

Korrekte Software: Grundlagen und Methoden

Vorlesung 11 vom 28.06.22

Funktionen und Prozeduren I

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2022

Fahrplan

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

Funktionen & Prozeduren

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

Beispiel: 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!; */
{
  int p;
  p = 1;
  while (0 < n)
    /** inv p == (n@pre-n)! ∧
    0 ≤ n; */ {
      p = p * n;
      n = n - 1;
    }
  return p;
}
```

- ▶ **Spezifikation** mit Vor-/Nachbedingung
- ▶ Keine logischen Variablen nötig, **\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:

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

- Abstrakte Syntax
- Größe von Feldern: **konstanter** Ausdruck
- **FunSpec** wird später erläutert

Rückgaben

Neue Anweisungen: Return-Anweisung

Stmt

$$\begin{aligned} s ::= & l = e \mid c_1; c_2 \mid \{ \} \mid \text{if } (b) \ c_1 \ \text{else} \ c_2 \\ & \mid \text{while } (b) \ /\!\! ** \ \text{inv } P \ /\!\! * / \ c \mid /\!\! ** \ \{P\} \ /\!\! * / \\ & \mid \text{return } a^? \end{aligned}$$

Rückgabewerte

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

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

- ▶ Lösung 1: verbieten!

- ▶ MISRA-C (Guidelines for the use of the C language in critical systems):

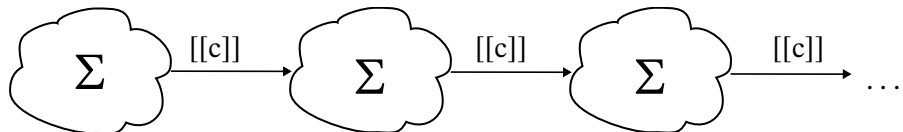
Rule 14.7 (required)

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

- ▶ Nicht immer möglich, unübersichtlicher Code ...
- ▶ Lösung 2: Erweiterung der Semantik

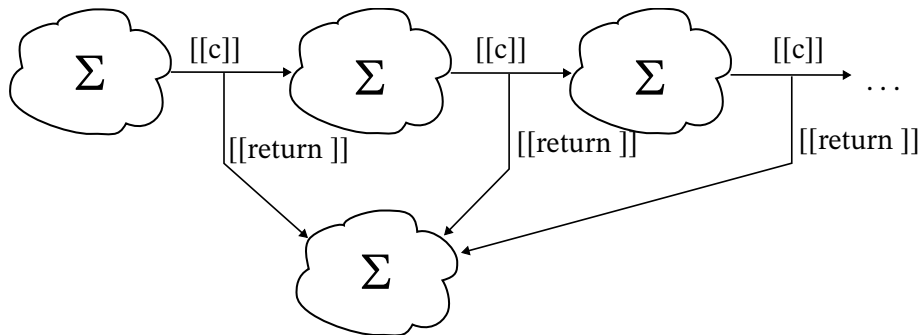
Denotationale Semantik der return-Anweisung

► Alt: $\Sigma \rightarrow \Sigma$



Denotationale Semantik der return-Anweisung

► Alt: $\Sigma \rightarrow \Sigma$



► **Neu:** $\Sigma \rightarrow (\Sigma + \Sigma \times \mathbf{V})$

Erweiterte Semantik

- ▶ Denotat einer Anweisung: $\Sigma \mapsto (\Sigma \cup \Sigma \times \mathbf{V})$
- ▶ 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: $\Sigma \mapsto (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- ▶ 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**: $\mathbf{V}_U \stackrel{\text{def}}{=} \mathbf{V} + \{*\}$

Komposition mit Rückgabewerten

- ▶ Komposition zweier Anweisungen $f, g : \Sigma \rightarrow (\Sigma \cup \Sigma \times \mathbf{V}_U)$.
- ▶ Im Allgemeinen: Wenn $f, g : A \rightarrow A + B$ dann $g \circ_S f : A \rightarrow A + B$:

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

- ▶ Als Mengen: $f, g \subseteq (A \times (A + B)) \cong A \times A + A \times B$

$$g \circ_S f = \{(a, x) \mid \exists a' \in A. (a, a') \in f \wedge (a', x) \in g\} \cup \{(a, b) \mid (a, b) \in f, b \in B\}$$

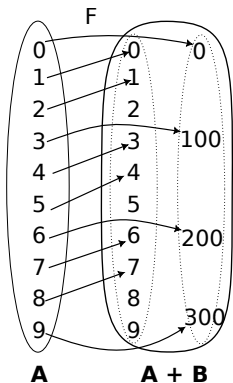
- ▶ **Frage:** Ist das gleiche wie Komposition von partiellen Funktionen?

Arbeitsblatt 11.1: Komposition mit Rückgabewerten

Sei $\mathbf{A} = \{0, \dots, 9\}$

$\mathbf{B} = \{0, 100, 200, 300\}$

Betrachte $F : \mathbf{A} \rightarrow \mathbf{A} + \mathbf{B}$



- 1 Wie ist die Funktion F links in Mengenschreibweise definiert?
- 2 Wie sind folgende Funktionen (als Mengen) definiert:
 $F_2 \stackrel{\text{def}}{=} F \circ_S F$
 $F_3 \stackrel{\text{def}}{=} F \circ_S F \circ_S F$
- 3 Was ist die **Bedeutung** von F_3 ?

Semantik von Anweisungen

$$\llbracket \cdot \rrbracket_c : \mathbf{Stmt} \rightarrow \Sigma \rightarrow (\Sigma \cup \Sigma \times \mathbf{V}_U)$$

$$\llbracket x = e \rrbracket_c = \{(\sigma, \sigma[l \mapsto a]) \mid (\sigma, l) \in \llbracket x \rrbracket_{\mathcal{L}}, (\sigma, a) \in \llbracket e \rrbracket_{\mathcal{A}}\}$$

$$\llbracket c_1; c_2 \rrbracket_c = \llbracket c_2 \rrbracket_c \circ_S \llbracket c_1 \rrbracket_c \quad \text{Komposition wie oben}$$

