

Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 12.07.22
Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2022

13:38:36 2022-07-19

1 [45]



Fahrplan

- ▶ Einführung
- ▶ Operationale Semantik
- ▶ Denotationale Semantik
- ▶ Äquivalenz der Operationalen und Denotationalen Semantik
- ▶ Der Floyd-Hoare-Kalkül I
- ▶ Der Floyd-Hoare-Kalkül II: Invarianten
- ▶ Korrektheit des Floyd-Hoare-Kalküls
- ▶ Strukturierte Datentypen
- ▶ Verifikationsbedingungen
- ▶ Vorwärts mit Floyd und Hoare
- ▶ Funktionen und Prozeduren I
- ▶ Funktionen und Prozeduren II
- ▶ Referenzen und Speichermodelle
- ▶ Ausblick und Rückblick

Korrekte Software

2 [45]



Motivation

- ▶ Warum Referenzen?
 - ▶ Nötig für *call by reference*
 - ▶ Funktionen können sonst nur **globale** Seiteneffekte haben
 - ▶ Effizienz
- ▶ Kurze Begriffsklärung:
 - ▶ Referenzen: getypt, eingeschränkte Arithmetik
 - ▶ Zeiger: ungetypt, Zeigerarithmetik

Korrekte Software

3 [45]



Referenzen in C

- ▶ Pointer in C ("pointer type"):
 - ▶ Schwach getypt (**void *** kompatibel mit allen Zeigertypen, Typumwandlung)
 - ▶ Eingeschränkte Zeigerarithmetik (Addition, Subtraktion)
 - ▶ Felder werden durch Zeigerarithmetik implementiert
- ▶ Pointer sind *first-class-values*
- ▶ C-Standard läßt das Speichermodell relativ offen
 - ▶ Repräsentation von Objekten

Korrekte Software

4 [45]



Referenzen in anderen Sprachen

- ▶ Java:
 - ▶ (Fast) alles ist eine Referenz
 - ▶ Schwach getypt (Subtyping und Typumwandlung)
- ▶ Haskell, SML, OCaml:
 - ▶ Stark getypt (typsicher)
- ▶ Scriptsprachen (Python, Ruby):
 - ▶ Ähnlich Java

Korrekte Software

5 [45]



Ausdrücke

- ▶ Neue Operatoren: Addressoperator (&) und Dereferenzierung (*)
 - Lexp** $l ::= \text{Idt} \mid l[a] \mid !. \text{Idt} \mid *a$
 - Aexp** $a ::= \text{Z} \mid \text{C} \mid \text{Lexp} \mid \&l \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$
 - Bexp** $b ::= \dots$
 - Exp** $e ::= \text{Aexp} \mid \text{Bexp}$
 - Stmt** $c ::= \dots$
 - Type** $t ::= \text{void} \mid \text{char} \mid \text{int} \mid *t \mid \text{struct Idt}^? \{ \text{Decl}^+ \} \mid t \text{ Idt}[a]$

Korrekte Software

6 [45]



Das Problem mit Zeigern

- ▶ **Aliasing**: Verschiedene Bezeichner (**Lexp**) für die gleiche Lokation $l \in \text{Loc}$

```
int a;  
int *x;  
  
x = &a;  
a = 0;  
// {a = 0}  
*x = 7;  
// {a = 7} (*)
```

- ▶ Wert von a ändert sich **ohne dass a erwähnt** wird.
- ▶ An der Stelle (*) zwei Bezeichner für die gleiche Lokation: a und *x
- ▶ Großes Problem für Semantik und Hoare-Kalkül.
- ▶ Modellierung der Zuweisung durch Substitution nicht mehr möglich

Korrekte Software

7 [45]



Erweiterung des Zustandsmodells

- ▶ Bisheriger Zustand $\Sigma \stackrel{\text{def}}{=} \text{Loc} \rightarrow \mathbf{V}$ mit
 - ▶ **Locations** (siehe Vorlesung 8)
 - ▶ Werte: $\mathbf{V} = \mathbb{Z}$
- ▶ Ansatz reicht nicht mehr: Werte müssen auch Locations sein:
 - $\mathbf{V} \stackrel{\text{def}}{=} \mathbb{Z} + \text{Loc}$
 - und somit sind Zustände $\Sigma \stackrel{\text{def}}{=} \text{Loc} \rightarrow (\mathbb{Z} + \text{Loc})$
- ▶ Was ist die Semantik von **Loc**? \rightarrow Speichermodelle

