Korrekte Software: Grundlagen und Methoden

Vorlesung 8 vom 27.5.21 / 1.6.2021 / 3.6.2021

Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

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

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-Werte (l-values), abstrakte Speicheradressen
- Neuer Basisdatentyp **C** für Zeichen
- Erweiterte Grammatik:

Lexp $l ::= \mathbf{Idt} \mid l[a] \mid l.\mathbf{Idt}$

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

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

Exp $e := \mathbf{Aexp} \mid \mathbf{Bexp}$

Werte und Zustände

- Zustände bilden **strukturierte** Adressen auf Werte (wie vorher) ab.

Systemzustände

- **Locations:** $\mathbf{Loc} ::= \mathbf{Idt} \mid \mathbf{Loc}[\mathbb{Z}] \mid \mathbf{Loc}.\mathbf{Idt}$
- Werte: $\mathbf{V} = \mathbb{Z} \uplus \mathbf{C}$
- Zustände: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$

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

Beispiel

Programm

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

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

Zustand

$$
\begin{array}{ll}
x[0].c[0] \mapsto 1 & x[1].c[0] \mapsto 3 \\
x[0].c[1] \mapsto 2 & x[1].c[1] \mapsto 4 \\
x[0].b.name[0] \mapsto \text{‘}n' & x[1].b.name[0] \mapsto \text{‘}n' \\
x[0].b.name[1] \mapsto \text{‘}a' & x[1].b.name[1] \mapsto \text{‘}a' \\
x[0].b.name[2] \mapsto \text{‘}m' & x[1].b.name[2] \mapsto \text{‘}m' \\
x[0].b.name[3] \mapsto \text{‘}e' & x[1].b.name[3] \mapsto \text{‘}e' \\
x[0].b.name[4] \mapsto \text{‘}1' & x[1].b.name[4] \mapsto \text{‘}2' \\
x[0].b.name[5] \mapsto \text{‘}\backslash 0' & x[1].b.name[5] \mapsto \text{‘}\backslash 0'
\end{array}
$$

Operationale Semantik: L-Werte

- **Lexp** m wertet zu **Loc** l aus: $\langle m, \sigma \rangle \rightarrow_{Lexp} l \mid \bot$

$$\frac{x \in \textbf{Idt}}{\langle x, \sigma \rangle \rightarrow_{Lexp} x}$$

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} l \neq \bot \qquad \langle a, \sigma \rangle \rightarrow_{Aexp} i \neq \bot}{\langle m[a], \sigma \rangle \rightarrow_{Lexp} l[i]}$$

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} l \qquad \langle a, \sigma \rangle \rightarrow_{Aexp} i \qquad i = \bot \text{ oder } l = \bot}{\langle m[a], \sigma \rangle \rightarrow_{Lexp} \bot}$$

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} l \neq \bot}{\langle m.i, \sigma \rangle \rightarrow_{Lexp} l.i}$$

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} \bot}{\langle m.i, \sigma \rangle \rightarrow_{Lexp} \bot}$$

Operationale Semantik: Ausdrücke

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

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} l \qquad l \in Dom(\sigma)}{\langle m, \sigma \rangle \rightarrow_{Aexp} \sigma(l)}$$

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} l \qquad l \notin Dom(\sigma)}{\langle m, \sigma \rangle \rightarrow_{Aexp} \bot} \qquad \frac{\langle m, \sigma \rangle \rightarrow_{Lexp} \bot}{\langle m, \sigma \rangle \rightarrow_{Aexp} \bot}$$

- Auswertung für **C**:

$$\frac{}{\langle c :: \mathbf{C}, \sigma \rangle \rightarrow_{Aexp} \mathrm{Ord}(c)}$$

wobei $\mathrm{Ord} : \mathbf{C} \rightarrow \mathbf{Z}$ eine bijektive Funktion ist, die jedem Character eine Ordinalzahl zuweist (zum Beispiel ASCII Wert).

Operationale Semantik: Zuweisungen

- Zuweisungen sind nur definiert für elementare Typen:

$$\frac{\langle m :: \tau, \sigma \rangle \rightarrow_{Lexp} l \qquad \langle e :: \tau, \sigma \rangle \rightarrow v \qquad \tau \text{ elementarer Typ}}{\langle m = e, \sigma \rangle \rightarrow_{Stmt} \sigma[l \mapsto v]}$$

