

Korrekte Software: Grundlagen und Methoden
Vorlesung 10 vom 23.06.20
Modellierung und Spezifikation

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Fahrplan

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

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    // {( $\forall j. 0 \leq j < i \rightarrow a[j] \leq a[r]$ )  $\wedge 0 \leq i < n \wedge 0 \leq r < n$ }
12 }
13 // {( $\forall j. 0 \leq j < n \rightarrow a[j] \leq a[r]$ )  $\wedge 0 \leq r < n$ }
```

Beispiel: Sortierte Felder

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

```
int a[8];  
// { $\forall j. 0 \leq j \leq n < 8. a[j] \leq a[j + 1]$ }
```

- ▶ Alternativ würden man auch gerne ein Prädikat definieren können

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

- ▶ ... und damit beweisen dass:

```
// { $\forall a \forall n. sorted(a, n) \longrightarrow \forall i, j. 0 \leq i \leq j \leq n \longrightarrow a[i] \leq a[j]$ }
```

Generelles Problem: Modellbildung

Source code

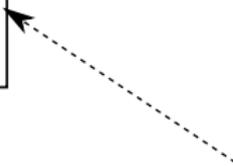
```
i= 0;  
while (i< n) {  
  a[i]= i;  
  i= i+1;  
}
```

a.out

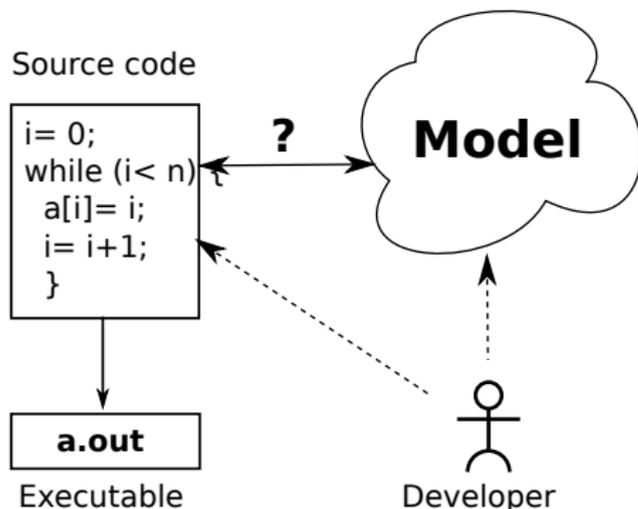
Executable



Developer



Generelles Problem: Modellbildung



Modell ist **abstrakte**
Repräsentation:

- ▶ Verhalten des Programmes kann kürzer beschrieben werden
- ▶ Einfachere Beweise

Modell ist **treue**
Repräsentation:

- ▶ Eigenschaften des Modelles gelten auch für das Programm

Was brauchen wir?

- ▶ Expressive **logische Sprache** (**Assn**)
- ▶ Konzeptbildung auf der Modellebene
 - ▶ Reichere Typen (bspw. Repräsentation von Feldern durch Listen)
 - ▶ Mehr Funktionen (bspw. auf Listen)
- ▶ Beispiele:
 - ▶ Separate Modellierungssprache, bspw. UML/OCL
 - ▶ Modellierungskonzepte in der Annotationsprache (ACSL, JML)

Modellierung von Typen: Integer

- ▶ 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 ∧
18         //   res = -1 → ∀j. 0 ≤ j < len → buf[j] ≠ val }
19 }
```