Korrekte Software

8 [45]

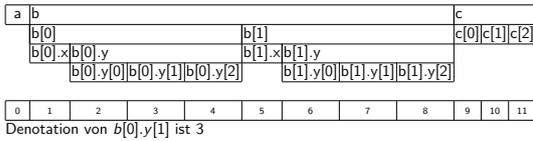


Speichermodelle I: Compiler

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in konkretes **Speicherlayout**:

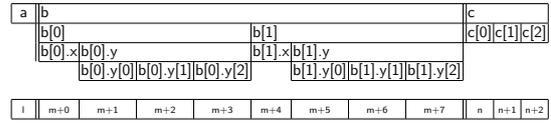


Speichermodelle II: C-Standard

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in abstraktes **Speicherlayout**:

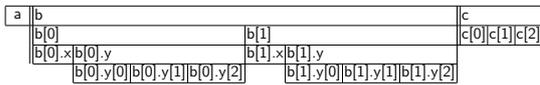


Speichermodelle III: Symbolisch

Beispieldeklarationen:

```
int a;
struct {
  int x;
  int y[3]} b[2];
int c[3];
```

Übersetzung in **symbolische Adressen**:



Speichermodelle im Überblick

- Speichermodell I:
 - Ausführbar, aber **spezifisch** für Compiler und **Architektur**
 - Uneingeschränkte **Zeigerarithmetik**: $\text{Loc} = \mathbb{N}$
- Speichermodell II:
 - Spezifiziert durch den **Standard**
 - **Eingeschränkte** Form der **Zeigerarithmetik** $p + n$ mit p Zeiger, $n \in \mathbb{N}$
- Speichermodell III:
 - **Mischung** von Syntax und Semantik: $a.x$ kann L-Ausdruck, aber auch Location bezeichnen.

Arbeitsblatt 13.1: Jetzt mit Zeigern!

Hier eine weitere Folge von Deklarationen:

```
int *a[1];
struct {
  int p[2];
  struct {
    int x;
    int y;} q[2];
} b;
```

- Skizziert hier das Speichermodell — konkret, abstrakt, symbolisch.
- Welches sind die jeweiligen Adressen (**Loc**)?
- Was sind die Denotationen für $a[1]$, $b.p[1]$, $b.q[0]$, $b.q[1].y$?
- Welche davon sind definiert/undefiniert?

Arbeitsblatt 13.2: Jetzt mit noch mehr Zeigern!

Hier eine weitere Folge von Deklarationen:

```
int *a[1];
struct {
  int p[2];
  struct {
    int x;
    int y;} *q[2];
} b;
```

- Skizziert hier das Speichermodell — konkret, abstrakt, symbolisch.
- Welches sind die jeweiligen Adressen (**Loc**)?
- Was sind die Denotationen für $a[1]$, $b.p[1]$, $b.q[0]$, $(*b.q[1]).y$?
- Welche davon sind definiert/undefiniert?

Erweiterung der Semantik

- Problem: **Loc** haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
 - $x = x+1$ — Links: Adresse der Variablen, rechts: Wert an dieser Adresse
 - Einführung von Dereferenzierung und Adressoperator verschärft dieses Problem
- Lösung in C: "Except when it is (...) the operand of the unary & operator, the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)" *C99 Standard*, §6.3.2.1 (2)
- Nicht spezifisch für C

Erweiterung der Semantik: Lexp

$$[-]_{\mathcal{L}} : \text{Env} \rightarrow \text{Lexp} \rightarrow \Sigma \rightarrow \text{Loc}$$

