Korrekte Software: Grundlagen und Methoden

Vorlesung 12 vom 05.07.22

Funktionen und Prozeduren II

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

Modellierung und Spezifikation von Funktionen

Wir brauchen:

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

Modellierung und Spezifikation von Funktionen

Wir brauchen:

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

Modellierung und Spezifikation von Funktionen

Wir brauchen:

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

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe
- Prozeduraufrufe (mit Zuweisung eines Rückgabewertes)

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

Zur Erinnerung: Semantik von Funktionsdefinitionen

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

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

  $$[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ ds\ c]\!]_{fd}\ \Gamma\ v_1, \ldots, v_n = [\![(t_1\ p_1\ v_1, t_2\ p_2\ v_2, \ldots, t_n\ p_n\ v_n, ds)\ c]\!]_{blk}\ \Gamma$$

- **Aufruf** der Funktion $f(e_1, \ldots, e_n)$ mit Argumenten $e_1, \ldots, e_n$:
  - **Auswertung** der Argumente $v_i = [\![e_i]\!]_{\mathcal{A}}$
  - Einsetzen in die **Semantik** $[\![f]\!]_{fd}(v_1, \ldots, v_n)$

Seiteneffekte bei Funktionsaufrufe

- Seiteneffekte:
  - Funktionen mit Seiteneffekten in zusammengesetzten Ausdrücken sind problematisch
  - In Java ist die Auswertungsreihenfolge fest (links nach rechts)
  - In C unspezifiziert (!)
- Deshalb keine Funktionen in zusammengesetzten Ausdrücken.
- Funktionsaufrufe nur in zwei Formen:
  - Als reine Prozeduren (**ldt**(a^*) vom Typ **void**
  - Als direkte Zuweisung ($l =$ **ldt**(a^*)) des Rückgabewertes
- Call by name, call by value, call by reference. . . ?

Arbeitsblatt 12.1: Funktionsaufrufe

Wie werden Parameter in folgenden Programmiersprachen übergeben?

- **C:**
- **Java**:
- **Haskell**:
- **Python**:
- **Other:** (specify)

Funktionsaufrufe

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

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup \mathbf{Loc}$$

- Wir haben hier einen **Namensraum** für Funktionen und Variablen.

Funktionsaufrufe

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

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup (\mathbf{Loc} + (\mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)))$$

- Wir haben hier einen **Namensraum** für Funktionen und Variablen.

Semantik von Funktionsaufrufen

- Gegegeben Funktionsbezeichner f, Semantik ist

$$\Gamma(f) = \{(\underbrace{(v_1, \ldots, v_n)}_{\textit{Parameterwerte}}, (\underbrace{\quad\sigma\quad}_{\textit{Anfangszustand}}, \underbrace{(\sigma', a)}_{\textit{Endzustand,Rückgabewert}}))\}$$

- Damit:

$$[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, \sigma') \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\ \Gamma\}$$
$$[\![x = f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, \sigma'[x \mapsto a]) \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$

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

Semantik von Funktionsaufrufen

- Gegegeben Funktionsbezeichner f, Semantik ist

$$\Gamma(f) = \{(\underbrace{(v_1, \ldots, v_n)}_{\textit{Parameterwerte}}, (\underbrace{\sigma}_{\textit{Anfangszustand}}, \underbrace{(\sigma', a)}_{\textit{Endzustand,Rückgabewert}}))\}$$

- Damit:

$$[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, \sigma') \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\ \Gamma\}$$
$$[\![x = f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, \sigma'[x \mapsto a]) \mid ((v_1, \ldots, v_n), \sigma, (\sigma', a)) \in \Gamma(f) \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}$$

- Aufruf einer Prozedur $[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}$ ignoriert Rückgabewert
  - Somit: Kombination mit Zuweisung
- Wir modellieren nur call-by-value.
  - C kennt nur call by value, allerdings sind Referenzen auch Werte (kommt noch)
  - Modellierung erlaubt Felder als Werte, im **Gegensatz** zu C.

Umgebung für den Kalkül

- Für Funktionsaufrufe gibt es eine **Umgebung**:

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup (\mathbf{Loc} + (\mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)))$$

- Deshalb muss für den Kalkül eine **Umgebung** Γ Funktionsbezeichnern ihre **Spezifikation** (Vor- und Nachbedingung, sowie Parameter) zuordnen:

$$\mathbf{Env}_{\text{fun}} = \mathbf{Idt} \rightharpoonup (\mathbf{Idt}^N \times \mathbf{Assn} \times \mathbf{Assn})$$

- $\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)$, für $f(x_1, \ldots, x_n)$/** pre P post Q */
- Korrektheit gilt immer nur im **Kontext** einer Umgebung, dadurch kann jede Funktion separat verifiziert werden (**Modularität**).
  - Umgebung wird zusätzliches Argument der Regeln.
  - Notation: $\Gamma \vdash \{P\}\, c\, \{Q \mid Q_R\}$.