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 ∧
18         //   res = -1 → ∀j. 0 ≤ j < len → buf[j] ≠ val }
19 }
```

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

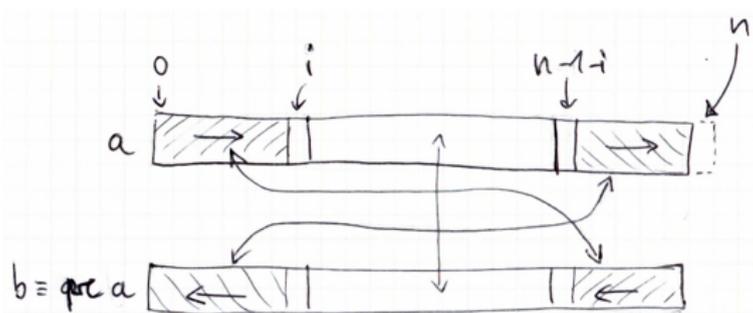
Typen: Felder

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

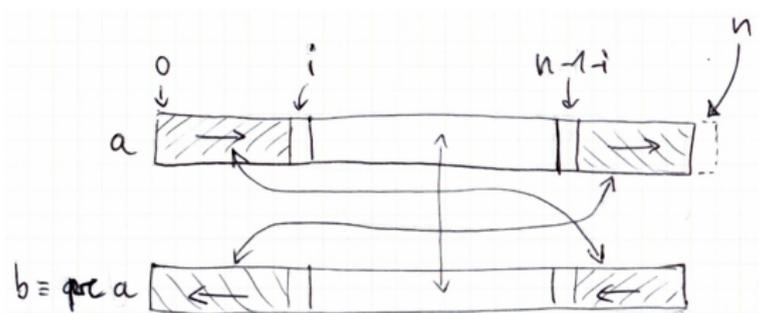
Ein längeres Beispiel: reverse in-place

```
1 i = 0;
2 // { $\forall i. 0 \leq i < n \rightarrow a[i] = b[i]$ }
3 while (i < n/2) {
4     // ???
5     tmp = a[n-1-i];
6     a[n-1-i] = a[i];
7     a[i] = tmp;
8     i = i + 1;
9 }
10 // { $\forall j. 0 \leq j < n \rightarrow a[j] = b[n-1-j]$ }
```

reverse-in-place: die Invariante



reverse-in-place: die Invariante



Mathematisch:

$$\left\{ \begin{array}{l} \forall j. 0 \leq j < i \longrightarrow a[j] = b[n-1-j] \wedge \\ \forall j. n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge \\ \forall j. i \leq j \leq n-1-i \longrightarrow a[j] = b[j] \end{array} \right\}$$

Ein längeres Beispiel: reverse in-place

```
1  i = 0;
2  // { $\forall i. 0 \leq i < n \rightarrow a[i] = b[i]$ }
3  while (i < n/2) {
4      // {  $\forall j. 0 \leq j < i \rightarrow a[j] = b[n-1-j] \wedge$   

   //       $\forall j. n-1-i < j < n \rightarrow a[j] = b[n-1-j] \wedge$   

   //       $\forall j. i \leq j \leq n-1-i \rightarrow a[j] = b[j]$  }
5      tmp = a[n-1-i];
6      a[n-1-i] = a[i];
7      a[i] = tmp;
8      i = i+1;
9  }
10 // { $\forall j. 0 \leq j < n \rightarrow a[j] = b[n-1-j]$ }
```

Arbeitsblatt 10.1: Jetzt seit ihr dran

- ▶ Berechnet die Beweisverpflichtungen aus der While-Schleife bei reverse-in-place:

$$I \wedge b \longrightarrow \text{awp}(c, I)$$

- ▶ Dazu berechnet ihr $\text{awp}(c, I)$, mit
 $c =$

```
tmp = a[n-1-i];  
a[n-1-i] = a[i];  
a[i] = tmp;  
i = i+1;
```

$$I = \{ \forall j. 0 \leq j < i \longrightarrow a[j] = b[n-1-j] \wedge \\ \forall j. n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge \\ \forall j. i \leq j \leq n-1-i \longrightarrow a[j] = b[j] \}$$

- ▶ Ihr braucht noch nichts zu beweisen...

Vereinfacht mit Modellbildung

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

Interaktion mit der Substitution

- ▶ $\text{set}(a, i, v)$ ist der **funktionale Update** an Index i mit dem Wert v :

$$\text{set}([], i, v) == []$$

$$\text{set}(a : as, 0, v) == v : as$$

$$i > 0 \longrightarrow \text{set}(a : as, i, v) == a : \text{set}(as, i - 1, v)$$

$$i < 0 \longrightarrow \text{set}(as, i, v) == as$$

- ▶ Damit ist

$$\text{seq}(a, n)[v/a[i]] = \text{set}(\text{seq}(a, n), i, v)$$

Reverse-in-Place mit Listen

