

Korrekte Software: Grundlagen und Methoden
Vorlesung 12 vom 09.07.20
Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2020

Prüfungstermine

- ▶ Mo, 20.07.2020: Präsenzprüfungen (9:00- 16:30, je zur vollen Stunde)
- ▶ Di, 21.07.2020: Onlineprüfungen (9:00- 13:00, alle 30 Minuten)
- ▶ Mo, 24.08.2020: Präsenzprüfungen (9:00- 16:30, je zur vollen Stunde)
- ▶ Di, 25.08.2020: Onlineprüfungen (9:00- 13:00, alle 30 Minuten)

Prüfungsmodalitäten

- ▶ Anmeldung über stud.ip.
- ▶ Präsenzprüfungen:
 - ▶ Im Raum 4380
 - ▶ Bitte **nicht vor dem Prüfungsraum versammeln** (sondern kurz vorher hochkommen)
 - ▶ **Wichtig:** Ausweispapiere mitbringen, unten am MZH ausweisen. Eingang ins MZH nur über die Ostseite (zur Enrique-Schmidt-Straße).
- ▶ Onlineprüfung:
 - ▶ Über Zoom, gleiche Meeting-Id wie gewohnt.
 - ▶ Wir lassen euch zur Prüfung in das Meeting, alle anderen bleiben draussen.
 - ▶ Eine Kamera ist **zwingend** erforderlich.
 - ▶ Die Prüfung muss in einem ruhigen Raum stattfinden. Es darf sich keine weitere Person im Raum befinden. Hilfsmittel sind nicht zugelassen.

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

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

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

Referenzen in anderen Sprachen

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

Ausdrücke

- ▶ Neue Operatoren: Addressoperator ($\&a$) und Derefenzierung ($*l$)

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

Aexp $a ::= \text{Z} \mid \text{C} \mid \text{Lexp} \mid \&l$
 $\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2 \mid \text{Idt}(\text{Exp}^*)$

Bexp $b ::= \dots$

Exp $e ::= \text{Aexp} \mid \text{Bexp}$

Stmt $c ::= \dots$

Type $t ::= \text{char} \mid \text{int} \mid *t \mid \text{struct Idt}^? \{ \text{Decl}^+ \} \mid t \text{ Idt}[a]$

Das Problem mit Zeigern

- ▶ Bisheriges Speichermodell: $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}$

- ▶ **Aliasing:**

Verschiedene Bezeichner (**Lexp**) für die gleiche Lokation $l \in \mathbf{Loc}$

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

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

Erweiterung des Zustandsmodells

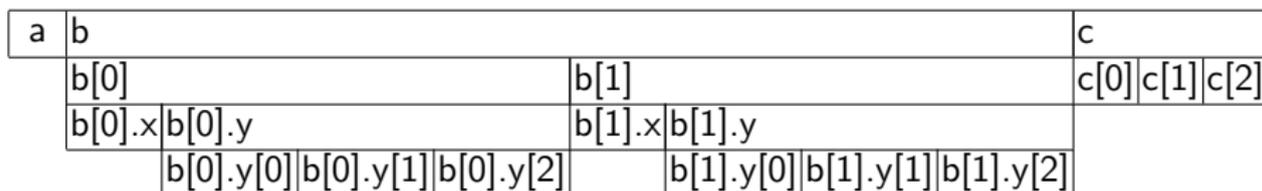
- ▶ Bisheriger Zustand $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightarrow \mathbf{V}$ mit
 - ▶ **Locations:** $\mathbf{Loc} ::= \mathbf{Idt} \mid \mathbf{Loc}[\mathbb{Z}] \mid \mathbf{Loc}.\mathbf{Idt}$
 - ▶ Werte: $\mathbf{V} = \mathbb{Z}$
- ▶ Ansatz reicht nicht mehr:
 - ❶ Werte müssen auch Locations sein: $\mathbf{V} \stackrel{def}{=} \mathbb{Z} + \mathbf{Loc}$
 - ❷ **Idt** als Location nicht ausreichend für Referenzen und Funktionen
- ▶ Man kann den Zustand **modellbasiert** oder **axiomatisch** beschreiben.

