Korrekte Software: Grundlagen und Methoden

Vorlesung 12 vom 29.06.21

Spezifikation von Funktionen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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

Funktionen & Prozeduren

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

Beispiel: Rekursion

```
int factorial(int n)
/** pre  0≤ n;
    post  \result = n!; */
{
  if (n == 0) return 1;
  else return n * factorial(n-1);
}
```

```
int factorial(int n)
/** pre  0≤ n;
    post  \result = n!; */
{
  return n == 0 ? 1 : n * factorial(n-1);
}
```

Beispiel: Reverse mittels Swap

```
int swap(int a[], int i, int j)
/** pre  i < a_len ∧ j < a_len;
    post  a[i]=a[j]@pre ∧ a[j]=a[i]@pre; */
    {
      int buf = a[j];
      a[j] = a[i];
      a[i] = buf;
    }
    return;
```

```
int rev(int a[], int a_len)
/** pre  0 < a_len;
    post  ...; */
{
  int i;

  i= 0;
  while (i< a_len/2)
    /** inv  ...; */
    {
    swap(a[], i, a_len-i);
    i= i+1;
  }
  return;
}
```

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen
3. Spezifikation von Funktionsdefinitionen

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen
3. Spezifikation von Funktionsdefinitionen
4. Beweisregeln für Funktionsdefinitionen

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen
3. Spezifikation von Funktionsdefinitionen
4. Beweisregeln für Funktionsdefinitionen
5. Semantik des Funktionsaufrufs

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
2. Semantik von Funktionsdefinitionen
3. Spezifikation von Funktionsdefinitionen
4. Beweisregeln für Funktionsdefinitionen
5. Semantik des Funktionsaufrufs
6. Beweisregeln für Funktionsaufrufe

Von Anweisungen zu Funktionen

- Erweiterung unserer Kernsprache um Funktionsdefinition und Deklarationen:

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

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

Rückgaben

Neue Anweisungen: Return-Anweisung

Stmt

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

Rückgabewerte

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

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

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

Rule 14.7 (required)

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

  - Nicht immer möglich, unübersichtlicher Code ...

- Lösung 2: Erweiterung der Semantik von $\Sigma \rightharpoonup \Sigma$ zu $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$

Erweiterte Semantik

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

Erweiterte Semantik

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

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

- Und als Mengen/partielle Funktionen formuliert:

$$g \circ_S f = \{(\sigma, \rho') \mid \exists \sigma'. (\sigma, \sigma') \in f \wedge (\sigma', \rho') \in g\} \cup \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in f\}$$

Semantik von Anweisungen

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

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

$$[\![c_1; c_2]\!]_{\mathcal{C}} = [\![c_2]\!]_{\mathcal{C}} \circ_S [\![c_1]\!]_{\mathcal{C}} \qquad \text{Komposition wie oben}$$

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

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

$$[\![\textbf{return }\ e]\!]_{\mathcal{C}} = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\}$$

$$[\![\textbf{return}]\!]_{\mathcal{C}} = \{(\sigma, (\sigma, *))\}$$

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

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

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1. $$[\![x = 3; x = 4]\!]_{\mathcal{C}} =$$

2. $$[\![x = 3; \textbf{return } x; x = 4]\!]_{\mathcal{C}} =$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1. $$[\![x = 3; x = 4]\!]_C = [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C$$

2. $$[\![x = 3; \textbf{return } x; x = 4]\!]_C =$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}\end{aligned}$$

2.

$$[\![x = 3; \textbf{return } x; x = 4]\!]_C =$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\ &= \{(\sigma, \sigma[x \mapsto 4])\}\end{aligned}$$

2.

$$[\![x = 3; \textbf{return } x; x = 4]\!]_C =$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\ &= \{(\sigma, \sigma[x \mapsto 4])\}\end{aligned}$$

2.

$$[\![x = 3; \textbf{return } x; x = 4]\!]_C = [\![x = 4]\!]_C \circ_S ([\![\textbf{return } x]\!]_C \circ_S [\![x = 3]\!]_C)$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\ &= \{(\sigma, \sigma[x \mapsto 4])\}\end{aligned}$$

2.