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

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

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

$$[*e]_{\mathcal{L}} = [e]_{\mathcal{L}}$$

Erweiterung der Semantik: Aexp(1)

$$\llbracket - \rrbracket_A : \text{Env} \rightarrow \text{Aexp} \rightarrow \Sigma \rightarrow \mathbf{V}$$

$$\llbracket n \rrbracket_A = \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbb{N}$$

$$\llbracket e \rrbracket_A = \{(\sigma, v \mid (\sigma, l) \in \llbracket e \rrbracket_C, (l, v) \in \sigma)\} \quad e \in \text{Lexp und } e \text{ kein Array-Typ}$$

$$\llbracket e \rrbracket_A = \{(\sigma, l) \mid (\sigma, l) \in \llbracket e \rrbracket_C\} \quad e \in \text{Lexp und } e \text{ ist Array-Typ}$$

$$\llbracket \&e \rrbracket_A = \{(\sigma, l) \mid (\sigma, l) \in \llbracket e \rrbracket_C\} \quad \llbracket e \rrbracket_C$$

- Es ist wichtig, die semantischen Funktionen $\llbracket e \rrbracket_A$ und $\llbracket e \rrbracket_C$ zu unterscheiden!

Erweiterung der Semantik: Aexp(2)

$$\llbracket - \rrbracket_A : \text{Env} \rightarrow \text{Aexp} \rightarrow \Sigma \rightarrow \mathbf{V}$$

$$\llbracket a_0 + a_1 \rrbracket_A = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_A\}$$

$$\llbracket a_0 - a_1 \rrbracket_A = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_A\}$$

$$\llbracket a_0 * a_1 \rrbracket_A = \{(\sigma, n_0 \cdot n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_A\}$$

$$\llbracket a_0 / a_1 \rrbracket_A = \{(\sigma, n_0 \div n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_A \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_A \wedge n_1 \neq 0\}$$

Beispiel

```
int a, *x;
*x = 5 * a;
```

Gegeben folgende Werte, was ist die Semantik?

$$\Gamma \stackrel{\text{def}}{=} \langle a \mapsto h_1, x \mapsto h_2 \rangle$$

$$\sigma_1 \stackrel{\text{def}}{=} \langle h_1 \mapsto 10, h_2 \mapsto h_1 \rangle$$

$$\begin{aligned} \llbracket *x = 5 * a \rrbracket_C(\sigma_1) &= \sigma_1[\llbracket *x \rrbracket_C(\sigma_1) \mapsto \sigma_1[\llbracket 5 * a \rrbracket_A(\sigma_1)]] \\ &= \sigma_1[\llbracket x \rrbracket_A(\sigma_1) \mapsto \sigma_1[\llbracket 5 \rrbracket_A(\sigma_1) \cdot \llbracket a \rrbracket_A(\sigma_1)]] \\ &= \sigma_1[\sigma_1[\llbracket x \rrbracket_C(\sigma_1)] \mapsto 5 \cdot \sigma_1[\llbracket a \rrbracket_C(\sigma_1)]] \\ &= \sigma_1[\sigma_1(h_2) \mapsto 5 \cdot \sigma_1(h_1)] \\ &= \sigma_1[h_1 \mapsto 5 \cdot 10] = \langle h_1 \mapsto 50, h_2 \mapsto h_1 \rangle \end{aligned}$$

Arbeitsblatt 13.3: Pop-Quiz

Gegeben folgende Funktionen:

```
int f(int *x)
{
  int a;
  a = *x;
  *x = a + 1;
  return a;
}
```

```
int a[3] = {0, 0, 0};
void g()
{
  int x = 1;
  a[x] = f(&x);
}
```

Was ist der Wert des Feldes a am Ende von g?

- 1 a == {0, 0, 1}
- 2 a == {0, 0, 2}
- 3 a == {0, 1, 0}
- 4 a == {0, 2, 0}

Korrektur der Semantik

- Das Pop-Quiz demonstriert den **Nichtdeterminismus** der C-Semantik.
 - Es ist **unbestimmt**, ob die linke oder rechte Seite der Zuweisung zuerst ausgewertet wird.
- Unsere Teilsprache soll den **deterministischen** Teil von C umfassen.
- Deshalb beim Funktionsaufruf mit Zuweisung links nur **zustandsfreie** Ausdrücke l:

$$\forall \sigma, \sigma'. \llbracket l \rrbracket_C(\sigma) = \llbracket l \rrbracket_C(\sigma')$$

- Gilt insbesondere für Bezeichner $l \in \text{Idt}$, deshalb:

$$\text{Stmt } c ::= \dots \mid \text{Idt} = \text{Idt}(a^*) \mid \dots$$

- Bei anderen Zuweisungen $l = e$ sind beide Seiten **zustandsabhängig**, aber frei von **Seiteneffekten**.