Speichermodelle I: Konkret (Compiler)

Beispieldeklarationen:

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

Übersetzung in konkretes **Speicherlayout**:



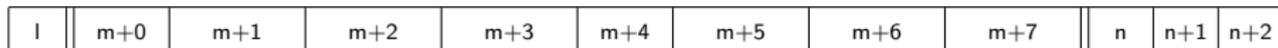
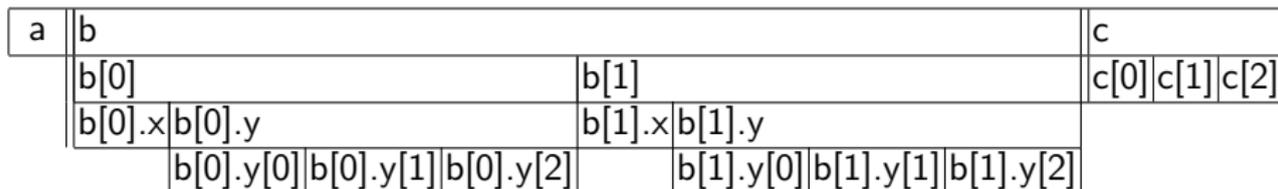
Denotation von $b[0].y[1]$ ist 3

Speichermodelle II: Abstrakt (C-Standard)

Beispieldeklarationen:

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

Übersetzung in abstraktes **Speicherlayout**:



Denotation von $b[0].y[1]$ ist $m + 3$, mit m **unbestimmte** Adresse

Speichermodelle III: Symbolisch

Beispieldeklarationen:

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

Übersetzung in symbolische Adressen:

a	b						c				
	b[0]			b[1]			c[0]	c[1]	c[2]		
	b[0].x		b[0].y		b[1].x		b[1].y				
	b[0].y[0]		b[0].y[1]		b[0].y[2]		b[1].y[0]		b[1].y[1]		b[1].y[2]

Denotation von $b[0].y[1]$ ist $m[0].y[1]$, mit m unbestimmte Adresse

Abstrakte Zeigerarithmetik

- ▶ Adressen sind ein abstrakter Datentyp **Loc** so dass:
 - ▶ Es gibt **unbestimmte** Adressen
 - ▶ Operation *off* addiert Offset (Feldzugriff)
 - ▶ Operation *fld* selektiert Feld (**struct**)
 - ▶ Problem: Gleichheit und Ungleichheit

$$\textit{off} : \mathbf{Loc} \rightarrow \mathbf{Z} \rightarrow \mathbf{Loc}$$

$$\textit{off}(l, 0) = l$$

$$\textit{off}(\textit{off}(l, a), b) = \textit{off}(l, a + b)$$

$$\textit{off}(l, a) = l \implies a = 0$$

$$\textit{off}(l, a) = \textit{off}(l, b) \implies a = b$$

$$\textit{fld} : \mathbf{Loc} \rightarrow \mathbf{Idt} \rightarrow \mathbf{Loc}$$

$$\textit{fld}(l, f) \neq l$$

$$\textit{fld}(l, f) = \textit{fld}(l, g) \implies f = g$$

$$\textit{fld}(l, f) = \textit{fld}(m, f) \implies l = m$$

$$f \neq g \implies \textit{fld}(l, f) \neq \textit{fld}(m, g)$$

Arbeitsblatt 12.1: 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 $a [1]$, $b.p [1]$, $(*b.q [0]).x$, $(*b.q [1]).y$?
- ▶ Welche davon sind definiert/undefiniert?

