Korrekte Software: Grundlagen und Methoden
Vorlesung 8 vom 29.05.24
Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Organisatorisches

Die Vorlesung am 05.06.2024 muss wegen Abwesenheit der Veranstalter **entfallen**!

Am 06.06. geht es weiter.

Fahrplan

Motivation

- Immer nur ganze Zahlen ist doch etwas langweilig.
- Weitere Basisdatentypen von C (Felder, Zeichenketten, Strukturen)
- Noch rein funktional, keine Referenzen
- Nicht behandelt, aber nur syntaktischer Zucker: **enum**
- Prinzipiell: keine **union**

Arrays

- Beispiele:

```
int six[6] = {1,2,3,4,5,6};
int a[3][2];
int b[][] = { {1, 0},
              {3, 7},
              {5, 8} }; /* Ergibt Array [3][2] */
```

  - b [2][1] liefert 8, b [1][0] liefert 3
- Index startet mit 0, *row-major order*
- In C0: Felder als echte Objekte (in C: Felder ≅ Zeiger)
- Allgemeine Form:

```
typ name[groesse1][groesse2]...[groesseN] =
    { ... }
```

  - Alle Felder haben **feste Größe**.

Zeichenketten

- Zeichenketten sind in C (und C0) Felder von **char**, die mit einer Null abgeschlossen werden.
- Beispiel:

```
char hallo[6] = {'h', 'a', 'l', 'l', 'o', '\0' }
```

- Nützlicher syntaktischer Zucker:

```
char hallo[] = "hallo";
```

- Auswertung: hallo [4] liefert o

Strukturen

- Strukturen haben einen *structure tag* (optional) und Felder:

```
struct Vorlesung {
  char dozenten[2][30];
  char titel[30];
  int cp;
} ksgm;

struct Vorlesung pi3;
```

- Zugriff auf Felder über Selektoren:

```
int i = 0;
char name1[] = "Serge Autexier";
while (i < strlen(name1)) {
  ksgm.dozenten[0][i] = name1[i];
  i = i + 1;
}
```

- Rekursive Strukturen nur über Zeiger erlaubt (kommt noch)

C0: Erweiterte Ausdrücke

- **Lexp** beschreibt L-Ausdrücke (l-values), abstrakte Speicheradressen
- Neuer Basisdatentyp **C** für Zeichen
- Erweiterte Grammatik:

  Lexp $l ::=$ **Idt** $\mid l[a] \mid l.$**Idt**

  Aexp $a ::=$ **Z** $\mid$ **C** $\mid$ **Lexp** $\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$

  Bexp $b ::=$ **1** $\mid$ **0** $\mid a_1 == a_2 \mid a_1 < a_2 \mid !\, b \mid b_1 \,\&\&\, b_2 \mid b_1 \,||\, b_2$

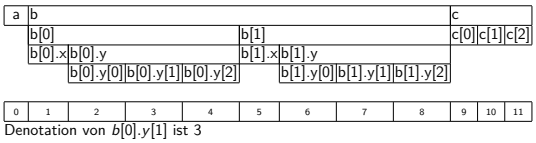
  Exp $e :=$ **Aexp** $\mid$ **Bexp**
- Keine strukturierten **Werte** (Arrays, Strukturen)
- Semantik: strukturiertes Speichermodell
  - **Idt** $\rightharpoonup$ **V** nicht mehr ausreichend

Speichermodelle I: Compiler

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in konkretes **Speicherlayout**:



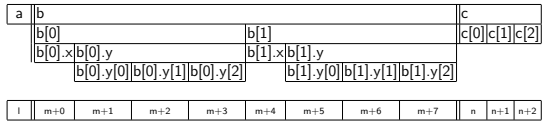
Denotation von $b[0].y[1]$ ist 3

Speichermodelle II: C-Standard

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in abstraktes **Speicherlayout**:



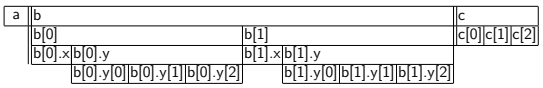
Denotation von $b[0].y[1]$ ist $m+3$, mit m **unbestimmter** Adresse

Speichermodelle III: Symbolisch

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in **symbolische Adressen**:



Denotation von $b[0].y[1]$ ist $b[0].y[1]$

Speichermodelle im Überblick

- Speichermodell I:
  - Ausführbar, aber **spezifisch** für Compiler und **Architektur**
  - Uneingeschränkte **Zeigerarithmetik**: $\mathbf{Loc} = \mathbb{N}$
- Speichermodell II:
  - Nahe dem **C-Standard**
  - **Eingeschränkte** Form der **Zeigerarithmetik** $p + n$ mit p Zeiger, $n \in \mathbb{N}$
- Speichermodell III:
  - **Symbolische** Adressierung

Werte und Zustände

- Systemzustand bildet **strukturierte** Adressen auf Werte ab.

