Korrekte Software: Grundlagen und Methoden
Vorlesung 11 vom 28.06.22
Funktionen und Prozeduren I

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2022

Fahrplan

Funktionen & Prozeduren

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

Beispiel: Rekursion

Fakultät

```
// {N == n ∧ 0 ≤ n}
p= 1;
while (0< n)
  /** inv p == (N–n)! ∧ 0≤ n; */ {
  p= p* n;
  n= n–1;
  }
// {p == N!}
```

Verkapselt als Funktion

```
int factorial(int n)
/** pre  0≤ n;
    post \result == n!; */
{
 int p;
 p= 1;
 while (0< n)
   /** inv p == (n@pre–n)! ∧
0≤ n; */ {
   p= p* n;
   n= n–1;
   }
 return n;
}
```

- **Spezifikation** mit Vor-/Nachbedingung
- Keine logischen Variablen nötig, **\result**für Ergebnis
- Lokale Variablen, von außen nicht sichtbar (scoping)

Modellierung und Spezifikation von Funktionen

Wir brauchen:

1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
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

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

Rückgabewerte

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

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

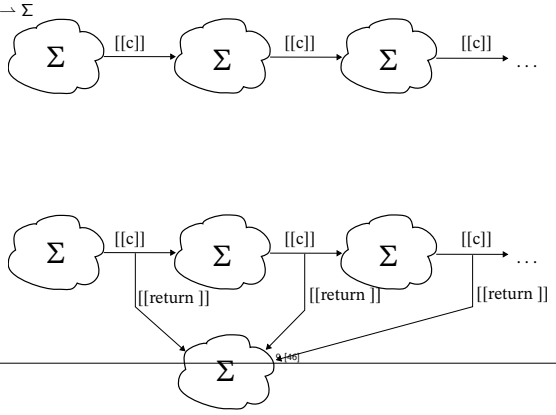
- Lösung 1: verbieten!
  - MISRA-C (Guidelines for the use of the C language in critical systems):

> 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

Denotationale Semantik der return-Anweisung

- Alt: $\Sigma \rightharpoonup \Sigma$



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

Erweiterte Semantik

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

Komposition mit Rückgabewerten

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

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

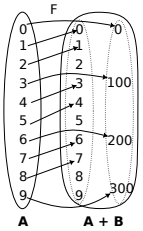
- Als Mengen: $f, g \subseteq (A \times (A + B)) \cong A \times A + A \times B$

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

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

Arbeitsblatt 11.1: Komposition mit Rückgabewerten

Sei $\mathbf{A} = \{0, \ldots, 9\}$
$\mathbf{B} = \{0, 100, 200, 300\}$
Betrachte $F : \mathbf{A} \rightharpoonup \mathbf{A} + \mathbf{B}$



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

Semantik von Anweisungen

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

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

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

Arbeitsblatt 11.2: Semantik mit Rückgabe

Berechnet die Denotate der folgenden Programme:

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

Erweiterung des Floyd-Hoare-Kalküls

- Neue Semantik: $\mathbf{Stmt} \to \Sigma \rightharpoonup (\Sigma \cup \Sigma \times \mathbf{V}_U)$
- Wie passt das zu unseren Floyd-Hoare-Tripeln $\models \{P\}\, c\, \{Q\}$?
- Problem: Tripel behandel **einen** Nachzustand, jetzt haben wir **zwei** ...
- Lösung: **Erweiterung** des Tripels um eine explizite Nachbedingung für den **Rückgabezustand**

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

Erweiterung des Floyd-Hoare-Kalküls

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

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.

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

Erweiterung des Floyd-Hoare-Kalküls: return

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

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

Zusammenfassung: Erweiterter Floyd-Hoare-Kalkül

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

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

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

$$\frac{P \longrightarrow P' \quad \vdash \{P'\}\,c\,\{Q' \mid R'\} \quad Q' \longrightarrow Q \quad R' \longrightarrow R}{\vdash \{P\}\,c\,\{Q \mid R\}} \qquad \frac{}{\vdash \{Q\}\ \textbf{return}\ \{P \mid Q\}}$$

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

Arbeitsblatt 11.3: Kurzbeispiel

Verifiziert folgendes Kurzbeispiel:

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


Lösungsblatt 11.3: Kurzbeispiel

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

Beweisverpflichtung:

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

Approximative schwächste Vorbedingung

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

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

Approximative schwächste Vorbedingung (Revisited)

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

Approximative Verifikationsbedingungen (Revisited)

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

Beispiel: Fakultät

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

Beispiel: Fakultät (Beweisverpflichtungen I)

Unvereinfacht:

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

Vereinfacht:

$$(1.1)\quad 0 \le n \longrightarrow 1 = 0! \quad \checkmark$$
$$(1.2)\quad 0 \le n \longrightarrow 0 < 1 \quad \checkmark$$
$$(3)\quad false \longrightarrow false \quad \checkmark$$

Beispiel: Fakultät (Schleifenrumpf)

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


Beispiel: Fakultät (Beweisverpflichtung II)

Unvereinfacht:

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

Neue **Vereinfachungsregel**:

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

Damit Vereinfachung:

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

Was fällt uns auf?

- Die Invariante ist $p = (c-1)! \wedge 0 < c$
- Da fehlt $c - 1 \le n$ — wie können wir $c - 1 = n$ am Ende beweisen?
- Mit der Schleifenbedingung 1 gilt **jede** Nachbedingung.
- Bei der Rückgabe ist $c == n$ — vereinfacht den Beweis.
- Essenziell: Schleife wird **nicht** verlassen.
- Nachbedigung *false* forciert dass Programm nur mit **return** verlassen wird.