Axiomatisches Zustandsmodell

- ▶ Der Zustand ist ein abstrakter Datentyp Σ mit zwei Operationen und folgenden Gleichungen:

$$read : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{V}$$

$$upd : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{V} \rightarrow \Sigma$$

$$\mathbf{V} \stackrel{def}{=} \mathbb{Z} + \mathbf{Loc}$$

$$read(upd(\sigma, l, v), l) = v$$

$$l \neq m \implies read(upd(\sigma, l, v), m) = read(\sigma, m)$$

$$upd(upd(\sigma, l, v), l, w) = upd(\sigma, l, w)$$

$$l \neq m \implies upd(upd(\sigma, l, v), m, w) = upd(upd(\sigma, m, w), l, v)$$

- ▶ Diese Gleichungen sind **vollständig**.

Axiomatisches Speichermodell

- ▶ Es gibt einen **leeren** Speicher, und neue (“frische”) Adressen:

$$\text{empty} : \Sigma$$

$$\text{fresh} : \Sigma \rightarrow \mathbf{Loc}$$

$$\text{rem} : \Sigma \rightarrow \mathbf{Loc} \rightarrow \Sigma$$

- ▶ *fresh* modelliert **Allokation**, *rem* modelliert **Deallokation**
- ▶ *dom* beschreibt den **Definitionsbereich**:

$$\text{dom}(\sigma) = \{l \mid \exists v. \text{read}(\sigma, l) = v\}$$

$$\text{dom}(\text{empty}) = \emptyset$$

- ▶ Eigenschaften von *empty*, *fresh* und *rem*:

$$\text{fresh}(\sigma) \notin \text{dom}(\sigma)$$

$$\text{dom}(\text{rem}(\sigma, l)) = \text{dom}(\sigma) \setminus \{l\}$$

$$l \neq m \implies \text{read}(\text{rem}(\sigma, l), m) = \text{read}(\sigma, m)$$

Erweiterung der Semantik: Umgebung

- ▶ Für Funktionen brauchen wir eine **Umgebung** (Environment):

$$\begin{aligned}\mathbf{Env} &= \mathbf{Idt} \rightarrow \llbracket \mathbf{FunDef} \rrbracket \\ &= \mathbf{Idt} \rightarrow \mathbf{V}^N \rightarrow \Sigma \rightarrow (\Sigma \times \mathbf{V}_u)\end{aligned}$$

- ▶ Diese muss erweitert werden für Variablen:

$$\mathbf{Env} = \mathbf{Idt} \rightarrow (\llbracket \mathbf{FunDef} \rrbracket \uplus \mathbf{Loc})$$

- ▶ Insbesondere: gleicher Namensraum für Funktionen und Variablen (*C99 Standard*, §6.2.3)

Kurze Frage

- ▶ Wieso modellieren wir **Loc** nicht als Datentyp (so wie bisher):

$$l ::= \mathbf{ldt} \mid l[\mathbf{Z}] \mid l.\mathbf{ldt}$$

Dann wäre $\text{off}(l, n) \stackrel{\text{def}}{=} l[n]$, $\text{fld}(l, i) \stackrel{\text{def}}{=} l.i$.

Kurze Frage

- ▶ Wieso modellieren wir **Loc** nicht als Datentyp (so wie bisher):

$$l ::= \text{ldt} \mid l[\mathbf{Z}] \mid l.\text{ldt}$$

Dann wäre $\text{off}(l, n) \stackrel{\text{def}}{=} l[n]$, $\text{fld}(l, i) \stackrel{\text{def}}{=} l.i$.

- ▶ $\llbracket a \rrbracket$ wäre immer a . Damit funktionieren drei Dinge nicht:

- ① Wir können globale nicht von lokale Variablen unterscheiden.
- ② Beim rekursiven Aufruf wird keine neue Instanz erzeugt.
- ③ Generell funktioniert call-by-reference nicht, z.B.

```
void f(int *x)
{
  int a;
  a = *x;
}
```

```
void g()
{
  int a;
  f(&a);
}
```