Systemzustände

- **Basisadressen:** $\mathbf{Addr} \stackrel{def}{=} \mathbb{N}$
- **Locations:** $\mathbf{Loc} \stackrel{def}{=} \mathbf{Addr} \times \mathbb{N}$ (Basisadresse mit Offset)
- Werte: $\mathbf{V} = \mathbb{Z}$ (Einbettung von $\mathbf{C}$ in $\mathbb{Z}$)
- Zustände: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

- Wir betrachten nur Zugriffe vom Typ $\mathbf{Z}$ oder $\mathbf{C}$ (**elementare Typen**)
- Nützliche Abstraktion des tatsächlichen C-Speichermodells

Beispiel

Programm

```
struct A {
 int c[2];
 struct B {
  int len;
  char name[20];
 } b;
};

struct A x[] = {
 {{1,2},
  {5, {'n','a','m','e','1','\0'}}
 },
 {{3,4},
  {5, {'n','a','m','e','2','\0'}}
 }
};
```

Zustand

1. Die Basisadresse von x sei m.
2. Dann:

$\langle m, 0\rangle \mapsto 1$	$\langle m, 23\rangle \mapsto 3$
$\langle m, 1\rangle \mapsto 2$	$\langle m, 24\rangle \mapsto 4$
$\langle m, 2\rangle \mapsto 5$	$\langle m, 25\rangle \mapsto 5$
$\langle m, 3\rangle \mapsto$ 'n'	$\langle m, 26\rangle \mapsto$ 'n'
$\langle m, 4\rangle \mapsto$ 'a'	$\langle m, 27\rangle \mapsto$ 'a'
$\langle m, 5\rangle \mapsto$ 'm'	$\langle m, 28\rangle \mapsto$ 'm'
$\langle m, 6\rangle \mapsto$ 'e'	$\langle m, 29\rangle \mapsto$ 'e'
$\langle m, 7\rangle \mapsto$ '1'	$\langle m, 30\rangle \mapsto$ '2'
$\langle m, 8\rangle \mapsto$ '\0'	$\langle m, 31\rangle \mapsto$ '\0'

Umgebung und Typen

- Unsere Sprache kennt folgende **Typen**:

$$\begin{aligned}\mathbf{Type} &::= \mathbf{char} \mid \mathbf{int} \mid \mathbf{Struct} \mid \mathbf{Array}\\ \mathbf{Struct} &::= \mathbf{struct}\ \mathbf{Idt}^?\ \{(\mathbf{Type}\ \mathbf{Idt})^+\}\\ \mathbf{Array} &::= \mathbf{Type}\ \mathbf{Idt}[\mathbf{Aexp}]\end{aligned}$$

- Die Umgebung Γ ordnet **Bezeichnern** $x \in \mathbf{Idt}$ einen **Typ** und eine **Basisadresse** zu.
- $\Gamma \vdash e : t$ Ausdruck e hat Typ t im Kontext Γ
- Semantik benötigt Γ als **zusätzlichen** (statischen!) Parameter

Größen und Offsets

- **Größe** eines Typen (vgl. **sizeof** in C, aber vereinfacht):

$$\begin{aligned}sizeof(\mathbf{char}) &\stackrel{def}{=} 1\\ sizeof(\mathbf{int}) &\stackrel{def}{=} 1\\ sizeof(\mathbf{struct}\ s\ \{t_1 f_1; \ldots t_n f_n\}) &\stackrel{def}{=} \sum_{i=1}^{n} sizeof(t_i)\\ sizeof(t\ i[n]) &\stackrel{def}{=} sizeof(t) \cdot n\end{aligned}$$

- Offset $off_t(a)$ eines Feldes $a \in \mathbf{Idt}$ für einen Strukturtyp t:

$$off_{\mathbf{struct}\ i\ \{t_1\ f_1;\ldots;t_n\ f_n\}}(f_k) \stackrel{def}{=} \sum_{i=1}^{k-1} sizeof(t_i) \text{für } 1 \le k \le n$$