Arbeitsblatt 13.4: Kurze Semantik

Gegeben folgende Deklarationen:

```
struct p { int x; int a; struct p *q;
          int y; } p;
```

mit folgender Umgebung und Zustand

$$\Gamma \stackrel{\text{def}}{=} \langle p \mapsto h_1, a \mapsto h_2, q \mapsto h_3, h_1 \neq h_2, \sigma_1 \stackrel{\text{def}}{=} \langle h_1.x \mapsto 3, h_1.y \mapsto 5, h_2 \mapsto 7, h_3 \mapsto h_1 \rangle$$

Berechnet die denotationale Semantik (erst für beliebige Zustände, dann für σ_1) von

$$\begin{aligned} \llbracket a = a + (*q).x \rrbracket_C &=? \\ \llbracket a = a + (*q).x \rrbracket_C(\sigma_1) &=? \end{aligned}$$

Und jetzt?

- Zustand erweitert, so dass wir Zeiger modellieren können.
- Semantik entsprechend erweitert.
- Was machen wir mit dem Hoare-Kalkül, speziell der **Zuweisung**?
- Vorherige Modellierung — Zuweisung durch Substitution modelliert — nicht mehr ausreichend.
- Daher: **explizite Zustandsprädikate**

Explizite Zustandsprädikate

- Zusicherungen (**Assn**) sind zustandsabhängige Prädikate
 - Mit anderen Worten, Prädikate über Programmvariablen.
- Axiomatische Beschreibung des Zustandes erforderte neue Modellierung auf der Ebene der Prädikate
- Explizite Zustandsprädikate modellieren das Lesen und die Änderung des Zustandes **explizit** mit Operationen read und upd.

Explizite Zustandsprädikate

- Enthalten keine * oder &, und unterscheiden **strikt** zwischen **Lexp** und **Aexp**.
- Erweiterung von **Aexpv** um **read**, neue Sorte **State** mit Operation **upd**:
 - Assn_s** $b ::= \mathbf{1} \mid \mathbf{0} \mid a_1 == a_2 \mid a_1 < a_2 \mid !b \mid b_1 \&\& b_2 \mid \dots$
 - Lexp_s** $l ::= \dots \mid *a$
 - Aexp_s** $a ::= \text{read}(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid f \mid \&\&f \mid \dots \mid e \text{ @pre} \mid \dots$
 - State** $S ::= s \mid r \mid \text{upd}(S, l, e)$
- Zustandsvariablen *StateVar*: aktueller Zustand *s*, Vorzustand *r*
- Im Gegensatz zur Semantik rechnen wir mit **symbolischen Namen**

Gleichungen für explizite Zustandsprädikate

- Für die Operationen **read**, **upd** gelten folgende Gleichungen:

$$\begin{aligned} \forall l, v, \sigma. \text{read}(\text{upd}(\sigma, l, v), l) &= v \\ \forall l, m, v, \sigma. l \neq m &\implies \text{read}(\text{upd}(\sigma, l, v), m) = \text{read}(\sigma, m) \\ \forall l, v, w, \sigma. \text{upd}(\text{upd}(\sigma, l, v), l, w) &= \text{upd}(\sigma, l, w) \\ \forall l, m, v, w, \sigma. l \neq m &\implies \text{upd}(\text{upd}(\sigma, l, v), m, w) = \text{upd}(\text{upd}(\sigma, m, w), l, v) \end{aligned}$$

- Diese Gleichungen sind **vollständig**.