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.

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

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.
  - ... aber was ist die Semantik von Deklarationen?

Deklarationen und ihre Semantik

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

Abstraktes Speichermodell

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

$\nu : \Sigma \to \mathbf{Loc}$	$\nu(\sigma) \notin dom(\sigma)$	Neue Lokation
$rem : \Sigma \times \mathbf{Loc} \to \Sigma$	$l \notin dom(rem(\sigma, l))$	Deallokation

  - Ferner ist **Loc** abgeschlossen über:

$$l \in \mathbf{Loc}, i \in \mathbf{Idt} \Longrightarrow l.i \in \mathbf{Loc}$$
$$l \in \mathbf{Loc}, n \in \mathbb{Z} \Longrightarrow l[n]\mathbf{Loc}$$

Die Umgebung Γ

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

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

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

Semantik von Deklarationen

- Zuerst: lokale Variablen.

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

- Neue Variable alloziieren, Block ausführen, Variable wieder deallokieren.
  - Ist nicht ganz korrekt: wir müssen neue Variable initialisieren (ggf. mit unbestimmten Wert), ansonsten wird sie nicht Teil von $\mathit{dom}(\sigma)$, dem Definitionsbereich von σ.
  - Gilt insbesondere für die Funktionsparameter, die mit dem aktuellen Wert des Funktion initialisiert werden.

Semantik von Blöcken

- Blöcke bestehen aus Deklarationen und einer Anweisung c:

$$[\![-]\!]_{blk} : \mathbf{Blk} \to \mathbf{Env} \to \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$
$$[\![((\mathit{typ}\ \mathit{idt}) : \mathit{ds})\ c]\!]_{blk}\Gamma = \text{local}(\lambda l.\, [\![\mathit{ds}\ c]\!]_{blk}(\Gamma[\mathit{idt} \mapsto l]))$$
$$[\![[]\ c]\!]_{blk}\Gamma = \{(\sigma, (\sigma', v)) \mid (\sigma, (\sigma', v)) \in [\![c]\!]_C\Gamma\}$$

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

Arbeitsblatt 11.4: Semantik mit Return

Gegeben folgende Funktion f:

```
int f(int y)
{
 int x;
 x= 7;
 if (y == 0) return x;
 x= x+4;
}
```

Was ist die denotationale Semantik von f?

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

Spezifikation von Funktionen

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

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

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

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

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;
}
```


Arbeitsblatt 11.5: Suche

Spezifiziert die Suche nach dem maximalen Element:

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

  j= 0;
  r= 0;
  while (j< a_len)
    /** inv (∀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;
}
```


Gültigkeit von Spezifikationen

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

$$\models \mathit{fd}\ \mathbf{pre}\ p\ \mathbf{post}\ q \iff (\forall v_1, \ldots, v_n.\, [\![P]\!]_{\mathcal{B}}\Gamma(\sigma) \implies [\![Q]\!]_{\mathcal{B}sp}\Gamma(\sigma, [\![\mathit{fd}]\!]_{fd}\Gamma(v_1 \ldots v_n)(\sigma)))$$

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

$$\models \mathit{fd}\ \mathbf{pre}\ p\ \mathbf{post}\ q \iff$$
$$\forall v_1, \ldots, v_n, \sigma_0.\ \models \{\lambda\sigma.\, [\![P]\!]_{\mathcal{B}}\Gamma(\sigma) \wedge \sigma_0 = \sigma\}\ [\![\mathit{fd}]\!]_{blk}\Gamma(v_1, \ldots, v_n)\ \{\mathit{false} \mid \lambda\sigma.[\![Q]\!]_{\mathcal{B}sp}\Gamma(\sigma_0, \sigma)\}$$

Beispiel: Rekursion

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

  if (n == 0) {
    return 1;
    }
  else {
    x = factorial(n−1);
    return n ∗ x;
    }
}
```

Semantik von Spezifikationen

- Vorbedingung: Auswertung als $[\![sp]\!]_{\mathcal{B}}\,\Gamma$ 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

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

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

Zusammenfassung

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

Verifikation

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

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

- Berechnung von awp und wvc:

$$\text{awp}(\Gamma, f(x_1,\ldots,x_n)/\!*\!*\ \text{pre } P \text{ post } Q\ *\!/\ \{ds\ c\}) \stackrel{def}{=} \text{awp}(\Gamma', c, false, Q[x_i\,\texttt{@pre}\,/x_i])$$
$$\text{wvc}(\Gamma, f(x_1,\ldots,x_n)/\!*\!*\ \text{pre } P \text{ post } Q\ *\!/\ \{ds\ c\}) \stackrel{def}{=} \{P \wedge x_i = x_i\,\texttt{@pre} \Longrightarrow P'\} \cup \text{wvc}(\Gamma', c, false, Q[x_i\,\texttt{@pre}\,/x_i])$$
$$\Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1,\ldots,x_n.\,(P,Q)]$$
$$P' \stackrel{def}{=} \text{awp}(\Gamma', c, false, Q[x_i\,\texttt{@pre}\,/x_i])$$

Beispiel

Gegeben folgende kurze Funktion vom Arbeitsblatt 11.1:

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

- Verifikationsbedingung:

$$(1)\ true \wedge x\,\texttt{@pre} = x \Longrightarrow x\,\texttt{@pre} < x+1 (1)\ x < x+1 \quad \checkmark$$

Zusammenfassung

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