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

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

- 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 l 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:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  //
16  r = fac(x- 1);
17  //
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  //
16  r = fac(x- 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    //
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    //
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post   \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     //
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   //
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre  0 ≤ x;
3      post \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

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

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x$ **@pre**$!$

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$

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

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

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

(1.1) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x = 0$
$\longrightarrow 1 = x\,\texttt{@pre}!$ ✓

(1.2) $0 \leq x \wedge x = x\,\texttt{@pre} \wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$

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

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x$ **@pre**$!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

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

Beispiel: die Fakultätsfunktion, rekursiv

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1}
14    }
15  // {0 ≤ x − 1}
16  r = fac(x− 1);
17  // {r = (x − 1)!}
18  // {r · x = x @pre!}
19  return r* x;
20  // {false | \result = x @pre!}
21 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x$ **@pre**$!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

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

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

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben

Beobachtungen

- Bei der Verifikation von f muss die Spezifikation von f Teil des Kontextes sein.
- Wir brauchen **gar keine** Invariante — ist durch die Nachbedingung gegeben
- Rekursion benötigt keine Extrabehandlung
  - Termination von rekursiven Funktionen wird extra gezeigt

Beobachtungen

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

Frame Rule

- Konstanzregel (Rule of Constancy):

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

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

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe mit Zuweisung eines Rückgabewertes

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

Approximative schwächste Vorbedingung & Verifikationsbedingung

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

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

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

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) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1$
$\wedge x = x$ **@pre**$)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(1.3) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow x = x$ **@pre**

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

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@ pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1$
$\wedge x = x$ **@pre**$)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(1.3) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow x = x$ **@pre** ✓

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

Beispiel: die Fakultätsfunktion

```
1  int fac(int x)
2  /** pre   0 ≤ x;
3      post  \result = x!; */
4  {
5   int r = 0;
6
7   // {(x = 0 ∧ 1 = x @pre!) ∨ (x ≠ 0 ∧ 0 ≤ x − 1 ∧ x = x @pre)}
8   if (x == 0) {
9     // {1 = x @pre!}
10    return 1;
11    // {0 ≤ x − 1 ∧ x = x @pre | \result = x @pre!}
12    } else {
13    // {0 ≤ x − 1 ∧ x = x @pre}
14    }
15  // {0 ≤ x − 1 ∧ x = x @pre}
16  /** const x= x@ pre */ r = fac(x− 1);
17  // {r · x = x @pre!}
18  return r* x;
19  // {false | \result = x @pre!}
20 }
```

Verifikationsbedingungen:

(1) $0 \leq x \wedge x = x$ **@pre**
$\longrightarrow (x = 0 \wedge 1 = x$ **@pre**$!)$
$\vee (x \neq 0 \wedge 0 \leq x - 1$
$\wedge x = x$ **@pre**$)$

(1.1) $0 \leq x \wedge x = x$ **@pre** $\wedge x = 0$
$\longrightarrow 1 = x!$ ✓

(1.2) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow 0 \leq x - 1$ ✓

(1.3) $0 \leq x \wedge x = x$ **@pre** $\wedge x \neq 0$
$\longrightarrow x = x$ **@pre** ✓

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

Arbeitsblatt 12.2: Fakultät endrekursiv

Hier nochmal die Fakultät (endrekursiv und buggy):

```
int factorial(int n)
/** pre ???;
    post ???; */
{
  int f;

  f= fact(0, n);
  return f;

}

int fact(int acc, int n)
/** pre ???;
    post ???; */
{
  int r;

  if (n == 0) return 1;
  r= fact(acc* n, n-1);
  return r;
}
```

1. Annotiert das Programm mit Vor/Nachbedingungen.
2. Findet und berichtigt die Fehler.
3. Berechnet die Verifikationsbedingungen.
4. Beweist die Verifikationsbedingungen.

Arbeitsblatt 12.3: Binäre Produke

Das Binärprodukt, rekursiv:

```
int binprod(int m, int n)
/** pre ?;
    post ?;
 */
{
  int r;

  if (n == 0) return 0;
  r= binprod(2* m, n/2);
  return r+ m*(n%2);
}
```

1. Annotiert das Programm mit Vor/Nachbedingungen
2. Berechnet die Verifikationsbedingungen
3. Beweist die Verfiktionsbedingungen

Arbeitsblatt 12.3: Binäre Produke

Das Binärprodukt, rekursiv:

```
int binprod(int m, int n)
/** pre 0 ≤ n;
    post \result = m* n;
 */
{
  int r;

  if (n == 0) return 0;
  r= binprod(2* m, n/2);
  return r+ m*(n%2);
}
```

1. Annotiert das Programm mit Vor/Nachbedingungen
2. Berechnet die Verifikationsbedingungen
3. Beweist die Verfikationsbedingungen

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