Semantik expliziter Zustandsprädikate

- Semantik der expliziten Zustandsprädikate:

$$\begin{aligned} \llbracket \cdot \rrbracket_{Bsp} : \mathbf{Env} &\rightarrow \mathbf{Assn}_s \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbb{B} \\ \llbracket \cdot \rrbracket_{Aexp} : \mathbf{Env} &\rightarrow \mathbf{Aexp}_s \rightarrow (\Sigma \rightarrow \mathbf{V}) \\ \llbracket \cdot \rrbracket_{Lexp} : \mathbf{Env} &\rightarrow \mathbf{Lexp}_s \rightarrow (\Sigma \rightarrow \mathbf{Loc}) \end{aligned}$$

Hoare-Triple

- Floyd-Hoare-Tripel: $\Gamma \models \{P\} c \{Q \mid R\}$ mit $P, Q, R \in \mathbf{Assn}$ **Prädikate**
- Floyd-Hoare-Kalkül: $\Gamma \vdash \{P\} c \{Q \mid R\}$ mit $P, Q, R \in \mathbf{Assn}_s$ **explizite Zustandsprädikate**

- Wir brauchen also eine Übersetzung **Assn** nach **Assn_s** welche die Semantik erhält:

$$\begin{aligned} (-)^\dagger : \mathbf{Lexp} &\rightarrow \mathbf{Lexp}_s & (-)^\# : \mathbf{Aexpv} &\rightarrow \mathbf{Aexp}_s & (-)^* : \mathbf{Assn} &\rightarrow \mathbf{Assn}_s \\ \forall l \in \mathbf{Lexp}. \llbracket l \rrbracket_{L}^\dagger &= \llbracket l \rrbracket_{Lsp} & \forall a \in \mathbf{Aexpv}. \llbracket a \rrbracket_A^\# &= \llbracket a^\# \rrbracket_{Aexp} & \forall a \in \mathbf{Assn}. \llbracket a \rrbracket_B^* &= \llbracket a^* \rrbracket_{Bsp} \end{aligned}$$

Konversion in explizite Zustandsprädikaten

Alte Zuweisungsregel

$$\frac{}{\Gamma \vdash \{Q[e/x]\} x = e \{Q \mid R\}}$$

Umwandlung von *Q*, *x*, *e* in Zustandsprädikate:

- Eine **Lexp** *l* auf der rechten Seite *e* wird durch **read**(*σ*, *l*) ersetzt.
- &** dient lediglich dazu, diese Konversion zu **verhindern**.
- *** **erzwingt** diese Konversion, auch auf der linken Seite *x*.
- Beispiel: $*a = * \&b$;

Formal: Konversion in Zustandsprädikate

$$\begin{aligned} (-)^\dagger : \mathbf{Lexp} &\rightarrow \mathbf{Lexp}_s & (-)^\# : \mathbf{Aexp} &\rightarrow \mathbf{Aexp}_s & (-)^* : \mathbf{Assn} &\rightarrow \mathbf{Assn}_s \\ i^\dagger &= i \quad (i \in \mathbf{Idt}) & e^\# &= \text{read}(s, e^\dagger) \quad (e \in \mathbf{Lexp}) & (\forall x. b)^* &= \forall x. b^* \\ l.id^\dagger &= l^\dagger.id & n^\# &= n & (b_1 \wedge b_2)^* &= b_1^* \wedge b_2^* \\ l[e]^\dagger &= l^\dagger[e^\#] & v^\# &= v \quad (v \text{ logische Variable}) & (a_1 == a_2)^* &= a_1^\# = a_2^\# \\ *j^\dagger &= j^\# & (\&e)^\# &= e^\dagger & & \\ (e_1 + e_2)^\# &= e_1^\# + e_2^\# & \backslash \text{result}^\# &= \backslash \text{result} & & \\ e \text{ @pre}^\# &= e^\#[s/r] & & & & \end{aligned}$$

Angepasste Regeln des Hoare-Kalküls

$$\frac{}{\Gamma \vdash \{Q[\text{upd}(s, x^\dagger, e^\#)/s]\} x = e \{Q \mid R\}}$$

- Beispiel 1: $(**x == 3)^* = \text{read}(s, (**x)^\dagger) = 3$
 - $= \text{read}(s, x^\#) = 3$
 - $= \text{read}(s, \text{read}(s, x)) = 3$
- Beispiel 2: $\Gamma \vdash \{P\} x = \&a \{**x = 3 \mid R\}$
 - $P = (\text{read}(s, \text{read}(s, x)) = 3)[\text{upd}(s, x^\dagger, (\&a)^\#)/s]$
 - $= (\text{read}(s, \text{read}(s, x)) = 3)[\text{upd}(s, x, a^\dagger)/s]$
 - $= \text{read}(\text{upd}(s, x, a), \text{read}(\text{upd}(s, x, a), x)) = 3$
 - $= \text{read}(\text{upd}(s, x, a), a) = 3$
 - $= \text{read}(s, a) = 3$
 - $= (a == 3)^*$