In allen anderen Fällen ($\bot$, keine/unterschiedliche elementare Typen)

$$\langle m = e, \sigma \rangle \rightarrow_{Stmt} \bot$$

- Die restlichen Regeln bleiben

Denotationale Semantik

- Denotation für **Lexp**:

$$\llbracket - \rrbracket_{\mathcal{L}} : \textbf{Lexp} \rightarrow (\Sigma \rightharpoonup \textbf{Loc})$$

$$\llbracket x \rrbracket_{\mathcal{L}} = \{(\sigma, x) \mid \sigma \in \Sigma\}$$
$$\llbracket m[a] \rrbracket_{\mathcal{L}} = \{(\sigma, l[i]) \mid (\sigma, l) \in \llbracket m \rrbracket_{\mathcal{L}}, (\sigma, i) \in \llbracket a \rrbracket_{\mathcal{A}}\}$$
$$\llbracket m.i \rrbracket_{\mathcal{L}} = \{(\sigma, l.i) \mid (\sigma, l) \in \llbracket m \rrbracket_{\mathcal{L}}\}$$

- Denotation für **Characters** $c \in \mathbf{C}$:

$$\llbracket c \rrbracket_{\mathcal{A}} = \{(\sigma, \mathrm{Ord}(c)) | \sigma \in \Sigma\}$$

- Denotation für **Zuweisungen**:

$$\llbracket m = e \rrbracket_{\mathcal{C}} = \{(\sigma, \sigma[l \mapsto v]) \mid (\sigma, l) \in \llbracket m \rrbracket_{\mathcal{L}}, (\sigma, v) \in \llbracket e \rrbracket_{\mathcal{A}}\}$$

Floyd-Hoare-Kalkül

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

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

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

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

  Anmerkung: l und e enthalten **keine** logischen Variablen.

  - Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
  - Problem: Feldzugriffe

Von der Substitution zur Ersetzung

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

$$
\begin{aligned}
true[e/l] &:= true \\
false[e/l] &:= false \\
(a_1 = a_2)[e/l] &:= (a_1[e/l] = a_2[e/l]) \\
(b_1 \wedge b_2)[e/l] &:= (b_1[t/x] \wedge b_2[e/l]) \\
(\forall v.\, b)[e/l] &:= \forall v.\, (b[e/l])
\end{aligned}
$$

$$n[e/l] := n \qquad (n \in \mathbf{Z} \uplus \mathbf{C})$$

$$l'[e/l] := l'[e/l] \begin{cases} e & \text{falls } l = l' \\ l' & \text{sonst} \end{cases} \qquad (l' \in \textbf{Lexp})$$

$$(a_1 + a_2)[e/l] := a_1[e/l] + a_2[e/l]$$

$$\ldots$$

Beispiel Problemsituationen:

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

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}
```

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

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 ∧ 7 = −1) ∨ (i ≠ 1 ∧ a[1] = 7}{a[1] = 7}
a[i] = −1;
// {a[1] = 7}
```

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

Arbeitsblatt 8.1: 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

```
// {0 ≤ n}
// {∀j.0 ≤ j < 0 ⟶ a[j] = j ∧ 0 ≤ n}
i= 0;
// {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n}
while (i< n) {
  // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i ≤ n ∧ i < n}
  // {∀j.0 ≤ j < i ⟶ a[j] = j ∧ i + 1 ≤ n}
  // {∀j.0 ≤ j < i ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
  //     ∧((i = i ∧ i = i) ∨ (i ≠ i ∧ a[i] = i)) ∧ i + 1 ≤ n}
  // {∀j.0 ≤ j < i + 1 ⟶ ((i = j ∧ i = j) ∨ (j ≠ i ∧ a[j] = j))
  //     ∧i + 1 ≤ n}
  a[i]= i;
  // {(∀j.0 ≤ j < i + 1 ⟶ a[j] = j) ∧ i + 1 ≤ n}
  i= i+1;
  // {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n}
  }
// {(∀j.0 ≤ j < i ⟶ a[j] = j) ∧ i ≤ n ∧ i ≥ n}
// {∀j.0 ≤ j < n ⟶ a[j] = j}
```

- Wichtiges Theorem:

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

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
// {(∀j. 0 ≤ j < 0 ⟶ a[j] ≤ a[0]) ∧ 0 ≤ 0 ∧ 0 ≤ 0 < n}
i= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[0]) ∧ 0 ≤ i ∧ 0 ≤ 0 < n}
r= 0;
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ∧ 0 ≤ r < n}
while (i< n) {
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i < n ∧ 0 ≤ r < n}
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  if (a[r] < a[i]) {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ a[r] < a[i]}
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[i]) ∧ a[i] ≤ a[i] ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[i]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ i < n}
    r= i;
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  else {
    // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n ∧ ¬(a[r] < a[i])}
    // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
    }
  // {(∀j. 0 ≤ j < i + 1 ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i + 1 ≤ n ∧ 0 ≤ r < n}
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
}
// {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n ∧ n ≤ i}
// {(∀j. 0 ≤ j < n ⟶ a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```


Vorgehensweise

```
// {I}
while (b) {
  // {I ∧ b}
  c
  // {I}
}
// {I ∧ ¬b}
// {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