$$\llbracket \{ \} \rrbracket_c = \mathbf{Id}_{\Sigma} \quad \mathbf{Id}_{\Sigma} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\begin{aligned} \llbracket \text{if } (b) \ c_0 \ \text{else} \ c_1 \rrbracket_c &= \{(\sigma, \rho') \mid (\sigma, \text{true}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma, \rho') \in \llbracket c_0 \rrbracket_c\} \\ &\quad \cup \{(\sigma, \rho') \mid (\sigma, \text{false}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma, \rho') \in \llbracket c_1 \rrbracket_c\} \\ &\quad \text{mit } \rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U \end{aligned}$$

$$\llbracket \text{return } e \rrbracket_c = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in \llbracket e \rrbracket_{\mathcal{A}}\}$$

$$\llbracket \text{return} \rrbracket_c = \{(\sigma, (\sigma, *))\}$$

$$\llbracket \text{while } (b) \ c \rrbracket_c = \text{fix}(\Gamma)$$

$$\begin{aligned} \Gamma(\psi) &\stackrel{\text{def}}{=} \{(\sigma, \rho') \mid (\sigma, \text{true}) \in \llbracket b \rrbracket_{\mathcal{B}} \wedge (\sigma, \rho') \in \psi \circ_S \llbracket c \rrbracket_c\} \\ &\quad \cup \{(\sigma, \sigma) \mid (\sigma, \text{false}) \in \llbracket b \rrbracket_{\mathcal{B}}\} \end{aligned}$$

Arbeitsblatt 11.2: Semantik mit Rückgabe

Berechnet die Denotate der folgenden Programme:

①

$$\llbracket x = 3; x = 4 \rrbracket_C = ?$$

②

$$\llbracket x = 3; \text{return } x; x = 4 \rrbracket_C = ?$$

Erweiterung des Floyd-Hoare-Kalküls

- ▶ Neue Semantik: $\mathbf{Stmt} \rightarrow \Sigma \rightarrow (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- ▶ Wie passt das zu unseren Floyd-Hoare-Tripeln $\models \{P\} c \{Q\}$?

Erweiterung des Floyd-Hoare-Kalküls

- ▶ Neue Semantik: $\mathbf{Stmt} \rightarrow \Sigma \rightarrow (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- ▶ Wie passt das 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**

$$\models \{P\} c \{Q \mid Q_R\}$$

Erweiterung des Floyd-Hoare-Kalküls

$$\llbracket \cdot \rrbracket_c : \mathbf{Stmt} \rightarrow \Sigma \rightarrow (\Sigma \cup \Sigma \times \mathbf{V}_U)$$

Hoare-Tripel: zusätzliche Spezifikation für **Rückgabewert**.

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

c ist **partiell korrekt**, wenn für alle Zustände σ , die P erfüllen:

- ▶ die Ausführung von c mit σ in σ' regulär terminiert, so dass σ' die Spezifikation Q erfüllt,
- ▶ oder die Ausführung von c in σ' mit dem Rückgabewert v terminiert, so dass (σ', v) die Rückgabespezifikation Q_R erfüllt.

$$\begin{aligned} \models \{P\} c \{Q \mid Q_R\} &\iff \\ &\forall \sigma. (\sigma, true) \in \llbracket P \rrbracket_B \wedge (\exists \sigma'. (\sigma, \sigma') \in \llbracket c \rrbracket_c \implies (\sigma', true) \in \llbracket Q \rrbracket_B) \\ &\quad \vee \\ &\quad (\exists \sigma', v. (\sigma, (\sigma', v)) \in \llbracket c \rrbracket_c \implies (\sigma', true) \in \llbracket Q_R \rrbracket_B) \end{aligned}$$

Erweiterung des Floyd-Hoare-Kalküls: return

$$\frac{}{\vdash \{Q\} \text{ return } \{P \mid Q\}}$$

$$\frac{}{\vdash \{Q[e/\backslash\text{result}]\} \text{ return } e \{P \mid Q\}}$$

- ▶ 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 $\backslash\text{result}$ enthält.
- ▶ Bei **return** mit Argument ersetzt der Rückgabewert den $\backslash\text{result}$ in der Rückgabespezifikation.

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

$$\frac{}{\vdash \{P\} \{\} \{P \mid Q_R\}} \quad \frac{\vdash \{P\} c_1 \{R \mid Q_R\} \quad \vdash \{R\} c_2 \{Q \mid Q_R\}}{\vdash \{P\} c_1; c_2 \{Q \mid Q_R\}}$$

$$\frac{}{\vdash \{Q[e/x]\} I = e \{Q \mid Q_R\}} \quad \frac{\vdash \{P \wedge b\} c \{P \mid Q_R\}}{\vdash \{P\} \text{while } (b) \ c \{P \wedge \neg b \mid Q_R\}}$$

$$\frac{\vdash \{P \wedge b\} c_1 \{Q \mid Q_R\} \quad \vdash \{P \wedge \neg b\} c_2 \{Q \mid Q_R\}}{\vdash \{P\} \text{if } (b) \ c_1 \text{ else } c_2 \{Q \mid Q_R\}}$$

$$\frac{P \longrightarrow P' \quad \vdash \{P'\} c \{Q' \mid R'\} \quad Q' \longrightarrow Q \quad R' \longrightarrow R}{\vdash \{P\} c \{Q \mid R\}}$$

$$\frac{}{\vdash \{Q\} \text{return } \{P \mid Q\}}$$

$$\frac{}{\vdash \{Q[e/\backslash \text{result}]\} \text{return } e \{P \mid Q\}}$$

Arbeitsblatt 11.3: Kurzbeispiel

Verifiziert folgendes Kurzbeispiel:

```
{ // {x = X}  
  // ???  
  x = x + 1;  
  // ???  
  return x;  
  // {false | \result == X + 1}  
}
```

Lösungsblatt 11.3: Kurzbeispiel

```
{ // {x = X}
  //
  x = x + 1;
  //
  return x;
  // {false | \result = X + 1}
}
```

Lösungsblatt 11.3: Kurzbeispiel

```
{ // {x = X}  
  //  
  x = x + 1;  
  // {x = X + 1}  
  return x;  
  // {false | \result = X + 1}  
}
```

Lösungsblatt 11.3: Kurzbeispiel

```
{ // {x = X}
  // {x + 1 = X + 1}
  x = x + 1;
  // {x = X + 1}
  return x;
  // {false | \result = X + 1}
}
```

Lösungsblatt 11.3: Kurzbeispiel

```
{ // {x = X}
  // {x + 1 = X + 1}
  x = x + 1;
  // {x = X + 1}
  return x;
  // {false | \result = X + 1}
}
```

Beweisverpflichtung:

$$x = X \implies x + 1 = X + 1 \quad \checkmark$$

Approximative schwächste Vorbedingung

- ▶ Erweiterung zu $\text{awp}(c, Q, Q_R)$ und $\text{wvc}(c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
- ▶ Es wird immer eine **Rückgabespezifikation** Q_R benötigt.
- ▶ Es gilt:

$$\bigwedge \text{wvc}(c, Q, Q_R) \implies \vdash \{ \text{awp}(c, Q, Q_R) \} c \{ Q \mid Q_R \}$$

Approximative schwächste Vorbedingung (Revisited)

$\text{awp}(\{ \}, Q, Q_R)$	$\stackrel{\text{def}}{=} Q$
$\text{awp}(l = e, Q, Q_R)$	$\stackrel{\text{def}}{=} Q[e/l]$
$\text{awp}(c_1; c_2, Q, Q_R)$	$\stackrel{\text{def}}{=} \text{awp}(c_1, \text{awp}(c_2, Q, Q_R), Q_R)$
$\text{awp}(\text{if } (b) \ c_0 \ \text{else } c_1, Q, Q_R)$	$\stackrel{\text{def}}{=} (b \wedge \text{awp}(c_0, Q, Q_R)) \vee (\neg b \wedge \text{awp}(c_1, Q, Q_R))$
$\text{awp}(/\!\!*\{q\} \!*/, Q, Q_R)$	$\stackrel{\text{def}}{=} q$
$\text{awp}(\text{while } (b) \ /\!\!*\text{inv } i \!*/ \ c, Q_R)$	$\stackrel{\text{def}}{=} i$
$\text{awp}(\text{return } e, Q, Q_R)$	$\stackrel{\text{def}}{=} Q_R[e/ \backslash \text{result}]$
$\text{awp}(\text{return}, Q, Q_R)$	$\stackrel{\text{def}}{=} Q_R$

Approximative Verifikationsbedingungen (Revisited)

$$\begin{aligned} \text{wvc}(\{\}, Q, Q_R) &\stackrel{\text{def}}{=} \emptyset \\ \text{wvc}(l = e, Q, Q_R) &\stackrel{\text{def}}{=} \emptyset \\ \text{wvc}(c_1; c_2, Q, Q_R) &\stackrel{\text{def}}{=} \text{wvc}(c_1, \text{awp}(c_2, Q, Q_R), Q_R) \cup \text{wvc}(c_2, Q, Q_R) \\ \text{wvc}(\text{if } (b) \text{ } c_1 \text{ else } c_2, Q, Q_R) &\stackrel{\text{def}}{=} \text{wvc}(c_1, Q, Q_R) \cup \text{wvc}(c_2, Q, Q_R) \\ \text{wvc}(/** \{q\} */, Q, Q_R) &\stackrel{\text{def}}{=} \{q \implies Q\} \\ \text{wvc}(\text{while } (b) /** \text{inv } i */ c, Q, Q_R) &\stackrel{\text{def}}{=} \text{wvc}(c, i, Q_R) \cup \{i \wedge b \implies \text{awp}(c, i, Q_R)\} \\ &\quad \cup \{i \wedge \neg b \implies Q\} \\ \text{wvc}(\text{return } e, Q, Q_R) &\stackrel{\text{def}}{=} \emptyset \end{aligned}$$

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:

$$(1) \quad 0 \leq n \longrightarrow 1 = (1 - 1)! \wedge 0 < 1$$

$$(3) \quad p = (c - 1)! \wedge 0 < c \wedge \neg true \longrightarrow false$$

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

$$(1) \quad 0 \leq n \longrightarrow 1 = (1 - 1)! \wedge 0 < 1$$

$$(3) \quad p = (c - 1)! \wedge 0 < c \wedge \neg \text{true} \longrightarrow \text{false}$$

Vereinfacht:

$$(1.1) \quad 0 \leq n \longrightarrow 1 = 0!$$

$$(1.2) \quad 0 \leq n \longrightarrow 0 < 1$$

$$(3) \quad \text{false} \longrightarrow \text{false}$$

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

$$(1) \quad 0 \leq n \longrightarrow 1 = (1 - 1)! \wedge 0 < 1$$

$$(3) \quad p = (c - 1)! \wedge 0 < c \wedge \neg \text{true} \longrightarrow \text{false}$$