Weitere geänderte Regeln

$$\frac{\frac{\Gamma \vdash \{Q[e^\#/\backslash \text{result}]\} \text{return } e \{P \mid Q\}}{\Gamma(f) = \forall x_1, \dots, x_n. (P, Q)}}{\Gamma \vdash \{P[t_i^\#/x_i] \wedge y_i, \text{ @pre} = y_i\} l = f(t_1, \dots, t_n) \{Q[t_i^\#/x_i][l^\#/\backslash \text{result}] \mid Q_R\}} \frac{\Gamma \vdash \{P^*\} c \{false \mid Q^*\}}{\Gamma \vdash f(x_1, \dots, x_n)/** \text{pre } P \text{ post } Q^* / \{ds \ c\}}$$

Rückwärtsrechnung mit Zeigern I

$$\begin{aligned} \text{awp}(\Gamma, f(x_1, \dots, x_n) / ** \text{pre } P \text{ post } Q * / \{ds \ c\}) &\stackrel{\text{def}}{=} \text{awp}(\Gamma, c, \text{false}, Q^*[x_i \text{ @pre } /x_i]) \\ \text{wvc}(\Gamma, f(x_1, \dots, x_n) / ** \text{pre } P \text{ post } Q * / \{ds \ c\}) &\stackrel{\text{def}}{=} \{P^* \wedge x_i = x_i \text{ @pre} \implies P'\} \\ &\quad \cup \text{wvc}(\Gamma, c, \text{false}, Q^*[x_i \text{ @pre } /x_i]) \\ P' &\stackrel{\text{def}}{=} \text{awp}(\Gamma, c, \text{false}, Q^*[x_i \text{ @pre } /x_i]) \end{aligned}$$

Sei $\Gamma(f) = \forall x_1, \dots, x_n. (P, Q)$

$$\begin{aligned} \text{awp}(\Gamma, / ** \text{const } R * / l = f(t_1, \dots, t_n), U, U_R) &\stackrel{\text{def}}{=} R^* \wedge P^*[t_i^* /x_i], l \notin \text{FV}(R) \\ \text{wvc}(\Gamma, / ** \text{const } R * / l = f(t_1, \dots, t_n), U, U_R) &\stackrel{\text{def}}{=} \{R^* \wedge Q[t_i^* /x_i] [l / \backslash \text{result}] \longrightarrow U\}, \\ &\quad l \notin \text{FV}(R) \end{aligned}$$

Approximative schwächste Vorbedingung mit Zeigern II

$$\begin{aligned} \text{awp}(\Gamma, \{ \}, Q, Q_R) &\stackrel{\text{def}}{=} Q \\ \text{awp}(\Gamma, l = e, Q, Q_R) &\stackrel{\text{def}}{=} Q[\text{upd}(s, l^\dagger, e^\#) /s] \\ \text{awp}(\Gamma, c_1; c_2, Q, Q_R) &\stackrel{\text{def}}{=} \text{awp}(\Gamma, c_1, \text{awp}(c_2, Q, Q_R), Q_R) \\ \text{awp}(\Gamma, \text{if } (b) \ c_0 \ \text{else } c_1, Q, Q_R) &\stackrel{\text{def}}{=} (b^* \wedge \text{awp}(\Gamma, c_0, Q, Q_R)) \vee (\neg b^* \wedge \text{awp}(\Gamma, c_1, Q, Q_R)) \\ \text{awp}(\Gamma, / ** \{q\} * /, Q, Q_R) &\stackrel{\text{def}}{=} q^* \\ \text{awp}(\Gamma, \text{while } (b) \ / ** \text{inv } i * / c, Q_R) &\stackrel{\text{def}}{=} i^* \\ \text{awp}(\Gamma, \text{return } e, Q, Q_R) &\stackrel{\text{def}}{=} Q_R[e^\# / \backslash \text{result}] \\ \text{awp}(\text{return}, Q, Q_R) &\stackrel{\text{def}}{=} Q_R \end{aligned}$$