Erweiterung der Semantik: Problem

- ▶ Problem: **L**oc haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
 - ▶ $x = x+1$ — Links: Adresse der Variablen, rechts: Wert an dieser Adresse
- ▶ 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

$$\llbracket - \rrbracket_{\mathcal{L}} : \mathbf{Env} \rightarrow \mathbf{Lexp} \rightarrow \Sigma \rightarrow \mathbf{Loc}$$

$$\llbracket x \rrbracket_{\mathcal{L}} \Gamma = \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\}$$

$$\llbracket \text{lexp}[a] \rrbracket_{\mathcal{L}} \Gamma = \{(\sigma, \text{off}(l, i)) \mid (\sigma, l) \in \llbracket \text{lexp} \rrbracket_{\mathcal{L}} \Gamma, (\sigma, i) \in \llbracket a \rrbracket_{\mathcal{A}} \Gamma\}$$

$$\llbracket \text{lexp}.f \rrbracket_{\mathcal{L}} \Gamma = \{(\sigma, \text{fld}(l, f)) \mid (\sigma, l) \in \llbracket \text{lexp} \rrbracket_{\mathcal{L}} \Gamma\}$$

$$\llbracket *e \rrbracket_{\mathcal{L}} \Gamma = \llbracket e \rrbracket_{\mathcal{A}} \Gamma$$

Erweiterung der Semantik: Aexp(1)

$$\llbracket - \rrbracket_{\mathcal{A}} : \mathbf{Env} \rightarrow \mathbf{Aexp} \rightarrow \Sigma \rightarrow \mathbf{V}$$

$$\llbracket n \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbb{N}$$

$$\llbracket e \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, \text{read}(\sigma, l)) \mid (\sigma, l) \in \llbracket e \rrbracket_{\mathcal{L}} \Gamma\}$$

$e \in \mathbf{Lexp}$ und e kein Array-Typ

$$\llbracket e \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, l) \mid (\sigma, l) \in \llbracket e \rrbracket_{\mathcal{L}} \Gamma\}$$

$e \in \mathbf{Lexp}$ und e ist Array-Typ

$$\llbracket \&e \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, l) \mid (\sigma, l) \in \llbracket e \rrbracket_{\mathcal{L}} \Gamma\}$$

Erweiterung der Semantik: Aexp(2)

$$\llbracket - \rrbracket_{\mathcal{A}} : \mathbf{Env} \rightarrow \mathbf{Aexp} \rightarrow \Sigma \rightarrow \mathbf{V}$$

$$\llbracket a_0 + a_1 \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_{\mathcal{A}} \Gamma \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_{\mathcal{A}} \Gamma\}$$

$$\llbracket a_0 - a_1 \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_{\mathcal{A}} \Gamma \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_{\mathcal{A}} \Gamma\}$$

$$\llbracket a_0 * a_1 \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_{\mathcal{A}} \Gamma \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_{\mathcal{A}} \Gamma\}$$

$$\llbracket a_0 / a_1 \rrbracket_{\mathcal{A}} \Gamma = \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in \llbracket a_0 \rrbracket_{\mathcal{A}} \Gamma \wedge (\sigma, n_1) \in \llbracket a_1 \rrbracket_{\mathcal{A}} \Gamma \\ \wedge n_1 \neq 0\}$$

Erweiterung der Semantik: Stmt

$$\llbracket x = e \rrbracket_{\mathcal{L}} \Gamma = \{ (\sigma, \text{upd}(\sigma, l, a)) \mid (\sigma, l) \in \llbracket x \rrbracket_{\mathcal{L}} \Gamma \wedge (\sigma, a) \in \llbracket e \rrbracket_{\mathcal{A}} \Gamma \}$$

$$\llbracket x = f(t_1, \dots, t_n) \rrbracket_{\mathcal{L}} \Gamma = \{ (\sigma, \text{upd}(\sigma', l, v)) \mid (\sigma, (\sigma', v)) \in \Gamma(f)(v_1, \dots, v_n) \wedge (\sigma, v_i) \in \llbracket t_i \rrbracket_{\mathcal{A}} \Gamma \wedge (\sigma, l) \in \llbracket x \rrbracket_{\mathcal{L}} \Gamma \}$$

Arbeitsblatt 12.2: 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}`

Arbeitsblatt 12.3: Kurze Semantik

Gegeben folgende Deklarationen:

```
struct {  
    int x;  
    int y; } p[5];  
int a;
```

mit folgender Umgebung

$$\Gamma \stackrel{\text{def}}{=} \langle p \mapsto l_1, a \mapsto l_2 \rangle, l_1 \neq l_2$$

Berechnet die denotationale Semantik von

```
a = a + p[3].x;
```

Und jetzt?

