

Korrekte Software: Grundlagen und Methoden

Vorlesung 7 vom 22.05.16: Strukturierte Datentypen: Strukturen und Felder

Serge Autexier, Christoph L  th

Universit  t Bremen

Sommersemester 2018

10:10:42 2018-07-31

1 [21]



Fahrplan

- Einf  hrung
- Operationale Semantik
- Denotationale Semantik
-   quivalenz der Operationalen und Denotationalen Semantik
- Die Floyd-Hoare-Logik
- Invarianten und die Korrektheit des Floyd-Hoare-Kalk  ls
- **Strukturierte Datentypen**
- Modellierung und Spezifikation
- Verifikationsbedingungen
- Vorw  rts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen
- Ausblick und R  ckblick

Korrekte Software

2 [21]



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**

Korrekte Software

3 [21]



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 Gr   e**.

Korrekte Software

4 [21]



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' }
```
- N  tzlicher syntaktischer Zucker:

```
char hallo[] = "hallo";
```
- Auswertung: `hallo[4]` liefert `o`

Korrekte Software

5 [21]



Strukturen

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

```
struct Vorlesung {
    char dozenten[2][30];
    char titel[30];
    int cp;
} ksgm;
```
- Zugriff auf Felder   ber Selektoren:

```
int i = 0;
char name1[] = "Serge Autexier";
while (i < strlen(name1)) {
    ksgm.dozenten[0][i] = name1[i];
    i = i + 1;
}
```
- Rekursive Strukturen nur   ber Zeiger erlaubt (kommt noch)

Korrekte Software

6 [21]



C0: Erweiterte Ausdr  cke

- **Lexp** beschreibt L-Werte (l-values), abstrakte Speicheradressen
- Neuer Basisdatentyp **C** f  r Zeichen
- Erweiterte Grammatik:

```
Lexp ::= Idt | /[a] | /.Idt
Aexp a ::= Z | C | Lexp | a1 + a2 | a1 - a2 | a1 * a2 | a1 / a2
Bexp b ::= 1 | 0 | a1 == a2 | a1 < a2 | ! b | b1 && b2 | b1 || b2
Exp e ::= Aexp | Bexp
```

Korrekte Software

7 [21]



Werte und Zust  nde

- Zust  nde bilden **strukturierte** Adressen auf Werte (wie vorher) ab.

Systemzust  nde

- **Locations:** $\text{Loc} ::= \text{Idt} \mid \text{Loc}[\mathbb{Z}] \mid \text{Loc.Idt}$
- Werte: $\mathbf{V} = \mathbb{Z}$
- Zust  nde: $\Sigma \stackrel{\text{def}}{=} \text{Loc} \rightarrow \mathbf{V}$

- Wir betrachten nur Zugriffe vom Typ **Z** oder **C** (**elementare Typen**)
- N  tzliche Abstraktion des tats  chliche C-Speichermodells

Korrekte Software

8 [21]



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:

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

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

- Zuweisungen sind nur definiert für elementare Typen:

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

- 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 **Idt**
 - 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) }
```

- Wichtiges Theorem:

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



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}
```

Korrekte Software

17 [21]



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 $r = -1$?

Korrekte Software

18 [21]



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)}
```

Korrekte Software

19 [21]



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)}
```

Korrekte Software

20 [21]



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

Korrekte Software

21 [21]