$$\begin{aligned}[\![x = 3; \textbf{return } x; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S ([\![\textbf{return } x]\!]_C \circ_S [\![x = 3]\!]_C) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \\ &\quad (\{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![x]\!]_A\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\})\end{aligned}$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\ &= \{(\sigma, \sigma[x \mapsto 4])\}\end{aligned}$$

2.

$$\begin{aligned}[\![x = 3; \textbf{return } x; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S ([\![\textbf{return } x]\!]_C \circ_S [\![x = 3]\!]_C) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \\ &\quad (\{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![x]\!]_A\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S (\{(\sigma, (\sigma, \sigma(x)))\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\})\end{aligned}$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S [\![x = 3]\!]_C \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\ &= \{(\sigma, \sigma[x \mapsto 4])\}\end{aligned}$$

2.

$$\begin{aligned}[\![x = 3; \textbf{return } x; x = 4]\!]_C &= [\![x = 4]\!]_C \circ_S ([\![\textbf{return } x]\!]_C \circ_S [\![x = 3]\!]_C) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \\ &\quad (\{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![x]\!]_A\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S (\{(\sigma, (\sigma, \sigma(x)))\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, (\sigma[x \mapsto 3], \underbrace{\sigma[x \mapsto 3](x)}_{3}))\}\end{aligned}$$

Arbeitsblatt 12.1: Jetzt seid ihr mal dran...

Berechnet die Denotate der folgenden Programme:

1.

$$\begin{aligned}[\![x = 3; x = 4]\!]_{\mathcal{C}} &= [\![x = 4]\!]_{\mathcal{C}} \circ_S [\![x = 3]\!]_{\mathcal{C}} \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\} \\ &= \{(\sigma, \sigma[x \mapsto 4])\}\end{aligned}$$

2.

$$\begin{aligned}[\![x = 3; \textbf{return } x; x = 4]\!]_{\mathcal{C}} &= [\![x = 4]\!]_{\mathcal{C}} \circ_S ([\![\textbf{return } x]\!]_{\mathcal{C}} \circ_S [\![x = 3]\!]_{\mathcal{C}}) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \\ &\quad (\{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![x]\!]_{\mathcal{A}}\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S (\{(\sigma, (\sigma, \sigma(x)))\} \circ_S \{(\sigma, \sigma[x \mapsto 3])\}) \\ &= \{(\sigma, \sigma[x \mapsto 4])\} \circ_S \{(\sigma, (\sigma[x \mapsto 3], \underbrace{\sigma[x \mapsto 3](x)}_{3}))\} \\ &= \{(\sigma, (\sigma[x \mapsto 3], 3))\}\end{aligned}$$

Semantik von Funktionsdefinitionen

$$[\![.]\!]_{fd} : \mathbf{FunDef} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$

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

$$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!]_{fd} v_1, \ldots, v_n = \\ \{(\sigma[p_1 \mapsto v_1] \ldots, [p_n \mapsto v_n], (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![blk]\!]_{blk}\}\}$$

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.

Semantik von Blöcken und Deklarationen

Blöcke bestehen aus Deklarationen und einer Anweisung.

$$[\![.]\!]_{blk} : \mathbf{Blk} \to \Sigma \rightharpoonup (\Sigma \times V_U)$$
$$[\![decls\ stmts]\!]_{blk} \stackrel{def}{=} \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![stmts]\!]_{\mathcal{C}}\}$$

- Von $[\![stmts]\!]_{\mathcal{C}}$ sind nur **Rückgabezustände** interessant.
  - Kein „fall-through“
  - Was passiert ohne **return** am Ende?
- Keine Initialisierungen, Deklarationen haben (noch) keine Semantik.

Spezifikation von Funktionen

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

$$\textbf{FunSpec} ::= \texttt{/**}\ \textbf{pre}\ \textbf{Assn}\ \textbf{post}\ \textbf{Assn}\ \texttt{*/}$$

