Korrekte Software: Grundlagen und Methoden
Vorlesung 11 vom 02.07.20
Spezifikation von Funktionen

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

- Einführung
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Der Floyd-Hoare-Kalkül
- Invarianten und die 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: 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;
}
```

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


Beispiel: Rekursion

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


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:

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

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

Rückgaben

Neue Anweisungen: Return-Anweisung

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

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: $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V})$ $\Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- 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:

$$\begin{aligned} g \circ_S f = &\{(\sigma, \rho') \mid (\sigma, \sigma') \in f \wedge (\sigma', \rho') \in g\} \\ &\cup \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in f\} \end{aligned}$$

Semantik von Anweisungen

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

$$\begin{aligned}
[\![x = e]\!]_{\mathcal{C}} &= \{(\sigma, \sigma[a/l]) \mid (\sigma, l) \in [\![x]\!]_{\mathcal{L}}, (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\} \\
[\![c_1; c_2]\!]_{\mathcal{C}} &= [\![c_2]\!]_{\mathcal{C}} \circ_S [\![c_1]\!]_{\mathcal{C}} \qquad \text{Komposition wie oben} \\
[\![\{\,\}]\!]_{\mathcal{C}} &= \mathbf{Id}_\Sigma \qquad \mathbf{Id}_\Sigma := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
[\![\mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1]\!]_{\mathcal{C}} &= \{(\sigma, \rho') \mid (\sigma, \mathit{true}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \rho') \in [\![c_0]\!]_{\mathcal{C}}\} \\
&\quad \cup \{(\sigma, \rho') \mid (\sigma, \mathit{false}) \in [\![b]\!]_{\mathcal{B}} \wedge (\sigma, \rho') \in [\![c_1]\!]_{\mathcal{C}}\} \\
&\quad \text{mit } \rho' \in \Sigma \cup \Sigma \times \mathbf{V}_U \\
[\![\mathbf{return}\ \ e]\!]_{\mathcal{C}} &= \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\} \\
[\![\mathbf{return}]\!]_{\mathcal{C}} &= \{(\sigma, (\sigma, *))\}
\end{aligned}$$

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

Arbeitsblatt 11.1: Jetzt seid ihr mal dran...

- Berechnet die Denotate der folgenden Programme:
-

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

-

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

Semantik von Funktionsdefinitionen

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

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

$$\begin{aligned}
&[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!]_{\mathcal{D}_{fd}} v_1, \ldots, v_n = \\
&\qquad \{(\sigma, (\sigma', v)) \mid (\sigma[v_1/p_1, \ldots, v_n/p_n], (\sigma', v)) \in \mathcal{D}_{blk}[\![blk]\!]\}
\end{aligned}$$

- 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.

$$\mathcal{D}_{blk}[\![.]\!] : \mathbf{Blk} \to \Sigma \rightharpoonup (\Sigma \times V_U)$$

$$\mathcal{D}_{blk}[\![decls\ stmts]\!] \stackrel{def}{=} \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![stmts]\!]_{\mathcal{C}}\}$$

- 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:

$$\mathbf{FunSpec} ::= /\!**\ \mathbf{pre}\ \mathbf{Assn}\ \mathbf{post}\ \mathbf{Assn}\ */$$

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

`\old(e)`	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)! ∧ c ≤ n+1 ∧ 0 < c} */ {
    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] ≤ \result}; */
{
  int x; int j;

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


Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre {0 < a_len};
    post {\result = max(seq(a, a_len))}; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /** inv {j > 0 ⟶ x = max(seq(a, j)) ∧ j ≤ a_len}; */
    {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```


Ziel: Gültigkeit von Spezifikationen

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

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

  - 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;
}
```

