Korrekte Software: Grundlagen und Methoden
Vorlesung 10 vom 13.06.22
Funktionen und Prozeduren I

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

Funktionen & Prozeduren

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

Beispiel: Rekursion

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
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
```

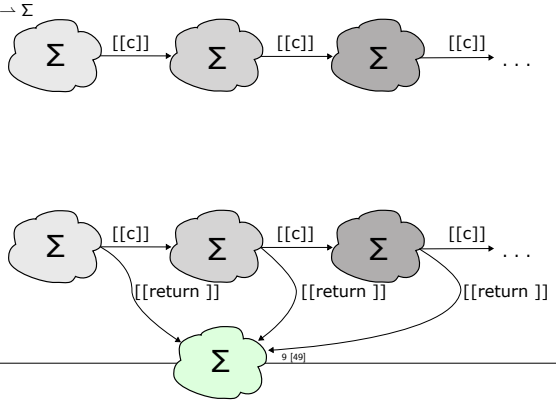
- Lösung 1: verbieten!
  - MISRA-C (Guidelines for the use of the C language in critical systems):

> 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: $\Sigma \rightharpoonup \Sigma$



- **Neu**: $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$

Erweiterte Semantik

- Denotat einer Anweisung: $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V})$ $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- Abbildung von Ausgangszustand Σ auf:
  - Sequentieller **Folgezustand** oder **Rückgabewert** und **Rückgabezustand**;
  - Σ und $\Sigma \times \mathbf{V}$ sind **disjunkt**.
- Was ist mit **void**?
  - **Erweiterte Werte**: $\mathbf{V}_U \stackrel{def}{=} \mathbf{V} + \{*\}$

Komposition 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 $g \circ_S f : A \rightharpoonup A + B$:

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

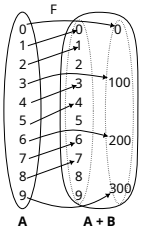
- Als Mengen: $f, g \subseteq (A \times (A + B)) \cong A \times A + A \times B$

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

- **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

$$[\![.]\!]_C : \mathbf{Stmt} \to \mathbf{Env} \to \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$$

$$[\![x = e]\!]^\Gamma_C = \{(\sigma, \sigma[l \mapsto a]) \mid (\sigma, l) \in [\![x]\!]_\mathcal{L}, (\sigma, a) \in [\![e]\!]_A\}$$
$$[\![c_1; c_2]\!]^\Gamma_C = [\![c_2]\!]^\Gamma_C \circ_S [\![c_1]\!]^\Gamma_C \quad \text{Komposition wie oben}$$
$$[\![\{\,\}]\!]^\Gamma_C = \mathbf{Id}_\Sigma \qquad \mathbf{Id}_\Sigma := \{(\sigma, \sigma) | \sigma \in \Sigma\}$$
$$[\![\textbf{if } (b)\ c_0\ \textbf{else}\ c_1]\!]^\Gamma_C = \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]^\Gamma_\mathcal{B} \wedge (\sigma, \rho') \in [\![c_0]\!]^\Gamma_C\} \cup \{(\sigma, \rho') \mid (\sigma, \mathit{false}) \in [\![b]\!]^\Gamma_\mathcal{B} \wedge (\sigma, \rho') \in [\![c_1]\!]^\Gamma_C\}$$
mit $\rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U$

$$[\![\textbf{return}\ e]\!]^\Gamma_C = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]^\Gamma_A\}$$
$$[\![\textbf{return}]\!]^\Gamma_C = \{(\sigma, (\sigma, *))\}$$

$$[\![\textbf{while } (b)\ c]\!]^\Gamma_C = \mathit{fix}(\Psi),\ \Psi(\varphi) \stackrel{def}{=} \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]^\Gamma_\mathcal{B} \wedge (\sigma, \rho') \in \varphi \circ_S [\![c]\!]^\Gamma_C\} \cup \{(\sigma, \sigma) \mid (\sigma, \mathit{false}) \in [\![b]\!]^\Gamma_\mathcal{B}\}$$

Arbeitsblatt 10.2: Semantik mit Rückgabe

Berechnet die Denotate der folgenden Programme:

1. $$[\![x = 3; x = 4]\!]^\Gamma_C = \textbf{?}$$
2. $$[\![x = 3; \textbf{return } x; x = 4]\!]^\Gamma_C = \textbf{?}$$

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 $\models \{P\}\, c\, \{Q\}$?
- 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