Operationale Semantik: L-Ausdrücke

- $m \in$ **Lexp** wertet zu $l \in$ **Loc** aus: $\langle m, \sigma\rangle \to^{\Gamma}_{Lexp} l$

$$\frac{x \in \mathbf{Idt}, x \in \Gamma}{\langle x, \sigma\rangle \to^{\Gamma}_{Lexp} \langle \Gamma(x), 0\rangle}$$

$$\frac{\Gamma \vdash m : t[\,] \qquad \langle m, \sigma\rangle \to^{\Gamma}_{Lexp} l \qquad \langle e, \sigma\rangle \to^{\Gamma}_{Aexp} n}{\langle m[e], \sigma\rangle \to^{\Gamma}_{Lexp} l + n \cdot sizeof(t)}$$

$$\frac{\Gamma \vdash m : t \qquad \langle \sigma, m\rangle \to^{\Gamma}_{Lexp} l}{\langle m.a, \sigma\rangle \to^{\Gamma}_{Lexp} l + off_t(a)}$$

- Notation: für $a = \langle b, m\rangle \in$ **Loc**, $n \in \mathbb{N}$, $a + n \stackrel{def}{=} \langle b, m + n\rangle$

Operationale Semantik: Ausdrücke und Zuweisungen

- Ein L-Wert als Ausdruck wird ausgewertet, indem er ausgelesen wird:

$$\frac{\langle m, \sigma\rangle \to^{\Gamma}_{Lexp} l \qquad m \in Dom(\sigma)}{\langle m, \sigma\rangle \to^{\Gamma}_{Aexp} \sigma(l)}$$

- Zuweisung:

$$\frac{\langle m, \sigma\rangle \to^{\Gamma}_{Lexp} l \qquad \langle e, \sigma\rangle \to^{\Gamma}_{Aexp} v}{\langle m = e, \sigma\rangle \to^{\Gamma}_{Stmt} \sigma[l \mapsto v]}$$

- Die restlichen Regeln bleiben (mit Γ garnieren)

Denotationale Semantik

- Denotation für **Lexp**:

$$[\![-]\!]_{\mathcal{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$[\![x]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, \langle \Gamma(x), 0\rangle) \mid x \in \Gamma\}$$
$$[\![m[e]]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + n \cdot sizeof(t)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, (\sigma, n) \in [\![e]\!]^{\Gamma}_{\mathcal{A}}, \Gamma \vdash m : t[x]\}$$
$$[\![m.a]\!]^{\Gamma}_{\mathcal{L}} = \{(\sigma, l + off_t(a)) \mid \Gamma \vdash m : t, (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}\}$$

- Denotation für **Ausdrücke** ($m \in$ **Lexp**):

$$[\![m]\!]^{\Gamma}_{\mathcal{A}} = \{(\sigma, \sigma(l)) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, l \in Dom(\sigma)\}$$

- Denotation für **Zuweisungen**:

$$[\![m = e]\!]^{\Gamma}_{\mathcal{C}} = \{(\sigma, \sigma[l \mapsto v]) \mid (\sigma, l) \in [\![m]\!]^{\Gamma}_{\mathcal{L}}, (\sigma, v) \in [\![e]\!]^{\Gamma}_{\mathcal{A}}\}$$

Arbeitsblatt 8.1: Semantik

Gegeben folgende Deklaration:

```
struct p {
  int a[5];
  struct q {
    int a;
    int b;
    } b[5];
  int c[2];
  } x;
```

Für $\Gamma \stackrel{def}{=} \langle x \mapsto l\rangle, \sigma = \emptyset$ berechne die Semantik von folgendem Programmfragment:

```
x.a[2]= 7;
x.c[1]= 9;
x.b[3].b= 17;
```

Berechne dazu die Größen von p und q und die Offsets von a, b, c.

Links oder Rechts?

- In beiden Semantiken hat ein **Lexp** unterschiedliche Semantik auf der **linken** oder **rechten** Seite einer Zuweisung.
  - x = x+1 — Links: Lokation, rechts: Wert (im Zustand an dieser Stelle)
- Lösung in C: "Except when it is (. . .) the operand of the unary & operator, the left operand of the . operator or an assigment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)" *C99 Standard*, §6.3.2.1 (2)
- Nicht spezifisch für C

