Korrekte Software: Grundlagen und Methoden
Vorlesung 11 vom 22.06.21
Modellierung und Spezifikation

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Fahrplan

Beispiel: Suche nach dem maximalen Element

```
// {0 < n}
i= 0;
r= 0;
while (i< n) {
  if (a[r] < a[i]) {
    r= i;
    }
  else {
    }
  i= i+1;
  // {(∀j. 0 ≤ j < i ⟶ a[j] ≤ a[r]) ∧ 0 ≤ i ≤ n ∧ 0 ≤ r < n}
  }
// {(∀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];
// {∀j. 0 ≤ j ≤ n < 8.a[j] ≤ a[j + 1]}
```

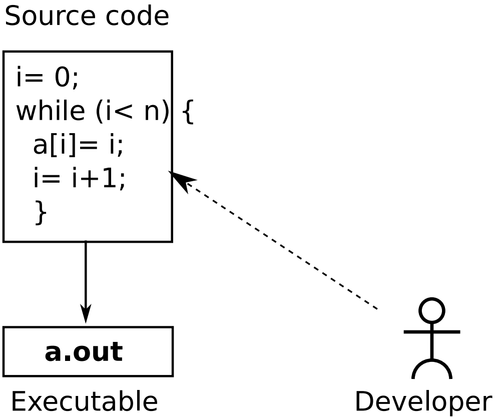
- Alternativ würden man auch gerne ein Prädikat definieren können

```
//{∀a. sorted(a, 0) ⟷ true}
//{∀a∀i. i ≥ 0 ⟶ (sorted(a, i + 1) ⟷ (a[i] ≤ a[i + 1] ∧ sorted(a, i)))}
```

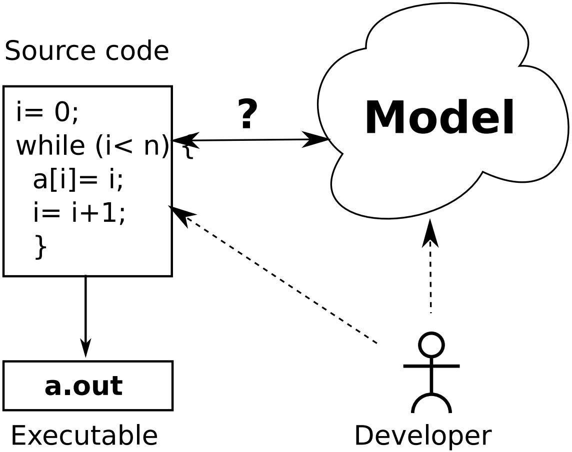
- ... und damit beweisen dass:

```
//{∀a∀n. sorted(a, n) ⟶ ∀i, j. 0 ≤ i ≤ j ≤ n ⟶ a[i] ≤ a[j]}
```

Generelles Problem: Modellbildung



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 Annotationssprache (JML, ACSL)

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

```
int binary_search(int val, int buf[], unsigned len)
{
  // {0 ≤ len}
  int low, high, mid, res;
  low = 0; high = len;
  while (low < high) {
    mid= (low+ high)/2;
    if (buf[mid] < val)
      low = mid + 1;
    else
      high = mid;
  }
  if (low < len && buf[low] == val)
    res= low;
  else
    res= −1;
  // { res ≠ −1 ⟶ buf[res] = val ∧
       res = −1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val }
}
```

Binäre Suche, korrekt

```
int binary_search(int val, int buf[], unsigned len)
{
  // {0 ≤ len}
  int low, high, mid, res;
  low = 0; high = len;
  while (low < high) {
    mid= low+ (high-low)/2;
    if (buf[mid] < val)
      low = mid + 1;
    else
      high = mid;
  }
  if (low < len && buf[low] == val)
    res= low;
  else
    res= -1;
  // { res ≠ -1 ⟶ buf[res] = val ∧
        res = -1 ⟶ ∀j.0 ≤ j < len ⟶ buf[j] ≠ val }
}
```

