Korrekte Software: Grundlagen und Methoden
Vorlesung 10 vom 11.06.19
Modellierung und Spezifikation

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2019



## Fahrplan





## 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}
```



## Beispiel: Sortierte Felder

- Wie formulieren wir, dass ein Array sortiert ist? Ggf. bis zu einem bestimmten Punkt $n$ sortiert ist?

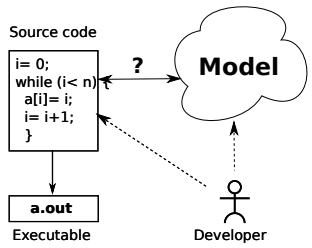
```
int a[8];
// {∀0 ≤ j ≤ n < 6.a[j] ≤ a[j + 1]}
```

- Alternativ würden man auch gerne ein Prädikat definieren können

$//\{\forall a.sorteduntil(a, 0) \longleftrightarrow true\}$
$//\{\forall a. \forall i.i \geq 0 \longrightarrow (sorteduntil(a, i + 1) \longleftrightarrow (a[i] \leq a[i + 1] \wedge sorteduntil(a, i)))\}$



## Generelles Problem: Modellbildung






## Was brauchen wir?

- Expressive **logische Sprache** (**Assn**)
- Konzeptbildung auf der Modellebene
  - Funktionen
  - Typen
- Beispiele:
  - Separate Modellierungssprache, bspw. UML/OCL
  - Modellierungskonzepte in der Annotationssprache (ACSL, JML)



## Modellierung von Typen: Integers

- Vereinfachung: **int** wird abgebildet auf $\mathbb{Z}$
- Das **kann** sehr falsch sein
- Manchmal **unerwartete** Effekte
- Behebung: statisch auf **Überlauf** prüfen
  - Nachteil: Plattformspezifisch



## Binäre Suche

```
1  int binary_search(int val, int buf[], unsigned len)
2  {
3    // {0 ≤ len}
4    int low, high, mid, res;
5    low = 0; high = len;
6    while (low < high) {
7      mid= (low+ high)/2;
8      if (buf[mid] < val)
9        low = mid + 1;
10     else
11       high = mid;
12   }
13   if (low < len && buf[low] == val)
14     res= low;
15   else
16     res= −1;
17   // { res ≠ −1 ⟶ buf[res] = val ∧                    }
       //     res = −1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val
18 }
```

## Binäre Suche, korrekt

```
1  int binary_search(int val, int buf[], unsigned len)
2  {
3    // {0 ≤ len}
4    int low, high, mid, res;
5    low = 0; high = len;
6    while (low < high) {
7      mid= low+ (high−low)/2;
8      if (buf[mid] < val)
9        low = mid + 1;
10     else
11       high = mid;
12   }
13   if (low < len && buf[low] == val)
14     res= low;
15   else
16     res= −1;
17   // { res ≠ −1 ⟶ buf[res] = val ∧                    }
       //     res = −1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val
18 }
```



## Typen: reelle Zahlen

- Vereinfachung: **double** wird abgebildet auf $\mathbb{R}$
- Auch hier **Fehler** und **unerwartete Effekte** möglich:
  - Kein Überlauf, aber **Rundungsfehler**
  - Fließkommazahlen: Standard IEEE 754-2008
- Mögliche Abhilfe:
  - Spezifikation der Abweichung von **exakter** (ideeller) Berechnung



## Typen: labelled records

- Passen gut zu Klassen (Klassendiagramme in der UML)
- Bis auf Methoden: impliziter Parameter self
  - Werden nicht behandelt



## Typen: Felder

- Was repräsentiert **Felder**?
- **Sequenzen** (Listen)
- Modellierungssprache:
  - Annotation + **OCL**



## Ein längeres Beispiel: reverse in-place

```
1  i= 0;
2  // {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
3  while (i< n) {
4    // ???
5    tmp= a[n−1−i];
6    a[n−1−i]= a[i];
7    a[i]= tmp;
8    i= i+1;
9  }
10 // {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```



## Ein längeres Beispiel: reverse in-place

```
1  i= 0;
2  // {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
3  while (i< n/2) {
4    // { ∀j.0 ≤ j < i ⟶ a[j] = b[n − 1 − j]∧
       //       ∀j.n − 1 − i < j < n ⟶ a[j] = b[n − 1 − j]∧
       //       ∀j.i ≤ j ≤ n − 1 − i ⟶ a[j] = b[j] }
5    tmp= a[n−1−i];
6    a[n−1−i]= a[i];
7    a[i]= tmp;
8    i= i+1;
9  }
10 // {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```



## Vereinfacht mit Modellbildung

- seq$(a, n)$ ist ein Feld der Länge $n$ repräsentiert als Liste (Sequenz)
- Aktionen auf Sequenzen:
  - rev$(a)$ — Reverse
  - $a[i : j]$ — Slicing (à la Python)
  - $++$ — Konkatenation



## Ein längeres Beispiel, vereinfacht

