maximales Element mit Zusicherung: Beweisverpflichtungen

Vereinfacht (Teil 1):

$$
\begin{aligned}
&(1.1)\quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \wedge n \le i \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < n \longrightarrow a[j] \le a[r]) \\
&(1.2)\quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \wedge n \le i \\
&\qquad \longrightarrow 0 \le r < n \\
&(2.1)\quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[i] \le a[r] \\
&\qquad \longrightarrow (\forall j.\, 0 \le j < i + 1 \longrightarrow a[j] \le a[r]) \\
&(2.2)\quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[i] \le a[r] \\
&\qquad \longrightarrow 0 \le r < n \\
&(2.3)\quad (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[i] \le a[r] \\
&\qquad \longrightarrow 0 \le i + 1 \le n
\end{aligned}
$$

Maximales Element mit Zusicherung: Beweisverpflichtungen

Vereinfacht (Teil 2):

$$
\begin{array}{ll}
(3.1) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[r] < a[i] \\
 & \longrightarrow (\forall j.\, 0 \le j < i + 1 \longrightarrow a[j] \le a[i]) \\
(3.2) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[r] < a[i] \\
 & \longrightarrow 0 \le i < n \\
(3.3) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i < n \wedge a[r] < a[i] \\
 & \longrightarrow 0 \le i + 1 \le n \\
(4.1) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \longrightarrow \\
 & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \\
(4.2) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \longrightarrow\ 0 \le r < n \\
(4.3) & (\forall j.\, 0 \le j < i \longrightarrow a[j] \le a[r]) \wedge 0 \le r < n \wedge 0 \le i \le n \longrightarrow\ 0 \le i < n \\
(5.1) & 0 < n \longrightarrow (\forall j.\, 0 \le j < 0 \longrightarrow a[j] \le a[0]) \\
(5.2) & 0 < n \longrightarrow 0 \le 0 < n \\
(5.3) & 0 < n \longrightarrow 0 \le 0 \le n
\end{array}
$$

Beweismethoden

- Um $P_1 \wedge \ldots \wedge P_n \longrightarrow Q$ zu zeigen, nehmen wir $P_1, \ldots, P_n$ an und zeigen Q.
- Dabei nutzen wir **u.a.** folgende Regeln:

Wenn P, dann P (Trivial)
Wenn P und $l = t$, dann $P[t/l]$ (Substitution)
$x \leq x$ (Reflexität)
Wenn $x \leq y$ und $y \leq z$, dann $x \leq z$ (Transitivität)
Wenn $x \leq y$ und $y \leq x$, dann $x = y$ (Antisymmetrie)
Wenn $x < y$, dann $x \leq y + 1$ oder $x + 1 \leq y$ (Inc)
Wenn $\forall x.\, P$, dann $P[t/x]$ (Instantiierung)
Wenn *false*, dann P (Ex falso)
Wenn $a \leq b$ und $x \leq y$, dann $a + x \leq b + y$ und Variation mit $x = 0$ etc.
Umformungen mit $(0, +)$ und $(1, \cdot)$
Domänenspezifische Regeln

Arbeitsblatt 9.4: Beweisverpflichtungen Beweisen

Betrachtet die vereinfachten Verifikationsbedingungen. Wie würdet ihr sie beweisen? Was für Methoden verwendet ihr?

- Welches sind **triviale** Beweise?
- Welches sind **einfache** Beweise?
- Welche erfordern längere Argumentation?

Arbeitsblatt 9.5: Kopien

Dieses Programm kopiert ein Array:

```
i= 0;
while (i< m)
  /** inv ??? */ {
  b[m-1-i]= a[i];
  i= i+1;
  }
```

1. Spezifiziert die Funktionalität.
2. Findet die Invariante.
3. Berechnet die Verifikationsbedingungen (VCs) und schwächste Vorbedingung.
4. Beweist die VCs.

Noch ein Beispiel: kleinstes Null-Element

- Folgendes Programm sucht in a nach dem **ersten** Null-Element:

```
void find_zero()
{
i= 0;
r= -1;
while (i< n)
  if (r== -1 && a[i] == 0) {
    r= i;
    }
  else {
    }
  }
}
```

- Wie **spezifizieren** wir das?

Noch ein Beispiel: kleinstes Null-Element

- Folgendes Programm sucht in a nach dem **ersten** Null-Element:

```
void find_zero()
{
i= 0;
r= -1;
while (i< n)
  if (r== -1 && a[i] == 0) {
    r= i;
    }
  else {
    }
  }
}
```

- Wie **spezifizieren** wir das?
- Welche **Beweisverpflichtungen** entstehen?

Noch ein Beispiel: kleinstes Null-Element

- Folgendes Programm sucht in a nach dem **ersten** Null-Element:

```
void find_zero()
{
i= 0;
r= -1;
while (i< n)
  if (r== -1 && a[i] == 0) {
    r= i;
    }
  else {
    }
  }
}
```

- Wie **spezifizieren** wir das?
- Welche **Beweisverpflichtungen** entstehen?
- Wie **beweisen** wir diese?

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.

Korrekte Software: Grundlagen und Methoden

Vorlesung 10 vom 13.06.22

Funktionen und Prozeduren I

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Funktionen & Prozeduren

- **Funktionen** sind das zentrale Modularisierungskonzept von C
  - Kleinste Einheit
  - NB. Prozeduren sind nur Funktionen vom Typ **void**
- In objektorientierten Sprachen: Methoden
  - Funktionen mit (implizitem) erstem Parameter **this**
- Wie behandeln wir Funktionen?

Beispiel: Rekursion

Fakultät

```
// {N == n ∧ 0 ≤ n}
p= 1;
while (0< n)
  /** inv p == (N-n)! ∧ 0≤ n; */ {
  p= p* n;
  n= n-1;
  }
// {p == N!}
```

Verkapselt als Funktion

```
int factorial(int n)
/** pre  0≤ n;
    post \result == n!; */
{
 /** N == n; */
 int p;
 p= 1;
 while (0< n)
   /** inv p == (N-n)! ∧ 0≤ n; */ {
   p= p* n;
   n= n-1;
   }
 return p;
}
```

- **Spezifikation** mit Vor-/Nachbedingung
- **\result** für Ergebnis
- Lokale Variablen von außen nicht sichtbar (scoping)

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
- Neue Anweisungen: Return-Anweisung

$$
\begin{aligned}
\textbf{FunDef} &::= \textbf{FunHeader}\ \textbf{FunSpec}^{+}\ \textbf{Blk} \\
\textbf{FunHeader} &::= \textbf{Type}\ \textbf{Idt}(\textbf{Decl}^{*}) \\
\textbf{Decl} &::= \textbf{Type}\ \textbf{Idt} \\
\textbf{Blk} &::= \{\textbf{Decl}^{*}\ \textbf{Stmt}\} \\
\textbf{Stmt}\ c &::= \ l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
&\quad \mid \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ P\ {*}/\ c \mid /{*}{*}\ \{P\}\ {*}/ \\
&\quad \mid \textbf{return}\ a^{?}
\end{aligned}
$$

- Abstrakte Syntax (konkrete Syntax mischt **Type** und **Idt**, Kommata bei Argumenten, ...)
- **FunSpec** wird später erläutert

I. Semantik: return

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):

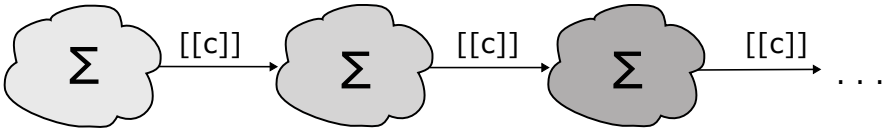
> **Rule 14.7 (required)**
>
> A function shall have a single point of exit at the end of the function.

  - Nicht immer möglich, unübersichtlicher Code ...

- Lösung 2: Erweiterung der Semantik

Denotationale Semantik der return-Anweisung

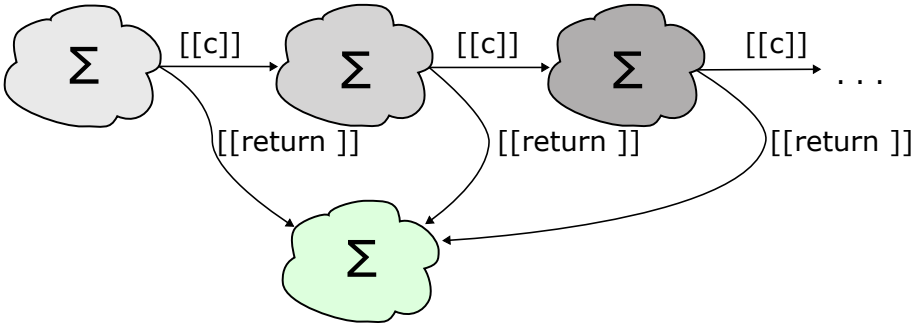
- Alt: $\Sigma \rightharpoonup \Sigma$



-

Denotationale Semantik der return-Anweisung

- Alt: $\Sigma \rightharpoonup \Sigma$



- **Neu**: $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$

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 mit Rückgabewerten

- Komposition zweier Anweisungen $f, g : \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$.
- Im Allgemeinen: Wenn $f, g : A \rightharpoonup A + B$ dann $g \circ_S f : A \rightharpoonup A + B$:

$$g \circ_S f(\sigma) \stackrel{def}{=} \begin{cases} g(a') & f(a) = a' \\ b & f(a) = b \end{cases}$$

- Als Mengen: $f, g \subseteq (A \times (A + B)) \cong A \times A + A \times B$

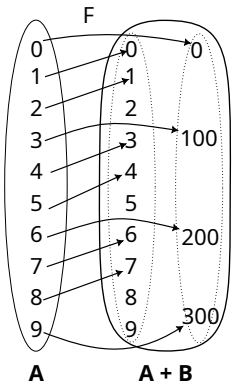
$$g \circ_S f = \{(a, x) \mid \exists a' \in A.\, (a, a') \in f \wedge (a', x) \in g\} \cup \{(a, b) \mid (a, b) \in f, b \in B\}$$

- **Frage:** Ist das gleiche wie Komposition von partiellen Funktionen?

Arbeitsblatt 10.1: Komposition mit Rückgabewerten

Sei $\mathbf{A} = \{0, \ldots, 9\}$
$\mathbf{B} = \{0, 100, 200, 300\}$

Betrachte $F : \mathbf{A} \rightharpoonup \mathbf{A} + \mathbf{B}$



1. Wie ist die Funktion F links in Mengenschreibweise definiert?
2. Wie sind folgende Funktionen (als Mengen) definiert:
   $F_2 \stackrel{def}{=} F \circ_S F$?
   $F_3 \stackrel{def}{=} F \circ_S F \circ_S F$?
3. Was ist die **Bedeutung** von F_3?

Semantik von Anweisungen

$$[\![.]\!]_{\mathcal{C}} : \mathbf{Stmt} \to \mathbf{Env} \to \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$$

$$[\![x = e]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, \sigma[l \mapsto a]) \mid (\sigma, l) \in [\![x]\!]_{\mathcal{L}}, (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\}$$

$$[\![c_1; c_2]\!]^{\Gamma}_{\mathcal{C}} = [\![c_2]\!]^{\Gamma}_{\mathcal{C}} \circ_S [\![c_1]\!]^{\Gamma}_{\mathcal{C}} \qquad \text{Komposition wie oben}$$

$$[\![\{\,\}]\!]^{\Gamma}_{\mathcal{C}} = \mathbf{Id}_\Sigma \qquad \mathbf{Id}_\Sigma := \{(\sigma, \sigma) | \sigma \in \Sigma\}$$

$$\begin{aligned}[\![\textbf{if } (b)\ c_0\ \textbf{ else }\ c_1]\!]^{\Gamma}_{\mathcal{C}} = {} & \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]^{\Gamma}_{\mathcal{B}} \wedge (\sigma, \rho') \in [\![c_0]\!]^{\Gamma}_{\mathcal{C}}\} \\ & \cup \{(\sigma, \rho') \mid (\sigma, \mathit{false}) \in [\![b]\!]^{\Gamma}_{\mathcal{B}} \wedge (\sigma, \rho') \in [\![c_1]\!]^{\Gamma}_{\mathcal{C}}\} \\ & \text{mit } \rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U\end{aligned}$$

$$[\![\textbf{return }\ e]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]^{\Gamma}_{\mathcal{A}}\}$$

$$[\![\textbf{return}]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, (\sigma, *))\}$$

$$\begin{aligned}[\![\textbf{while } (b)\ c]\!]^{\Gamma}_{\mathcal{C}} = \mathit{fix}(\Psi),\ \Psi(\varphi) \stackrel{\mathit{def}}{=} {} & \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]^{\Gamma}_{\mathcal{B}} \wedge (\sigma, \rho') \in \varphi \circ_S [\![c]\!]^{\Gamma}_{\mathcal{C}}\} \\ & \cup \{(\sigma, \sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]^{\Gamma}_{\mathcal{B}}\}\end{aligned}$$

Arbeitsblatt 10.2: Semantik mit Rückgabe

Berechnet die Denotate der folgenden Programme:

1. $$[\![x = 3; x = 4]\!]^{\Gamma}_{\mathcal{C}} = \textbf{?}$$

2. $$[\![x = 3; \textbf{return}\ \ x; x = 4]\!]^{\Gamma}_{\mathcal{C}} = \textbf{?}$$

Erweiterung des Floyd-Hoare-Kalküls

- Wie passt die neue Semantik $[\![.]\!]^{\Gamma}_{C} : \mathbf{Stmt} \to \mathbf{Env} \to \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$ zu unseren Floyd-Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?