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

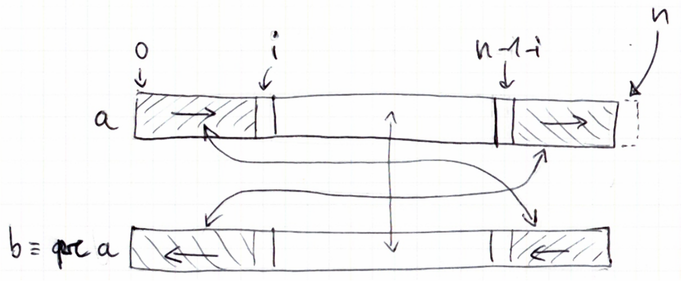
Typen: Felder

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

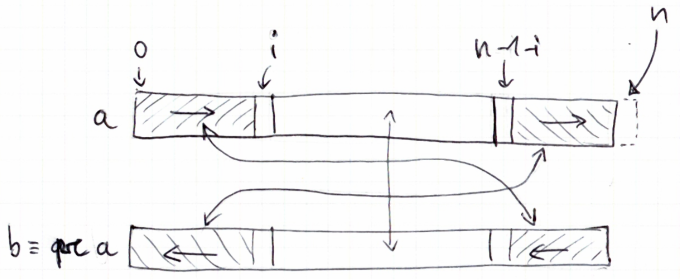
Ein längeres Beispiel: reverse in-place

```
i= 0;
// {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
while (i< n/2) {
   // ???
   tmp= a[n-1-i];
   a[n-1-i]= a[i];
   a[i]= tmp;
   i= i+1;
}
// {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```

reverse-in-place: die Invariante



reverse-in-place: die Invariante



Mathematisch:

$$\begin{aligned}\{\ & \forall j. 0 \leq j < i \longrightarrow a[j] = b[n-1-j] \wedge \\ & \forall j. n-1-i < j < n \longrightarrow a[j] = b[n-1-j] \wedge \\ & \forall j. i \leq j \leq n-1-i \longrightarrow a[j] = b[j] \quad \}\end{aligned}$$

Ein längeres Beispiel: reverse in-place