- ▶ Zustand erweitert, so dass wir Zeiger modellieren können.
- ▶ Semantik entsprechend erweitert.
- ▶ Was machen wir mit dem Hoare-Kalkül, speziell der **Zuweisung**?

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 erfordert neue Modellierung auf der Ebene der Prädikate
- ▶ Explizite Zustandsprädikate modellieren die Zustandsoperationen *read* und *upd* **explizit**

Explizite Zustandsprädikate

- ▶ Erweiterung von **Aexpv** um *read*, neue Sorte **State** mit Operation *upd*:

Lexp_s $l ::= \dots \mid *a$

Assn_s $b ::= \dots$

Aexp_s $a ::= \text{read}(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid l \mid \&t \mid \dots \mid \backslash\text{old}(e) \mid \dots$

State $S ::= \text{StateVar} \mid \text{upd}(S, l, e)$

- ▶ Zustandsvariablen *StateVar*:
 - ▶ Aktueller Zustand σ , Vorzustand ρ_{old} , Zwischenzustände $\rho_0, \rho_1, \rho_2, \dots$
 - ▶ Explizite Zustandsprädikate enthalten kein $*$ oder $\&$
 - ▶ Im Gegensatz zur Semantik rechnen wir mit **symbolischen Namen**
 - ▶ Damit Semantik:

$$\mathcal{B}_{sp}[\cdot] : \mathbf{Env} \rightarrow \mathbf{Assn}_s \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbb{B}$$

$$\mathcal{A}_{sp}[\cdot] : \mathbf{Env} \rightarrow \mathbf{Aexp}_s \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbf{V}$$

Hoare-Triple

$$\Gamma \models \{P\} c \{Q|R\}$$

- ▶ $P, Q, R \in \mathbf{Assn}_s$ sind **explizite Zustandsprädikate**
- ▶ Deklarationen (**Decl**) allozieren für jede Variable eine Location (*fresh*), und ordnen diese in Γ dem Namen zu.
- ▶ Gültigkeit von Hoare-Tripeln (partielle, totale Korrektheit) wie vorher

Floyd-Hoare-Kalkül

Alte Regel

$$\frac{}{\Gamma \vdash \{Q[\text{upd}(\sigma, x, e)/\sigma]\} x = e \{Q|R\}}$$

- ▶ Ein **Lexp** l auf der rechten Seite e wird durch $\text{read}(\sigma, l)$ ersetzt.¹
- ▶ $\&$ dient lediglich dazu, diese Konversion zu **verhindern**.
- ▶ $*$ **erzwingt** diese Konversion, auch auf der linken Seite x .
- ▶ Beispiel: $*a = *\&b;$

¹Außer l ist ein Array-Typ.

Formal: Konversion in Zustandsprädikate

$$(-)^\dagger : \mathbf{Lexp} \rightarrow \mathbf{Lexp}_s$$

$$i^\dagger = i \quad (i \in \mathbf{Idt})$$

$$l.id^\dagger = l^\dagger.id$$

$$l[e]^\dagger = l^\dagger[e^\#]$$

$$*l^\dagger = l^\#$$

$$(-)^\# : \mathbf{Aexp} \rightarrow \mathbf{Aexp}_s$$

$$e^\# = \text{read}(\sigma, e^\dagger) \quad (e \in \mathbf{Lexp})$$

$$n^\# = n$$

$$v^\# = v \quad (v \text{ logische Variable})$$

$$\&e^\# = e^\dagger$$

$$e_1 + e_2^\# = e_1^\# + e_2^\#$$

$$\backslash \mathbf{result}^\# = \backslash \mathbf{result}$$

$$\backslash \mathbf{old}(e)^\# = \backslash \mathbf{old}(e)$$

Angepasste Regeln des Hoare-Kalküls

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

$$\frac{}{\Gamma \vdash \{Q[e^\#/\backslash\text{result}]\} \text{return } e \{P|Q\}}$$

Arbeitsblatt 12.4: 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.