Vereinfacht:

$$(1.1) \quad 0 \leq n \longrightarrow 1 = 0! \quad \checkmark$$

$$(1.2) \quad 0 \leq n \longrightarrow 0 < 1 \quad \checkmark$$

$$(3) \quad \text{false} \longrightarrow \text{false} \quad \checkmark$$

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv p = (c- 1)!  $\wedge$  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  $p = (c-1)! \wedge 0 < c$ ; */ {
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     // { $p = (c-1)! \wedge 0 < c$ }
17 }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv p = (c- 1)!  $\wedge$  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)!  $\wedge$  0 < c + 1}
15     c= c+1;
16     // {p = (c - 1)!  $\wedge$  0 < c}
17 }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv  $p = (c-1)! \wedge 0 < c$ ; */ {
3      //
4      p = p * c;
5      //
6      if (c == n) {
7          //
8          return p;
9          //
10     }
11     else {
12         //  $\{p = ((c+1)-1)! \wedge 0 < c+1\}$ 
13     }
14     //  $\{p = ((c+1)-1)! \wedge 0 < c+1\}$ 
15     c = c + 1;
16     //  $\{p = (c-1)! \wedge 0 < c\}$ 
17 }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv p = (c- 1)!  $\wedge$  0 < c; */ {
3      //
4      p= p*c;
5      //
6      if (c == n) {
7          //
8          return p;
9          // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1 | \result == n!}
10     }
11     else {
12         // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
13     }
14     // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
15     c= c+1;
16     // {p = (c - 1)!  $\wedge$  0 < c}
17 }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv p = (c- 1)!  $\wedge$  0 < c; */ {
3      //
4      p= p*c;
5      //
6      if (c == n) {
7          // {p = n!}
8          return p;
9          // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1 | \result == n!}
10     }
11     else {
12         // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
13     }
14     // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
15     c= c+1;
16     // {p = (c - 1)!  $\wedge$  0 < c}
17 }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv p = (c- 1)!  $\wedge$  0 < c; */ {
3      //
4      p= p*c;
5      // {(c = n  $\wedge$  p = n!)  $\vee$  (c  $\neq$  n  $\wedge$  p = ((c + 1) - 1)!  $\wedge$  0 < c + 1)}
6      if (c == n) {
7          // {p = n!}
8          return p;
9          // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1 | \result == n!}
10     }
11     else {
12         // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
13     }
14     // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
15     c= c+1;
16     // {p = (c - 1)!  $\wedge$  0 < c}
17 }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2  while (1) /** inv p = (c- 1)!  $\wedge$  0 < c; */ {
3      // {(c = n  $\wedge$  p  $\cdot$  c = n!)  $\vee$  (c  $\neq$  n  $\wedge$  p  $\cdot$  c = ((c + 1) - 1)!  $\wedge$  0 < c + 1)}
4      p = p * c;
5      // {(c = n  $\wedge$  p = n!)  $\vee$  (c  $\neq$  n  $\wedge$  p = ((c + 1) - 1)!  $\wedge$  0 < c + 1)}
6      if (c == n) {
7          // {p = n!}
8          return p;
9          // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1 | \result == n!}
10     }
11     else {
12         // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
13     }
14     // {p = ((c + 1) - 1)!  $\wedge$  0 < c + 1}
15     c = c + 1;
16     // {p = (c - 1)!  $\wedge$  0 < c}
17 }
```