Vorbedingung **pre** sp; $\overbrace{\Sigma}^{\text{Vorzustand}} \to \mathbb{B}$

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

e**@pre**	Wert von e im **Vorzustand**
result	**Rückgabewert** der Funktion

Beispiel: Fakultät

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

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

Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre  \array (a, a_len) ∧ 0 < a_len;
    post ∀i. 0≤ i< a_len ⟶ a[i]≤ a[ \result ] ∧ 0≤  \result < a_len; */
{
  int r; int j;

  j= 0;
  r= 0;
  while (j< a_len)
    /** inv (∀j. 0≤ j < i⟶a[j]≤ a[r])∧ 0≤ i≤ n ∧ 0≤ r< n; */
    {
    if (a[j]> x) { r= j; }
    j= j+1;
    }
  return r;
}
```

Ziel: Gültigkeit von Spezifikationen

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

$$\textbf{pre } p \textbf{ post } q \models fd$$
$$\iff \forall v_1, \ldots, v_n. [\![fd]\!]_{fd}\, \Gamma\ v_1 \ldots v_n \in [\![\textbf{pre } p \textbf{ post } q]\!]_{\mathcal{B}sp}\, \Gamma$$

- Γ enthält globale Definitionen, insbesondere andere Funktionen.
  - Warum?

Beispiel: Reverse mittels Swap

```
int rev(int a[], int a_len)
/** pre 0 < a_len;
    post ...; */
{
  int i;

  i= 0;
  while (i< a_len/2)
    /** inv ...; */
    {
    swap(a[], i, a_len-i);
    i= i+1;
  }
  return;
}
```

```
void swap(int a[], int i, int j)
/** pre  ∃l. \array(a, l) ∧ i< l ∧ j< l;
    post a[i]=a[j]@pre ∧ a[j]=a[i]@pre; */
{
  int buf;