```
i= 0;
r= −1;
while (i< n) {
  if (a[i] == 0) {
    r= i;
  }
  else {
  }
  i= i+1;
}
// {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}
```

12 $// \;\{\overbrace{0 \le i < i+1 \wedge a[i] = 0 \wedge 0 \le i+1 \le n \wedge a[i] = 0}^{B(i) \wedge C}\}$

13 $// \;\{(\overbrace{i = -1}^{\neg A(i)} \wedge \overbrace{0 \le i+1 \le n \wedge a[i] = 0)}^{C} \vee (\overbrace{0 \le i < i+1 \wedge a[i] = 0}^{B(i)} \wedge \overbrace{0 \le i+1 \le n \wedge a[i] = 0)}^{C}\}$

14 $// \;\{(i = -1 \vee (0 \le i < i+1 \wedge a[i] = 0)) \wedge 0 \le i+1 \le n \wedge a[i] = 0\}$

15 $// \;\{(\underbrace{i \ne -1}_{A(i)} \longrightarrow \underbrace{0 \le i < i+1 \wedge a[i] = 0)}_{B(i)} \wedge \underbrace{0 \le i+1 \le n \wedge a[i] = 0}_{C}\}$

```
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

```
/** { 0 ≤ n } */
/** { 0 ≤ 0 ≤ n } */
i= 0;
/** { 0 ≤ i ≤ n } */
/** { (−1 ≠ −1 ⟶ 0 ≤ −1< i ∧ a[−1] == 0) ∧ 0 ≤ i ≤ n } */
r= −1;
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
while (i< n) {
  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i < n } */
  /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
  if (a[i] == 0) {
    /** { (r ≠ −1 ⟶ 0 ≤ r < i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
    /** { 0 ≤ i+1 ≤ n ∧ a[i] == 0 } */
    /** { (i ≠ −1 ⟶ 0 ≤ i< i+1 ∧ a[i] == 0) ∧ 0≤ i+1≤ n } */
    r= i;
    /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0≤ i+1≤ n } */
    }
  else {
     /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n ∧ a[i] ≠ 0 } */
     /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
    }
  /** { (r ≠ −1 ⟶ 0 ≤ r< i+1 ∧ a[r] == 0) ∧ 0 ≤ i+1 ≤ n } */
  i= i+1;
  /** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n } */
  }
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ ¬(i < n) } */
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ 0 ≤ i ≤ n ∧ i ≥ n } */
/** { (r ≠ −1 ⟶ 0 ≤ r< i ∧ a[r] == 0) ∧ i == n } */
/** { r ≠ −1 ⟶ 0 ≤ r< n ∧ a[r] == 0 } */
```


Allgemeine Regel bei Ersetzungen?

Wie sieht nun die allgemeine Regel aus für

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

```
int a[3];
int i;
a[0] = 3;
a[1] = 7;
a[2] = 9;
a[a[2]−a[1]] = −1;
// {a[2] = −1}
```

```
int a[3];
int i;
i = 8;
a[0] = 3;
a[1] = i;
a[2] = 9;
a[a[2]−a[1]] = −1;
// {a[1] = −1}
```


Allgemeine Regel bei Ersetzungen (Nur Arrays)

Wie sieht nun die allgemeine Regel aus für

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

1. Wenn l Programmvariable ist, wie gewohnt substituieren
2. Wenn $l = a[s]$:
   - 2.1 Vorkommen der Form $a[t]$ in **Literalen** $L(a[t])$ und s und t beide in $\mathbb{Z}$ oder **Idt**,
     - dann ersetze $L(a[t])$ durch $L(e)$, falls $s = t$
   - 2.2 Vorkommen der Form $a[t]$ in **Literalen** $L(a[t])$ und s oder t sind nicht aus $\mathbb{Z}$,
     - dann ersetze $L(a[t])$ durch $(t = s \wedge L(e)) \vee (t \neq s \wedge L(a[t]))$

2.2 könnt ihr immer machen, 2.1 ist eine Optimierung

- Das ist jetzt immer noch nicht die ganz allgemeine Form, aber für unsere Belange reicht das.

Arbeitsblatt 8.2: Längeres Beispiel: Suche nach dem ersten Null-Element

Ausgehend von dem vorherigem Beispiel, annotiert folgendes

```
// {0 ≤ n}
i= 0;
r= −1;
/* — beforeloop — */
while (i< n) {
  /* — startloop — */
  if (r== −1 && a[i] == 0) {
    r= i;
    }
  else {
  }
  /* — afterif — */
  i= i+1;
  /* — endloop — */
}
/* — afterloop — */
/** {(r ≠ −1 ⟶ (0 ≤ r< n ∧ a[r] == 0 ∧ (∀ int j . 0 ≤ j < r ⟶ a[j] ≠ 0)))
    ∧ (r==−1 ⟶ (∀ int j . 0 ≤ j < n ⟶ a[j] ≠ 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

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