Ein Beispiel mit Zeigern

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

Ein Beispiel mit Zeigern

```
void foo(){
  int x, y, *z;

  /** { 0< 5 } */
  /** { 0< read(upd(... , x , 5), x) } */
  /** { 0< read(upd(upd(upd(s, z, x), x, 0), x , 5), x) } */
  /** { 0< read(upd(upd(upd(s, z, x), x, 0), read(upd(s, z, x), z), 5), x) } */
  z = &x;
  /** { 0< read(upd(upd(s, x, 0), read(s, z), 5), x) } */
  /** { 0< read(upd(upd(s, x, 0), read(s, z), 5), x) } */
  /** { 0< read(upd(upd(s, x, 0), read(upd(s, x, 0), z), 5), x) } */
  x = 0;
  /** { 0< read(upd(s, read(s, z), 5), x) } */
  *z = 5;
  /** { 0< read(s, x) } */
  /** { 0< read(s, x) } */
  /** { 0< read(upd(s, y, read(s, x), y) } */
  y = x;
  /** { 0< read(s, y) } */
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
    int x;

    //
    //
    x= 7;
    //
    *p= 99;
    //
    // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
    int x;

    //
    //
    x= 7;
    //
    *p= 99;
    // {read(s, x) = 7}
    // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
  int x;

  //
  //
  x= 7;
  // {read(upd(s, read(s, p), 99), x) = 7}
  *p= 99;
  // {read(s, x) = 7}
  // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
  int x;

  //
  // {read(upd(upd(s, x, 7), read(upd(s, x, 7), p), 99), x) = 7}
  x= 7;
  // {read(upd(s, read(s, p), 99), x) = 7}
  *p= 99;
  // {read(s, x) = 7}
  // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
  int x;

  // {read(upd(upd(s, x, 7), read(s, p), 99), x) = 7}
  // {read(upd(upd(s, x, 7), read(upd(s, x, 7), p), 99), x) = 7}
  x = 7;
  // {read(upd(s, read(s, p), 99), x) = 7}
  *p = 99;
  // {read(s, x) = 7}
  // {x = 7}
}
```

Ein problematisches Beispiel

```
void foo(int *p)
{
    int x;

    // {read(upd(upd(s, x, 7), read(s, p), 99), x) = 7}
    // {read(upd(upd(s, x, 7), read(upd(s, x, 7), p), 99), x) = 7}
    x = 7;
    // {read(upd(s, read(s, p), 99), x) = 7}
    *p = 99;
    // {read(s, x) = 7}
    // {x = 7}
}
```

- ▶ Können **weder** beweisen, dass $read(s, p) = x$ **noch** $read(s, p) \neq x$
- ▶ Erfordert Spezifikation: wenn $*p$ auf ein **gültiges** Objekt zeigt, dann $*p \neq x$ da x **lokale** Variable.
- ▶ Generelles Problem — was ist mit

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

- ▶ Können weder beweisen, dass $*p = *q$ noch $*p \neq *q$

Weitere Beispiele: Felder

```
int findmax(int a[], int a_len)
  /** pre \array(a, a_len)  $\wedge$   $0 < a\_len$ ; */
  /** post  $\forall i. 0 \leq i < a\_len \rightarrow a[i] \leq \text{result}$ ; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j < a_len)
    /** inv  $(\forall i. 0 \leq i < j \rightarrow a[i] \leq x) \wedge j \leq a\_len$ ; */
    {
      if (a[j] > x) 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 $\backslash\text{valid}(x)$

$\backslash\text{valid}(x) \iff \text{read}(\sigma, x^\dagger)$ 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:

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

Was noch fehlt...

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

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

- ▶ 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