Erweiterung des Floyd-Hoare-Kalküls

- Wie passt die neue Semantik $[\![.]\!]^{\Gamma}_{C} : \mathbf{Stmt} \to \mathbf{Env} \to \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$ zu unseren Floyd-Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?
- Problem: Tripel behandel **einen** Nachzustand, jetzt haben wir **zwei** ...
- Lösung: **Erweiterung** des Tripels um eine explizite Nachbedingung für den **Rückgabezustand**

$$\Gamma \models \{P\}\, c\, \{Q \mid Q_R\}$$

- Neue Konstante **\result** für das Resultat der Funktion

Erweiterung der Spezifikationen

- Erweiterung von **Aexpv** um **\result**
- Erweiterung von $[\![.]\!]_{\mathcal{B}sp}$ und $[\![.]\!]_{\mathcal{A}sp}$
  - Vorher: Zustand

$$[\![.]\!]_{\mathcal{B}sp} : \mathbf{Env} \to \mathbf{Intprt} \to \mathbf{Assn} \rightharpoonup \Sigma \to \mathbb{B}$$
$$[\![.]\!]_{\mathcal{A}sp} : \mathbf{Env} \to \mathbf{Intprt} \to \mathbf{Aexpv} \rightharpoonup \Sigma \to \mathbf{V}$$

  - Jetzt: Zustand und **Rückgabewert**

$$[\![.]\!]_{\mathcal{B}sp} : \mathbf{Env} \to \mathbf{Intprt} \to \mathbf{Assn} \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbb{B}$$
$$[\![.]\!]_{\mathcal{A}sp} : \mathbf{Env} \to \mathbf{Intprt} \to \mathbf{Aexpv} \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbf{V}$$

- **\result** darf nur in **Nachbedingungen** von Funktionen vom Typ ungleich **void** auftreten.

Semantik von Spezifikationen

$$[\![.]\!]_{\mathcal{B}sp} : \mathbf{Env} \to \mathbf{Intprt} \to \mathbf{Assn} \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbb{B}$$

$$[\![.]\!]_{\mathcal{A}sp} : \mathbf{Env} \to \mathbf{Intprt} \to \mathbf{Aexpv} \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbf{V}$$

$$\begin{aligned}[\![b_1 \;\&\&\; b_2]\!]^{\Gamma,I}_{\mathcal{B}sp} = {} & \{((\sigma, v), \mathit{true}) \mid ((\sigma, v), \mathit{true}) \in [\![b_1]\!]^{\Gamma,I}_{\mathcal{B}sp} \wedge ((\sigma, v), \mathit{true}) \in [\![b_2]\!]^{\Gamma,I}_{\mathcal{B}sp}\} \\ & \cup \{((\sigma, v), \mathit{false}) \mid ((\sigma, v), \mathit{false}) \in [\![b_1]\!]^{\Gamma,I}_{\mathcal{B}sp} \vee ((\sigma, v), \mathit{false}) \in [\![b_2]\!]^{\Gamma,I}_{\mathcal{B}sp}\}\end{aligned}$$

$$\ldots$$

$$[\![a_1 = a_2]\!]^{\Gamma,I}_{\mathcal{B}sp} = \{((\sigma, v), v_1 = v_2) \mid ((\sigma, v), v_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathcal{A}sp}, (\sigma, (\sigma', v_2)) \in [\![a_2]\!]^{\Gamma,I}_{\mathcal{A}sp}\}$$

$$[\![a_1 + a_2]\!]^{\Gamma,I}_{\mathcal{A}sp} = \{((\sigma, v), v_1 + v_2) \mid ((\sigma, v), v_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathcal{A}sp}, (\sigma, (\sigma', v_2)) \in [\![a_2]\!]^{\Gamma,I}_{\mathcal{A}sp}\}$$

$$\ldots$$

$$[\![l]\!]^{\Gamma,I}_{\mathcal{A}sp} = \{((\sigma, v), \sigma(m)) \mid (\sigma, m) \in [\![l]\!]^{\Gamma,I}_{\mathcal{L}sp}\}, l \in \mathbf{Lexp}$$

$$[\![x]\!]^{\Gamma,I}_{\mathcal{A}sp} = I(x), x \in \mathrm{dom}(I)$$

$$[\![\backslash\mathbf{result}]\!]^{\Gamma,I}_{\mathcal{A}sp} = \{((\sigma, v), v)\}$$

Erweiterung der Hoare-Tripel

$$\Gamma \models \{P\}\, c\, \{Q \mid Q_R\}$$

- Vorbedingung P wird im Vorzustand ausgewertet.
- Nachbedingung Q wird im Folgezustand ausgewertet.
- Nachbedingung Q_R wird im Rückgabezustand ausgewertet
- Umgebung Γ und Interpretation I müssen für beide gleich sein.
- I ist beliebig, Γ ist gegeben.
- **result** darf nur in Rückgabebedingungen von Funktionen vom Typ ungleich **void** auftreten.

Erweiterung der Hoare-Tripel

Partielle Korrektheit ($\Gamma \models \{P\}\, c\, \{Q \mid Q_R\}$)

c ist **partiell korrekt**, wenn für alle Zustände σ, die P erfüllen:

- die Ausführung von c mit σ in σ' regulär terminiert, so dass σ' die Spezifikation Q erfüllt,
- oder die Ausführung von c in σ' mit dem Rückgabewert v terminiert, so dass (σ', v) die Rückgabespezifikation Q_R erfüllt.

$$\Gamma \models \{P\}\, c\, \{Q \mid Q_R\} \iff$$
$$\forall I.\, \forall \sigma.\, ((\sigma, *), \mathit{true}) \in [\![P]\!]^{\Gamma,I}_{\mathcal{B}sp} \implies (\exists \sigma'.\, (\sigma, \sigma') \in [\![c]\!]^{\Gamma}_{\mathcal{C}} \implies ((\sigma', *), \mathit{true}) \in [\![Q]\!]^{\Gamma,I}_{\mathcal{B}sp})$$
$$\vee$$
$$(\exists \sigma', v.\, (\sigma, (\sigma', v)) \in [\![c]\!]^{\Gamma}_{\mathcal{C}} \implies ((\sigma', v), \mathit{true}) \in [\![Q_R]\!]^{\Gamma,I}_{\mathcal{B}sp})$$

Erweiterung des Floyd-Hoare-Kalküls: return

$$\frac{}{\vdash \{Q\}\ \textbf{return}\ \{P \mid Q\}} \qquad \frac{}{\vdash \{Q[e/\backslash\textbf{result}]\}\ \textbf{return}\ \ e\ \{P \mid Q\}}$$

- Bei **return** wird die Rückgabespezifikation Q zur Vorbedingung, die reguläre Nachfolgespezifikation wird ignoriert, da die Ausführung von **return** kein Nachfolgezustand hat.
- **return** ohne Argument darf nur bei einer Nachbedingung Q auftreten, die kein **\result** enthält.
- Bei **return** mit Argument ersetzt der Rückgabewert den **\result** in der Rückgabespezifikation.

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

$$\frac{}{\vdash \{P\}\,\{\}\,\{P \mid Q_R\}} \qquad \frac{\vdash \{P\}\,c_1\,\{R \mid Q_R\} \quad \vdash \{R\}\,c_2\,\{Q \mid Q_R\}}{\vdash \{P\}\,c_1; c_2\,\{Q \mid Q_R\}}$$

$$\frac{}{\vdash \{Q[e/x]\}\,l = e\,\{Q \mid Q_R\}} \qquad \frac{\vdash \{P \wedge b\}\,c\,\{P \mid Q_R\}}{\vdash \{P\}\ \textbf{while}\ (b)\ c\,\{P \wedge \neg b \mid Q_R\}}$$

$$\frac{\vdash \{P \wedge b\}\,c_1\,\{Q \mid Q_R\} \quad \vdash \{P \wedge \neg b\}\,c_2\,\{Q \mid Q_R\}}{\vdash \{P\}\ \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2\,\{Q \mid Q_R\}}$$

$$\frac{P \longrightarrow P' \quad \vdash \{P'\}\,c\,\{Q' \mid R'\} \quad Q' \longrightarrow Q \quad R' \longrightarrow R}{\vdash \{P\}\,c\,\{Q \mid R\}}$$

$$\frac{}{\vdash \{Q\}\ \textbf{return}\,\{P \mid Q\}} \qquad \frac{}{\vdash \{Q[e/\backslash\textbf{result}]\}\ \textbf{return}\ e\,\{P \mid Q\}}$$

Arbeitsblatt 10.3: Kurzbeispiel

Verifiziert folgendes Kurzbeispiel:

```
{  // {x = X}
   // ???
   x= x+1;
   // ???
   return x;
   // {false | \result == X + 1}
}
```

Lösungsblatt 10.3: Kurzbeispiel

```
{  // {x = X}
   //
   x= x+1;
   //
   return x;
   // {false | \result = X + 1}
}
```

Lösungsblatt 10.3: Kurzbeispiel

```
{  // {x = X}
   //
   x= x+1;
   // {x = X + 1}
   return x;
   // {false | \result = X + 1}
}
```

Lösungsblatt 10.3: Kurzbeispiel

```
{  // {x = X}
   // {x + 1 = X + 1}
   x= x+1;
   // {x = X + 1}
   return x;
   // {false | \result = X + 1}
}
```

Lösungsblatt 10.3: Kurzbeispiel

```
{  // {x = X}
   // {x + 1 = X + 1}
   x= x+1;
   // {x = X + 1}
   return x;
   // {false | \result = X + 1}
}
```

Beweisverpflichtung:

$$x = X \Longrightarrow x + 1 = X + 1 \quad \checkmark$$

Zusatzfrage

Was ist der Unterschied zwischen den Nachbedingungen

- $\{\mathit{false} \mid \backslash\mathbf{result} = X + 1\}$
- $\{x = X + 1 \mid \backslash\mathbf{result} = X + 1\}$
- $\{\mathit{true} \mid \backslash\mathbf{result} = X + 1\}$

Genauer: wie muss die Funktion beschaffen sein, um diese Nachbedingungen zu erfüllen?

Zusatzfrage

Was ist der Unterschied zwischen den Nachbedingungen

i. $\{\mathit{false} \mid \backslash\mathbf{result} = X + 1\}$

ii. $\{x = X + 1 \mid \backslash\mathbf{result} = X + 1\}$

iii. $\{\mathit{true} \mid \backslash\mathbf{result} = X + 1\}$

Genauer: wie muss die Funktion beschaffen sein, um diese Nachbedingungen zu erfüllen?

i. Ende des Rumpfes wird nie regulär erreicht, und es wird $x + 1$ zurückgegeben

ii. Wenn Ende des Rumpfes erreicht wird, muss $x = X + 1$ sein, oder es wird $x + 1$ zurückgegeben

iii. Entweder Ende des Rumpfes wird erreicht, oder es wird $x + 1$ zurückgegeben.

Approximative schwächste Vorbedingung

- Erweiterung zu $\mathrm{awp}(c, Q, Q_R)$ und $\mathrm{wvc}(c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
- Es wird immer eine **Rückgabespezifikation** Q_R benötigt.
- Die Rückgabespezifikation wird immer durchgereicht, aber:
- **return** ersetzt die schwächste Vorbedingung durch die Rückgabespezifikation.
- Es gilt:

$$\bigwedge \mathrm{wvc}(c, Q, Q_R) \Longrightarrow \vdash \{\mathrm{awp}(c, Q, Q_R)\}\, c\, \{Q \mid Q_R\}$$

Approximative schwächste Vorbedingung (Revisited)

$$
\begin{aligned}
\mathsf{awp}(\{\,\},\, Q,\, Q_R) &\stackrel{def}{=} Q \\
\mathsf{awp}(l = e,\, Q,\, Q_R) &\stackrel{def}{=} Q[e/l] \\
\mathsf{awp}(c_1; c_2,\, Q,\, Q_R) &\stackrel{def}{=} \mathsf{awp}(c_1, \mathsf{awp}(c_2, Q, Q_R), Q_R) \\
\mathsf{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1,\, Q,\, Q_R) &\stackrel{def}{=} (b \wedge \mathsf{awp}(c_0, Q, Q_R)) \vee (\neg b \wedge \mathsf{awp}(c_1, Q, Q_R)) \\
\mathsf{awp}(/{*}{*}\ \{q\}\ {*}/,\, Q,\, Q_R) &\stackrel{def}{=} q \\
\mathsf{awp}(\textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c,\, Q_R) &\stackrel{def}{=} i \\
\mathsf{awp}(\textbf{return}\ e,\, Q,\, Q_R) &\stackrel{def}{=} Q_R[e/\backslash\textbf{result}] \\
\mathsf{awp}(\textbf{return},\, Q,\, Q_R) &\stackrel{def}{=} Q_R
\end{aligned}
$$