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{aligned}(2) \quad & p = (c - 1)! \wedge 0 < c \wedge \text{true} \\ & \longrightarrow (c = n \wedge p \cdot c = n!) \\ & \quad \vee (c \neq n \wedge p \cdot c = ((c + 1) - 1)! \wedge 0 < c + 1)\end{aligned}$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{aligned}(2) \quad & p = (c - 1)! \wedge 0 < c \wedge \text{true} \\ & \longrightarrow (c = n \wedge p \cdot c = n!) \\ & \quad \vee (c \neq n \wedge p \cdot c = ((c + 1) - 1)! \wedge 0 < c + 1)\end{aligned}$$

Neue **Vereinfachungsregel**:

10 Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

$$\blacktriangleright P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$$

Damit Vereinfachung:

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{aligned}(2) \quad & p = (c - 1)! \wedge 0 < c \wedge \text{true} \\ & \longrightarrow (c = n \wedge p \cdot c = n!) \\ & \quad \vee (c \neq n \wedge p \cdot c = ((c + 1) - 1)! \wedge 0 < c + 1)\end{aligned}$$

Neue **Vereinfachungsregel**:

10 Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

$$\blacktriangleright P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$$

Damit Vereinfachung:

$$(2.1) \quad p = (c - 1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n!$$

$$(2.2) \quad p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c!$$

$$(2.3) \quad p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c + 1$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{aligned}(2) \quad & p = (c - 1)! \wedge 0 < c \wedge \text{true} \\ & \longrightarrow (c = n \wedge p \cdot c = n!) \\ & \quad \vee (c \neq n \wedge p \cdot c = ((c + 1) - 1)! \wedge 0 < c + 1)\end{aligned}$$

Neue **Vereinfachungsregel**:

10 Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

$$\blacktriangleright P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$$

Damit Vereinfachung:

$$\begin{aligned}(2.1) \quad & p = (c - 1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! \quad \checkmark ((c - 1)! \cdot c = c!) \\ (2.2) \quad & p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! \\ (2.3) \quad & p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c + 1\end{aligned}$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{aligned}(2) \quad & p = (c - 1)! \wedge 0 < c \wedge \text{true} \\ & \longrightarrow (c = n \wedge p \cdot c = n!) \\ & \quad \vee (c \neq n \wedge p \cdot c = ((c + 1) - 1)! \wedge 0 < c + 1)\end{aligned}$$

Neue **Vereinfachungsregel**:

10 Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

$$\blacktriangleright P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$$

Damit Vereinfachung:

$$(2.1) \quad p = (c - 1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! \quad \checkmark \quad ((c - 1)! \cdot c = c!)$$

$$(2.2) \quad p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! \quad \checkmark \quad ((c - 1)! \cdot c = c!)$$

$$(2.3) \quad p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c + 1$$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

$$\begin{aligned}(2) \quad & p = (c - 1)! \wedge 0 < c \wedge \text{true} \\ & \longrightarrow (c = n \wedge p \cdot c = n!) \\ & \quad \vee (c \neq n \wedge p \cdot c = ((c + 1) - 1)! \wedge 0 < c + 1)\end{aligned}$$

Neue **Vereinfachungsregel**:

10 Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

$$\blacktriangleright P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$$

Damit Vereinfachung:

$$\begin{aligned}(2.1) \quad & p = (c - 1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n! \quad \checkmark ((c - 1)! \cdot c = c!) \\ (2.2) \quad & p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c! \quad \checkmark ((c - 1)! \cdot c = c!) \\ (2.3) \quad & p = (c - 1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c + 1 \quad \checkmark (c < c + 1)\end{aligned}$$

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.
- ▶ Nachbedingung *false* forciert dass Programm nur mit **return** verlassen wird.

