

Korrekte Software: Grundlagen und Methoden
Vorlesung 7 vom 22.05.16: Strukturierte Datentypen: Strukturen
und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2018

Fahrplan

- ▶ Einführung
- ▶ Operationale Semantik
- ▶ Denotationale Semantik
- ▶ Äquivalenz der Operationalen und Denotationalen Semantik
- ▶ Die Floyd-Hoare-Logik
- ▶ Invarianten und die Korrektheit des Floyd-Hoare-Kalküls
- ▶ **Strukturierte Datentypen**
- ▶ Modellierung und Spezifikation
- ▶ Verifikationsbedingungen
- ▶ Vorwärts mit Floyd und Hoare
- ▶ Funktionen und Prozeduren
- ▶ Referenzen
- ▶ 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 $\cong$ 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[5] = { '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 $/ ::= \text{Idt} \mid /[a] \mid /. \text{Idt}$

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

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 $e ::= \text{Aexp} \mid \text{Bexp}$

Werte und Zustände

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

Systemzustände

- ▶ **Locations:** $\text{Loc} ::= \text{Idt} \mid \text{Loc}[\mathbb{Z}] \mid \text{Loc}.\text{Idt}$
 - ▶ Werte: $\mathbf{V} = \mathbb{Z}$
 - ▶ Zustände: $\Sigma \stackrel{\text{def}}{=} \mathbf{Loc} \rightarrow \mathbf{V}$
-
- ▶ Wir betrachten nur Zugriffe vom Typ **Z** oder **C** (**elementare Typen**)
 - ▶ Nützliche Abstraktion des tatsächliche 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

$x[0].c[0] \mapsto 1$	$x[1].c[0] \mapsto 3$
$x[0].c[1] \mapsto 2$	$x[1].c[1] \mapsto 4$
$x[0].b.name[0] \mapsto 'n'$	$x[1].b.name[0] \mapsto 'n'$
$x[0].b.name[1] \mapsto 'a'$	$x[1].b.name[1] \mapsto 'a'$
$x[0].b.name[2] \mapsto 'm'$	$x[1].b.name[2] \mapsto 'm'$
$x[0].b.name[3] \mapsto 'e'$	$x[1].b.name[3] \mapsto 'e'$
$x[0].b.name[4] \mapsto '1'$	$x[1].b.name[4] \mapsto '2'$
$x[0].b.name[5] \mapsto '\0'$	$x[1].b.name[5] \mapsto '\0'$

Operationale Semantik: L-Werte

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

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

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

$$\frac{\langle m, \sigma \rangle \rightarrow_{Lexp} l \quad \langle a, \sigma \rangle \rightarrow_{Aexp} \perp}{\langle m[a], \sigma \rangle \rightarrow_{Lexp} \perp}$$

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

Operationale Semantik: Ausdrücke und Zuweisungen

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

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

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

- Zuweisungen sind nur definiert für elementare Typen:

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

- Die restlichen Regeln bleiben

Denotationale Semantik

- Denotation für **Lexp**:

$$\mathcal{L}[\![x]\!] = \{(\sigma, x) \mid \sigma \in \Sigma\}$$

$$\mathcal{L}[\![m[a]]\!] = \{(\sigma, l[i]) \mid (\sigma, l) \in \mathcal{L}[\![m]\!], (\sigma, i) \in \mathcal{A}[\![a]\!]\}$$

$$\mathcal{L}[\![m.i]\!] = \{(\sigma, m.i) \mid (\sigma, l) \in \mathcal{L}[\![m]\!]\}$$

- Denotation für **Zuweisungen**:

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

Floyd-Hoare-Kalkül

- ▶ Die Regeln des Floyd-Hoare-Kalküls berechnen geltende Zusicherungen
- ▶ Nötige Änderung: Substitution in Zusicherungen
 - ▶ Jetzt werden **Lexp** ersetzt, keine **ldt**
 - ▶ Gleichheit und Ungleichheit von **Lexp** nicht immer entscheidbar
 - ▶ Problem: Feldzugriffe