Approximative Verifikationsbedingungen (Revisited)

$$
\begin{aligned}
\mathrm{wvc}(\{\,\}, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(l = e, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, Q, Q_R), Q_R) \cup \mathrm{wvc}(c_2, Q, Q_R) \\
\mathrm{wvc}(\textbf{if } (b)\ c_1\ \textbf{else}\ c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(c_1, Q, Q_R) \cup \mathrm{wvc}(c_2, Q, Q_R) \\
\mathrm{wvc}(/{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} \{q \Longrightarrow Q\} \\
\mathrm{wvc}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(c, i, Q_R) \cup \{i \wedge b \Longrightarrow \mathrm{awp}(c, i, Q_R)\} \\
&\qquad\qquad \cup \{i \wedge \neg b \Longrightarrow Q\} \\
\mathrm{wvc}(\textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}
$$

Beispiel: Fakultät

```
1  { // {0 ≤ n}
2   //
3   p= 1;
4   //
5   c= 1;
6   //
7   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
8       p= p*c;
9       if (c == n) {
10        return p;
11        }
12      c= c+1;
13      }
14  // {false | \result == n!}
15 }
```

Beispiel: Fakultät

```
1  { // {0 ≤ n}
2   //
3   p= 1;
4   //
5   c= 1;
6   //
7   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
8       p= p*c;
9       if (c == n) {
10        return p;
11        }
12      c= c+1;
13      }
14  // {false | \result == n!}
15 }
```

Beispiel: Fakultät

```
1  { // {0 ≤ n}
2   //
3   p= 1;
4   //
5   c= 1;
6   // {p = (c − 1)! ∧ 0 < c}
7   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
8       p= p*c;
9       if (c == n) {
10        return p;
11        }
12      c= c+1;
13      }
14  // {false | \result == n!}
15 }
```

Beispiel: Fakultät

```
1  { // {0 ≤ n}
2   //
3   p= 1;
4   // {p = (1 − 1)! ∧ 0 < 1}
5   c= 1;
6   // {p = (c − 1)! ∧ 0 < c}
7   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
8       p= p*c;
9       if (c == n) {
10        return p;
11        }
12      c= c+1;
13      }
14  // {false | \result == n!}
15 }
```

Beispiel: Fakultät

```
1  { // {0 ≤ n}
2   // {1 = (1 − 1)! ∧ 0 < 1}
3   p= 1;
4   // {p = (1 − 1)! ∧ 0 < 1}
5   c= 1;
6   // {p = (c − 1)! ∧ 0 < c}
7   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
8       p= p*c;
9       if (c == n) {
10        return p;
11        }
12      c= c+1;
13      }
14  // {false | \result == n!}
15 }
```

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

$$(1) \quad 0 \leq n \longrightarrow 1 = (1-1)! \wedge 0 < 1$$
$$(3) \quad p = (c-1)! \wedge 0 < c \wedge \neg \mathit{true} \longrightarrow \mathit{false}$$

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

$$(1) \quad 0 \leq n \longrightarrow 1 = (1-1)! \wedge 0 < 1$$
$$(3) \quad p = (c-1)! \wedge 0 < c \wedge \neg \mathit{true} \longrightarrow \mathit{false}$$

Vereinfacht:

$$(1.1) \quad 0 \leq n \longrightarrow 1 = 0!$$
$$(1.2) \quad 0 \leq n \longrightarrow 0 < 1$$
$$(3) \quad \mathit{false} \longrightarrow \mathit{false}$$

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

$$(1) \quad 0 \leq n \longrightarrow 1 = (1-1)! \wedge 0 < 1$$
$$(3) \quad p = (c-1)! \wedge 0 < c \wedge \neg \mathit{true} \longrightarrow \mathit{false}$$

Vereinfacht:

$$(1.1) \quad 0 \leq n \longrightarrow 1 = 0! \quad \checkmark$$
$$(1.2) \quad 0 \leq n \longrightarrow 0 < 1 \quad \checkmark$$
$$(3) \quad \mathit{false} \longrightarrow \mathit{false} \quad \checkmark$$

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c- 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     //
6     if (c == n) {
7       //
8       return p;
9       //
10      }
11    else {
12      //
13      }
14    //
15    c= c+1;
16    //
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c- 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     //
6     if (c == n) {
7       //
8       return p;
9       //
10      }
11    else {
12      //
13      }
14    //
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c- 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     //
6     if (c == n) {
7       //
8       return p;
9       //
10      }
11    else {
12      //
13      }
14    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     //
6     if (c == n) {
7       //
8       return p;
9       //
10      }
11    else {
12      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
13      }
14    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     //
6     if (c == n) {
7       //
8       return p;
9       // {p = ((c + 1) − 1)! ∧ 0 < c + 1 | \result == n!}
10      }
11    else {
12      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
13      }
14    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     //
6     if (c == n) {
7       // {p = n!}
8       return p;
9       // {p = ((c + 1) − 1)! ∧ 0 < c + 1 | \result == n!}
10      }
11    else {
12      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
13      }
14    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     //
4     p= p*c;
5     // {(c = n ∧ p = n!) ∨ (c ≠ n ∧ p = ((c + 1) − 1)! ∧ 0 < c + 1)}
6     if (c == n) {
7       // {p = n!}
8       return p;
9       // {p = ((c + 1) − 1)! ∧ 0 < c + 1 | \result == n!}
10      }
11    else {
12      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
13      }
14    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     // {(c = n ∧ p · c = n!) ∨ (c ≠ n ∧ p · c = ((c + 1) − 1)! ∧ 0 < c + 1)}
4     p= p*c;
5     // {(c = n ∧ p = n!) ∨ (c ≠ n ∧ p = ((c + 1) − 1)! ∧ 0 < c + 1)}
6     if (c == n) {
7       // {p = n!}
8       return p;
9       // {p = ((c + 1) − 1)! ∧ 0 < c + 1 | \result == n!}
10      }
11    else {
12      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
13      }
14    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
15    c= c+1;
16    // {p = (c − 1)! ∧ 0 < c}
17    }
```

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$
\begin{aligned}
(2) \quad & p = (c-1)! \wedge 0 < c \wedge \mathit{true} \\
\longrightarrow \quad & (c = n \wedge p \cdot c = n!) \\
& \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1)
\end{aligned}
$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{array}{lll} (2) & p = (c-1)! \wedge 0 < c \wedge \mathit{true} \\ & \longrightarrow & (c = n \wedge p \cdot c = n!) \\ & & \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1) \end{array}$$

Neue **Vereinfachungsregel**:

10. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

Damit Vereinfachung:

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{array}{lll} (2) & p = (c-1)! \wedge 0 < c \wedge \mathit{true} \\ & \longrightarrow & (c = n \wedge p \cdot c = n!) \\ & & \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1) \end{array}$$

Neue **Vereinfachungsregel**:

10. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

Damit Vereinfachung:

$$\begin{array}{ll} (2.1) & p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! \\ (2.2) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! \\ (2.3) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c+1 \end{array}$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{array}{lll} (2) & p = (c-1)! \wedge 0 < c \wedge \mathit{true} \\ & \longrightarrow & (c = n \wedge p \cdot c = n!) \\ & & \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1) \end{array}$$

Neue **Vereinfachungsregel**:

10. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

Damit Vereinfachung:

$$\begin{array}{lll} (2.1) & p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! & \checkmark\ ((c-1)! \cdot c = c!) \\ (2.2) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! \\ (2.3) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c+1 \end{array}$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{array}{lll} (2) & p = (c-1)! \wedge 0 < c \wedge \mathit{true} \\ & \longrightarrow & (c = n \wedge p \cdot c = n!) \\ & & \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1) \end{array}$$

Neue **Vereinfachungsregel**:

10. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

Damit Vereinfachung:

$$\begin{array}{lll} (2.1) & p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! & \checkmark\ ((c-1)! \cdot c = c!) \\ (2.2) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! & \checkmark\ ((c-1)! \cdot c = c!) \\ (2.3) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c+1 \end{array}$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{array}{lll} (2) & p = (c-1)! \wedge 0 < c \wedge \mathit{true} \\ & \longrightarrow & (c = n \wedge p \cdot c = n!) \\ & & \vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1) \end{array}$$

Neue **Vereinfachungsregel**:

10. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

Damit Vereinfachung:

$$\begin{array}{lll} (2.1) & p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! & \checkmark\ ((c-1)! \cdot c = c!) \\ (2.2) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! & \checkmark\ ((c-1)! \cdot c = c!) \\ (2.3) & p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c+1 & \checkmark\ (c < c+1) \end{array}$$

Was fällt uns auf?

- Die Invariante ist $p = (c - 1)! \wedge 0 < c$

Was fällt uns auf?

- Die Invariante ist $p = (c-1)! \wedge 0 < c$
- Da fehlt $c - 1 \leq n$ — wie können wir $c - 1 = n$ am Ende beweisen?
- Mit der Schleifenbedingung 1 gilt **jede** Nachbedingung.
- Bei der Rückgabe ist $c == n$ — vereinfacht den Beweis.
- Essenziell: Schleife wird **nicht** verlassen.
- Nachbedigung *false* forciert dass Programm nur mit **return** verlassen wird.

II. Semantik von Funktionsdefinitionen

Semantik von Funktionsdefinitionen

$$[\![.]\!]_{fd} : \mathbf{FunDef} \to \mathbf{Env} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$

Das Denotat einer Funktion ist eine Anweisung, die über den tatsächlichen Werten für die Funktionsargumente parametriert ist.

$$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ ds\ c]\!]^{\Gamma}_{fd}\ v_1, \ldots, v_n =$$
$$[\![\underbrace{(t_1\ p_1\ v_1, t_2\ p_2\ v_2, \ldots, t_n\ p_n\ v_n, ds)}_{\text{Deklarationen}}\ c]\!]^{\Gamma}_{blk}$$

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden ...
  - ... aber was ist die Semantik von Deklarationen?
- Deklarationen **erzeugen** lokale Variablen

Allokation und Deallokation

Systemzustände

- **Basisadressen:** $\mathbf{Addr} \stackrel{def}{=} \mathbb{N}$
- **Locations:** $\mathbf{Loc} \stackrel{def}{=} \mathbf{Addr} \times \mathbb{N}$ (Basisadresse mit Offset)
- Werte: $\mathbf{V} = \mathbb{Z}$ (Einbettung von $\mathbf{C}$ in $\mathbb{Z}$)
- Zustände: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

- Neue Operationen auf dem Systemzustand:

$\nu : \Sigma \to \mathbf{Addr}$	$\nu(\sigma) \notin \mathrm{dom}(\sigma)$	Allokation
$\mathrm{rem} : \Sigma \times \mathbf{Addr} \to \Sigma$	$a \notin \mathrm{dom}(\mathrm{rem}(\sigma, a))$	Deallokation

Semantik von Deklarationen

- Lokale Variablen: Adresse wird der **Umgebung** hinzugefügt

$$\text{local} : (\mathbf{Addr} \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U) \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$
$$\text{local}(b) = \{(\sigma, (\text{rem}(\sigma', \nu(\sigma)), v)) \mid (\sigma, (\sigma', v)) \in b(\nu(\sigma))\}$$

- Neue Variable allokieren, Block ausführen, Variable wieder deallokieren.
  - Ist nicht ganz korrekt: wir müssen neue Variable initialisieren (ggf. mit unbestimmten Wert), ansonsten wird sie nicht Teil von $\text{dom}(\sigma)$, dem Definitionsbereich von σ.
  - Gilt insbesondere für die Funktionsparameter, die mit dem aktuellen Wert des Funktion initialisiert werden.

Semantik von Blöcken

- Blöcke bestehen aus Deklarationen und einer Anweisung c:

$$[\![-]\!]_{blk} : \mathbf{Blk} \to \mathbf{Env} \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$
$$[\![(t\ x; ds)\ c]\!]^{\Gamma}_{blk} = \mathrm{local}(\lambda l.\, [\![ds\ c]\!]^{\Gamma[x \mapsto l]}_{blk})$$
$$[\![c]\!]^{\Gamma}_{blk} = \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![c]\!]^{\Gamma}_{\mathcal{C}}\}$$

- Rekursive Definition (über der Liste der Deklarationen)
- Semantik der Deklationen zusammen mit dem Rumpf.
- Von $[\![c]\!]^{\Gamma}_{\mathcal{C}}$ sind nur **Rückgabezustände** interessant.
  - Kein „fall-through“
  - Was passiert ohne **return** am Ende?

Semantik von Funktionsdefinitionen

- Grundprinzip (mit nur einen Parameter)

$$[\![f(t\ x)\ blk]\!]_{fd} = \lambda v.\, [\![\begin{array}{l} t\ x; \\ x = v; \\ blk]\!] \end{array}$$

Semantik von Funktionsdefinitionen

- Grundprinzip (mit nur einen Parameter)

$$[\![f(t\ x)\ blk]\!]_{fd} = \lambda v.\, [\![\ t\ x;\ x = v;\ blk]\!]$$

- In voller Schönheit:

$$[\![f(t\ x, ps)\ blk]\!]^{\Gamma}_{fd} = \lambda v.\ \mathrm{local}(\lambda l.\, \{(\sigma, (\sigma', r)) \mid (\sigma[l \mapsto v], (\sigma', r)) \in [\![f(ps)\ blk]\!]^{\Gamma[x \mapsto l]}_{fd}\})$$

$$[\![f()\ blk]\!]^{\Gamma}_{fd} = [\![blk]\!]^{\Gamma}_{blk}$$

Arbeitsblatt 10.4: Semantik mit Return

Gegeben folgende Funktion f:

```
int f(int y)
{
 int x;
 x= 7;
 if (y == 0) return x;
 x= x+4;
}
```

Was ist die denotationale Semantik von f?

$$[\![f]\!]^{\Gamma}_{fd} = ?$$

Arbeitsblatt 10.4: Semantik mit Return

Gegeben folgende Funktion f:

```
int f(int y)
{
 int x;
 x= 7;
 if (y == 0) return x;
 x= x+4;
}
```

Was ist die denotationale Semantik von f?

$$[\![f]\!]^{\Gamma}_{fd} = \lambda v.\, \{(\sigma, \ldots) \mid \ldots\}$$

III. Spezifikation und Verifikation von Funktionen

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)
  - Logische Variablen müssen deklariert werden
- Syntaktisch:

$$\textbf{FunSpec} ::= /^{**}\ \textbf{pre}\ \textbf{Assn}\ \textbf{post}\ \textbf{Assn}\ ^{*}/$$

Vorbedingung	**pre** sp;	**Env** $\rightarrow$	**Intprt** $\rightarrow$	$\Sigma \rightarrow \mathbb{B}$
Nachbedingung	**post** sp;	**Env** $\rightarrow$	**Intprt** $\rightarrow$	$(\Sigma \times \mathbf{V}_U) \rightarrow \mathbb{B}$
Rückgabewert	\result			

Gültigkeit von Spezifikationen

- Ziel ist eine **Semantik von Spezifikationen** $[\![.]\!]_{\mathcal{B}sp}$ zu definieren, um damit **semantische Gültigkeit** zu definieren:

$$\begin{aligned}&\Gamma \models f(x_1, \ldots, x_n) \textbf{ pre } P \textbf{ post } Q \ blk\\ \iff &\forall x_1, \ldots, x_n. \Gamma \models \{[\![P]\!]^{\Gamma,I}_{\mathcal{B}sp}\} \ [\![blk]\!]^{\Gamma}_{fd}(x_1, \ldots, x_n) \ \{false \mid [\![Q]\!]^{\Gamma,I}_{\mathcal{B}sp}\}\end{aligned}$$

- Spezifikationen beziehen sich auf **Parameter**, nicht auf lokale Variablen
  - Parameter sind in den Spezifikationen logische Variablen.
- Nicht ganz präzise (*blk* hat keine Parameter, nur *fd*)
- Parameter der Funktion werden zu semantischen Parametern.

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!};
 */
{
 int p;
 int c;

 p= 1;
 c= 1;
 while (c<= n) /** inv p == (c− 1)! ∧ 0≤ c ∧ c−1≤ n; */ {
    p= p*c;
    c= c+1;
    }
 return p;
}
```

Arbeitsblatt 10.5: Ganzzahlige Division

Spezifiziert die ganzzahlige Division:

```
1  int div(int a, b)
2  /**
3    pre ???
4    post ???
5  {
6   r= a;
7   q= 0;
8   while (b <= r) {
9     r= r-b;
10    q= q+1;
11    }
12  return q;
13 }
```

Beispiel: Fakultät (Revisited)

```
int fac(int n)
/** int N;
    pre  0 ≤ n;
    post \result == n!};
 */
{
 int p;

 /** { 0 ≤ n ∧ N == n } */
 p= 1;
 while (n< 0)
    /** inv n!* p == N!
          ∧ n≤ N ∧ 0 ≤ n; */ {
    p= p* n;
    n= n−1;
    }
 return p;
}
```

- Problem:
  Parameter $n \neq$ lokaler Variable n
  - In der Spezifikation: Parameter
  - Im Rumpf: Programmvariable
- Deshalb: logische Variable N, deren Wert am Anfang mit n gleichgesetzt wird.

Verifikation

- Ziel: Regel zur Verifikation einer annotierten Funktion.
- Unterscheidung des Parameterwerts x von der lokalen Variablen x durch x **@pre**
  - x **@pre** ist eine logische Variable (Erweiterung von **ldt** für logische Variablen)
  - In der Spezifikation wird x durch x **@pre** ersetzt.
  - Im Vorzustand gilt x **@pre** $= x$.
  - Verhindert, dass der **Parameter** x bei der Zuweisung substituiert wird.
- Verifikationsregel:

$$\frac{\vdash \{P \land x_i = x_i \,\texttt{@pre}\}\ c\ \{\mathit{false} \mid Q[x_i \,\texttt{@pre}\,/x_i]\}}{\vdash f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ c\}}$$

Verifikationsbedingungen

- Berechnung von awp und wvc:

$$\text{awp}(f(x_1,\ldots,x_n)/\text{** pre } P \text{ post } Q \text{ */ } \{ds\ c\}) \stackrel{def}{=} \text{awp}(c, \mathit{false}, Q[x_i\ \textbf{@pre}\ /x_i])$$

$$\begin{aligned}\text{wvc}((x_1,\ldots,x_n)/\text{** pre } P \text{ post } Q \text{ */ } \{ds\ c\}) \stackrel{def}{=} &\{P \wedge x_i = x_i\ \textbf{@pre} \Longrightarrow P'\} \\ &\cup\ \text{wvc}(c, \mathit{false}, Q[x_i\ \textbf{@pre}\ /x_i])\end{aligned}$$

$$P' \stackrel{def}{=} \text{awp}(c, \mathit{false}, Q[x_i\ \textbf{@pre}\ /x_i])$$

- Funktionsaufrufe in c (insbesondere Rekursion) wird noch nicht behandelt.

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

```
int inc(int x)
/** pre true;
    post x< \result ; */
{ //
  x= x+1;
  //
  return x;
  //
}
```

- Verifikationsbedingung:

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

```
int inc(int x)
/** pre true;
    post x< \result ; */
{ //
  x= x+1;
  //
  return x;
  // {false | x @pre < \result}
}
```

- Verifikationsbedingung:

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

```
int inc(int x)
/** pre true;
    post x< \result ; */
{ //
  x= x+1;
  // {x @pre < x}
  return x;
  // {false | x @pre < \result}
}
```

- Verifikationsbedingung:

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

```
int inc(int x)
/** pre true;
    post x< \result ; */
{ // {x @pre < x + 1}
  x= x+1;
  // {x @pre < x}
  return x;
  // {false | x @pre < \result}
}
```

- Verifikationsbedingung:

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

```
int inc(int x)
/** pre true;
    post x< \result ; */
{ // {x @pre < x + 1}
  x= x+1;
  // {x @pre < x}
  return x;
  // {false | x @pre < \result}
}
```

- Verifikationsbedingung:

$$(1)\ \mathit{true} \wedge x\ \mathbf{@pre} = x \Longrightarrow x\ \mathbf{@pre} < x + 1$$

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

```
int inc(int x)
/** pre true;
    post x< \result ; */
{ // {x @pre < x + 1}
  x= x+1;
  // {x @pre < x}
  return x;
  // {false | x @pre < \result}
}
```

- Verifikationsbedingung:

$$(1)\ x < x + 1 \quad \checkmark$$

Zusammenfassung

- Die **return**-Anweisung bricht den sequentiellen Kontrollfluss, mit weitreichenden Folgen für Semantik und Hoare-Kalkül.
- Die Semantik wird um einen Rückgabezustand erweitert, der Hoare-Kalkül um eine Rückgabespezfikation.
- Funktionen können wir mit Vor-/Nachbedingungen spezifizieren, und mit den Regeln des erweiterten Hoare-Kalküls verifizieren.
- Die Funktionsparameter sind für die Spezifikation unveränderliche logische Variablen, und innerhalb des Funktionsrumpfes veränderliche lokale Variablen.
- Um das auseinderzuhalten führen wie die Notation x **@pre** ein.
- Problem: Wie behandeln wir eigentlich **Funktionsaufrufe**?

Korrekte Software: Grundlagen und Methoden

Vorlesung 11 vom 20.06.22

Funktionen und Prozeduren II

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

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

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe
- Prozeduraufrufe (mit Zuweisung eines Rückgabewertes)

$$
\begin{array}{rll}
\textbf{Aexp} & a ::= & \mathbf{Z} \mid \mathbf{C} \mid \textbf{Lexp} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \\
\textbf{Bexp} & b ::= & \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,||\, b_2 \\
\textbf{Exp} & e := & \textbf{Aexp} \mid \textbf{Bexp} \\
\textbf{Stmt} & c ::= & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
 & & \mid\ \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ a\ {*}/\ c \mid /{**}\ \{a\}\ {*}/ \\
 & & \mid\ \textbf{Idt}(a^*) \\
 & & \mid\ l = \textbf{Idt}(a^*) \\
 & & \mid\ \textbf{return}\ a^?
\end{array}
$$

Zur Erinnerung: Semantik von Funktionsdefinitionen

$$[\![.]\!]_{fd} : \mathbf{FunDef} \to \mathbf{Env} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$

- Das Denotat einer Funktion ist eine Anweisung, die über den tatsächlichen Werten für die Funktionsargumente parametriert ist.

  $$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ ds\ c]\!]_{fd}\ \Gamma\ v_1, \ldots, v_n = [\![(t_1\ p_1\ v_1, t_2\ p_2\ v_2, \ldots, t_n\ p_n\ v_n, ds)\ c]\!]_{blk}\ \Gamma$$

- **Aufruf** der Funktion $f(e_1, \ldots, e_n)$ mit Argumenten $e_1, \ldots, e_n$:
  - **Auswertung** der Argumente $v_i = [\![e_i]\!]_{\mathcal{A}}$
  - Einsetzen in die **Semantik** $[\![f]\!]_{fd}(v_1, \ldots, v_n)$

Seiteneffekte bei Funktionsaufrufe

- Seiteneffekte:
  - Funktionen mit Seiteneffekten in zusammengesetzten Ausdrücken sind problematisch
  - In Java ist die Auswertungsreihenfolge fest (links nach rechts)
  - In C unspezifiziert (!)
- Deshalb keine Funktionen in zusammengesetzten Ausdrücken.
- Funktionsaufrufe nur in zwei Formen:
  - Als reine Prozeduren ($\textbf{ldt}(a^*)$) vom Typ **void**
  - Als direkte Zuweisung ($l = \textbf{ldt}(a^*)$) des Rückgabewertes
- Call by name, call by value, call by reference. . . ?

Arbeitsblatt 11.1: Funktionsaufrufe

Wie werden Parameter in folgenden Programmiersprachen übergeben?

- **C:**
- **Java**:
- **Haskell**:
- **Python**:
- **Other:** (specify)

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von f dem Bezeichner f zuordnen.
- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- Deshalb muss die **Umgebung** erweitert werden:

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup \mathbf{Loc}$$

- Wir haben hier den selben einen **Namensraum** für Funktionen und Variablen.

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von f dem Bezeichner f zuordnen.
- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- Deshalb muss die **Umgebung** erweitert werden:

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup (\mathbf{Loc} + (\mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)))$$

- Wir haben hier den selben einen **Namensraum** für Funktionen und Variablen.

Semantik von Funktionsaufrufen

- Gegegeben Funktionsbezeichner f, Semantik ist

$$\Gamma(f) = \{(\underbrace{(v_1, \ldots, v_n)}_{\textit{Parameterwerte}}, (\underbrace{\sigma}_{\textit{Anfangszustand}}, \underbrace{(\sigma', a)}_{\textit{Endzustand,Rückgabewert}}))\}$$

- Damit:

$$[\![f(t_1, \ldots, t_n)]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, \sigma') \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]^{\Gamma}_{\mathcal{A}}\}$$
$$[\![x = f(t_1, \ldots, t_n)]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, \sigma'[x \mapsto a]) \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]^{\Gamma}_{\mathcal{A}}\}$$

- Aufruf einer Prozedur $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}$ ignoriert Rückgabewert

Semantik von Funktionsaufrufen

- Gegegeben Funktionsbezeichner f, Semantik ist

$$\Gamma(f) = \{(\underbrace{(v_1, \ldots, v_n)}_{\textit{Parameterwerte}}, (\underbrace{\sigma}_{\textit{Anfangszustand}}, \underbrace{(\sigma', a)}_{\textit{Endzustand, Rückgabewert}}))\}$$

- Damit:

$$[\![f(t_1, \ldots, t_n)]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, \sigma') \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]^{\Gamma}_{\mathcal{A}}\}$$
$$[\![x = f(t_1, \ldots, t_n)]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, \sigma'[x \mapsto a]) \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]^{\Gamma}_{\mathcal{A}}\}$$

- Aufruf einer Prozedur $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}$ ignoriert Rückgabewert
  - Somit: Kombination mit Zuweisung
- Wir modellieren nur call-by-value.
  - C kennt nur call by value, allerdings sind Referenzen auch Werte (kommt noch)

Umgebung für den Kalkül

- Für Funktionsaufrufe gibt es eine **Umgebung**:

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup (\mathbf{Loc} + (\mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)))$$

- Deshalb muss für den Kalkül eine **Umgebung** Δ Funktionsbezeichnern ihre **Spezifikation** (Vor- und Nachbedingung, sowie Parameter) zuordnen:

$$\mathbf{Env}_{\mathsf{fun}} = \mathbf{Idt} \rightharpoonup (\mathbf{Idt}^N \times \mathbf{Assn} \times \mathbf{Assn})$$

- $\Delta(f) = \forall x_1, \ldots, x_n.\, (P, Q)$, für $f(x_1, \ldots, x_n)$/** pre P post Q */
- Korrektheit gilt immer nur im **Kontext** einer Umgebung, dadurch kann jede Funktion separat verifiziert werden (**Modularität**).
  - Umgebung wird zusätzliches Argument der Regeln.
  - Notation: $\Delta \vdash \{P\}\, c\, \{Q \mid Q_R\}$.