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


Beispiel: Rekursion

```
int factorial(int n)
/** pre   {n ≥ 0}
    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** kann nicht in Funktionen vom Typ **void** auftreten.

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

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

Gültigkeit von Spezifikationen

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

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

  - 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 ($\models \{P\}\, c\, \{Q|Q_R\}$)

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.

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

Erweiterung des Floyd-Hoare-Kalküls: return

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

- 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.

Erweiterung des Floyd-Hoare-Kalküls: Spezifikation

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

- 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 **\old**(x_i)).
  - Deswegen wird in Q im Hoare-Tripel ersetzt
- Variablen unterhalb von **\old**(.) werden bei der Substitution (Zuweisungsregel) **nicht ersetzt!**
- **\old**(.) wird beim Weakening von der Vorbedingung P ersetzt
- Sequentielle Nachbedingung von c ist *false*

Beispiel: Suche

```
int findmax(int a[], int a_len)
/** pre {0 < a_len};
    post {\result = max(seq(a, a_len))}; */
{ ...
```

$$\frac{\dfrac{(\Gamma \wedge 0 < a\_len) \Longrightarrow P'[a/\backslash\textbf{old}(a), a\_len/\backslash\textbf{old}(a\_len)]}{\Gamma \vdash \{P'\}\ c\ \{\mathit{false}|\backslash\textbf{result} = \max(\text{seq}(\backslash\textbf{old}(a), \backslash\textbf{old}(a\_len)))\}}}{\begin{array}{l}\Gamma \vdash\ \mathit{findmax}(\mathit{int}\ a[], \mathit{int}\ a\_len) \\ \quad /{**}\ \text{pre}\ \{0 < a\_len\} \\ \quad\quad \text{post}\ \{\backslash\textbf{result} = \max(\text{seq}(a, a\_len))\}{*}/\ \{\ldots\}\end{array}}$$

- Wobei P' noch Ausdrücke **\old**(a_len) enthalten kann,
- die dann ersetzt werden zu a_len in $P'[a/\backslash\textbf{old}(a), a\_len/\backslash\textbf{old}(a\_len)]$

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

$$\frac{}{\Gamma \vdash \{P\}\ \{\}\ \{P|Q_R\}} \qquad \frac{\Gamma \vdash \{P\}\ c_1\ \{R|Q_R\} \quad \Gamma \vdash \{R\}\ c_2\ \{Q|Q_R\}}{\Gamma \vdash \{P\}\ c_1; c_2\ \{Q|Q_R\}}$$

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

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

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

Erweiterter Floyd-Hoare-Kalkül II

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

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

Approximative schwächste Vorbedingung

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

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

- Berechnung von awp und wvc:

$$\begin{array}{r}\text{awp}(\Gamma, f(x_1,\ldots,x_n)/{**}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \\ \text{awp}(\Gamma', blk, \mathit{false}, Q[\backslash\textbf{old}(x_i)/x_i]) \\ \text{wvc}(\Gamma, f(x_1,\ldots,x_n)/{**}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \\ \{(\Gamma \wedge P) \Longrightarrow P'[x_i/\backslash\textbf{old}(x_i)]\} \cup \text{wvc}(\Gamma', blk, \mathit{false}, Q[\backslash\textbf{old}(x_i)/x_i]) \\ \Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1,\ldots,x_n.\,(P, Q)] \\ P' \stackrel{def}{=} \text{awp}(\Gamma', blk, Q[\backslash\textbf{old}(x_i)/x_i], Q[\backslash\textbf{old}(x_i)/x_i])\end{array}$$

Approximative schwächste Vorbedingung (Revisited)

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

Approximative Verifikationsbedingungen (Revisited)

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

Beispiel: Fakultät

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c− 1)!; */ {
     // {p == (c − 1)! ∧ true}
     // ↯
     // {(c == n ∧ p == n!) ∨ (c ≠ n ∧ p ∗ c = c!)}
     if (c == n) { return p; } else {}
     // {p ∗ c == c!}
     p= p∗c;
     // {p == c!}
     // {p == ((c + 1) − 1)!}
     c= c+1;
     // {p == (c − 1)!}
     }
}
```