```
i= 0;
// {∀i.0 ≤ i < n ⟶ a[i] = b[i]}
while (i< n/2) {
   // { ∀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]  }
   tmp= a[n−1−i];
   a[n−1−i]= a[i];
   a[i]= tmp;
   i= i+1;
}
// {∀j.0 ≤ j < n ⟶ a[i] = b[n − 1 − i]}
```

Arbeitsblatt 11.1: Jetzt seid ihr dran

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

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

- Dazu berechnet ihr $\mathrm{awp}(c, I)$, 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...

Überblick: Approximative schwächste Vorbedingung

$$
\begin{aligned}
\mathrm{awp}(\{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(I = e, P) &\stackrel{def}{=} P[e/I] \quad (\textbf{Genauer}\text{: Folie 24 aus VL 8 bzw. nächste Folie}) \\
\mathrm{awp}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{awp}(c_1, \mathrm{awp}(c_2, P)) \\
\mathrm{awp}(\textbf{if } (b)\ c_0 \ \textbf{else}\ c_1, P) &\stackrel{def}{=} Q \quad \text{wenn } \mathrm{awp}(c_0, P) = b \wedge Q, \mathrm{awp}(c_1, P) = \neg b \wedge Q \\
\mathrm{awp}(\textbf{if } (b)\ c_0 \ \textbf{else}\ c_1, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\
\mathrm{awp}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \mathrm{q} \\
\mathrm{awp}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv}\ \ i\ {*}/\ c, P) &\stackrel{def}{=} i \\
\mathrm{wvc}(\{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(I = e, P) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(c_1; c_2, P) &\stackrel{def}{=} \mathrm{wvc}(c_1, \mathrm{awp}(c_2, P)) \cup \mathrm{wvc}(c_2, P) \\
\mathrm{wvc}(\textbf{if } (b)\ c_0 \ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\
\mathrm{wvc}(/{*}{*}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \{q \longrightarrow P\} \\
\mathrm{wvc}(\textbf{while } (b)\ /{*}{*}\ \textbf{inv}\ \ i\ {*}/\ c, P) &\stackrel{def}{=} \mathrm{wvc}(c, i) \cup \{i \wedge b \longrightarrow \mathrm{awp}(c, i)\} \cup \{i \wedge \neg b \longrightarrow P\}
\end{aligned}
$$

Allgemeine Regel bei Ersetzungen (Nur Arrays)

Wie sieht nun die allgemeine Regel aus für

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

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.

Vereinfacht mit Modellbildung

- $\mathrm{seq}(a, n)$ ist ein Feld der Länge n repräsentiert als Liste (Sequenz)

$$n < 0 \longrightarrow \mathrm{seq}(a, n) = []$$
$$n \geq 0 \longrightarrow \mathrm{seq}(a, n) = \mathrm{seq}(a, n-1) {+\!\!+} (a[n] : [])$$

- Aktionen auf Sequenzen:
  - $a : as$, $[]$ — Listenkonstruktoren
  - $\mathrm{rev}(a)$ — Reverse
  - $a[i : j]$ — Slicing (à la Python)
  - $+\!\!+$ — Konkatenation
  - $[n]$ — Kurzform für $n : []$

Interaktion mit der Substitution

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

$$
\begin{aligned}
\mathrm{set}([], i, v) &== [] \\
\mathrm{set}(a : as, 0, v) &== v : as \\
i > 0 \longrightarrow \mathrm{set}(a : as, i, v) &== a : \mathrm{set}(as, i-1, v) \\
i < 0 \longrightarrow \mathrm{set}(as, i, v) &== as
\end{aligned}
$$

- Damit ist

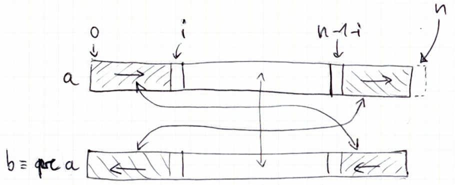
$$\mathrm{seq}(a, n)[v/a[i]] = \mathrm{set}(\mathrm{seq}(a, n), i, v)$$

- und es gilt

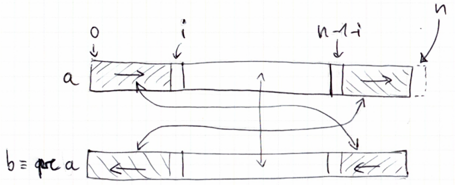
$$as = \mathrm{seq}(a, n) \wedge i \geq 0 \Longrightarrow \mathrm{set}(as, i, x) == as[0 : i]{+}{+}[x]{+}{+}as[i+1 : n]$$

Reverse-in-Place mit Listen

```
i= 0;
// {bs = seq(a, n)}
while (i< n/2)
  /** inv
  */ {
  tmp= a[n-1-i];
  a[n-i-1]= a[i];
  a[i]= tmp;
  i= i+1;
}
// {as = seq(a, n) ⟹ rev(as) = bs}
```



Reverse-in-Place mit Listen




```
i= 0;
// {bs = seq(a, n)}
while (i< n/2)
   /** inv   as = seq(a, n) ⟹
              rev(as[n − i : n])++as[i : n − i]++rev(as[0 : i]) = bs
   */ {
   tmp= a[n−1−i];
   a[n−i−1]= a[i];
   a[i]= tmp;
   i= i+1;
}
// {as = seq(a, n) ⟹ rev(as) = bs}
```

- Damit vereinfachte VCs und vereinfachter Beweis.

Reverse-in-Place mit Listen