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

$$\frac{\Delta(f) = \forall x_1, \ldots, x_n.\,(P, Q)}{\Delta \vdash \{P[t_i/x_i]\}\; l = f(t_1, \ldots, t_n)\; \{Q[t_i/x_i][l/\backslash\textbf{result}] \mid Q_R\}}$$

- Δ muss f mit der Vor-/Nachbedingung P, Q enthalten
- In P und Q werden Parameter x_i durch Argumente t_i ersetzt.
- $\backslash\textbf{result}$ in Q wird durch l ersetzt

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  //
16  r = fac(x- 1);
17  //
18  //
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  //
16  r = fac(x- 1);
17  //
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  //
16  r = fac(x- 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**$!$

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x\,\texttt{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\texttt{@pre}!) \vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x = 0 \longrightarrow 1 = x\,\texttt{@pre}!$

(1.2) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x \neq 0 \longrightarrow 0 \leq x - 1$

(2) $r = (x-1)! \longrightarrow r \cdot x = x\,\texttt{@pre}!$

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x$ **@pre**$!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**$!$

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x$ **@pre**$!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**$!$

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x$ **@pre**$!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**$!$

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@pre**!

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben
- Rekursion benötigt keine Extrabehandlung
  - Termination von rekursiven Funktionen wird extra gezeigt

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben
- Rekursion benötigt keine Extrabehandlung
  - Termination von rekursiven Funktionen wird extra gezeigt
- Der Aufruf einer Funktion **ersetzt** die momentane Nachbedingung — das ist ein Problem!

Frame Rule

- Konstanzregel (Rule of Constancy):

$$\frac{\Delta \vdash \{P\}\, c\, \{Q \mid Q_R\}}{\Delta \vdash \{P \wedge R\}\, c\, \{Q \wedge R \mid Q_R\}}$$

- Nebenbedingung:
  - c verändert keine Variablen in R, **oder**
  - für **keine** der Programm-Variablen x, die in R vorkommen, gibt es eine Zuweisung $x = \ldots$ in c
- Das ist eine **neue Regel**, die **bewiesen** werden muss.
- Schwierig zu handhaben bei Rückwärtsrechnung: R muss **annotiert** werden.

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe mit Zuweisung eines Rückgabewertes

$$
\begin{aligned}
\textbf{Stmt}\ c ::= \ & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
\mid\ & \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ a\ {*}/\ c \mid /{**}\ \{a\}\ {*}/ \\
\mid\ & \textbf{Idt}(a^*) \\
\mid\ & /{**}\,\textbf{const}\ R\ {*}/\ l = \textbf{Idt}(a^*) \\
\mid\ & \textbf{return}\ a^?
\end{aligned}
$$

Approximative schwächste Vorbedingung & Verifikationsbedingung für Funktionsaufrufe

$$\text{Sei } \Delta(f) = \forall x_1, \ldots, x_n.\, (P, Q)$$

$$\text{awp}(\Delta, /{**}\ \textbf{const}\ \ R\ {*}/ l = f(t_1, \ldots, t_n), U, U_R) \stackrel{def}{=} R \wedge P[t_i/x_i] \quad \text{wenn } l \notin FV(R)$$

$$\text{wvc}(\Delta, /{**}\ \textbf{const}\ \ R\ {*}/ l = f(t_1, \ldots, t_n), U, U_R) \stackrel{def}{=} \{R \wedge Q[t_i/x_i][l/\backslash\textbf{result}] \longrightarrow U\} \quad \text{wenn } l \notin FV(R)$$

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@ pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x\,\texttt{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\texttt{@pre}!) \vee (x \neq 0 \wedge 0 \leq x - 1 \wedge x = x\,\texttt{@pre})$

(1.1) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x = 0 \longrightarrow 1 = x!$

(1.2) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x \neq 0 \longrightarrow 0 \leq x - 1$

(1.3) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x \neq 0 \longrightarrow x = x\,\texttt{@pre}$

(2) $x = x\,\texttt{@pre} \wedge r = (x-1)! \longrightarrow r \cdot x = x\,\texttt{@pre}!$

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@ pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1$
$\wedge x = x$ **@pre**$)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$

(1.3) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow x = x$ **@pre**

(2) $x = x$ **@pre** $\wedge r = (x - 1)!$
$\longrightarrow r \cdot x = x$ **@pre**$!$

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@ pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1$
$\wedge x = x$ **@pre**$)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(1.3) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow x = x$ **@pre**

(2) $x = x$ **@pre** $\wedge r = (x - 1)!$
$\longrightarrow r \cdot x = x$ **@pre**$!$

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x\,\texttt{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\texttt{@pre}!) \vee (x \neq 0 \wedge 0 \leq x - 1 \wedge x = x\,\texttt{@pre})$

(1.1) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x = 0 \longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x \neq 0 \longrightarrow 0 \leq x - 1$ ✓

(1.3) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x \neq 0 \longrightarrow x = x\,\texttt{@pre}$ ✓

(2) $x = x\,\texttt{@pre} \wedge r = (x-1)! \longrightarrow r \cdot x = x\,\texttt{@pre}!$

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@ pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1$
$\wedge x = x$ **@pre**$)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(1.3) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow x = x$ **@pre** ✓

(2) $x = x$ **@pre** $\wedge r = (x - 1)!$
$\longrightarrow r \cdot x = x$ **@pre**$!$ ✓

Arbeitsblatt 11.2: Fakultät endrekursiv

Hier nochmal die Fakultät (endrekursiv und buggy):

```
int factorial(int n)
/** pre ???;
    post ???; */
{
  int f;

  f= fact(0, n);
  return f;

}

int fact(int acc, int n)
/** pre ???;
    post ???; */
{
  int r;

  if (n == 0) return 1;
  r= fact(acc* n, n-1);
  return r;
}
```

1. Annotiert das Programm mit Vor/Nachbedingungen.
2. Findet und berichtigt die Fehler.
3. Berechnet die Verifikationsbedingungen.
4. Beweist die Verifikationsbedingungen.

Zusammenfassung

- Funktionen sind **zentrales Modularisierungskonzept**
- Behandlung von Funktionen erfordert **vielfältige Erweiterungen**
- Erweiterung der **Semantik:**
  - Erweiterung der Semantik um **Rückgabezustand** $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$
  - Die Semantik einer Funktion ist **parametrisiert** $\mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$
- Erweiterung der **Spezifikationen:**
  - Spezifikation von Funktionen: **Vor-/Nachzustand** statt logischer Variablen
- Erweiterung des **Hoare-Kalküls**:
  - **Gesonderte Nachbedingung** für Rückgabewert/Endzustand
  - Aufruf einer Funktion **ersetzt** Vor/Nachbedingung, daher **Framing**
- **Einschränkungen**: nur call-by-value
- Fazit: **ohne Referenzen** sind Funktionen wenig brauchbar

Korrekte Software: Grundlagen und Methoden

Vorlesung 12 vom 27.06.24

Referenzen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Organisatorisches

- Prüfungstermine:
- 04.07.2024, 16:00
- August/Anfang September

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Motivation

- Warum Referenzen?
  - Nötig für *call by reference*
  - Funktionen können sonst nur **globale** Seiteneffekte haben
  - Effizienz
- Kurze Begriffsklärung:
  - Referenzen: getypt, eingeschränkte Arithmetik
  - Zeiger: ungetypt, Zeigerarithmetik

Refenzen in C

- Pointer in C (“pointer type”):
  - Schwach getypt (**void** $*$ kompatibel mit allen Zeigertypen, Typumwandlung)
  - Eingeschränkte Zeigerarithmetik (Addition, Subtraktion)
  - Felder werden durch Zeigerarithmetik implementiert
- Pointer sind *first-class-values*
- C-Standard läßt das Speichermodell relativ offen
  - Repräsentation von Objekten

Referenzen in anderen Sprachen

- Java:
  - (Fast) alles ist eine Referenz
  - Schwach getypt (Subtyping und Typumwandlung)
- Haskell, SML, OCaml:
  - Stark getypt (typsicher)
- Scriptsprachen (Python, Ruby):
  - Ähnlich Java

Ausdrücke

- Neue Typen: Zeiger (Pointer)

$$\textbf{Type}\ \ t ::= \textbf{void} \mid \textbf{char} \mid \textbf{int} \mid *t \mid \textbf{struct}\ \ \textbf{Idt}^{?}\ \{\textbf{Decl}^{+}\} \mid t\ \ \textbf{Idt}[a]$$

- Neue Operatoren: Addressoperator (&) und Derefenzierung ($*$)

$$\textbf{Lexp}\ \ l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \mid *a$$

$$\textbf{Aexp}\ \ a ::= \textbf{Z} \mid \textbf{C} \mid l \mid \&l \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$$

$$\textbf{Bexp}\ \ b ::= \ldots$$

$$\textbf{Exp}\ \ e := \textbf{Aexp} \mid \textbf{Bexp}$$

$$\textbf{Stmt}\ \ c ::= \ldots$$

Das Problem mit Zeigern

- **Aliasing:** Verschiedene Bezeichner (**Lexp**) für die gleiche Lokation $l \in$ **Loc**

```
int a;
int *x;

x= &a;
a= 99;
// {a = 99}
*x= 7;                    // (*)
// {a = 7}
```

- Wert von a ändert sich **ohne dass a erwähnt** wird.
- An der Stelle (*) zwei Bezeichner für die gleiche Lokation: a und *x
- Großes Problem für Semantik und Hoare-Kalkül.
- Modellierung der Zuweisung durch Substitution nicht mehr möglich

Erweiterung des Zustandmodells

- Bisheriger Zustand $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$ mit
  - **Locations**: $\mathbf{Loc} = \mathbf{Addr} \times \mathbb{N}$ (siehe Vorlesung 8)
  - Werte: $\mathbf{V} = \mathbb{Z}$
- Ansatz reicht nicht mehr.
- Locations müssen auch Werte sein:

$$\mathbf{V} \stackrel{def}{=} \mathbb{Z} + \mathbf{Loc}$$

und somit sind Zustände $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup (\mathbb{Z} + \mathbf{Loc})$

Erweiterung der Semantik

- Denotation für **L-Ausdrücke**:

$$[\![-]\!]_{\mathcal{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$[\![x]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, \langle \Gamma(x), 0 \rangle) \mid x \in \Gamma\}$$

$$[\![m[e]]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + n \cdot \mathit{sizeof}(t)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, (\sigma, n) \in [\![e]\!]^{\Gamma}_{\mathcal{A}}, \Gamma \vdash m : t[x]\}$$

$$[\![m.a]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + \mathit{off}_t(a)) \mid \Gamma \vdash m : t, (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}\}$$

$$[\![*e]\!]^{\Gamma}_{\mathcal{L}} = [\![e]\!]^{\Gamma}_{\mathcal{A}} \quad \text{mit } \Gamma \vdash e : *t$$

- Denotation für **Ausdrücke** ($m \in \mathbf{Lexp}$):

$$[\![m]\!]^{\Gamma}_{\mathcal{A}} = \{(\sigma, \sigma(l)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, l \in \mathit{Dom}(\sigma)\}$$

$$[\![\&m]\!]^{\Gamma}_{\mathcal{A}} = \{(\sigma, l) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, l \in \mathrm{dom}(\sigma)\}$$

- Es ist wichtig, die semantischen Funktionen $[\![.]\!]_{\mathcal{A}}$ und $[\![.]\!]_{\mathcal{L}}$ zu unterscheiden!

Erweiterung der Semantik

- Denotation für **L-Ausdrücke**:

$$[\![-]\!]_{\mathcal{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$[\![x]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, \langle \Gamma(x), 0 \rangle) \mid x \in \Gamma\}$$

$$[\![m[e]]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + n \cdot \mathit{sizeof}(t)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, (\sigma, n) \in [\![e]\!]^{\Gamma}_{\mathcal{A}}, \Gamma \vdash m : t[x]\}$$

$$[\![m.a]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + \mathit{off}_t(a)) \mid \Gamma \vdash m : t, (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}\}$$

$$[\![*e]\!]^{\Gamma}_{\mathcal{L}} = [\![e]\!]^{\Gamma}_{\mathcal{A}} \quad \text{mit } \Gamma \vdash e : *t$$

- Denotation für **Ausdrücke** ($m \in \mathbf{Lexp}$):

$$[\![m]\!]^{\Gamma}_{\mathcal{A}} = \{(\sigma, \sigma(l)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, l \in \mathit{Dom}(\sigma)\}$$

$$[\![\&m]\!]^{\Gamma}_{\mathcal{A}} = [\![m]\!]^{\Gamma}_{\mathcal{L}}$$

- Es ist wichtig, die semantischen Funktionen $[\![.]\!]_{\mathcal{A}}$ und $[\![.]\!]_{\mathcal{L}}$ zu unterscheiden!