```
1  i= 0;
2  // {bs = seq(a, n)}
3  while (i < n/2)
4      /** inv
5       */ {
6      tmp= a[n-1-i];
7      a[n-i-1]= a[i];
8      a[i]= tmp;
9      i= i+1;
10 }
11 // {as = seq(a, n)  $\implies$  rev(as) = bs}
```

- ▶ Damit vereinfachte VCs und vereinfachter Beweis.

Reverse-in-Place mit Listen

```
1  i = 0;
2  // {bs = seq(a, n)}
3  while (i < n/2)
4      /** inv   as = seq(a, n)  $\implies$ 
           rev(as[n - i : n]) ++ as[i : n - i] ++ rev(as[0 : i]) = bs
5      */ {
6      tmp = a[n - 1 - i];
7      a[n - i - 1] = a[i];
8      a[i] = tmp;
9      i = i + 1;
10 }
11 // {as = seq(a, n)  $\implies$  rev(as) = bs}
```

- ▶ Damit vereinfachte VCs und vereinfachter Beweis.

Arbeitsblatt 10.2: Beweise mit Listen

- ▶ Beweist durch **strukturelle Induktion** auf Sequenzen:

$$\text{rev}(as++bs) == \text{rev}(bs)++\text{rev}(as)$$

- ▶ Strukturelle Induktion heißt:

① Induktionsbasis: zeige Aussage für $as \stackrel{\text{def}}{=} []$.

② Induktionsschritt: Annahme der Aussage, zeige Aussage für $as \stackrel{\text{def}}{=} a : as$

- ▶ Beweis durch Umformung, Anwendung der Gleichungen für rev , $++$

$$\text{rev}([]) == []$$

$$\text{rev}(x : xs) == \text{rev}(xs)++[x]$$

$$[]++ys == ys$$

$$(x : xs)++ys == x : (xs++ys)$$

Fazit

- ▶ Die Abstraktion ermöglicht wesentlich **kürzere** Vorbedingungen und Verifikationsbedingungen.
- ▶ Die Beweise auf Ebene der Listen sind wesentlich **einfacher**.
- ▶ Die Theorie der Listen ist wesentlich **reicher**.

Formelsprache mit Quantoren

- ▶ Wir brauchen Programmausdrücken wie **Aexp**
- ▶ Wir müssen neue Funktionen verwenden können
 - ▶ Etwa eine Fakultätsfunktion
- ▶ 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

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

Was brauchen wir?

- ▶ Definiere Terme als Variablen und Funktionen bestimmter 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** $n!, \sum_{i=1}^n i, \dots$

- ▶ Implikation, **Äquivalenzen**, Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v. b, \exists v. b$

- ▶ Formal:

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

Aexpv $a ::= \mathbf{Z} \mid \text{Idt} \mid \text{Var} \mid \mathbf{C} \mid \text{Lexp}$
 $\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$
 $\mid f(e_1, \dots, e_n)$

Assn $b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2$
 $\mid !b \mid b_1 \&\& b_2 \mid b_1 || b_2$
 $\mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \dots, e_n)$
 $\mid \forall v. b \mid \exists v. b$

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**~~ $n!, \sum_{i=1}^n i, \dots$

- ▶ Implikation, **Äquivalenzen**, Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v. b, \exists v. b$

- ▶ Formal:

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

Aexpv $a ::= \mathbf{Z} \mid \text{Idt} \mid \text{Var} \mid \mathbf{C} \mid \text{Lexp}$
 $\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$
 $\mid f(e_1, \dots, e_n)$

Assn $b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2$
 $\mid !b \mid b_1 \&\& b_2 \mid b_1 || b_2$
 $\mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \dots, e_n)$
 $\mid \forall v. b \mid \exists v. b$

Zusicherungen (Assertions)

- ▶ Erweiterung von **Aexp** and **Bexp** durch

- ▶ **Logische** Variablen **Var**

$v := N, M, L, U, V, X, Y, Z$

- ▶ Funktionen und Prädikate selbst definieren