```
i= 0;
// {bs = seq(a, n)}
while (i< n/2)
  /** inv  as = seq(a, n) ==> rev(as[n - i : n])++as[i : n - i]++ rev(as[0 : i]) = bs
  */ {
  // { as = seq(a, n)[0 : i]++[a[n - 1 - i]]++Seq(a, n)[i + 1 : n - 1 - i]++[a[i]]++ seq(a, n)[n - i : n] }
        ==> rev(as[n - (i + 1) : n])++as[(i + 1) : n - (i + 1)]++ rev(as[0 : (i + 1)]) = bs
  // {as = set(set(seq(a, n), i, a[n - 1 - i]), n - i - 1, a[i]) ==> rev(as[n - (i + 1) : n])++as[(i + 1) : n - (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
  tmp= a[n-1-i];
  // {as = set(set(seq(a, n), i, tmp), n - i - 1, a[i]) ==> rev(as[n - (i + 1) : n])++as[(i + 1) : n - (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
  a[n-i-1]= a[i];
  // {as = set(seq(a, n), i, tmp) ==> rev(as[n - (i + 1) : n])++as[(i + 1) : n - (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
  a[i]= tmp;
  // {as = seq(a, n) ==> rev(as[n - (i + 1) : n])++as[(i + 1) : n - (i + 1)]++ rev(as[0 : (i + 1)]) = bs}
  i= i+1;
  // {as = seq(a, n) ==> rev(as[n - i : n])++as[i : n - i]++ rev(as[0 : i]) = bs}
}
// {as = seq(a, n) ==> rev(as) = bs}
```

Arbeitsblatt 11.2: Beweise mit Listen

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

$$\mathrm{rev}(as{+\!+}bs) == \mathrm{rev}(bs){+\!+}\,\mathrm{rev}(as)$$

- Strukturelle Induktion heißt:
  1. Induktionsbasis: zeige Aussage für $as \stackrel{def}{=} []$.
  2. Induktionsschritt: Annahme der Aussage, zeige Aussage für $as \stackrel{def}{=} a : as$
- Beweis durch Umformung, Anwendung der Gleichungen für rev, $++$

$$\begin{aligned} \mathrm{rev}([]) &== [] \\ \mathrm{rev}(x : xs) &== \mathrm{rev}(xs){+\!+}[x] \\ ys{+\!+}[] &== ys \end{aligned}$$

$$\begin{aligned} (x : xs){+\!+}ys &== x : (xs{+\!+}ys) \\ as{+\!+}(bs{+\!+}cs) &== (as{+\!+}bs){+\!+}cs \end{aligned}$$

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 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

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

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

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** $n!, \sum_{i=1}^{n} i, \ldots$
  - Implikation, **Äquivalenzen**, Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v.\, b, \exists v.\, b$
- Formal:

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

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**~~ $\cancel{n!, \sum_{i=1}^{n} i, \ldots}$
  - Implikation, **Äquivalenzen**, Quantoren $b_1 \longrightarrow b_2, b_1 \longleftrightarrow b_2, \forall v.\, b, \exists v.\, b$
- Formal:

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

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:

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

Die bisherigen Funktionen

Die bisherigen Funktionen selbst definiert:

$$
\begin{aligned}
n! &== factorial(n) \\
i \leq 0 \longrightarrow factorial(i) &== 1 \\
i > 0 \longrightarrow factorial(i) &== i \cdot factorial(i-1) \\
\sum_{i=a}^{b} i &== sum(a, b) \\
a > b \longrightarrow sum(a, b) &== 0 \\
a <= b \longrightarrow sum(a, b) &== a + sum(a+1, b)
\end{aligned}
$$

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:

$$
\begin{aligned}
a \leq b &\longrightarrow [a\,..\,b] == a : [a+1\,..\,b] \\
a > b &\longrightarrow [a\,..\,b] == []
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{foldl}(f, c, a : as) &== \mathrm{foldl}(f, f(c, a), as) \\
\mathrm{foldl}(f, c, []) &== c
\end{aligned}
$$

$$
\begin{aligned}
\sum_{i=a}^{b} e(i) &== \mathrm{foldl}(\lambda x\ i\ .x + e(i), 0, [a\,..\,b]) \\
\prod_{i=a}^{b} e(i) &== \mathrm{foldl}(\lambda x\ i\ .x \cdot e(i), 0, [a\,..\,b])
\end{aligned}
$$

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)	×	×	*Haskell*