$$[\![\cdot]\!]_{\mathcal{B}sp} : \textbf{Env} \to \textbf{Intprt} \to \textbf{Assn} \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbb{B}$$
$$[\![\cdot]\!]_{\mathcal{A}sp} : \textbf{Env} \to \textbf{Intprt} \to \textbf{Aexpv} \rightharpoonup (\Sigma \times \mathbf{V}_U) \to \mathbf{V}$$
$$[\![b_1 \;\&\&\; b_2]\!]^{\Gamma,I}_{\mathcal{B}sp} = \{((\sigma,v), true) \mid ((\sigma,v), true) \in [\![b_1]\!]^{\Gamma,I}_{\mathcal{B}sp} \wedge ((\sigma,v), true) \in [\![b_2]\!]^{\Gamma,I}_{\mathcal{B}sp}\} \cup \{((\sigma,v), false) \mid ((\sigma,v), false) \in [\![b_1]\!]^{\Gamma,I}_{\mathcal{B}sp} \vee ((\sigma,v), false) \in [\![b_2]\!]^{\Gamma,I}_{\mathcal{B}sp}\}$$
$$\ldots$$
$$[\![a_1 = a_2]\!]^{\Gamma,I}_{\mathcal{B}sp} = \{((\sigma,v), v_1 = v_2) \mid ((\sigma,v), v_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathcal{A}sp}, (\sigma,(\sigma',v_2)) \in [\![a_2]\!]^{\Gamma,I}_{\mathcal{A}sp}\}$$
$$[\![a_1 + a_2]\!]^{\Gamma,I}_{\mathcal{A}sp} = \{((\sigma,v), v_1 + v_2) \mid ((\sigma,v), v_1) \in [\![a_1]\!]^{\Gamma,I}_{\mathcal{A}sp}, (\sigma,(\sigma',v_2)) \in [\![a_2]\!]^{\Gamma,I}_{\mathcal{A}sp}\}$$
$$\ldots$$
$$[\![l]\!]^{\Gamma,I}_{\mathcal{A}sp} = \{((\sigma,v), \sigma(m)) \mid (\sigma,m) \in [\![l]\!]^{\Gamma,I}_{\mathcal{L}sp}\}, l \in \textbf{Lexp}$$
$$[\![x]\!]^{\Gamma,I}_{\mathcal{A}sp} = I(x), x \in \mathrm{dom}(I)$$
$$[\![\backslash\textbf{result}]\!]^{\Gamma,I}_{\mathcal{A}sp} = \{((\sigma,v), v)\}$$

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.

$$\Gamma \models \{P\}\, c\, \{Q \mid Q_R\} \iff \forall I.\, \forall \sigma.\, ((\sigma,*), true) \in [\![P]\!]^{\Gamma,I}_{\mathcal{B}sp} \implies (\exists \sigma'.\, (\sigma,\sigma') \in [\![c]\!]^{\Gamma}_{C} \implies ((\sigma',*), true) \in [\![Q]\!]^{\Gamma,I}_{\mathcal{B}sp}) \vee (\exists \sigma', v.\, (\sigma,(\sigma',v)) \in [\![c]\!]^{\Gamma}_{C} \implies ((\sigma',v), true) \in [\![Q_R]\!]^{\Gamma,I}_{\mathcal{B}sp})$$

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

Beweisverpflichtung:

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

Zusatzfrage

Was ist der Unterschied zwischen den Nachbedingungen

(i) $\{false \mid \backslash\textbf{result} = X + 1\}$

(ii) $\{x = X + 1 \mid \backslash\textbf{result} = X + 1\}$

(iii) $\{true \mid \backslash\textbf{result} = X + 1\}$

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 $\text{awp}(c, Q, Q_R)$ und $\text{wvc}(c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
- Es wird immer eine **Rückgabespezifikation** Q_R benötigt.
- Die Rückgabespezifikation wird immer durchgereicht, aber:
- **return** ersetzt die schwächste Vorbedingung durch die Rückgabespezifikation.
- Es gilt:

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

Approximative schwächste Vorbedingung (Revisited)

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

Approximative Verifikationsbedingungen (Revisited)

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

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

Vereinfacht:

(1.1) $0 \leq n \longrightarrow 1 = 0!$ ✓
(1.2) $0 \leq n \longrightarrow 0 < 1$ ✓
(3) $\mathit{false} \longrightarrow \mathit{false}$ ✓

Beispiel: Fakultät (Schleifenrumpf)