- ▶ Implikation, **Äquivalenzen**, Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v. b, \exists v. b$

- ▶ Formal:

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

Aexpv $a ::= \mathbf{Z} \mid \text{Idt} \mid \text{Var} \mid \mathbf{C} \mid \text{Lexp}$
 $\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$
 $\mid f(e_1, \dots, e_n)$

Assn $b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 != a_2 \mid a_1 <= a_2$
 $\mid !b \mid b_1 \&\&b_2 \mid b_1 || b_2$
 $\mid b_1 \longrightarrow b_2 \mid b_1 \longleftrightarrow b_2 \mid p(e_1, \dots, e_n)$
 $\mid \forall v. b \mid \exists v. b$

Die bisherigen Funktionen

Die bisherigen Funktionen selbst definiert:

$$n! == \text{factorial}(n)$$

$$i \leq 0 \longrightarrow \text{factorial}(i) == 1$$

$$i > 0 \longrightarrow \text{factorial}(i) == i \cdot \text{factorial}(i - 1)$$

$$\sum_{i=a}^b i == \text{sum}(a, b)$$

$$a > b \longrightarrow \text{sum}(a, b) == 0$$

$$a \leq b \longrightarrow \text{sum}(a, b) == a + \text{sum}(a + 1, b)$$

Kombination aus eingebautem **syntaktische Zucker** und eigenen **Definitionen**.

Die bisherigen Funktionen

- ▶ $\sum_{i=a}^b e$, $\prod_{i=a}^b e$ benötigen Funktionen **höherer Ordnung** und **anonyme Funktionen**:
- ▶ Ganz allgemein:

$$a \leq b \longrightarrow [a .. b] == a : [a + 1 .. b]$$

$$a > b \longrightarrow [a .. b] == []$$

$$\text{foldl}(f, c, a : as) == \text{foldl}(f, f(c, a), as)$$

$$\text{foldl}(f, c, []) == c$$

$$\sum_{i=a}^b e(i) == \text{foldl}(\lambda xi.x + e(i), 0, [a .. b])$$

$$\prod_{i=a}^b e(i) == \text{foldl}(\lambda xi.x \cdot e(i), 1, [a .. b])$$

Ein Zoo von Logiken

- Das grundlegende Dilemma:

Entscheidbarkeit \longleftrightarrow Ausdrucksmächtigkeit

- Der Logik-Zoo:

	Entscheidbar	Vollständig	
Aussagenlogik (OPL)	✓	✓	$(A \wedge B) \vee C$
Pressburger Arithmetik	✓	✓	$n < x \longrightarrow n + a < x + a$
Prädikatenlogik (PL)	✗	✓	$\forall x. \exists y. x = y$
Peano-Arithmetik	✗	✗	$n \cdot 0 = 0$
PL mit Ind. & Fkt.	✗	✗	$Z3$
Prädikatenlogik 2. Stufe	✗	✗	$\forall P. P(0) \longrightarrow \forall n. P(n)$
Logik höherer Stufe (HOL)	✗	✗	<i>Haskell</i>

- Auswahl der Logik: Kompromiss (*sweet spot*)

Erfüllung von Zusicherungen

- ▶ Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand σ ?
 - ▶ Auswertung (denotationale Semantik) ergibt *true*
- ▶ **Belegung** der logischen Variablen: $I : \mathbf{Var} \rightarrow (\mathbf{Z} \cup \mathbf{C})$
- ▶ Semantik von b unter der Belegung I : $\llbracket b \rrbracket_{\mathcal{B}_V}^I, \llbracket a \rrbracket_{\mathcal{A}_V}^I$

$$\llbracket I \rrbracket_{\mathcal{A}_V}^I = \{(\sigma, \sigma(i) \mid (\sigma, i) \in \llbracket I \rrbracket_{\mathcal{L}_V}^I, i \in \text{Dom}(\sigma)\}$$

Erfüllung von Zusicherungen

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

$$\begin{aligned} \llbracket \forall v. b \rrbracket_{Bv}^I &= \{(\sigma, true) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma, true) \in \llbracket b \rrbracket_{Bv}^{I[i/v]}\} \\ &\quad \cup \{(\sigma, false) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma, false) \in \llbracket b \rrbracket_{Bv}^{I[i/v]}\} \\ \llbracket \exists v. b \rrbracket_{Bv}^I &= \{(\sigma, true) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma, true) \in \llbracket b \rrbracket_{Bv}^{I[i/v]}\} \\ &\quad \cup \{(\sigma, false) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma, false) \in \llbracket b \rrbracket_{Bv}^{I[i/v]}\} \end{aligned}$$