Beispiel

```
int a, *x;

*x= 5*a;
```

Gegeben folgende Werte, was ist die Semantik?

$$\Gamma \stackrel{def}{=} \langle a \mapsto 1, x \mapsto 2 \rangle$$

$$\sigma_1 \stackrel{def}{=} \langle (1,0) \mapsto 10, (2,0) \mapsto (1,0) \rangle$$

$$[\![*x = 5 * a]\!]^{\Gamma}_{\mathcal{C}}(\sigma_1) = \sigma_1[[\![*x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1) \mapsto [\![5 * a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)]$$

Beispiel

```
int a, *x;

*x= 5*a;
```

Gegeben folgende Werte, was ist die Semantik?

$$\Gamma \stackrel{def}{=} \langle a \mapsto 1, x \mapsto 2 \rangle$$

$$\sigma_1 \stackrel{def}{=} \langle (1,0) \mapsto 10, (2,0) \mapsto (1,0) \rangle$$

$$\begin{aligned} [\![*x = 5 * a]\!]^{\Gamma}_{\mathcal{C}}(\sigma_1) &= \sigma_1[[\![*x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1) \mapsto [\![5 * a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\ &= \sigma_1[[\![x]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \mapsto [\![5]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \cdot [\![a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \end{aligned}$$

Beispiel

```
int a, *x;

*x= 5*a;
```

Gegeben folgende Werte, was ist die Semantik?

$$\Gamma \stackrel{def}{=} \langle a \mapsto 1, x \mapsto 2 \rangle$$

$$\sigma_1 \stackrel{def}{=} \langle (1,0) \mapsto 10, (2,0) \mapsto (1,0) \rangle$$

$$\begin{aligned}
[\![*x = 5 * a]\!]^{\Gamma}_{\mathcal{C}}(\sigma_1) &= \sigma_1[[\![*x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1) \mapsto [\![5 * a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\
&= \sigma_1[[\![x]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \mapsto [\![5]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \cdot [\![a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\
&= \sigma_1[\sigma_1(\underbrace{[\![x]\!]^{\Gamma}_{\mathcal{L}}}_{\Gamma(x)}(\sigma_1)) \mapsto 5 \cdot \sigma_1(\underbrace{[\![a]\!]^{\Gamma}_{\mathcal{L}}}_{\Gamma(a)})]
\end{aligned}$$

Beispiel

```
int a, *x;

*x= 5*a;
```

Gegeben folgende Werte, was ist die Semantik?

$$\Gamma \stackrel{def}{=} \langle a \mapsto 1, x \mapsto 2 \rangle$$

$$\sigma_1 \stackrel{def}{=} \langle (1,0) \mapsto 10, (2,0) \mapsto (1,0) \rangle$$

$$\begin{aligned}
[\![*x = 5 * a]\!]^{\Gamma}_{\mathcal{C}}(\sigma_1) &= \sigma_1[[\![*x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1) \mapsto [\![5 * a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\
&= \sigma_1[[\![x]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \mapsto [\![5]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \cdot [\![a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\
&= \sigma_1[\sigma_1(\underbrace{[\![x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1)}_{\Gamma(x)}) \mapsto 5 \cdot \sigma_1(\underbrace{[\![a]\!]^{\Gamma}_{\mathcal{L}}}_{\Gamma(a)})] \\
&= \sigma_1[\sigma_1(2,0) \mapsto 5 \cdot \sigma_1(1,0)]
\end{aligned}$$

Beispiel

```
int a, *x;

*x= 5*a;
```

Gegeben folgende Werte, was ist die Semantik?

$$\Gamma \stackrel{def}{=} \langle a \mapsto 1, x \mapsto 2 \rangle$$

$$\sigma_1 \stackrel{def}{=} \langle (1,0) \mapsto 10, (2,0) \mapsto (1,0) \rangle$$

$$\begin{aligned}
[\![*x = 5 * a]\!]^{\Gamma}_{\mathcal{C}}(\sigma_1) &= \sigma_1[[\![*x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1) \mapsto [\![5 * a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\
&= \sigma_1[[\![x]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \mapsto [\![5]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1) \cdot [\![a]\!]^{\Gamma}_{\mathcal{A}}(\sigma_1)] \\
&= \sigma_1[\sigma_1(\underbrace{[\![x]\!]^{\Gamma}_{\mathcal{L}}(\sigma_1)}_{\Gamma(x)}) \mapsto 5 \cdot \sigma_1(\underbrace{[\![a]\!]^{\Gamma}_{\mathcal{L}}}_{\Gamma(a)})] \\
&= \sigma_1[\sigma_1(2,0) \mapsto 5 \cdot \sigma_1(1,0)] \\
&= \sigma_1[(1,0) \mapsto 5 \cdot 10] = \langle (1,0) \mapsto 50, (2,0) \mapsto (1,0) \rangle
\end{aligned}$$

Arbeitsblatt 12.1: Aliasing-Semantik

```
{
  int a;
  int *x;

  // (0)
  x= &a;
  a= 99;
  // (1)
  *x= 7;
  // (2)
}
```

Berechnet mit Hilfe der formalen Semantik:

1. Die Umgebung Γ an der Stelle (0)
2. Den Zustand σ_1 an der Stelle (1)
3. Den Zustand σ_2 an der Stelle (2)

Arbeitsblatt 12.2: Pop-Quiz

Gegeben folgende Funktionen:

```
int f(int *x)
{
  *x= *x+1;
  return *x;
}
```

Was ist der Wert von f(&x)?

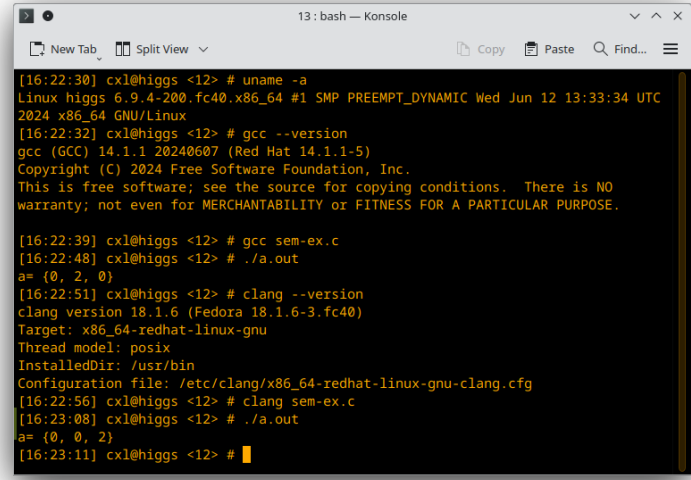
1. f(&x) == 1
2. f(&x) == 2

```
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}

Ergebnis des Pop-Quiz



Korrektur der Semantik

- Das Pop-Quiz demonstriert den **Nichtdeterminismus** der C-Semantik.
  - Es ist **unbestimmt**, ob die linke oder rechte Seite der Zuweisung zuerst ausgewertet wird.
  - In C ist die Zuweisung ein binärer Operator, keine Anweisung.
- Unsere Teilsprache soll den **deterministischen** Teil von C umfassen.
- Deshalb beim Funktionsaufruf mit Zuweisung links nur **zustandsfreie** Ausdrücke l:

$$\forall \sigma, \sigma'. [\![l]\!]^{\Gamma}_{\mathcal{L}}(\sigma) = [\![l]\!]^{\Gamma}_{\mathcal{L}}(\sigma')$$

- Gilt insbesondere für Bezeichner $l \in \mathbf{ldt}$, deshalb:

  Stmt $c ::= \ldots \mid \mathbf{ldt} = \mathbf{ldt}(a^*) \mid \ldots$

- Bei anderen Zuweisungen $l = e$ sind beide Seiten **zustandsabhängig**, aber frei von **Seiteneffekten**.

Arbeitsblatt 12.3: Kurze Semantik

Gegeben folgende Deklarationen:

```
struct p { int x;          int a;          struct p *q;
           int y; } p;
```

mit folgender Umgebung und Zustand

$$\Gamma \stackrel{def}{=} \langle p \mapsto 2, a \mapsto 3, q \mapsto 4 \rangle, \sigma_1 \stackrel{def}{=} \langle (2,0) \mapsto 3, (2,1) \mapsto 5, (3,0) \mapsto 7, (4,0) \mapsto (2,0) \rangle$$

Berechnet die denotationale Semantik

$$[\![a = a + (*q).x]\!]^{\Gamma}_{C}(\sigma_1) = ?$$

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.
- Explizite Zustandsprädikate modellieren das Lesen und die Änderung des Zustandes **explizit** mit Operationen read und upd.
- Vereinfachungsregeln analog zur Substitution.

Explizite Zustandsprädikate

- Enthalten keine $*$ oder &, und unterscheiden **strikt** zwischen **Lexp** und **Aexp**.
- Erweiterung von **Aexpv** um read, neue Sorte **State** mit Operation upd:

$$\textbf{Assn}_s \quad b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b2 \mid \ldots$$

$$\textbf{Lexp}_s \quad l ::= \ldots \mid \sout{*a}$$

$$\textbf{Aexp}_s \quad a ::= \mathrm{read}(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid \sout{l} \mid \sout{\&l} \mid \ldots \mid \ldots$$

$$\textbf{State} \quad S ::= s \mid \mathrm{upd}(S, l, e)$$

- Zustandsvariable s (logische Variable): aktueller Zustand
- Im Gegensatz zur Semantik rechnen wir mit **symbolischen Namen**

Gleichungen für explizite Zustandsprädikate

- Für die Operationen read, updgelten folgende Gleichungen:

$$\forall l, v, \sigma.\ \mathsf{read}(\mathsf{upd}(\sigma, l, v), l) = v$$
$$\forall l, m, v, \sigma.\ l \neq m \Longrightarrow \mathsf{read}(\mathsf{upd}(\sigma, l, v), m) = \mathsf{read}(\sigma, m)$$
$$\forall l, v, w, \sigma.\ \mathsf{upd}(\mathsf{upd}(\sigma, l, v), l, w) = \mathsf{upd}(\sigma, l, w)$$
$$\forall l, m, v, w, \sigma.\ l \neq m \Longrightarrow \mathsf{upd}(\mathsf{upd}(\sigma, l, v), m, w) = \mathsf{upd}(\mathsf{upd}(\sigma, m, w), l, v)$$

- Diese Gleichungen sind **vollständig.**

Semantik expliziter Zustandsprädikate

- Semantik der expliziten Zustandsprädikate:

$$[\![.]\!]_{\mathcal{B}sp} : \mathbf{Env} \to \mathbf{Assn}_s \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbb{B}$$
$$[\![.]\!]_{\mathcal{A}sp} : \mathbf{Env} \to \mathbf{Aexp}_s \rightharpoonup (\Sigma \times \mathbf{V}_U) \rightharpoonup \mathbf{V}$$
$$[\![.]\!]_{\mathcal{L}sp} : \mathbf{Env} \to \mathbf{Lexp}_s \rightharpoonup (\Sigma \times \mathbf{V}_U) \rightharpoonup \mathbf{Loc}$$

Hoare-Triple

- Floyd-Hoare-Tripel:

$$\Gamma \models \{P\}\, c\, \{Q \mid R\} \text{ mit } P, Q, R \in \mathbf{Assn} \text{ Prädikate}$$

- Floyd-Hoare-Kalkül:

$$\Delta \vdash \{P\}\, c\, \{Q \mid R\} \text{ mit } P, Q, R \in \mathbf{Assn}_s \text{ explizite Zustandsprädikate}$$

- Wir brauchen also eine Übersetzung **Assn** nach $\mathbf{Assn}_s$ welche die Semantik erhält:

$$(-)^\dagger : \mathbf{Lexp} \to \mathbf{Lexp}_s \qquad (-)^\# : \mathbf{Aexpv} \to \mathbf{Aexp}_s \qquad (-)^* : \mathbf{Assn} \to \mathbf{Assn}_s$$
$$\forall l \in \mathbf{Lexp}.\ [\![l]\!]^\Gamma_{\mathcal{L}} = [\![l^\dagger]\!]^\Gamma_{\mathcal{L}sp} \qquad \forall a \in \mathbf{Aexpv}.\ [\![a]\!]^\Gamma_{\mathcal{A}} = [\![a^\#]\!]^\Gamma_{\mathcal{A}sp} \qquad \forall a \in \mathbf{Assn}.\ [\![a]\!]^\Gamma_{\mathcal{B}} = [\![a^*]\!]^\Gamma_{\mathcal{B}sp}$$

Konversion in explizite Zustandsprädikaten

Alte Zuweisungsregel

$$\overline{\Delta \vdash \{Q[e/x]\}\, x = e\, \{Q \mid R\}}$$

Umwandlung von Q, x, e in Zustandsprädikate:

- Eine **Lexp** l auf der rechten Seite e wird durch $\mathrm{read}(\sigma, l)$ ersetzt.
- & dient lediglich dazu, diese Konversion zu **verhindern**.
- $*$ **erzwingt** diese Konversion, auch auf der linken Seite x.
- Beispiel: $*a = *\&b;$

Formal: Konversion in Zustandsprädikate

$$(-)^{\dagger} : \mathbf{Lexp} \to \mathbf{Lexp}_s$$

$$i^{\dagger} = i \quad (i \in \mathbf{Idt})$$

$$l.id^{\dagger} = l^{\dagger}.id$$

$$l[e]^{\dagger} = l^{\dagger}[e^{\#}]$$

$$*l^{\dagger} = l^{\#}$$

$$(-)^{\#} : \mathbf{Aexp} \to \mathbf{Aexp}_s$$

$$e^{\#} = \mathrm{read}(s, e^{\dagger}) \quad (e \in \mathbf{Lexp})$$

$$n^{\#} = n$$

$$v^{\#} = v \quad (v \text{ logische Variable})$$

$$(\&e)^{\#} = e^{\dagger}$$

$$(e_1 + e_2)^{\#} = {e_1}^{\#} + {e_2}^{\#}$$

$$\backslash\mathbf{result}^{\#} = \backslash\mathbf{result}$$

$$(-)^{*} : \mathbf{Assn} \to \mathbf{Assn}_s$$

$$(\forall x.\, b)^{*} = \forall x.\, b^{*}$$

$$(b_1 \,\&\&\, b_2)^{*} = {b_1}^{*} \,\&\&\, {b_2}^{*}$$

$$(a_1 == a_2)^{*} = {a_1}^{\#} == {a_2}^{\#}$$

$$\vdots$$

Angepasste Regeln des Hoare-Kalküls

$$\frac{}{\Delta \vdash \{Q[\mathrm{upd}(s, x^\dagger, e^\#)/s]\}\, x = e\, \{Q \mid R\}}$$

- Beispiel 1: $(*x == 3)^*$
$$\begin{aligned} &= \mathrm{read}(s, (*x)^\dagger) = 3 \\ &= \mathrm{read}(s, x^\#) = 3 \\ &= \mathrm{read}(s, \mathrm{read}(s, x)) = 3 \end{aligned}$$

- Beispiel 2: $\Delta \vdash \{P\}\, x = \&\, a\, \{*x = 3 \mid R\}$

$$\begin{aligned} P &= (\mathrm{read}(s, \mathrm{read}(s, x)) = 3)[\mathrm{upd}(s, x^\dagger, (\&\, a)^\#)/s] \\ &= (\mathrm{read}(s, \mathrm{read}(s, x)) = 3)[\mathrm{upd}(s, x, a^\dagger)/s] \\ &= \mathrm{read}(\mathrm{upd}(s, x, a), \mathrm{read}(\mathrm{upd}(s, \underline{x}, a), \underline{x})) = 3) \\ &= \mathrm{read}(\mathrm{upd}(s, \underline{x}, a), \underline{a}) = 3) \\ &= \mathrm{read}(s, a) = 3 \\ &= (a == 3)^* \end{aligned}$$

Weitere geänderte Regeln

$$\frac{}{\Delta \vdash \{Q[e^{\#}/\backslash\textbf{result}]\}\ \textbf{return}\ \ e\,\{P \mid Q\}}$$

$$\frac{\Delta(f) = \forall x_1, \ldots, x_n.\,(P, Q)}{\Delta \vdash \{P^*[t_i{}^{\#}/x_i]\}\ I = f(t_1, \ldots, t_n)\,\{Q^*[t_i{}^{\#}/x_i][I^{\#}/\backslash\textbf{result}] \mid Q_R\}}$$

$$\frac{\Delta \vdash \{P^*\}\ c\ \{\mathit{false} \mid Q^*\}}{\Delta \vdash f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{\mathit{ds}\ c\}}$$

Rückwärtsrechnung mit Zeigern I

$$\text{awp}(\Delta, f(x_1,\ldots,x_n)/{**}\ \textbf{pre}\ P\ \textbf{post}\ Q\ {*}/\ \{ds\ c\}) \stackrel{def}{=} \text{awp}(\Delta, c, \mathit{false}, Q^*[x_i\ \textbf{@pre}\ /x_i])$$

$$\begin{aligned}\text{wvc}(\Delta, f(x_1,\ldots,x_n)/{**}\ \textbf{pre}\ P\ \textbf{post}\ Q\ {*}/\ \{ds\ c\}) &\stackrel{def}{=} \{P^* \wedge x_i = x_i\ \textbf{@pre} \Longrightarrow P'\} \\ &\quad \cup\ \text{wvc}(\Delta, c, \mathit{false}, Q^*[x_i\ \textbf{@pre}\ /x_i]) \\ P' &\stackrel{def}{=} \text{awp}(\Delta, c, \mathit{false}, Q^*[x_i\ \textbf{@pre}\ /x_i])\end{aligned}$$

$$\text{Sei } \Delta(f) = \forall x_1,\ldots,x_n.\,(P,Q)$$

$$\text{awp}(\Delta, /{**}\,\textbf{const}\ \ R\ {*}/I = f(t_1,\ldots,t_n), U, U_R) \stackrel{def}{=} R^* \wedge P^*[{t_i}^*/x_i], I \notin \mathit{FV}(R)$$

$$\begin{aligned}\text{wvc}(\Delta, /{**}\,\textbf{const}\ \ R\ {*}/I = f(t_1,\ldots,t_n), U, U_R) &\stackrel{def}{=} \{R^* \wedge Q[{t_i}^*/x_i][I/\backslash\textbf{result}] \longrightarrow U\}, \\ &\quad I \notin \mathit{FV}(R)\end{aligned}$$

Approximative schwächste Vorbedingung mit Zeigern II

$$\mathrm{awp}(\Delta, \{\,\}, Q, Q_R) \stackrel{def}{=} Q$$

$$\mathrm{awp}(\Delta, l = e, Q, Q_R) \stackrel{def}{=} Q[\mathrm{upd}(s, l^\dagger, e^\#)/s]$$

$$\mathrm{awp}(\Delta, c_1; c_2, Q, Q_R) \stackrel{def}{=} \mathrm{awp}(\Delta, c_1, \mathrm{awp}(\Delta, c_2, Q, Q_R), Q_R)$$

$$\mathrm{awp}(\Delta, \textbf{if } (b)\ c_0\ \textbf{ else }\ c_1, Q, Q_R) \stackrel{def}{=} (b^* \wedge \mathrm{awp}(\Delta, c_0, Q, Q_R)) \vee (\neg b^* \wedge \mathrm{awp}(\Delta, c_1, Q, Q_R))$$

$$\mathrm{awp}(\Delta, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) \stackrel{def}{=} q^*$$

$$\mathrm{awp}(\Delta, \textbf{while } (b)\ /{*}{*}\ \textbf{inv }\ i\ {*}/\ c, Q_R) \stackrel{def}{=} i^*$$

$$\mathrm{awp}(\Delta, \textbf{return }\ e, Q, Q_R) \stackrel{def}{=} Q_R[e^\#/\backslash\textbf{result}]$$

$$\mathrm{awp}(\Delta, \textbf{return}, Q, Q_R) \stackrel{def}{=} Q_R$$

Approximative schwächste Vorbedingung mit Zeigern III

$$\begin{aligned}
\mathrm{wvc}(\Delta, \{\,\}, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Delta, l = e, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Delta, c_1; c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Delta, c_1, \mathrm{awp}(\Delta, c_2, Q, Q_R), Q_R) \\
&\quad \cup \mathrm{wvc}(\Delta, c_2, Q, Q_R) \\
\mathrm{wvc}(\Delta, \textbf{if } (b)\ c_1\ \textbf{ else }\ c_2, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Delta, c_1, Q, Q_R) \cup \mathrm{wvc}(\Delta, c_2, Q, Q_R) \\
\mathrm{wvc}(\Delta, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} \{q^* \Longrightarrow Q\} \\
\mathrm{wvc}(\Delta, \textbf{while } (b)\ /{*}{*}\ \textbf{inv }\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Delta, c, i^*, Q_R) \\
&\quad \cup \{i^* \wedge b^* \Longrightarrow \mathrm{awp}(\Delta, c, i^*, Q_R)\} \\
&\quad \cup \{i^* \wedge \neg b^* \Longrightarrow Q\} \\
\mathrm{wvc}(\Delta, \textbf{return }\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}$$

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.

Aliasing

- Das Beispiel mit Aliasing vom Anfang:

```
void foo(){
 int a, *x;

 /** { 5< 10 }
 /** { 5< read(upd(upd(upd(s, x, a), a, 0),                a     , 10), a) } */
 /** { 5< read(upd(upd(upd(s, x, a), a, 0), read(upd(s, x, a), x), 10), a) } */
 x= & a;
 /** { 5< read(upd(upd(s, a, 0), read(    s,         x), 10), a) } */
 /** { 5< read(upd(upd(s, a, 0), read(upd(s, a, 0), x), 10), a) } */
 a= 0;
 /** { 5< read(upd(s, read(s, x), 10), a) } */
 *x= 10;
 /** { 5< read(s, a) } */
}
```

- Aliasing wird **korrekt** behandelt.

Arbeitsblatt 12.5: Ein problematisches Beispiel

```
void foo(int *p)
/** post \result == 7; */
{
 int x;

 x= 7;
 *p= 99;
 return x;
}
```

Spezfikation des Speicherlayout

- Generelles Problem — was ist mit

```
void foo(int *p, int *q)
{ ... }
```

- Deutet auf ein Problem hin
- Entspricht einem “dangling pointer” (d.h. Pointer mit unspezifiziertem Wert)
- Können weder beweisen, dass $*p = *q$ noch $*p \neq *q$
- Muss (in den Vorbedingungen) spezifziert werden.
- Integration in die Logik: **separation logic**

Weitere Beispiele: Felder

```
int findmax(int a[], int a_len)
/** pre 0 \array(a, a_len) ∧ 0 < a_len;
    post ∀i. 0≤ i< a_len ⟶ a[i] ≤ \result ;
*/
{
 int x; int j;

 x= a[0]; j= 0;
 while (j< a_len)
   /** inv (∀i. 0≤ i< j ⟶ a[i]≤ x ∧ 0≤ j< a_len */ {
   if (a[j]< a_len) {
    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(l)

$\backslash\textbf{valid}(l) \iff \exists x.\, x = \text{read}(\sigma, l^{\dagger})$ für $l \in \textbf{Lexp}$

- Insbesondere: $\backslash\textbf{valid}(*x) \iff \text{read}(\sigma, \text{read}(\sigma, x))$ ist definiert.
- Felder als Parameter werden zu Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger ein Feld ist.
- $\backslash\textbf{array}(a, n)$ bedeutet: a ist ein Feld der Länge n, d.h.

$$\backslash\textbf{array}(a, n) \iff (\forall i.\, 0 \leq i < n \implies \backslash\textbf{valid}(a[i]))$$

- Gültigkeit kann abgeleitet werden:

$$\frac{\text{x= \&e}}{\backslash\text{valid(*x)}} \qquad \frac{\backslash\text{array(a, n)} \quad 0 \leq i \quad i < n}{\backslash\text{valid(a[i])}}$$

Zusammenfassung

- Um Referenzen (Pointer) in C behandeln zu können, benötigen wir ein erweitertes **Zustandsmodell**
- Referenzen werden zu Werten wie Zahlen oder Zeichen.
  - Arrays und Strukturen sind **keine** first-class values.
  - Großes Problem: **aliasing**
- Erweiterung der Semantik und der Hoare-Tripel nötig:
  - Vor/Nachbedingungen werden zu **expliziten Zustandsprädikaten**.
  - Zuweisung wird zu **Zustandsupdate**.
  - Problem:
    - Zustände werden **sehr groß**
    - Rückwärtsrechnung erzeugt schnell sehr große „unbestimmte“ Zustände, die nicht vereinfacht werden können
    - Hier ist Vorwärtsrechnung vorteilhaft

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Korrekte Software: Grundlagen und Methoden

Vorlesung 13 vom 04.07.24

Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten im Floyd-Hoare-Kalkül
- Korrektheit des Floyd-Hoare-Kalküls
- Strukturierte Datentypen
- Verifikationsbedingungen
- Funktionen und Prozeduren I
- Funktionen und Prozeduren II
- Referenzen
- Ausblick und Rückblick

Was gibt's heute?

- Rückblick
- Ausblick
- Feedback
- Prüfungsvorbereitung

I. Rückblick

Prüfungsrelevanz

- Was waren die **zentralen** theoretischen Konzepte?
- Wie sind sie formal definiert, was bedeuten sie?
- Wie hängen sie zusammen?