  buf = a[j];
  a[j] = a[i];
  a[i] = buf;
  return;
}
```

Beispiel: Rekursion

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

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

Semantik von Spezifikationen

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

Semantik von Spezifikationen

$$
\begin{aligned}
[\![.]\!]_{\mathcal{B}sp} &: \mathbf{Env} \to \mathbf{Assn} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B} \\
[\![.]\!]_{\mathcal{A}sp} &: \mathbf{Env} \to \mathbf{Aexpv} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V} \\
[\![!b]\!]_{\mathcal{B}sp}\ \Gamma &= \{((\sigma,(\sigma',v)),\mathit{true}) \mid ((\sigma,(\sigma',v)),\mathit{false}) \in [\![b]\!]_{\mathcal{B}sp}\Gamma\} \\
&\quad \cup \{((\sigma,(\sigma',v)),\mathit{false}) \mid ((\sigma,(\sigma',v)),\mathit{true}) \in [\![b]\!]_{\mathcal{B}sp}\Gamma\} \\
[\![x]\!]_{\mathcal{A}sp}\ \Gamma &= \{((\sigma,(\sigma',v)),\sigma'(x))\} \\
&\dots \\
[\![\mathrm{e@pre}]\!]_{\mathcal{B}sp}\ \Gamma &= \{((\sigma,(\sigma',v)),b) \mid (\sigma,b) \in [\![e]\!]_{\mathcal{B}}\ \Gamma\} \\
[\![\mathrm{e@pre}]\!]_{\mathcal{A}sp}\ \Gamma &= \{((\sigma,(\sigma',v)),a) \mid (\sigma,a) \in [\![e]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![\backslash\mathbf{result}]\!]_{\mathcal{A}sp}\ \Gamma &= \{((\sigma,(\sigma',v)),v)\}
\end{aligned}
$$

$$
[\![\mathbf{pre}\ p\ \mathbf{post}\ q]\!]_{\mathcal{B}sp}\ \Gamma = \{(\sigma,(\sigma',v)) \mid (\sigma,\mathit{true}) \in [\![p]\!]_{\mathcal{B}}\ \Gamma \wedge ((\sigma,(\sigma',v)),\mathit{true}) \in [\![q]\!]_{\mathcal{B}sp}\ \Gamma\}
$$

Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre } p \textbf{ post } q \models fd$$
$$\iff \forall v_1, \ldots, v_n.\ [\![fd]\!]_{fd}\ \Gamma\ v_1 \ldots v_n \in [\![\textbf{pre } p \textbf{ post } q]\!]_{\mathcal{B}sp}\ \Gamma$$

  - Γ enthält globale Definitionen, insbesondere andere Funktionen.

- Wie passt das zu den Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?

- Wie **beweisen** wir das?

Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der **semantischen Gültigkeit**.

$$\textbf{pre}\ p\ \textbf{post}\ q \models fd$$
$$\iff \forall v_1, \ldots, v_n.\ [\![fd]\!]_{fd}\ \Gamma\ v_1 \ldots v_n \in [\![\textbf{pre}\ p\ \textbf{post}\ q]\!]_{\mathcal{B}sp}\ \Gamma$$

  - Γ enthält globale Definitionen, insbesondere andere Funktionen.

- Wie passt das zu den Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?

- Wie **beweisen** wir das? **Erweiterung** des Hoare-Kalküls

Erweiterung des Floyd-Hoare-Kalküls

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

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

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

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

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

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

Erweiterung des Floyd-Hoare-Kalküls: return

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q\, \{\textbf{return} \mid P\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/\,\backslash\textbf{result}]\, \{\textbf{return}\ \ e \mid P\}}$$

- Bei **return** wird die Rückgabespezifikation Q zur Vorbedingung, die reguläre Nachfolgespezifikation wird ignoriert, da die Ausführung von **return** kein Nachfolgezustand hat.
- **return** ohne Argument darf nur bei einer Nachbedingung Q auftreten, die kein **\result** enthält.
- Bei **return** mit Argument ersetzt der Rückgabewert den **\result** in der Rückgabespezifikation.

Erweiterung des Floyd-Hoare-Kalküls: Spezifikation

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\,\texttt{@pre}] \quad \Gamma \vdash \{\Gamma\}\ P'\ \{c \mid \mathit{false}\}}{\Gamma \vdash f(x_1, \ldots, x_n) /{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{\mathit{ds}\ c\}}$$

- Die Parameter x_i werden in **post** Q per Konvention nur als x_i referenziert, aber es ist immer der Wert im **Vorzustand** gemeint (eigentlich x_i **@pre**).
  - Deswegen wird in Q im Hoare-Tripel ersetzt
- Variablen unterhalb von (.) **@pre** werden bei der Substitution (Zuweisungsregel) **nicht ersetzt!**
- (.) **@pre** wird beim Weakening von der Vorbedingung P ersetzt
- Sequentielle Nachbedingung von c ist *false*

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

$$\frac{}{\Gamma \vdash \{\Gamma\}\, P\, \{\{\} \mid P\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P\, \{c_1 \mid R\} \quad \Gamma \vdash \{\Gamma\}\, R\, \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{c_1; c_2 \mid Q\}}$$

$$\frac{}{\Gamma \vdash \{\Gamma\}\, Q[e/x]\, \{l = e \mid Q\}} \qquad \frac{\Gamma \vdash \{\Gamma\}\, P \wedge b\, \{c \mid P\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\textbf{while}\ (b)\ c \mid P \wedge \neg b\}}$$

$$\frac{\Gamma \vdash \{\Gamma\}\, P \wedge b\, \{c_1 \mid Q\} \quad \Gamma \vdash \{\Gamma\}\, P \wedge \neg b\, \{c_2 \mid Q\}}{\Gamma \vdash \{\Gamma\}\, P\, \{\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \mid Q\}}$$

$$\frac{(\Gamma \wedge P) \longrightarrow P' \quad \Gamma \vdash \{\Gamma\}\, P'\, \{c \mid Q'\} \quad (\Gamma \wedge Q') \longrightarrow Q \quad (\Gamma \wedge R') \longrightarrow R}{\Gamma \vdash \{\Gamma\}\, P\, \{c \mid Q\}}$$

Erweiterter Floyd-Hoare-Kalkül II

$$\frac{}{\Gamma \vdash \{\Gamma\}\ Q\,\{\textbf{return} \mid P\}} \qquad \frac{}{\Gamma \vdash \{\Gamma\}\ Q[e/\backslash\textbf{result}]\,\{\textbf{return}\ \ e \mid P\}}$$

$$\frac{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\ \textbf{@pre}] \quad \Gamma \vdash \{\Gamma\}\ P'\,\{c \mid \mathit{false}\}}{\Gamma \vdash f(x_1, \ldots, x_n)/{*}{*}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ c\}}$$

Arbeitsblatt 12.2: Kurzbeispiel

Verifiziert folgendes Kurzbeispiel:

```
int f(int x)
/** post \result = x+1; */
{
  // ???
  x= x+1;
  // ???
  return x;
  // ???
}
```

Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  //
  x= x+1;
  //
  return x;
  // {false | \result = x @pre +1}
}
```

Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  //
  x= x+1;
  // {x = x @pre +1}
  return x;
  // {false | \result = x @pre +1}
}
```

Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  // {x + 1 = x @pre +1}
  x= x+1;
  // {x = x @pre +1}
  return x;
  // {false | \result = x @pre +1}
}
```

Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  // {x + 1 = x @pre +1}
  x= x+1;
  // {x = x @pre +1}
  return x;
  // {false | \result = x @pre +1}
}
```

Weakening der Spezifikationsregel:

$$true \Longrightarrow (x + 1 = x\ \mathbf{@pre} + 1)[x/x\ \mathbf{@pre}]$$

Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  // {x + 1 = x @pre +1}
  x= x+1;
  // {x = x @pre +1}
  return x;
  // {false | \result = x @pre +1}
}
```

Weakening der Spezifikationsregel:

$$true \Longrightarrow (x + 1 = x\ \textbf{@pre} + 1)[x/x\ \textbf{@pre}]$$
$$x + 1 = x + 1$$

Lösungsblatt 12.2: Kurzbeispiel

```
int f(int x)
/** post \result = x+1; */
{
  // {x + 1 = x @pre +1}
  x= x+1;
  // {x = x @pre +1}
  return x;
  // {false | \result = x @pre +1}
}
```

Weakening der Spezifikationsregel:

$$true \Longrightarrow (x + 1 = x\ \textbf{@pre} + 1)[x/x\ \textbf{@pre}]$$
$$x + 1 = x + 1 \quad \checkmark$$

Approximative schwächste Vorbedingung

- Erweiterung zu $\mathrm{awp}(\Gamma, c, Q, Q_R)$ und $\mathrm{wvc}(\Gamma, c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
  - Es werden der **Kontext** Γ und eine **Rückgabespezifikation** Q_R benötigt.
- Es gilt:

$$\bigwedge \mathrm{wvc}(\Gamma, c, Q, Q_R) \Longrightarrow \Gamma \models \{\mathrm{awp}(c, Q, Q_R)\}\, c\, \{Q \mid Q_R\}$$

- Berechnung von awp und wvc:

$$\mathrm{awp}(\Gamma, f(x_1, \ldots, x_n)/{*}{*}\ \mathrm{pre}\ P\ \mathrm{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \mathrm{awp}(\Gamma', blk, false, Q[x_i\ \mathbf{@pre}\ /x_i])$$

$$\mathrm{wvc}(\Gamma, f(x_1, \ldots, x_n)/{*}{*}\ \mathrm{pre}\ P\ \mathrm{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \{(\Gamma \wedge P) \Longrightarrow P'[x_i/x_i\ \mathbf{@pre}]\} \cup \mathrm{wvc}(\Gamma', blk, false, Q[x_i\ \mathbf{@pre}\ /x_i])$$

$$\Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1, \ldots, x_n.\, (P, Q)]$$

$$P' \stackrel{def}{=} \mathrm{awp}(\Gamma', blk, false, Q[x_i\ \mathbf{@pre}\ /x_i])$$

Approximative schwächste Vorbedingung (Revisited)

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

Approximative Verifikationsbedingungen (Revisited)

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

Beispiel: Fakultät

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   //
7   p= 1;
8   //
9   c= 1;
10  //
11  while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
12     p= p*c;
13     if (c == n) {
14       return p;
15       }
16     c= c+1;
17     }
18     //
19 }
```

Beispiel: Fakultät

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   //
7   p= 1;
8   //
9   c= 1;
10  //
11  while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
12     p= p*c;
13     if (c == n) {
14       return p;
15       }
16     c= c+1;
17     }
18     // {false}
19 }
```

Beispiel: Fakultät

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   //
7   p= 1;
8   //
9   c= 1;
10  // {p = (c − 1)! ∧ 0 < c}
11  while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
12     p= p*c;
13     if (c == n) {
14       return p;
15       }
16     c= c+1;
17     }
18     // {false}
19 }
```

Beispiel: Fakultät

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   //
7   p= 1;
8   // {p = (1 − 1)! ∧ 0 < 1}
9   c= 1;
10  // {p = (c − 1)! ∧ 0 < c}
11  while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
12      p= p*c;
13      if (c == n) {
14        return p;
15        }
16      c= c+1;
17      }
18      // {false}
19 }
```