Analog für andere Typen.

Erfülltheit von Zusicherungen

Erfülltheit von Zusicherungen

$b \in \mathbf{Assn}$ ist in Zustand σ mit Belegung l erfüllt ($\sigma \models^l b$), gdw

$$\llbracket b \rrbracket'_{\mathcal{B}_V}(\sigma) = true$$

Formeln ohne Programmvariablen, ohne Arrays, ohne Strukturen

- ▶ Eine Formel $b \in \mathbf{Assn}$ 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 $\mathbf{Assn}^c \subseteq \mathbf{Assn}$ die Menge der geschlossenen Formeln

Lemma

Für eine geschlossene Formel b ist der Wahrheitswert $\llbracket b \rrbracket_{B_V}^I(\sigma)$ von b unabhängig von I und σ .

- ▶ Sei Γ eine endliche Menge von Formeln, dann definieren wir

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

Erfülltheit von Zusicherungen unter Kontext

Erfülltheit von Zusicherungen unter Kontext

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

$$\llbracket \Gamma \longrightarrow b \rrbracket_{B_V}^l(\sigma) = \text{true}$$

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 σ und alle Belegungen l die unter Kontext Γ P erfüllen, gilt:

wenn die Ausführung von c mit σ in σ' terminiert, **dann** erfüllen σ' und l im Kontext Γ auch Q .

$$\Gamma \models \{P\} c \{Q\} \iff \forall l. \forall \sigma. \Gamma, \sigma \models' P \wedge \exists \sigma'. (\sigma, \sigma') \in \llbracket c \rrbracket_c \implies \Gamma, \sigma' \models' Q$$

Floyd-Hoare-Kalkül mit Kontext

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

Floyd-Hoare-Kalkül mit Kontext

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

$$\frac{\Gamma \vdash \{A \wedge b\} c_0 \{B\} \quad \Gamma \vdash \{A \wedge \neg b\} c_1 \{B\}}{\Gamma \vdash \{A\} \mathbf{if} (b) c_0 \mathbf{else} c_1 \{B\}}$$

Floyd-Hoare-Kalkül mit Kontext

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

$$\frac{\Gamma \vdash \{A \wedge b\} c_0 \{B\} \quad \Gamma \vdash \{A \wedge \neg b\} c_1 \{B\}}{\Gamma \vdash \{A\} \mathbf{if} (b) c_0 \mathbf{else} c_1 \{B\}}$$

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

Floyd-Hoare-Kalkül mit Kontext

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

$$\frac{\Gamma \vdash \{A \wedge b\} c_0 \{B\} \quad \Gamma \vdash \{A \wedge \neg b\} c_1 \{B\}}{\Gamma \vdash \{A\} \text{if } (b) \ c_0 \ \text{else } c_1 \{B\}}$$

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

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

Floyd-Hoare-Kalkül mit Kontext

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

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

$$\llbracket \Gamma \longrightarrow (A' \longrightarrow A) \rrbracket'_{Bv}(\sigma) = true$$

Floyd-Hoare-Kalkül mit Kontext

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

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

$$\llbracket \Gamma \longrightarrow (A' \longrightarrow A) \rrbracket'_{B_V}(\sigma) = true$$

- ▶ $\llbracket \cdot \rrbracket'_{B_V}(\sigma)$ im Allgemeinen nicht berechenbar wegen

$$\begin{aligned} \llbracket \forall z v. b \rrbracket'_{B_V} &= \{(\sigma, 1) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma, 1) \in \llbracket b \rrbracket'_{B_V}^{[i/v]}\} \\ &\cup \{(\sigma, 0) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma, 0) \in \llbracket b \rrbracket'_{B_V}^{[i/v]}\} \end{aligned}$$

- ▶ Unvollständigkeit der Prädiktenlogik

Zusammenfassung

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