Semantik von Funktionsdefinitionen

$$\llbracket \cdot \rrbracket_{fd} : \mathbf{FunDef} \rightarrow \mathbf{V}^n \rightarrow \Sigma \rightarrow \Sigma \times \mathbf{V}_U$$

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

$$\begin{aligned} \llbracket f(t_1 \ p_1, t_2 \ p_2, \dots, t_n \ p_n) \ ds \ c \rrbracket_{fd} \ \Gamma \ v_1, \dots, v_n = \\ \llbracket \underbrace{(t_1 \ p_1 \ v_1, t_2 \ p_2 \ v_2, \dots, t_n \ p_n \ v_n, ds)}_{\text{Deklarationen}} \ c \rrbracket_{blk} \end{aligned}$$

- ▶ 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 und ihre Semantik

- ▶ Blöcke bestehen aus Deklaration und einem Rumpf — benötigen Semantik für Deklarationen
- ▶ Deklarationen erzeugen **lokale Variablen**
- ▶ Dadurch **Trennung** von Variablenname und Lokation (im Speicher)
- ▶ Vorher: **Idt** $\longrightarrow$ **V** jetzt: **Idt** $\longrightarrow$ **Loc** $\longrightarrow$ **V**
- ▶ Nötig bspw. für Rekursion
- ▶ Was sind Lokationen?
 - ① In C: Lokation $\cong$ Adressen $\longrightarrow$ Speichermodelle
 - ② Hier: abstraktes Speichermodell

Abstraktes Speichermodell

- ▶ Der Systemzustand ist $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}$
- ▶ **Loc** ist eine Menge so dass
 - ▶ **Loc** ist unbegrenzt;
 - ▶ Es gibt Operationen:

$$\begin{aligned}\nu &: \Sigma \rightarrow \mathbf{Loc} \\ \text{rem} &: \Sigma \times \mathbf{Loc} \rightarrow \Sigma\end{aligned}$$

$$\begin{aligned}\nu(\sigma) &\notin \text{dom}(\sigma) \\ l &\notin \text{dom}(\text{rem}(\sigma, l))\end{aligned}$$

Neue Lokation
Deallokation

- ▶ Ferner ist **Loc** abgeschlossen über:

$$\begin{aligned}l \in \mathbf{Loc}, i \in \mathbf{Idt} &\implies l.i \in \mathbf{Loc} \\ l \in \mathbf{Loc}, n \in \mathbb{Z} &\implies l[n] \in \mathbf{Loc}\end{aligned}$$

Die Umgebung Γ

- ▶ **Umgebung**: **statischer** Teil des Speichers
- ▶ Wird zur **Laufzeit** nicht benötigt
- ▶ Bildet **Variablenbezeichner** auf Lokationen ab.
- ▶ Modelliert als **partielle Funktion**:

$$\text{Env} = \text{Idt} \rightarrow \text{Loc}$$

- ▶ Wird später noch erweitert.
- ▶ Umgebung ist **zusätzlicher Parameter** für alle Definitionen

Semantik von Deklarationen

- ▶ Zuerst: lokale Variablen.

$$\begin{aligned} \text{local} &: (\mathbf{Loc} \rightarrow \Sigma \rightarrow \Sigma \times \mathbf{V}_U) \rightarrow \Sigma \rightarrow \Sigma \times \mathbf{V}_U \\ \text{local}(b) &= \{(\sigma, (\text{rem}(\nu(\sigma), \sigma'), \nu)) \mid (\sigma, (\sigma', \nu)) \in b(\nu(\sigma))\} \end{aligned}$$

- ▶ Neue Variable alloziieren, 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 :

$$\begin{aligned}\llbracket - \rrbracket_{blk} &: \mathbf{Blk} \rightarrow \mathbf{Env} \rightarrow \Sigma \rightarrow \Sigma \times \mathbf{V}_U \\ \llbracket ((\text{typ } \text{idt}) : ds) \ c \rrbracket_{blk} \Gamma &= \text{local}(\lambda l. \llbracket ds \ c \rrbracket_{blk}(\Gamma[\text{idt} \mapsto l])) \\ \llbracket [] \ c \rrbracket_{blk} \Gamma &= \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in \llbracket c \rrbracket_c \Gamma\}\end{aligned}$$

- ▶ Rekursive Definition (über der Liste der Deklationen)
- ▶ Semantik der Deklationen zusammen mit dem Rumpf.
- ▶ Von $\llbracket c \rrbracket_c$ sind nur **Rückgabezustände** interessant.
 - ▶ Kein „fall-through“
 - ▶ Was passiert ohne **return** am Ende?