Beispiel: Fakultät

```
1  int fac(int n)
2  /** pre 0 ≤ n;
3      post \result = n!; */
4  {
5   int p, c;
6   // {1 = (1 − 1)! ∧ 0 < 1}
7   p= 1;
8   // {p = (1 − 1)! ∧ 0 < 1}
9   c= 1;
10  // {p = (c − 1)! ∧ 0 < c}
11  while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
12      p= p*c;
13      if (c == n) {
14        return p;
15        }
16      c= c+1;
17      }
18      // {false}
19 }
```

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

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

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    else {
11      //
12      }
13    //
14    c= c+1;
15    //
16    }
```

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    else {
11      //
12      }
13    //
14    c= c+1;
15    // {p = (c − 1)! ∧ 0 < c}
16    }
```

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    else {
11      //
12      }
13    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
14    c= c+1;
15    // {p = (c − 1)! ∧ 0 < c}
16    }
```

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    else {
11      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
12      }
13    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
14    c= c+1;
15    // {p = (c − 1)! ∧ 0 < c}
16    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     //
4     p= p∗c;
5     //
6     if (c == n) {
7       // {p = n@pre!}
8       return p;
9       }
10    else {
11      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
12      }
13    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
14    c= c+1;
15    // {p = (c − 1)! ∧ 0 < c}
16    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c− 1)! ∧ 0< c; */ {
3     //
4     p= p∗c;
5     // {(c = n ∧ p = n@pre!) ∨ (c ≠ n ∧ p = ((c + 1) − 1)! ∧ 0 < c + 1)}
6     if (c == n) {
7       // {p = n@pre!}
8       return p;
9       }
10    else {
11      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
12      }
13    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
14    c= c+1;
15    // {p = (c − 1)! ∧ 0 < c}
16    }
```

Beispiel: Fakultät (Schleifenrumpf)

```
1
2   while (1) /** inv p = (c- 1)! ∧ 0< c; */ {
3     // {(c = n ∧ p · c = n@pre!) ∨ (c ≠ n ∧ p · c = ((c + 1) − 1)! ∧ 0 < c + 1)}
4     p= p*c;
5     // {(c = n ∧ p = n@pre!) ∨ (c ≠ n ∧ p = ((c + 1) − 1)! ∧ 0 < c + 1)}
6     if (c == n) {
7       // {p = n@pre!}
8       return p;
9       }
10    else {
11      // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
12      }
13    // {p = ((c + 1) − 1)! ∧ 0 < c + 1}
14    c= c+1;
15    // {p = (c − 1)! ∧ 0 < c}
16    }
```

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

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

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**:

9. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:
   - $P \longrightarrow (A \wedge B) \vee (C \wedge \neg B) \rightsquigarrow P \wedge B \longrightarrow A, P \wedge \neg B \longrightarrow C$

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

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

Neue **Vereinfachungsregel**:

9. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

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

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

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

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

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

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

Neue **Vereinfachungsregel**:

9. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

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

     (2.1) $p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n$ **@pre**! × Benötigt $n = n$ **@pre**

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

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

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

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

Neue **Vereinfachungsregel**:

9. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

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

  (2.1) $p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n$ **@pre**! × Benötigt $n = n$ **@pre**

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

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

Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

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

Neue **Vereinfachungsregel**:

9. Disjunktionen folgender Form in der Konklusion können vereinfacht werden:

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

  (2.1) $p = (c-1)! \wedge 0 < c \wedge c = n \longrightarrow p \cdot c = n$ **@pre**! × Benötigt $n = n$ **@pre**

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

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

Was fällt uns auf?

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

Was fällt uns auf?

- Die Invariante ist $p = (c-1)! \wedge 0 < c \wedge n = n$ **@pre**
- Da fehlt $c - 1 \leq n$ — wie können wir $c - 1 = n$ am Ende beweisen?
- Mit der Schleifenbedingung 1 gilt **jede** Nachbedingung.
- Austritt aus der Schleife mit $c == n$ — vereinfacht den Beweis.
- Aber: müssen in der Invariante **explizit** spezfizieren, dass n sich nicht ändert.