Approximative schwächste Vorbedingung mit Zeigern III

$$\begin{aligned} \text{wvc}(\Gamma, \{ \}, Q, Q_R) &\stackrel{\text{def}}{=} \emptyset \\ \text{wvc}(\Gamma, l = e, Q, Q_R) &\stackrel{\text{def}}{=} \emptyset \\ \text{wvc}(\Gamma, c_1; c_2, Q, Q_R) &\stackrel{\text{def}}{=} \text{wvc}(\Gamma, c_1, \text{awp}(\Gamma, c_2, Q, Q_R), Q_R) \cup \text{wvc}(\Gamma, c_2, Q, Q_R) \\ \text{wvc}(\Gamma, \text{if } (b) \ c_1 \ \text{else } c_2, Q, Q_R) &\stackrel{\text{def}}{=} \text{wvc}(\Gamma, c_1, Q, Q_R) \cup \text{wvc}(\Gamma, c_2, Q, Q_R) \\ \text{wvc}(\Gamma, / ** \{q\} * /, Q, Q_R) &\stackrel{\text{def}}{=} \{q^* \implies Q\} \\ \text{wvc}(\Gamma, \text{while } (b) \ / ** \text{inv } i * / c, Q, Q_R) &\stackrel{\text{def}}{=} \text{wvc}(\Gamma, c, i^*, Q_R) \cup \{i^* \wedge b^* \implies \text{awp}(\Gamma, c, i^*, Q_R)\} \\ &\quad \cup \{i^* \wedge \neg b^* \implies Q\} \\ \text{wvc}(\Gamma, \text{return } e, Q, Q_R) &\stackrel{\text{def}}{=} \emptyset \end{aligned}$$

Arbeitsblatt 13.5: Ein kurzes Beispiel

Betrachtet folgendes Beispiel:

```
void foo(){
  int x, y, z;
  x= 1;
  z= x;
  y= x;
  z= 5;
  // {0 < y}
```

- 1 Konvertiert das Prädikat $0 < y$ in ein explizites Zustandsprädikat.
- 2 Berechnet (rückwärts) die jeweils gültigen Zwischenzustände.
- 3 Vereinfacht nach jedem Schritt die Zwischenzustände.

Aliasing

- Das Beispiel mit Aliasing vom Anfang:

```
void foo(){
  int a, *x;
  /** { 5 < 10 }
  /** { 5 < read(upd(upd(upd(s, x, a), a, 0),
  /** { 5 < read(upd(upd(upd(s, x, a), a, 0), read(upd(s, x, a), x), 10), a) } */
  x = &a;
  /** { 5 < read(upd(upd(s, a, 0), read( s, x), 10), a) } */
  /** { 5 < read(upd(upd(s, a, 0), read(upd(s, a, 0), x), 10), a) } */
  a = 0;
  /** { 5 < read(upd(s, read(s, x), 10), a) } */
  *x = 10;
  /** { 5 < read(s, a) } */
}
```

- Aliasing wird **korrekt** behandelt.

Arbeitsblatt 13.6: Ein problematisches Beispiel

```
void foo(int *p)
/** post \result == 7; */
{
  int x;

  x= 7;
  *p= 99;
  return x;
}
```

Spezifikation des Speicherlayout

- Generelles Problem — was ist mit

```
void foo(int *p, int *q)
{ ... }
```

- Deutet auf ein Problem hin
- Entspricht einem "dangling pointer" (d.h. Pointer mit un spezifiziertem Wert)
- Können weder beweisen, dass $*p = *q$ noch $*p \neq *q$
- Muss (in den Vorbedingungen) spezifiziert werden.
- Integration in die Logik: **separation logic**

Weitere Beispiele: Felder

```
int findmax(int a[], int a_len)
/** pre 0 < array(a, a_len) & 0 < a_len;
  post \forall i. 0 \le i < a_len \to a[i] \le \result ;
*/
{
  int x; int j;

  x= a[0]; j= 0;
  while (j < a_len)
  /** inv (\forall i. 0 \le i < j \to a[i] \le x & 0 \le j < a_len) */ {
    if (a[j] < a_len) {
      x= a[j];
    }
    j= j+1;
  }
  return x;
}
```