Arbeitsblatt 11.4: 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 ?

$$\llbracket f \rrbracket_{fd} = ?$$

Arbeitsblatt 11.4: 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 ?

$$\llbracket f \rrbracket_{fd} = \lambda v. \{(\sigma, \dots) \mid \dots\}$$

Spezifikation von Funktionen

- ▶ Wir **spezifizieren** Funktionen durch **Vor-** und **Nachbedingungen**
 - ▶ Ähnlich den Hoare-Tripeln, aber vereinfachte Syntax
 - ▶ **Behavioural specification**, angelehnt an JML, OCL, ACSL (Frama-C)
- ▶ Syntaktisch:

FunSpec ::= **/** pre Assn post Assn */**

Vorbedingung **pre** sp; $\underbrace{\Sigma}_{\text{Vorzustand}} \rightarrow \mathbb{B}$

Nachbedingung **post** sp; $\underbrace{\Sigma}_{\text{Vorzustand}} \times \underbrace{(\Sigma \times \mathbf{V}_U)}_{\text{Nachzustand und Return-Wert}} \rightarrow \mathbb{B}$

e@pre Wert von e im **Vorzustand**

\result **Rückgabewert** der Funktion

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n! };
*/
{
    int p;
    int c;

    p= 1;
    c= 1;
    while (c≤ n) /** inv p == (c- 1)! ∧ 0≤ c ∧ c-1≤ n; */ {
        p= p*c;
        c= c+1;
    }
    return p;
}
```

Arbeitsblatt 11.5: Suche

Spezifiziert die Suche nach dem maximalen Element:

```
int findmax(int a[], int a_len)
/** pre ???
    post ??? */
{
    int r; int j;

    j= 0;
    r= 0;
    while (j< a_len)
        /** inv  $(\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]) \wedge 0 \leq i \leq n \wedge 0 \leq r < n$ ; */
        {
            if (a[j]> x) { r= j; }
            j= j+1;
        }
    return r;
}
```

Gültigkeit von Spezifikationen

- ▶ Ziel ist eine **Semantik von Spezifikationen** $\llbracket \cdot \rrbracket_{\mathcal{B}sp}$ zu definieren, um damit **semantische Gültigkeit** zu definieren:

$$\models fd \text{ pre } p \text{ post } q \iff (\forall v_1, \dots, v_n. \llbracket P \rrbracket_{\mathcal{B}} \Gamma(\sigma) \implies \llbracket Q \rrbracket_{\mathcal{B}sp} \Gamma(\sigma, \llbracket fd \rrbracket_{fd} \Gamma(v_1 \dots v_n)(\sigma)))$$

- ▶ Nicht ganz präzise wegen Partialität
- ▶ Spezifikationen beziehen sich nicht auf lokale Variablen (nur Parameter).
 - ▶ Nicht präzise, Parameter v_i müssen in Γ an die Namen gebunden werden.
- ▶ Nachbedingung wird in **Vor-** und **Nachzustand** ausgewertet.
- ▶ Kein Hoare-Tripel, aber wir können es zu einem machen:

$$\models fd \text{ pre } p \text{ post } q \iff \forall v_1, \dots, v_n, \sigma_0. \models \{ \lambda \sigma. \llbracket P \rrbracket_{\mathcal{B}} \Gamma(\sigma) \wedge \sigma_0 = \sigma \} \llbracket fd \rrbracket_{blk} \Gamma(v_1, \dots, v_n) \{ false \mid \lambda \sigma. \llbracket Q \rrbracket_{\mathcal{B}sp} \Gamma(\sigma_0, \sigma) \}$$

Beispiel: Rekursion

```
int factorial(int n)
/** pre  0 ≤ n;
    post  \result == n!; */
{
    int x;