- 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*

Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand σ?
  - Auswertung (denotationale Semantik) ergibt *true*
- Variablen denotieren:
  - Zahlen und Characters wie bisher
  - Arrays wie in $\mathrm{seq}(a, n)$
  - Listen über Zahlen/Character wie in $\mathrm{rev}(as), as[(i+1) : n-(i+1)]$
  - Es könnten auch Strukturen sein (Datentypen wie in Haskell)
    - Sei **T** die Menge aller anderen Werte wie Listen, Strukturen usw.

Erfüllung von Zusicherungen

- Wann gilt eine Zusicherung $b \in \mathbf{Assn}$ in einem Zustand σ?
  - Auswertung (denotationale Semantik) ergibt *true*
- Variablen denotieren:
  - Zahlen und Characters wie bisher
  - Arrays wie in $\mathrm{seq}(a, n)$
  - Listen über Zahlen/Character wie in $\mathrm{rev}(as)$, $as[(i+1) : n-(i+1)]$
  - Es könnten auch Strukturen sein (Datentypen wie in Haskell)
    - Sei $\mathbf{T}$ die Menge aller anderen Werte wie Listen, Strukturen usw.
- **Belegung** der logischen Variablen: $I : \mathbf{Var} \rightarrow (\mathbf{Z} \cup \mathbf{C} \cup \mathbf{Array} \cup \mathbf{T})$
- Semantik von b unter der Belegung I: $[\![b]\!]^I_{\mathcal{B}v}, [\![a]\!]^I_{\mathcal{A}v}$

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

Erfüllung von Zusicherungen

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

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

Analog für andere Typen.

Erfülltheit von Zusicherungen

Erfülltheit von Zusicherungen

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

$$[\![b]\!]^I_{\mathcal{B}v}(\sigma) = \mathit{true}$$

Formeln ohne Programmvariablen, ohne Arrays, ohne Strukturen

- Eine Formel $b \in \textbf{Assn}$ 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 $[\![b]\!]^I_{\mathcal{B}v}(\sigma)$ von b unabhängig von I und σ.

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

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

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 I erfüllt $(\Gamma, \sigma \models^I b)$, gdw

$$[\![(\bigwedge \Gamma) \longrightarrow b]\!]^I_{\mathcal{B}v}(\sigma) = \mathit{true}$$

Floyd-Hoare-Tripel mit Kontext

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

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

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

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

Floyd-Hoare-Kalkül mit Kontext

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

Floyd-Hoare-Kalkül mit Kontext

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

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

Floyd-Hoare-Kalkül mit Kontext

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

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

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

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 σ und Belegungen I dass $\Gamma \longrightarrow (A' \longrightarrow A)$ wahr bzw. dass

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

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 σ und Belegungen I dass $\Gamma \longrightarrow (A' \longrightarrow A)$ wahr bzw. dass

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

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

$$\begin{aligned}[\![\forall_{\mathbf{Z}} v.b]\!]^{I}_{\mathcal{B}v} = {} & \{(\sigma, 1) \mid \text{für alle } i \in \mathbf{Z} \text{ gilt } (\sigma, 1) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\} \\ & \cup \{(\sigma, 0) \mid \text{für ein } i \in \mathbf{Z} \text{ gilt } (\sigma, 0) \in [\![b]\!]^{I[i/v]}_{\mathcal{B}v}\}\end{aligned}$$

- Unvollständigkeit der Prädikatenlogik

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