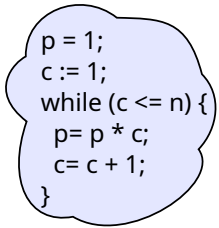
Semantik

- Operational — Auswertungsrelation $\langle c, \sigma \rangle \to \sigma'$
- Denotational — Partielle Funktion $[\![c]\!] : \Sigma \rightharpoonup \Sigma$
- Axiomatisch — Floyd-Hoare-Logik
- Welche Semantik wofür?
- Beweis: Äquivalenz von operationaler und denotationaler Semantik

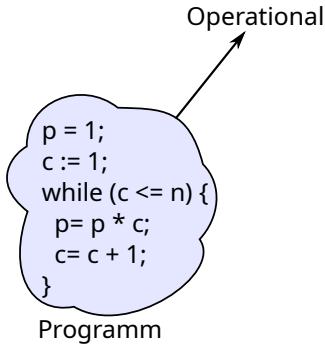
Floyd-Hoare-Logik

- Floyd-Hoare-Logik: partiell und total
- $\vdash \{P\}\, c\, \{Q\}$ vs. $\models \{P\}\, c\, \{Q\}$: Vollständigkeit, Korrektheit
- Die sechs Basisregeln
- Zuweisungsregel: vorwärts (Floyd) vs. rückwärts (Hoare)
- VCG: Schwächste Vorbedingung und stärkste Nachbedingung
- Beweis: Korrektheit und Vollständigkeit der Floyd-Hoare-Logik

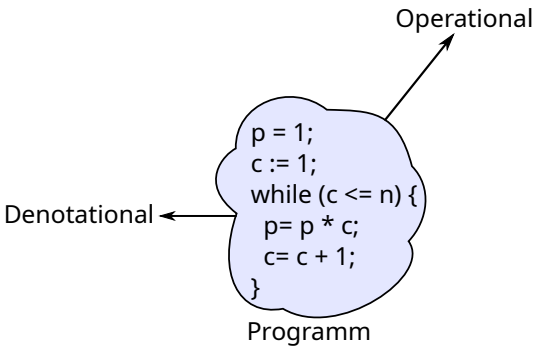
Zusammenhang der Semantiken



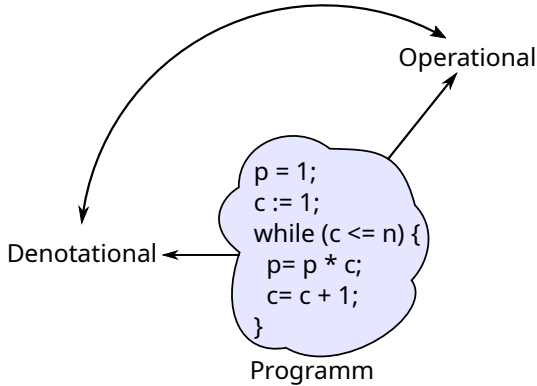
Zusammenhang der Semantiken



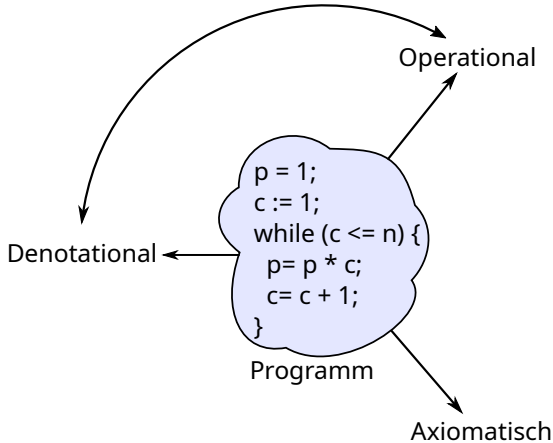
Zusammenhang der Semantiken



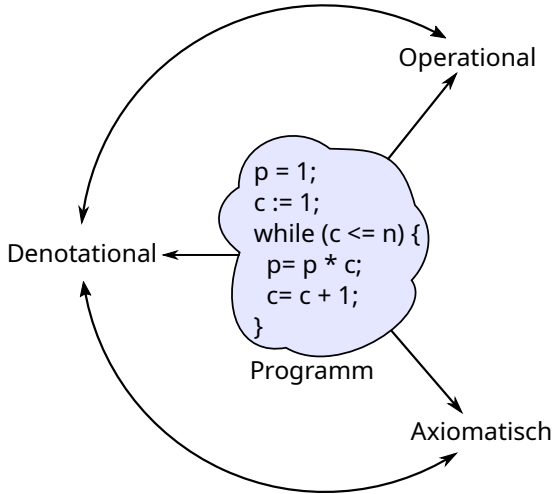
Zusammenhang der Semantiken



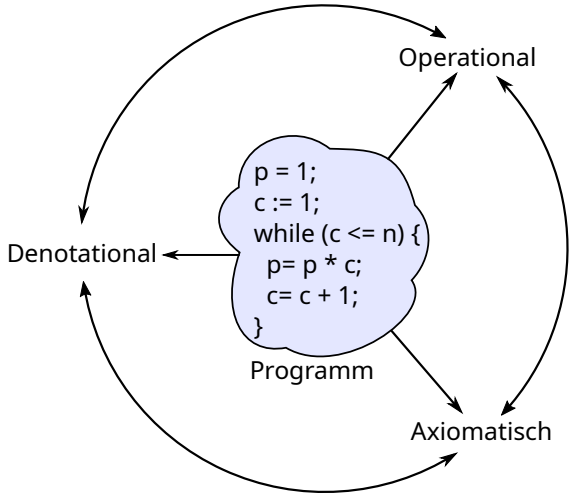
Zusammenhang der Semantiken



Zusammenhang der Semantiken



Zusammenhang der Semantiken



Erweiterungen der Programmiersprache

- Für jede Erweiterung:
  - Wie modellieren wir sie semantisch?
  - Wie ändern sich die Regeln der Logik?

1. Erweiterung der Programmiersprache

- Strukturen und Felder
  - Locations und strukturierte Werte **Lexp**
  - Zustand: endliche Abbildung von Locations auf Werte
  - Erweiterte Substitution in Zuweisungsregel
  - Sonstige Regeln bleiben

2. Erweiterung der Programmiersprache

- Prozeduren und Funktionen
  - Modellierung von **return**: Erweiterung zu $\Sigma \rightharpoonup \Sigma + \Sigma \times \mathbf{V}_U$
  - Modellierung von lokalen Variablen
  - 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
  - Grundproblem: Aliasing
  - Zustand enthält auch Locations im Wertebereich: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}, \mathbf{V} = \mathbb{Z} + \mathbf{Loc}$
  - Spezifikationen sind **explizite Zustandsprädikate** mit read und upd, Konversion $(-)^{\dagger}, (-)^{\#}$
  - Zuweisung nicht mehr mit Substitution, sondern explizit durch upd

II. Ausblick

Was geht noch?

- Die Sprache C
- Andere Programmiersprachen
- Logik und Spezifikation
- Success Stories

Die Sprache C: Was haben wir ausgelassen?

Semantik:

- Nichtdeterministische Semantik: Seiteneffekte, Sequence Points
  $\longrightarrow$ Umständlich zu modellieren, Effekt zweitrangig
- Implementationsabhängiges, unspezifiziertes und undefiniertes Verhalten
  $\longrightarrow$ Genauere Unterscheidung in der Semantik

Kontrollstrukturen:

- **switch** $\longrightarrow$ Ist im allgemeinen Fall ein **goto**
- **goto**, setjmp/longjmp
  $\longrightarrow$ Allgemeinfall: tiefe Änderung der Semantik (*continuations*)

Die Sprache C: Was haben wir ausgelassen?

Typen:

- Funktionszeiger $\longrightarrow$ Für "saubere" Benutzung gut zu modellieren
- Weitere Typen: **short**/**long int**, **double**/**float**, wchar_t, und Typkonversionen $\longrightarrow$ Fleißarbeit
- Fließkommazahlen $\longrightarrow$ Spezifikation nicht einfach
- **union** $\longrightarrow$ Kompliziert das Speichermodell
- **volatile** $\longrightarrow$ Bricht read/update-Gleichungen
- **typedef** $\longrightarrow$ Ärgernis für Lexer/Parser, sonst harmlos

Die Sprache C: Was haben wir ausgelassen?

Für **realistische C-Programme**:

- Compiler-Erweiterungen (`gcc`, `clang`)
- Büchereien (Standardbücherei, Posix, ...)
- Nebenläufigkeit

Andere Sprachen: Wie modelliert man Java?

- Die **Kernsprache** ist ähnlich zu C0.
- Java hat erschwerend:
  - dynamische Bindung,
  - Klassen mit gekapselten Zustand und Invarianten,
  - Nebenläufigkeit, und
  - Reflektion.
- Java hat dafür aber
  - ein einfacheres Speichermodell, und
  - eine wohldefinierte Ausführungsumgebung (die JVM).

Andere Sprachen: Wie modelliert man C++?

- Sehr **vorsichtig** (konservativ)
- Viele Features, fehlende formale Semantik, ...
- Mehrfachvererbung theoretisch anspruchsvoll
- Es gibt **keine** Formalismen/Werkzeuge, die C++ voll unterstützen
- Ansätze: Übersetzung nach C/LLVM, Behandlung dort

Andere Sprachen: Wie modelliert man PHP?

Gar nicht.

Logik und Spezifikation

- Wir **generieren** Verifikationsbedingungen, wie kann man sie **beweisen**?
- **Automatische Beweiser:**
  - **SAT-Checker** lösen Erfüllbarkeitsproblem der Aussagenlogik (MiniSAT, Chaff)
  - **SMT-Beweiser** beweisen Aussagen der Prädikatenlogik mit linearer Arithmetik, Funktionen und Induktion (Z3, Yices, CVC)
- **Interaktive Beweiser:**
  - Beweisführung durch Benutzer, **Überprüfung** durch Beweiser
  - Sehr **mächtige** Logiken, aber nicht vollautomatisch (Isabelle, Coq, Lean)

Beispiel: Z3

- SMT-Beweiser versuchen Gegenbeweis zu konstruieren
- Daher: um ϕ zu beweisen, versuchen wir $\neg\Phi$ zu widerlegen

Beweis einer VC:

$$x \geq 0 \wedge y > 0 \Longrightarrow x = 0 * y + x$$

Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
   (not (=> (and (>= x 0) (> y 0))
            (= x (+ (* 0 y) x))))
)
(check-sat)
```

Antwort:

```
unsat
```

Unerfüllbare VC:

$$x \geq 0 \wedge y > 0 \Longrightarrow x \geq y$$

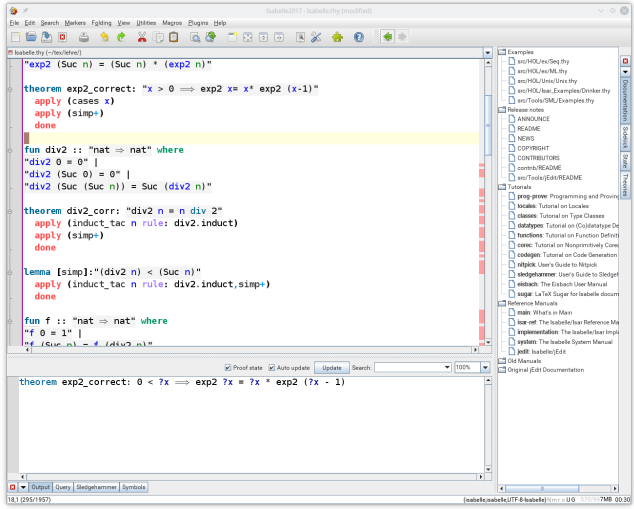
Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
   (not (=> (and (>= x 0) (> y 0))
            (>= x y)))
)
(check-sat)
```

Antwort:

```
sat
```

Beispiel: Isabelle



Korrekte Software in der Industrie

- Meist in speziellen Anwendungsgebieten: Luft-/Raumfahrt, Automotive, sicherheitskritische Systeme, Betriebssysteme
- Ansätze:
  1. Vollautomatisch: **statische Analyse** (Abstrakte Interpretation) für spezielle Aspekte: Freiheit von Ausnahmen und Unter/Überläufen, Programmsicherheit, Laufzeitverhalten (WCET) (nicht immer korrekt, meist vollständig)
     - Werkzeuge: `absint`
  2. Halbautomatisch: **Korrektheitsannotationen**, Überprüfung automatisch
     - Werkzeuge: Spark (ADA), Frama-C (C), JML (ESC/Java, Krakatao; Java), Boogie und Why (generisches VCG), VCC (C)
  3. Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
     - Beispiele: L4.verified, CompCert, SAMS

III. Prüfungsvorbereitung

Prüfungsvorbereitung

- Termine: Do 15.08.2024 / Fr 13.09.2024 (ab 10 Uhr) (Anmeldung über stud.ip)
- Mündliche Modulprüfung, 20–30 Minuten
- Schwerpunkte:
  - **Verständnis** des Stoffes, weniger Folien auswendig lernen
  - Zentrale theoretische Konzepte kennen und benennen
  - Stoff der Vorlesung und Übungsblätter, weniger eure Lösungen
- Bewertung
  - Sicherheit/Beherrschung des Stoffes
  - *covered ground*

IV. Feedback

Deine Meinung zählt

- Bitte die **Evaluation** auf stud.ip beantworten!
- Was war gut, was nicht?
- Arbeitsaufwand?
- Mehr **Theorie** oder mehr **Praxis**?
- Programmieraufgaben?
- Leichtgewichtiger Übungsbetrieb — mehr oder weniger?

Werbeblock

- **Studentische Tutoren** gesucht:
  - Praktische Informatik 3 — Einführung in die funktionale Programmierung
- Veranstaltung:
  - „Adventures in Programming Languages“
  - Praktische Einführung in Rust
  - 04.09.2024, 09:00 – 16:00
  - Details:
    `https://www.bremen-research.de/data-train/courses/course-details?event_id=70`

Tschüß!