```
1  i= 0;
2  // {bs = seq(a, n)}
3  while (i< n/2) {
4    // { as = seq(a, n) ⟹
       //       rev(as[n − i : n])++as[i : n − i]++rev(as[0 : i]) = bs}
5    tmp= a[n−1−i];
6    a[n−i−1]= a[i];
7    a[i]= tmp;
8    i= i+1;
9  }
10 // {as = seq(a, n) ⟹ rev(as) = bs}
```

## Formelsprache mit Quantoren

- Wir brauchen Programmausdrücken wie **Aexp**
- Wir müssen neue Funktionen verwenden können
  - Etwa eine Fakultätsfunkion
- Wir müssen neue Prädikate definieren können
  - rev, *sorted*, ...
- Wir müssen Formeln bilden können
  - Analog zu **Bexp**
  - Zusätzlich mit Implikation $\longrightarrow$, Äquivalenz $\longleftrightarrow$
  - Zusätzlich Quantoren über logische Variablen wie in

$$(\forall j.\, 0 \leq j < n \longrightarrow P[j]) \wedge P[n] \longrightarrow \forall j.\, 0 \leq j < n+1 \longrightarrow P[j]$$
$$\forall i.\, i \geq 0 \longrightarrow (sorteduntil(a, i+1) \longleftrightarrow (a[i] \leq a[i+1] \wedge sorteduntil(a, i)))$$



## Was brauchen wir?

- Definiere Terme als Variablen und Funktionen besimmter Stelligkeit
- Definiere Literale und Formeln
- Interpretation von Formeln
  - mit und ohne Programmvariablen



## Zusicherungen (Assertions)

- Erweiterung von **Aexp** and **Bexp** durch
  - **Logische** Variablen **Var** $v := N, M, L, U, V, X, Y, Z$
  - ~~Definierte Funktionen und Prädikate über **Aexp** $n!, \sum_{i=1}^{n} i, \ldots$~~
  - Funktionen und Prädikate selbst definieren
  - Implikation, Äquivalenzen und Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v.\, b, \exists v.\, b$
- Formal:

$$\begin{array}{rll} \textbf{Lexp} & l ::= & \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \\ \textbf{Aexpv} & a ::= & \textbf{Z} \mid \textbf{Idt} \mid \textbf{Var} \mid \textbf{C} \mid \textbf{Lexp} \\ & & \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\ & & \mid f(e_1, \ldots, e_n) \\ \textbf{Assn} & b ::= & \textbf{1} \mid \textbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2 \\ & & \mid !\, b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\ & & \mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \ldots, e_n) \\ & & \mid \forall v.\, b \mid \exists v.\, b \end{array}$$



## Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in$ **Assn** in einem Zustand $\sigma$?
  - Auswertung (denotationale Semantik) ergibt *true*
- **Belegung** der logischen Variablen: $I : \textbf{Var} \to (\textbf{Z} \cup \textbf{C})$
- Semantik von $b$ unter der Belegung $I$: $\mathcal{B}_v[\![b]\!]^I, \mathcal{A}_v[\![a]\!]^I$

$$\mathcal{A}_v[\![I]\!]^I = \{(\sigma, \sigma(i)) \mid (\sigma, i) \in \mathcal{L}_v[\![I]\!]^I, i \in Dom(\sigma)\}$$



## Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in$ **Assn** in einem Zustand $\sigma$?
  - Auswertung (denotationale Semantik) ergibt *true*
- **Belegung** der logischen Variablen: $I : \textbf{Var} \to (\textbf{Z} \cup \textbf{C} \cup \text{Array})$
- Semantik von $b$ unter der Belegung $I$:

$$\begin{aligned} \mathcal{B}_v[\![\forall v.b]\!]^I = \{&(\sigma, true) \mid \text{für alle } i \in \textbf{Z} \text{ gilt } (\sigma, true) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \\ \cup \{&(\sigma, false) \mid \text{für ein } i \in \textbf{Z} \text{ gilt } (\sigma, false) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \\ \mathcal{B}_v[\![\exists v.b]\!]^I = \{&(\sigma, true) \mid \text{für ein } i \in \textbf{Z} \text{ gilt } (\sigma, true) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \\ \cup \{&(\sigma, false) \mid \text{für alle } i \in \textbf{Z} \text{ gilt } (\sigma, false) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\} \end{aligned}$$

Analog für andere Typen.