Beispiel: Fakultät (berichtigt)

```
int fac(int n)
/** pre 0 ≤ n;
    post \result == n!; */
{
 int p, c;
 p= 1;
 c= 1;
 while (1) /** inv p == (c− 1)! ∧ 0< c; */ {
     // {p == (c − 1)! ∧ 0 < c ∧ true}
     // {p ∗ c == c! ∧ 0 < c}
     p= p∗c;
     // {p == c! ∧ 0 < c}
     // {(c == n ∧ p == n! ∧ 0 < c) ∨ (c ≠ n ∧ p == c! ∧ 0 < c)}
     if (c == n) {
       //** {c == n ∧ p == n! ∧ 0 < c} */
       // {c == n ∧ p == n!}
       return p;
     } else {}
     // {p == c! ∧ 0 < c}
     // {p == ((c − 1) + 1)! ∧ 0 < c + 1}
     c= c+1;
     // {p == (c − 1)! ∧ 0 < c}
     }
}
```


Zusammenfassung

- Funktionen sind **zentrales Modularisierungskonzept**
- Wir müssen Funktionen **modular** verifizieren können
- Erweiterung der **Semantik:**
  - Semantik von Deklarationen und Parameter — straightforward
  - Semantik von **Rückgabewerten** — Erweiterung der Semantik
- Erweiterung der **Spezifikationen:**
  - Spezifikation von Funktionen: **Vor-/Nachzustand** statt logischer Variablen
- Erweiterung des Hoare-Kalküls:
  - Environment, um andere Funktionen zu nutzen
  - Gesonderte Nachbedingung für Rückgabewert/Endzustand
- Es fehlt: **Funktionsaufruf** und **Parameterübergabe**

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)

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

Zur Erinnerung: Semantik von Funktionsdefinitionen

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

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

$$
\begin{aligned}
&[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!]_{\mathcal{D}_{fd}} = \\
&\quad \lambda v_1, \ldots, v_n.\, \{(\sigma, (\sigma', v)) \mid \\
&\qquad (\sigma, (\sigma', v)) \in \mathcal{D}_{blk}[\![blk]\!] \circ_S \{(\sigma, \sigma[v_1/p_1, \ldots, v_n/p_n])\}\}
\end{aligned}
$$

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.
- Von $\mathcal{D}_{blk}[\![blk]\!]$ sind nur **Rückgabezustände** interessant.
  - Kein „fall-through"

Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $[\![f]\!]_{\mathcal{D}_{fd}}$
- 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**!

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der **Definition** von f dem Bezeichner f zuordnen.
- Deshalb brauchen wir eine **Umgebung** (Environment):

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

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

Nebenbedingungen von Funktionsaufrufen

- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch **statische Analyse** verhindert werden
- **Reine Funktion** (pure function):
  - keine (sichtbaren) Seiteneffekte und Spezifikation der Form

    $Q[\backslash\mathbf{result}]$

    ...und Q enthält nur formale Parameter **innerhalb von** $\backslash\mathbf{old}(.)$

Semantik von Funktionsaufrufen

$$
\begin{aligned}
[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}\Gamma &= \{(\sigma,\sigma') \mid (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \\
&\qquad \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\} \\
[\![x = f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}\Gamma &= \{(\sigma,\sigma'[v/x]) \mid (\sigma,(\sigma',v)) \in \Gamma(f)(v_1,\ldots,v_n) \\
&\qquad \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma\}
\end{aligned}
$$

- Aufruf von Funktion $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{A}}$ ignoriert Endzustand
- Aufruf einer rein funktionalen Prozedur $[\![f(t_1,\ldots,t_n)]\!]_{\mathcal{C}}$ ohne Rückgabewert hat keinen Effekt
- Somit: Kombination mit Zuweisung
- Zuweisungen gehen nur anm Programmvariablen, Feldeinträge oder Struktur-Einträge vom Typ **Z** oder **C**.