    if (n == 0) {
        return 1;
    }
    else {
        x = factorial(n-1);
        return n * x;
    }
}
```

Semantik von Spezifikationen

- ▶ Vorbedingung: Auswertung als $\llbracket sp \rrbracket_{\mathcal{B}} \Gamma$ über dem Vorzustand
- ▶ Nachbedingung: Erweiterung von $\llbracket . \rrbracket_{\mathcal{B}}$ und $\llbracket . \rrbracket_{\mathcal{A}}$
 - ▶ Ausdrücke können in Vor- **oder** Nachzustand ausgewertet werden.
 - ▶ **\result** darf nicht in Funktionen vom Typ **void** auftreten.

Semantik von Spezifikationen

$$\llbracket \cdot \rrbracket_{\mathcal{B}sp} : \mathbf{Env} \rightarrow \mathbf{Assn} \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbb{B}$$

$$\llbracket \cdot \rrbracket_{\mathcal{A}sp} : \mathbf{Env} \rightarrow \mathbf{Aexpv} \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbf{V}$$

$$\begin{aligned} \llbracket !b \rrbracket_{\mathcal{B}sp} \Gamma = & \{((\sigma, (\sigma', v)), true) \mid ((\sigma, (\sigma', v)), false) \in \llbracket b \rrbracket_{\mathcal{B}sp} \Gamma\} \\ & \cup \{((\sigma, (\sigma', v)), false) \mid ((\sigma, (\sigma', v)), true) \in \llbracket b \rrbracket_{\mathcal{B}sp} \Gamma\} \end{aligned}$$

$$\llbracket x \rrbracket_{\mathcal{A}sp} \Gamma = \{((\sigma, (\sigma', v)), \sigma'(x))\}$$

...

$$\llbracket e@pre \rrbracket_{\mathcal{B}sp} \Gamma = \{((\sigma, (\sigma', v)), b) \mid (\sigma, b) \in \llbracket e \rrbracket_{\mathcal{B}} \Gamma\}$$

$$\llbracket e@pre \rrbracket_{\mathcal{A}sp} \Gamma = \{((\sigma, (\sigma', v)), a) \mid (\sigma, a) \in \llbracket e \rrbracket_{\mathcal{A}} \Gamma\}$$

$$\llbracket \backslash result \rrbracket_{\mathcal{A}sp} \Gamma = \{((\sigma, (\sigma', v)), v)\}$$

$$\llbracket pre \ p \ post \ q \rrbracket_{\mathcal{B}sp} \Gamma = \{(\sigma, (\sigma', v)) \mid (\sigma, true) \in \llbracket p \rrbracket_{\mathcal{B}} \Gamma \wedge ((\sigma, (\sigma', v)), true) \in \llbracket q \rrbracket_{\mathcal{B}sp} \Gamma\}$$

Zusammenfassung

- ▶ Funktionsspezifikationen bestehen aus Vorbedingung (**pre**) und Nachbedingung (**post**).
- ▶ In der Nachbedingung kann ein Ausdruck mit **e@pre** im **Vorzustand** ausgewertet werden (ersetzt logische Variablen).
- ▶ Funktionsparameter sind Parameter der Spezifikation, keine lokalen Variablen.
- ▶ Wir haben die **semantische Gültigkeit** von Funktionsspezifikationen definiert, und auf Hoare-Tripel zurückgeführt.
- ▶ Damit können wir die Regeln des Hoare-Kalküls zur Verifikation benutzen.

Verifikation

- ▶ Unterscheidung des Parameterwerts x von der lokalen Variablen x durch Notation x **@pre**
 - ▶ Im Vorzustand gilt x **@pre** $= x$.
 - ▶ Verhindert, dass der Parameter x bei der Zuweisung substituiert wird.
- ▶ Verifikationsregel:

$$\frac{P \wedge x_i = x_i \text{ @pre} \implies P' \quad \vdash \{P'\} c \{false \mid Q[x_i \text{ @pre} / x_i]\}}{\vdash f(x_1, \dots, x_n) / ** \text{ pre } P \text{ post } Q \text{ */ } \{ds \ c\}}$$

- ▶ Berechnung von **awp** und **wvc**:

$$\begin{aligned} \text{awp}(\Gamma, f(x_1, \dots, x_n) / ** \text{ pre } P \text{ post } Q \text{ */ } \{ds \ c\}) &\stackrel{\text{def}}{=} \text{awp}(\Gamma', c, false, Q[x_i \text{ @pre} / x_i]) \\ \text{wvc}(\Gamma, f(x_1, \dots, x_n) / ** \text{ pre } P \text{ post } Q \text{ */ } \{ds \ c\}) &\stackrel{\text{def}}{=} \{P \wedge x_i = x_i \text{ @pre} \implies P'\} \\ &\quad \cup \text{wvc}(\Gamma', c, false, Q[x_i \text{ @pre} / x_i]) \\ \Gamma' &\stackrel{\text{def}}{=} \Gamma[f \mapsto \forall x_1, \dots, x_n. (P, Q)] \\ P' &\stackrel{\text{def}}{=} \text{awp}(\Gamma', c, false, Q[x_i \text{ @pre} / x_i]) \end{aligned}$$

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:

$$(1) \text{ true} \wedge x \text{ @pre} = x \implies x \text{ @pre} < x + 1$$

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

- ▶ Funktionen können wir mit den Regeln des erweiterten Hoare-Kalküls verifizieren.
- ▶ Dabei ersetzen wir Funktionsparameter x in der Nachbedingung mit x **@pre**.
- ▶ Die dahinter liegenden weitgehenden Änderungen in der Semantik bleiben weitgehend unsichtbar.
- ▶ Probleme:
 - ▶ Felder als Parameter werden anders als in C behandelt (hier: wie in Java, in C: wie Referenzen).
 - ▶ Wie behandeln wir eigentlich **Funktionsaufrufe**?