## Erfülltheit von Zusicherungen

**Erfülltheit von Zusicherungen**

$b \in$ **Assn** ist in Zustand $\sigma$ mit Belegung $I$ erfüllt ($\sigma \models^I b$), gdw

$$\mathcal{B}_v[\![b]\!]^I(\sigma) = true$$



## Formeln ohne Programmvariablen, ohne Arrays, ohne Strukturen

- Eine Formel $b \in$ **Assn** ist pur, wenn sie weder Programmvariablen, noch Strukturen, noch Felder enthält (also keine Teilterme aus **Lexp** und **Idt**.
- Eine Formel ist geschlossen, wenn sie pur ist und keine freien logischen Variablen enthält.
- Sei $\textbf{Assn}^c \subseteq \textbf{Assn}$ die Menge der geschlossenen Formeln

**Lemma**

*Für eine geschlossene Formel b ist der Wahrheitswert $\mathcal{B}_v[\![b]\!]^I(\sigma)$ von b unabhängig von I und $\sigma$.*

- Sei $\Gamma$ eine endliche Menge von Formeln, dann definieren wir

$$\bigwedge \Gamma := \begin{cases} b_1 \wedge \ldots \wedge b_n & \text{für alle } b_i \in \Gamma, \Gamma \neq \emptyset \\ true & \text{falls } \Gamma = \emptyset \end{cases}$$



## Erfülltheit von Zusicherungen unter Kontext

**Erfülltheit von Zusicherungen unter Kontext**

Sei $\Gamma \subseteq \textbf{Assn}^c$ eine endliche Menge und $b \in$ **Assn**. Im Kontext $\Gamma$ ist $b$ in Zustand $\sigma$ mit Belegung $I$ erfüllt ($\Gamma, \sigma \models^I b$), gdw

$$\mathcal{B}_v[\![\Gamma \longrightarrow b]\!]^I(\sigma) = true$$

## Floyd-Hoare-Tripel mit Kontext

- Sei $\Gamma \in \mathbf{Assn}^c$ und $P, Q \subseteq \mathbf{Assn}$

**Partielle Korrektheit unter Kontext ($\Gamma \models \{P\}\, c\, \{Q\}$)**

$c$ ist **partiell korrekt**, wenn für alle Zustände $\sigma$ und alle Belegungen $I$ die unter Kontext $\Gamma$ $P$ erfüllen, gilt:
**wenn** die Ausführung von $c$ mit $\sigma$ in $\sigma'$ terminiert, **dann** erfüllen $\sigma'$ und $I$ im Kontext $\Gamma$ auch $Q$.

$$\Gamma \models \{P\}\, c\, \{Q\} \iff \forall I.\, \forall \sigma.\, \Gamma, \sigma \models^I P \wedge \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!] \implies \Gamma, \sigma' \models^I Q$$



## Floyd-Hoare-Kalkül mit Kontext

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

$$\frac{\Gamma \vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \Gamma \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\Gamma \vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\Gamma \vdash \{A \wedge b\}\, c\, \{A\}}{\Gamma \vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{\Gamma \vdash \{A\}\, c_1\, \{B\} \qquad \Gamma \vdash \{B\}\, c_2\, \{C\}}{\Gamma \vdash \{A\}\, c_1; c_2\, \{C\}}$$



## Floyd-Hoare-Kalkül mit Kontext

$$\frac{\Gamma \longrightarrow (A' \longrightarrow A) \qquad \Gamma \vdash \{A\}\, c\, \{B\} \qquad \Gamma \longrightarrow (B \longrightarrow B')}{\Gamma \vdash \{A'\}\, c\, \{B'\}}$$

und es muss gezeigt werden für alle Zustände $\sigma$ und Belegungen $I$ dass $\Gamma \longrightarrow (A' \longrightarrow A)$ wahr bzw. dass

$$\mathcal{B}_v[\![\Gamma \longrightarrow (A' \longrightarrow A)]\!]^I(\sigma) = \mathit{true}$$

(Analog für $\Gamma \longrightarrow (B \longrightarrow B')$).

**Problem**

$\mathcal{B}_v[\![.]\!]^I(\sigma)$ *im Allgemeinen nicht berechenbar wegen*

$$\mathcal{B}_v[\![\forall \mathbf{z} v.b]\!]^I = \{(\sigma, 1) \mid \textit{für alle } i \in \mathbf{Z} \textit{ gilt } (\sigma, 1) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\}$$
$$\cup \{(\sigma, 0) \mid \textit{für ein } i \in \mathbf{Z} \textit{ gilt } (\sigma, 0) \in \mathcal{B}_v[\![b]\!]^{I[i/v]}\}$$



## Zusammenfassung

- Spezifikation erfordert **Modellbildung**
- Herangehensweisen:
  - Modellbildung in der Annotation ("ghost-code")
  - Separate Modellierungssprache
- Erweiterung der Annotationssprache um logische Anteile
  - Quantoren, Typen, Kontexte
- Problem: Unvollständigkeit der Logik