```
1
2     while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3       // {(c = n ∧ p · c = n!) ∨ (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:

(2) $p = (c-1)! \wedge 0 < c \wedge \mathit{true}$
$\longrightarrow (c = n \wedge p \cdot c = n!)$
$\vee (c \neq n \wedge p \cdot c = ((c+1)-1)! \wedge 0 < c+1)$

Neue **Vereinfachungsregel**:

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:

(2.1) $p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n!$ ✓ $((c-1)! \cdot c = c!)$
(2.2) $p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow p \cdot c = c!$ ✓ $((c-1)! \cdot c = c!)$
(2.3) $p = (c-1)! \wedge 0 < c \wedge c \neq n \longrightarrow 0 < c+1$ ✓ $(c < c+1)$

Was fällt uns auf?

- Die Invariante ist $p = (c-1)! \wedge 0 < c$
- 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

$$\mathrm{local} : (\mathbf{Addr} \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U) \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$
$$\mathrm{local}(b) = \{(\sigma, (\mathrm{rem}(\sigma', \nu(\sigma)), v)) \mid (\sigma, (\sigma', v)) \in b(\nu(\sigma))\}$$

- 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 $\mathrm{dom}(\sigma)$, 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:

$$[\![-]\!]_{blk} : \mathbf{Blk} \to \mathbf{Env} \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$
$$[\![(t\ x; ds)\ c]\!]^\Gamma_{blk} = \mathrm{local}(\lambda l.\ [\![ds\ c]\!]^{\Gamma[x \mapsto l]}_{blk})$$
$$[\![c]\!]^\Gamma_{blk} = \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![c]\!]^\Gamma_C\}$$

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

Semantik von Funktionsdefinitionen

- Grundprinzip (mit nur einen Parameter)

$$[\![f(t\ x)\ blk]\!]_{fd} = \lambda v.\ [\![\ t\ x;\ x = v;\ blk\ ]\!]$$

- 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?

$$[\![f]\!]^\Gamma_{fd} = ? \lambda v.\ \{(\sigma, \ldots) \mid \ldots\}$$

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:

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

Vorbedingung	**pre** sp;	**Env** → **Intprt** →	$\Sigma \to \mathbb{B}$
Nachbedingung	**post** sp;	**Env** → **Intprt** →	$(\Sigma \times \mathbf{V}_U) \to \mathbb{B}$
Rückgabewert	\result		

Gültigkeit von Spezifikationen

- Ziel ist eine **Semantik von Spezifikationen** $[\![.]\!]_{\mathcal{B}sp}$ zu definieren, um damit **semantische Gültigkeit** zu definieren:

$$\Gamma \models f(x_1, \ldots, x_n) \text{ pre } P \text{ post } Q \; blk$$
$$\iff \forall x_1, \ldots, x_n. \Gamma \models \{[\![P]\!]^{\Gamma,I}_{\mathcal{B}sp}\} \; [\![blk]\!]^{\Gamma}_{fd}(x_1, \ldots, x_n) \{false \mid [\![Q]\!]^{\Gamma,I}_{\mathcal{B}sp}\}$$

- 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 **Idt** 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:

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

Verifikationsbedingungen

- Berechnung von awp und wvc:

$$\text{awp}(f(x_1, \ldots, x_n) /** \text{ pre } P \text{ post } Q \; */ \; \{ds \; c\}) \stackrel{def}{=} \text{awp}(c, false, Q[x_i \text{ @pre} / x_i])$$
$$\text{wvc}((x_1, \ldots, x_n) /** \text{ pre } P \text{ post } Q \; */ \; \{ds \; c\}) \stackrel{def}{=} \{P \wedge x_i = x_i \text{ @pre} \implies P'\}$$
$$\cup \; \text{wvc}(c, false, Q[x_i \text{ @pre} / x_i])$$
$$P' \stackrel{def}{=} \text{awp}(c, false, Q[x_i \text{ @pre} / x_i])$$

- 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 @pre < x + 1}
  x= x+1;
  // {x @pre < x}
  return x;
  // {false | x @pre < \result}
}
```

- Verifikationsbedingung:

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

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**?