Floyd-Hoare-Kalkül

- Die Regeln des Floyd-Hoare-Kalküls berechnen geltende Zusicherungen
- Nötige Änderung: Substitution in Zusicherungen wird zur Ersetzung von **Lexp**-Ausdrücken

$$\overline{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

  - Jetzt werden **Lexp** ersetzt, keine **Idt**

$$\overline{\vdash \{P[e/l]\}\, l = e\, \{P\}}$$

  Anmerkung: l und e enthalten **keine** logischen Variablen.
- Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
- Problem: Feldzugriffe

Von der Substitution zur Ersetzung

Assn $b ::= true \mid false \mid a_1 = a_2 \mid a_1 \le a_2 \mid p(e_1, \ldots, e_n) \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid b_1 \longrightarrow b_2 \mid \forall v.\, b \mid \exists v.\, b$ **(Literale)**

$true[e/l] \stackrel{def}{=} true$

$false[e/l] \stackrel{def}{=} false$

$(a_1 = a_2)[e/l] \stackrel{def}{=} (a_1[e/l] = a_2[e/l])$

$(b_1 \wedge b_2)[e/l] \stackrel{def}{=} (b_1[t/x] \wedge b_2[e/l])$

$(\forall v.\, b)[e/l] \stackrel{def}{=} \forall v.\, (b[e/l])$

$n[e/l] \stackrel{def}{=} n$ $\quad (n \in \mathbf{Z} \uplus \mathbf{C})$

$m[e/l] \stackrel{def}{=} \begin{cases} e & \text{falls } l \cong m \\ m\,\mathrm{sub}(m, l, e) & \text{sonst} \end{cases}$ $\quad (m \in \mathbf{Lexp})$

$(a_1 + a_2)[e/l] \stackrel{def}{=} a_1[e/l] + a_2[e/l]$

$\ldots$

$\mathrm{sub}(x, m, e) \stackrel{def}{=} x$

$\mathrm{sub}(l[i], m, e) \stackrel{def}{=} l[i[e/m]]$

$\mathrm{sub}(l.a, m, e) \stackrel{def}{=} (\mathrm{sub}(l, m, e)).a$

Beispiel Problemsituationen:
$(c[i].x[0])[5/c[1].x[0]] =?$
$(c[1].x[0])[8/c[1].x[j]] =?$
$(c[i].x[0])[8/c[1].x[j]] =?$

Gleichheit von L-Ausdrücken

- Das Substitutionslemma muss gelten:

$$[\![P[e/m]]\!]^{\Gamma}_{\mathcal{B}v}(\sigma) = [\![P]\!]^{\Gamma}_{\mathcal{B}v}(\sigma[[\![m]\!]^{\Gamma}_{\mathcal{L}} \mapsto [\![e]\!]^{\Gamma}_{\mathcal{A}v}(\sigma)])$$
$$[\![a[e/m]]\!]^{\Gamma}_{\mathcal{A}v}(\sigma) = [\![a]\!]^{\Gamma}_{\mathcal{A}v}(\sigma[[\![m]\!]^{\Gamma}_{\mathcal{L}} \mapsto [\![e]\!]^{\Gamma}_{\mathcal{A}v}(\sigma)])$$

- $l \cong m$ gdw. $[\![l]\!]^{\Gamma}_{\mathcal{L}} = [\![m]\!]^{\Gamma}_{\mathcal{L}}$

$$\frac{x \in \mathbf{Idt}}{x \cong x} \qquad \frac{l \cong m}{l.a \cong m.a} \qquad \frac{l \cong m, [\![i]\!]^{\Gamma}_{\mathcal{A}} = [\![j]\!]^{\Gamma}_{\mathcal{A}}}{l[i] \cong m[j]}$$

- Gleichheit von L-Ausdrücken reduziert zu Gleichheit von Feldindizes

Ersetzung leicht gemacht

Wie ersetzen wir in $P[e/l]$?

- Wenn l ein Identifier $x \in$ **Idt** ist, wie gewohnt substituieren
- Wenn $l = a[s]$ und P in der Form $a[t]$ in $L(a[t])$ (wobei L keine Quantoren enthält)
  1. Wenn s und t **beide** in $\mathbb{Z}$ oder **Idt**, ersetze $L(a[t])$ durch $L(e)$, falls $s = t$.
  2. Wenn s oder t nicht aus $\mathbb{Z}$ oder **Idt**, ersetze $L(a[t])$ durch $(t = s \wedge L(e)) \vee (t \neq s \wedge L(a[t]))$

  (2) ist immer möglich, (1) eine Optimierung
- Ansonsten $\forall i.\, P[e/i] = \forall i.\, (P[e/l])$ etc.
- Das ist jetzt immer noch nicht die ganz allgemeine Form (was ist mit geschachtelten Indizes $a[i][j]$?), aber für unsere Belange reicht das.