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Deklarationen und Parameter ✓
2. Semantik von Funktionsdefinitionen ✓
3. Spezifikation von Funktionsdefinitionen ✓
4. Beweisregeln für Funktionsdefinitionen ✓

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Deklarationen und Parameter ✓
2. Semantik von Funktionsdefinitionen ✓
3. Spezifikation von Funktionsdefinitionen ✓
4. Beweisregeln für Funktionsdefinitionen ✓
5. Semantik des Funktionsaufrufs

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Deklarationen und Parameter ✓
2. Semantik von Funktionsdefinitionen ✓
3. Spezifikation von Funktionsdefinitionen ✓
4. Beweisregeln für Funktionsdefinitionen ✓
5. Semantik des Funktionsaufrufs
6. Beweisregeln für Funktionsaufrufe

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe
- Prozeduraufrufe (mit Zuweisung eines Rückgabewertes)

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

Zur Erinnerung: Semantik von Funktionsdefinitionen

$$[\![.]\!]_{fd} : \mathbf{FunDef} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$

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

$$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!]_{fd} = \{((x_1, \ldots, x_n), \sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![blk]\!]_{blk} \circ_S \{(\sigma, \sigma[p_i \mapsto x_i]_{i=1,\ldots,n})\}\}$$

- Die Funktionsargumente sind lokale Deklarationen, die beim Aufruf initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.

Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $[\![f]\!]_{fd}$
- Call by name, call by value, call by reference...?
  - C kennt nur call by value (C-Standard 99, §6.9.1. (10))
  - Was ist mit **Seiteneffekten?** Wie können wir Werte **ändern**?
    - In C: Durch Übergabe von **Referenzen** als **Werte**
      $\Longrightarrow$ Erfordert Modellierung des Speichermodells (nächste Vorlesung)
    - Wir betrachten das hier/heute nicht, somit nur **reine Funktionen**!

Arbeitsblatt 12.3: Funktionsaufrufe

Wie werden Parameter in folgenden Programmiersprachen übergeben?

- **C:** Call-by-value für skalare Typen (arithmetische Typen und Referenzen), damit call-by-reference für aggregierte Typen (**struct**, Felder);
- **Java**:
- **Haskell**:
- **Python**:
- **Other:** (specify)

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von f dem Bezeichner f zuordnen.
- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- Deshalb brauchen wir eine **Umgebung** (Environment):

$$\begin{aligned}\mathbf{Env} &= \mathit{Id} \rightharpoonup [\![\mathbf{FunDef}]\!] \\ &= \mathit{Id} \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)\end{aligned}$$

- Das Environment ist **zusätzlicher Parameter** für alle Definitionen

Semantik von Funktionsaufrufen

$$[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}\Gamma = \{(\sigma,v) \mid ((a_1,\ldots,a_n),\sigma,(\sigma',v)) \in \Gamma(f) \wedge (\sigma,a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$
$$[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma,\sigma') \mid ((a_1,\ldots,a_n),\sigma,(\sigma',v)) \in \Gamma(f) \wedge (\sigma,a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$

- Aufruf von Funktion $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}$ ignoriert Endzustand
- Aufruf einer Prozedur $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}$ ignoriert Rückgabewert

Semantik von Funktionsaufrufen

$$[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}\Gamma = \{(\sigma,v) \mid ((a_1,\ldots,a_n),\sigma,(\sigma',v)) \in \Gamma(f) \wedge (\sigma,a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$
$$[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma,\sigma') \mid ((a_1,\ldots,a_n),\sigma,(\sigma',v)) \in \Gamma(f) \wedge (\sigma,a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$
$$[\![x = f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma,\sigma'[x \mapsto v]) \mid ((a_1,\ldots,a_n),\sigma,(\sigma',v)) \in \Gamma(f) \wedge (\sigma,a_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$

- Aufruf von Funktion $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}$ ignoriert Endzustand
- Aufruf einer Prozedur $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}$ ignoriert Rückgabewert
- Somit: Kombination mit Zuweisung

Erweiterung des Kontext

- Der **Kontext** Γ muss Funktionsbezeichnern ihre **Spezifikation** (Vor/Nachbedingung) zuordnen.
- $\Gamma(f) = \forall x_1, \ldots, x_n. (P, Q)$, für Funktion $f(x_1, \ldots, x_n)$ mit Vorbedingung P und Nachbedingung Q.
- Korrektheit gilt immer nur im **Kontext**, dadurch kann jede Funktion separat verifiziert werden (**Modularität**).