Beispiel

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

Beispiel: Problem

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

Erstes Beispiel: Ein Feld initialisieren

```
1 // {n ≤ 0}
2 i = 0;
3 while (i < n) {
4     a[i] = i;
5     i = i + 1;
6     // {(∀j. 0 ≤ j < i → a[j] = j) ∧ i ≤ n}
7 }
8 // {(∀j. 0 ≤ j < n → a[j] = j)}
```


Erstes Beispiel: Ein Feld initialisieren

```
1 // {n ≤ 0}
2 i = 0;
3 while (i < n) {
4     a[i] = i;
5     i = i + 1;
6     // {(∀j. 0 ≤ j < i → a[j] = j) ∧ i ≤ n}
7 }
8 // {∀j. 0 ≤ j < n → a[j] = j}
```

► Wichtiges Theorem:

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

Längeres Beispiel: Suche nach dem maximalen Element

```
1 // {0 < n}
2 i = 0;
3 r = 0;
4 while (i < n) {
5     if (a[r] < a[i]) {
6         r = i;
7     }
8     else {
9     }
10    i = i + 1;
11    // {(∀j. 0 ≤ j < i → a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
12 }
13 // {(∀j. 0 ≤ j < n → a[j] ≤ a[r]) ∧ 0 ≤ r < n}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1 // {0 ≤ n}
2 i = 0;
3 r = -1;
4 while (i < n) {
5     if (a[i] == 0) {
6         r = i;
7     }
8     else {
9     }
10    i = i + 1;
11    // {(r ≠ -1 → 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
12 }
13 // {r ≠ -1 → 0 ≤ r < n ∧ a[r] = 0}
```

Längeres Beispiel: Suche nach einem Null-Element

```
1 // {0 ≤ n}
2 i = 0;
3 r = -1;
4 while (i < n) {
5     if (a[i] == 0) {
6         r = i;
7     }
8     else {
9     }
10    i = i + 1;
11    // {(r ≠ -1 → 0 ≤ r < i ∧ a[r] = 0) ∧ 0 ≤ i ≤ n}
12 }
13 // {r ≠ -1 → 0 ≤ r < n ∧ a[r] = 0}
```

- Spezifikation zu schwach: wann ist $r = -1$?

Längeres Beispiel: Suche nach einem Null-Element

```
1 // {0 ≤ n}
2 i = 0;
3 r = -1;
4 while (i < n) {
5     if (a[i] == 0) {
6         r = i;
7     }
8     else {
9     }
10    i = i + 1;
11    // {(r ≠ -1 → 0 ≤ r < i ∧ a[r] = 0)
12        ∧ (r = -1 → ∀j. 0 ≤ j < i → a[j] ≠ 0)
13        ∧ 0 ≤ i ≤ n}
14 }
15 // {(r ≠ -1 → 0 ≤ r < n ∧ a[r] = 0)
16     ∧ (r = -1 → ∀j. 0 ≤ j < n → a[j] ≠ 0)}
```

Längeres Beispiel: Suche nach dem **ersten** Null-Element

```
1 // {0 ≤ n}
2 i = 0;
3 r = -1;
4 while (i < n) {
5     if (r == -1 && a[i] == 0) {
6         r = i;
7     }
8     else {
9     }
10    i = i + 1;
11    // {(r ≠ -1 → 0 ≤ r < i ∧ a[r] = 0 ∧ (∀j. 0 ≤ j < i → a[j] ≠ 0))
12        ∧ (r = -1 → ∀j. 0 ≤ j < i → a[j] ≠ 0)
13        0 ≤ i ≤ n}
14 }
15 // {(r ≠ -1 → 0 ≤ r < n ∧ a[r] = 0 ∧ (∀j. 0 ≤ j < r → a[j] ≠ 0))
16     ∧ (r = -1 → ∀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