Arbeitsblatt 8.2: Ersetzungen

Berechnet folgende Substitutionen (mit $P \stackrel{def}{=} \forall i.\, 0 \leq i \longrightarrow b.x[i].y < 15$):

1. $P[5/a.x[j].y] =?$
2. $P[3/x[7].y] =?$
3. $P[9/b.x[3].y] =?$
4. $(c[7 + b.x[i].i].y \leq i)[99/i] =?$
5. $(\forall i.\, 0 \leq i \longrightarrow c[7 + b.x[i].y].a = i)[17/b.x[j].y] =?$

Beispiel

```
int a[3];
// {true}
// {3 = 3}
a[2] = 3;
// {a[2] = 3}
// {4 · a[2] = 12}
a[1] = 4;
// {a[1] · a[2] = 12}
// {5 · a[1] · a[2] = 60}
a[0] = 5;
// {a[0] · a[1] · a[2] = 60}
```

$$\overline{\vdash \{P[e/l]\}\ l = e\ \{P\}}$$

Beispiel: Problem

```
int a[3];
int i;
// {0 ≤ i < 2}
// ↯
// {i ≠ 1}
a[0] = 3;
// {i ≠ 1}
// {i ≠ 1 ∧ 7 = 7)}
a[1] = 7;
// {i ≠ 1 ∧ a[1] = 7}
a[2] = 9;
// {i ≠ 1 ∧ a[1] = 7}
// {(i = 1 ∧ −1 = 7) ∨ (i ≠ 1 ∧ a[1] = 7)}{a[1] = 7}
a[i] = −1;
// {a[1] = 7}
```

$$\overline{\vdash \{P[e/l]\}\ l = e\ \{P\}}$$

Arbeitsblatt 8.3: Jetzt seid ihr dran

Annotiert die beiden folgenden Programme:

```
int a[2];
int b[2];
// {0 ≤ n ∧ 0 ≤ m ∧ n ≤ m}
a[0] = m;
//
b[0] = a[0] − n;
//
b[1] = a[0] + n
//
a[1] = b[0] ∗ b[1];
// {a[1] = m² − n²}
```

```
int a[3];
int i;
// {0 ≤ n}
i = 2;
//
a[i] = 3;
//
a[0] = n;
//
a[2] = a[2] ∗ a[0];
// {a[2] = 3 ∗ n}
```


Erstes Beispiel: Ein Feld initialisieren

```
1  // {0 ≤ n}
2  // {∀j.0 ≤ j < 0 ⟶ a[j] = j ∧ 0 ≤ n}
3  i= 0;
4  // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
5  while (i< n) {
6    // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
7    // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i + 1 ≤ n}
8    // {∀j.0 ≤ j < i ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
9    //      ∧((i = i ∧ i = i) ∨ (i ≠ i ∧ a[i] = i)) ∧ i + 1 ≤ n}
10   // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
11   //      ∧ i + 1 ≤ n}
12   a[i]= i;
13   // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
14   i= i+1;
15   // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
16   }
17 // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
18 // {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

$$(\forall j.\, 0 \leq j < n \longrightarrow P[j]) \wedge P[n] \Longrightarrow \forall j.\, 0 \leq j < n+1 \longrightarrow P[j]$$

Beispiel: Suche nach dem maximalen Element

```
1  // {0 < n}
2  // {∀j. 0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 ∧ 0 ≤ 0 < n}
3  i= 0;
4  // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ∧ 0 ≤ 0 < n}
5  r= 0;
6  // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
7  while (i< n) {
8    // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
9    // {∀j.0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
10   if (a[r] < a[i]) {
11     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
12     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ a[i] ≤ a[r] ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
13     // {∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
14     r= i;
15     // {∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
16     }
17   else {
18     // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
19     // {∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
20     }
21   // {∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
22   i= i+1;
23   // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
24 }
25 // {∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
26 // {∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```


Vorgehensweise

```
1  // {I}
2  while (b) {
3    // {I ∧ b}
4    c
5    // {I}
6  }
7  // {I ∧ ¬b}
8  // {Φ}
```

1. Finde/rate/formuliere Invariante I
2. Beweise $(I \wedge \neg b) \longrightarrow \Phi$
3. Zeige mittels Floyd-Hoare-Regeln, dass Invariante durch Schleifenrumpf c erhalten bleibt
4. Setze Beweis mit Floyd-Hoare Regeln vor der Schleife fort