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)}{\Gamma \vdash \{\Gamma\}\, P[t_i/x_i] \wedge y_i\,\texttt{@pre} = y_i\, \{I = f(t_1, \ldots, t_n) \mid Q[t_i/x_i][I/\backslash\texttt{result}]\}}$$

- Γ muss f mit der Vor-/Nachbedingung P, Q enthalten
- In P und Q werden Parameter x_i durch Argumente t_i ersetzt.
- $\backslash$**result** in Q wird durch I ersetzt
- Für alle Variablen y in Q, die mit y **@pre** referenziert werden, wird eine Gleichung $y = y$ **@pre** in die Vorbedingung eingefügt.
  - z.Zt. nur für global Variablen sinnvoll

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
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:

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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   //
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:

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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   //
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:

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**!

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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   //
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:

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**!

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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   //
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:

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**!

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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   //
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:

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**!

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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   //
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:

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**!

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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:

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**!

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(2) $r = (x-1)! \longrightarrow r \cdot x = x$ **@pre**$!$

Problem: Beweis von (2) benötigt Voraussetzung $x = x$ **@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**!

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**!

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**!

Beobachtung

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Der Aufruf einer Funktion **ersetzt** die momentane Nachbedingung — das ist ein Problem!
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben
- Rekursion benötigt keine Extrabehandlung
  - Termination von rekursiven Funktionen wird extra gezeigt

Frame Rule

- Konstanzregel (Rule of Constancy):

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P \wedge R\}\, c\, \{Q \wedge R\}}$$

- Nebenbedingung:
  - c verändert keine Variablen in R, **oder**
  - für alle Programm-Variablen x, die in R vorkommen, gibt es **keine** Zuweisung $x = \ldots$ in c
- Das ist eine **neue Regel**, die **bewiesen** werden muss
- Schwierig zu handhaben bei Rückwärts/Vorwärtsrechung
  - R muss **annotiert** weerden

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe mit Zuweisung eines Rückgabewertes

$$
\begin{aligned}
\textbf{Stmt}\ c ::= \ & l = e \mid c_1; c_2 \mid \{\,\} \mid \textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2 \\
\mid\ & \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ a\ {*}/\ c \mid /{**}\ \{a\}\ {*}/ \\
\mid\ & \textbf{Idt}(a^*) \\
\mid\ & /{**}\,\textbf{const}\ R\ {*}/\ l = \textbf{Idt}(a^*) \\
\mid\ & \textbf{return}\ a^?
\end{aligned}
$$

Approximative schwächste Vorbedingung & Verifikationsbedingung

$$\text{Sei } \Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)$$

$$\text{awp}(\Gamma, /{**}\ \textbf{const}\ \ R\ {*}/ I = f(t_1, \ldots, t_n), U, U_R) \stackrel{def}{=} R \wedge P[t_i/x_i] \quad \text{wenn } I \notin FV(R)$$

$$\text{wvc}(\Gamma, /{**}\ \textbf{const}\ \ R\ {*}/ I = f(t_1, \ldots, t_n), U, U_R) \stackrel{def}{=} \{R \wedge Q[t_i/x_i][I/\ \backslash\textbf{result}] \longrightarrow U\} \quad \text{wenn } I \notin FV(R)$$

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) $0 \leq x \wedge x = x\,\texttt{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\texttt{@pre}!) \vee (x \neq 0 \wedge 0 \leq x - 1 \wedge x = x\,\texttt{@pre})$

(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) $0 \leq x \wedge x = x\,\texttt{@pre} \longrightarrow (x = 0 \wedge 1 = x\,\texttt{@pre}!) \vee (x \neq 0 \wedge 0 \leq x - 1 \wedge x = x\,\texttt{@pre})$

(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) $x = x\,\texttt{@pre} \wedge r = (x-1)! \longrightarrow r \cdot x = x\,\texttt{@pre}!$ ✓

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