Beispiel: Reverse mittels Swap geht nicht...

```
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;
}
```

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


Kontext

- Wir benötigen ferner einen **Kontext** Γ, der Funktionsbezeichnern ihre **Spezifikation** (Vor/Nachbedingung) zuordnet.
- $\Gamma(f) = \forall x_1,\ldots,x_n.\,(P,Q)$, 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

$$
\frac{\Gamma(f) = \forall x_1,\ldots,x_n.\,(P,Q)}{\begin{array}{l}\Gamma \vdash \{P[t_i/x_i]\} \\ \qquad l = f(t_1,\ldots,t_n) \\ \qquad \{Q[t_i/x_i][l/\backslash\mathbf{result}]_{\mathbf{old}(Y)\to Y} | Q_R\}\end{array}}
$$

- Γ muss f mit der Vor-/Nachbedingung P, Q enthalten
- In P und Q werden Parameter x_i durch Argumente t_i ersetzt.
- In Q werden die x_i unterhalb von $\backslash\mathbf{old}(.)$ durch t_i ersetzt,
- Alle Ausdrücke der Form $\backslash\mathbf{old}(e)$ werden durch e ersetzt,
- $\backslash\mathbf{result}$ in Q wird durch l ersetzt

Beispiel: die Fakultätsfunktion, rekursiv

```
int fac(int x)
/** pre  0 ≤ x;
    post \result = \old(x!) */
{
 int r = 0;
 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r* x;
}
```

$$
\frac{\Gamma(fac) = \forall x_1,\ldots,x_n.\,(0 \le x, \backslash\mathbf{result} = \backslash\mathbf{old}(x!))}{\Gamma \vdash \{\qquad\qquad\} \; l = fac(2*y) \; \{\qquad\qquad | Q_R\}}
$$

Beobachtung

- Der Aufruf einer Funktion **ersetzt** die momentane Nachbedingung — das ist ein Problem bei Schleifen!
- Wir brauchen keine Invariante mehr — 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
- Ist aber schwierig zu handhaben als Teil von wvc()
  - Hier braucht man eine Behandlung ähnlich zum Einfügen von Zwischenbedingungen

Funktionsaufrufe und Rückgaben

Neue Ausdrücke und Anweisungen:

- Funktionsaufrufe mit Zuweisung eines Rückgabewertes

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

Approximative schwächste Vorbedingung & Verifikationsbedingung

$$\Gamma(f) = \forall x_1, \ldots, x_n. (P, Q)$$

$$\text{awp}(\Gamma, //{*}{*}\ \textbf{const}\ R\ {*}/ l = f(t_1, \ldots, t_n), Q, Q_R) \stackrel{def}{=} R \wedge P[t_i/x_i]$$
wenn $l \notin R$

$$\text{wvc}(\Gamma, //{*}{*}\ \textbf{const}\ R\ {*}/ l = f(t_1, \ldots, t_n), Q, Q_R) \stackrel{def}{=} \{R \wedge Q[t_i/x_i][l/\backslash\textbf{result}]_{\backslash\textbf{old}(Y)\to Y} \Longrightarrow Q\}$$
wenn $l \notin R$

Beispiel: die Fakultätsfunktion

```
// {y = 5 ∧ x = 2 ∗ y}
/** const y = 5 ∧ x = 2 ∗ y */
l = fac(x);
// {l = 10!}
```

```
int fac(int x)
/** pre  0 ≤ x;
    post \result = \old(x!) */{
 int r = 0;
 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r* x;
}
```

$$\text{awp}(\Gamma, //{*}{*}\ \textbf{const}\ y = 5 \wedge x = 2 * y\ {*}/\ l = fac(x), l = 10!, Q_R) \stackrel{def}{=} y = 5 \wedge x = 2 * y \wedge 0 \leq x$$

$$\text{wvc}(\Gamma, //{*}{*}\ \textbf{const}\ y = 5 \wedge x = 2 * y\ {*}/\ l = fac(x), l = 10!, Q_R) \stackrel{def}{=} \{y = 5 \wedge x = 2 * y \wedge l = x! \Longrightarrow l = 10!\}$$

Zusammenfassung

- Aufruf von Funktionen:
  - Funktionen ohne Seiteneffekt in Kombination mit Zuweisung
- Aufruf einer Funktion **ersetzt** Vor/Nachbedingung
- **Einschränkungen**
  - Keine Seiteneffekte
  - Keine Veränderungen von/Zuweisungen ganzen Strukturen oder Feldern
  - Prozeduren sind unbrauchbar/überflüssig
- Fazit: Funktionen sind nicht ganz so straightforward