Längeres Beispiel: Suche nach einem Null-Element

```
1  i= 0;
2  r= -1;
3  while (i< n) {
4    if (a[i] == 0) {
5      r= i;
6    }
7    else {
8    }
9    i= i+1;
10 }
11 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```

Merkt euch folgende korrekten logischen Umformungen:

- $(F \wedge H) \vee (G \wedge H)$ ist äquivalent zu $(F \vee G) \wedge H$
- $\neg F \vee G$ ist äquivalent zu $F \longrightarrow G$

Längeres Beispiel: Suche nach einem Null-Element

```
1  // {0 ≤ n}
2  // {(−1 ≠ −1 ⟶ 0 ≤ −1 < 0 ∧ a[−1] = 0) ∧ 0 ≤ 0 ≤ n}
3  i= 0;
4  // {(−1 ≠ −1 ⟶ 0 ≤ −1 < i ∧ a[−1] = 0) ∧ 0 ≤ i ≤ n}
5  r= −1;
6  // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
7  while (i< n) {
8    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ i < n}
9    // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
10   if (a[i] == 0) {
11     // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
                                          B(i)∧C
12     // {0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
           ¬A(i)        C          B(i)        C
13     // {(i = −1 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0) ∨ (0 ≤ i < i + 1 ∧ a[i] = 0 ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0)}
14     // {(i = −1 ∨ (0 ≤ i < i + 1 ∧ a[i] = 0)) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
15     // {(i ≠ −1 ⟶ 0 ≤ i < i + 1 ∧ a[i] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] = 0}
           A(i)        B(i)        C
16     r= i;
17     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
18     }
19   else {
20     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n ∧ a[i] ≠ 0}
21     // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
22   }
23   // {(r ≠ −1 ⟶ 0 ≤ r < i + 1 ∧ a[r] = 0) ∧ 0 ≤ i + 1 ≤ n}
24   i= i+1;
25   // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
26 }
27 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n)}
28 // {(r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] = 0) ∧ i = n}
29 // {r ≠ −1 ⟶ 0 ≤ r < n ∧ a[r] = 0}
```


Benutzte Logische Umformungen

- Zeilen 11-12:
  - $[D \wedge C] \Rightarrow [C]$ und
  - Erweiterung von C auf $B(i) \wedge C$, weil $C \vdash B(i)$ gilt.
- $[\varphi] \Rightarrow [\psi \vee \varphi]$ in der Form

$$[(B(i) \wedge C)] \Rightarrow [(\neg A(i) \wedge C) \vee (B(i) \wedge C))]$$

- DeMorgan:

$$[(\neg A(i) \wedge C) \vee (B(i) \wedge C))] \Rightarrow [(\neg A(i) \vee B(i)) \wedge C]$$

- Klassische Implikation:

$$[\neg U \vee V] \Leftrightarrow [U \Rightarrow V]$$

Längeres Beispiel: Suche nach einem Null-Element

```
10 /** { 0 ≤ n } */
11 /** { 0 ≤ 0 ≤ n } */
12 i= 0;
13 /** { 0 ≤ i ≤ n } */
14 /** { (−1 ≠ −1 ⟶ 0 ≤ −1< i ∧ a[−1] == 0) ∧ 0 ≤ i ≤ n } */
15 r= −1;
16 /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
17 while (i< n) {
18   /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i < n } */
19   /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
20   if (a[i] == 0) {
21     /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
22     /** { 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
23     /** { (i ≠ −1 ⟶ 0 ≤ i< i+1 ∧ a[i] == 0) ∧ 0≤ i+1≤ n } */
24     r= i;
25     /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0≤ i+1≤ n } */
26     }
27   else {
28     /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] ≠ 0 } */
29     /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
30     }
31   /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
32   i= i+1;
33   /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
34   }
35 /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n) } */
36 /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n } */
37 /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ i == n } */
38 /** { r ≠ −1 ⟶ 0 ≤ r< n ∧ a[r] == 0 } */
```


Zusammenfassung

- Strukturierte Datentypen (Felder und Structs) erfordern strukturierte Adressen
- Abstraktion über „echtem“ Speichermodell
- Änderungen in der Semantik und im Floyd-Hoare-Kalkül überschaubar
- . . . aber mit erheblichen Konsequenzen:
  - Substitution wird zur Ersetzung
  - Anwendung der Zuweisungsregel führt i.A. zu großen Formeln

Fahrplan