Felder und Zeiger revisited

- ▶ In C sind Zeiger und Felder schwach spezifiziert
- ▶ Insbesondere:
 - ▶ $a[j]= *(a+j)$ für a Array-Typ
 - ▶ Dereferenzierung von $**x$ nur definiert, wenn x "gültig" ist (d.h. auf ein Objekt zeigt) *C99 Standard*, §6.5.3.2(4)
- ▶ Bisher in den Hoare-Regeln ignoriert — **partielle** Korrektheit.
- ▶ Ist das sinnvoll? Nein, bekannte Fehlerquelle

Spezifikation von Zeigern und Feldern

Das Prädikat $\backslash\text{valid}(x)$

$\backslash\text{valid}(x) \iff \text{read}(\sigma, x^i)$ ist definiert

- ▶ Insbesondere: $\backslash\text{valid}(*x) \iff \text{read}(\sigma, \text{read}(\sigma, x))$ ist definiert.
- ▶ Felder als Parameter werden zu Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger ein Feld ist.
- ▶ $\backslash\text{array}(a, n)$ bedeutet: a ist ein Feld der Länge n , d.h.

$$\backslash\text{array}(a, n) \iff (\forall i. 0 \leq i < n \implies \backslash\text{valid}(a[i]))$$

- ▶ Gültigkeit kann abgeleitet werden:

$$\frac{x = \&e}{\backslash\text{valid}(*x)} \quad \frac{\backslash\text{array}(a, n) \quad 0 \leq i < n}{\backslash\text{valid}(a[i])}$$

Was noch fehlt. . .

- ▶ **Vorwärtsrechnung** mit expliziten Zustandsprädikaten.
- ▶ Statt Existenzquantoren über Variablenwerte **unbestimmte Zwischenzustände** ρ_1, ρ_2, \dots :

$$\frac{\rho_i \notin \text{FV}(P)}{\Gamma \vdash \{P\} x = e \{P[\rho_i/\sigma] \wedge \sigma = \text{upd}(\rho_i, x^i[\rho_i/\sigma], e^\#[\rho_i/\sigma]) \mid R\}}$$

- ▶ Zwischenzustände sind **existenzquantifiziert**, d.h. das Prädikat gilt für **irgendeinen** Zustand ρ_i (aber für alle σ).
- ▶ Schwächste **Vorbedingung** und stärkste **Nachbedingung**:
 - ▶ Ergibt sich aus den Hoare-Regeln.
 - ▶ Erfordert durchgängige und aggressive **Vereinfachung**.

Zusammenfassung

- ▶ Um Referenzen (Pointer) in C behandeln zu können, benötigen wir ein erweitertes **Zustandsmodell**
- ▶ Referenzen werden zu Werten wie Zahlen oder Zeichen.
 - ▶ Arrays und Strukturen sind **keine** first-class values.
 - ▶ Großes Problem: **aliasing**
- ▶ Erweiterung der Semantik und der Hoare-Tripel nötig:
 - ▶ Vor/Nachbedingungen werden zu **expliziten Zustandsprädikaten**.
 - ▶ Zuweisung wird zu **Zustandsupdate**.
 - ▶ Problem:
 - ▶ Zustände werden **sehr groß**
 - ▶ Rückwärtsrechnung erzeugt schnell sehr große „unbestimmte“ Zustände, die nicht vereinfacht werden können
 - ▶ Hier ist Vorwärtsrechnung vorteilhaft

Fahrplan

- ▶ Einführung
- ▶ Operationale Semantik
- ▶ Denotationale Semantik
- ▶ Äquivalenz der Operationalen und Denotationalen Semantik
- ▶ Der Floyd-Hoare-Kalkül I
- ▶ Der Floyd-Hoare-Kalkül II: Invarianten
- ▶ Korrektheit des Floyd-Hoare-Kalküls
- ▶ Strukturierte Datentypen
- ▶ Verifikationsbedingungen
- ▶ Vorwärts mit Floyd und Hoare
- ▶ Funktionen und Prozeduren I
- ▶ Funktionen und Prozeduren II
- ▶ **Referenzen und Speichermodelle**
- ▶ Ausblick und Rückblick