Korrekte Software: Grundlagen und Methoden

Vorlesung 13 vom 06.07.21

Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2021

Prüfungstermine

- Fr, 16.07.2021: Onlineprüfungen (9:00- 16:00, alle 30 Minuten)
- Di, 27.07.2021: Onlineprüfungen (9:00- 12:00, alle 30 Minuten)
- Mi, 28.07.2021: Onlineprüfungen (10:00- 17:00, alle 30 Minuten)
- Do, 02.09.2021: Onlineprüfungen (9:00- 17:00, alle 30 Minuten)
- Fr, 03.09.2021: Onlineprüfungen (9:00- 17:00, alle 30 Minuten)

Prüfungsmodalitäten

- Anmeldung über stud.ip.
- Onlineprüfung:
  1. Für die Prüfung müsst ihr euch **zwingend** mit eurem **Uni-Bremen-Zoom-Account** anmelden, sonst kommt ihr nicht in den Zoom-Raum. Bitte ruhig zehn Minuten vor der Prüfung den Zoom-Raum betreten, ihr könnt dann im Warteraum warten.
  2. Haltet einen Lichtbildausweis (Studentenausweis, Perso, Führerschein o.ä.) bereit, um Eure Identität nachzuweisen.
  3. Die Kamera ist Pflicht, und bitte während der der gesamten Prüfung eingeschaltet zu lassen. (Ihr könnt gerne einen virtuellen Hintergrund wählen, wenn ihr wollt, aber ohne Kamera können wir keine Prüfung abnehmen.)
  4. Bitte sucht euch einen ruhigen Ort, ohne Hintergrundgeräusche (insbesondere Stimmen) im Hintergrund. Wir wollen hören können, ob ihr mit jemand anders redet.

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
- 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

Refenzen 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 (typsicher)
- Scriptsprachen (Python, Ruby):
  - Ähnlich Java

Ausdrücke

- Neue Operatoren: Addressoperator (&) und Derefenzierung ($*$)

$$
\begin{array}{rl}
\textbf{Lexp} & l ::= \textbf{Idt} \mid l[a] \mid l.\textbf{Idt} \mid *a \\
\textbf{Aexp} & a ::= \textbf{Z} \mid \textbf{C} \mid \textbf{Lexp} \mid \&l \\
 & \quad \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid \textbf{Idt}(\textbf{Exp}^*) \\
\textbf{Bexp} & b ::= \ldots \\
\textbf{Exp} & e := \textbf{Aexp} \mid \textbf{Bexp} \\
\textbf{Stmt} & c ::= \ldots \\
\textbf{Type} & t ::= \textbf{char} \mid \textbf{int} \mid *t \mid \textbf{struct}\ \ \textbf{Idt}^?\ \{\textbf{Decl}^+\} \mid t\ \textbf{Idt}[a]
\end{array}
$$

Das Problem mit Zeigern

- Bisheriges Speichermodell: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$
- **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 Zustandmodells

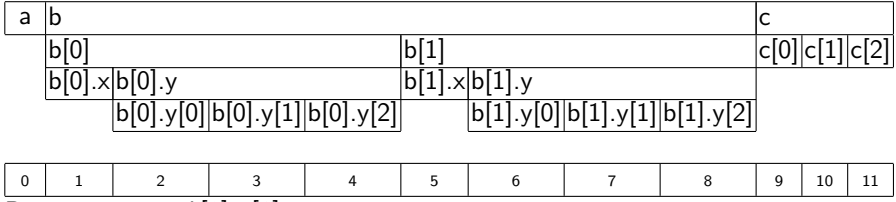
- Bisheriger Zustand $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbf{V}$ mit
  - **Locations:** $\mathbf{Loc} ::= \mathbf{Idt} \mid \mathbf{Loc}[\mathbb{Z}] \mid \mathbf{Loc}.\mathbf{Idt}$
  - Werte: $\mathbf{V} = \mathbb{Z}$
- Ansatz reicht nicht mehr:
  - (i) Werte müssen auch Locations sein: $\mathbf{V} \stackrel{def}{=} \mathbb{Z} + \mathbf{Loc}$
  - (ii) **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**:



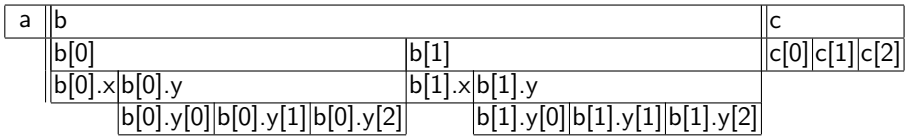
Denotation von $b[0].y[1]$ ist 3

Speichermodelle II: Abstrakt (C-Standard)

Beispieldeklarationen:

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

Übersetzung in abstraktes **Speicherlayout**:



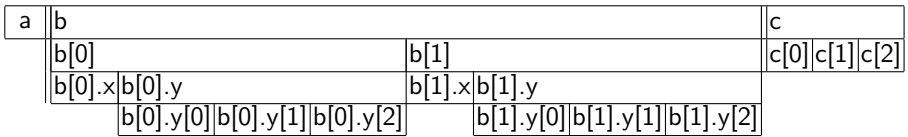
Denotation von $b[0].y[1]$ ist $m + 3$, mit m **unbestimmte** Adresse

Speichermodelle III: Symbolisch

Beispieldeklarationen:

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

Übersetzung in symbolische Adressen:



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

$$off : \mathbf{Loc} \to \mathbf{Z} \to \mathbf{Loc}$$

$$off(l, 0) = l$$
$$off(off(l, a), b) = off(l, a + b)$$
$$off(l, a) = l \Longrightarrow a = 0$$
$$off(l, a) = off(l, b) \Longrightarrow a = b$$

$$fld : \mathbf{Loc} \to \mathbf{Idt} \to \mathbf{Loc}$$

$$fld(l, f) \neq l$$
$$fld(l, f) = fld(l, g) \Longrightarrow f = g$$
$$fld(l, f) = fld(m, f) \Longrightarrow l = m$$
$$f \neq g \Longrightarrow fld(l, f) \neq fld(m, g)$$

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?

Axiomatisches Zustandsmodell

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

$$\begin{aligned} \mathit{read} &: \Sigma \to \mathbf{Loc} \rightharpoonup \mathbf{V} \\ \mathit{upd} &: \Sigma \to \mathbf{Loc} \to \mathbf{V} \rightharpoonup \Sigma \\ \mathbf{V} &\stackrel{\mathit{def}}{=} \mathbb{Z} + \mathbf{Loc} \end{aligned}$$

$$\begin{aligned} \mathit{read}(\mathit{upd}(\sigma, l, v), l) &= v \\ l \neq m \Longrightarrow \mathit{read}(\mathit{upd}(\sigma, l, v), m) &= \mathit{read}(\sigma, m) \\ \mathit{upd}(\mathit{upd}(\sigma, l, v), l, w) &= \mathit{upd}(\sigma, l, w) \\ l \neq m \Longrightarrow \mathit{upd}(\mathit{upd}(\sigma, l, v), m, w) &= \mathit{upd}(\mathit{upd}(\sigma, m, w), l, v) \end{aligned}$$

- Diese Gleichungen sind **vollständig.**

Axiomatisches Speichermodell

- Es gibt einen **leeren** Speicher, und neue ("frische") Adressen:

$$\begin{aligned} \mathit{empty} &: \Sigma \\ \mathit{fresh} &: \Sigma \to \mathbf{Loc} \\ \mathit{rem} &: \Sigma \to \mathbf{Loc} \to \Sigma \end{aligned}$$

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

$$\begin{aligned} \mathit{dom}(\sigma) &= \{l \mid \exists v.\, \mathit{read}(\sigma, l) = v\} \\ \mathit{dom}(\mathit{empty}) &= \emptyset \end{aligned}$$

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

$$\begin{aligned} \mathit{fresh}(\sigma) &\notin \mathit{dom}(\sigma) \\ \mathit{dom}(\mathit{rem}(\sigma, l)) &= \mathit{dom}(\sigma) \setminus \{l\} \\ l \neq m \implies \mathit{read}(\mathit{rem}(\sigma, l), m) &= \mathit{read}(\sigma, m) \end{aligned}$$

Erweiterung der Semantik: Umgebung

- Für Funktionen brauchten wir eine **Umgebung** (Environment):

$$\begin{aligned}\mathbf{Env} &= \mathbf{Idt} \rightharpoonup [\![\mathbf{FunDef}]\!] \\ &= \mathbf{Idt} \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)\end{aligned}$$

- Diese muss erweitert werden für Variablen:

$$\mathbf{Env} = \mathbf{Idt} \rightharpoonup ([\![\mathbf{FunDef}]\!] \uplus \mathbf{Loc})$$

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

Kurze Frage

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

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

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

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

- $[\![a]\!]$ wäre immer a. Damit funktionieren drei Dinge nicht:
  1. Wir können globale nicht von lokalen 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: **Loc** haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
  - `x = x+1` — Links: Addresse der Variablen, rechts: Wert an dieser Adresse
- Lösung in C: "Except when it is (...) the operand of the unary `&` oprerator, the left operand of the `.` operator or an assigment 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

$$[\![-]\!]_{\mathcal{L}} : \mathbf{Env} \to \mathbf{Lexp} \to \Sigma \rightharpoonup \mathbf{Loc}$$

$$\begin{aligned}
[\![x]\!]_{\mathcal{L}}\ \Gamma &= \{(\sigma, \Gamma(x)) \mid \sigma \in \Sigma\} \\
[\![lexp[a]]\!]_{\mathcal{L}}\ \Gamma &= \{(\sigma, \mathit{off}(l, i)) \mid (\sigma, l) \in [\![lexp]\!]_{\mathcal{L}}\ \Gamma, (\sigma, i) \in [\![a]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![lexp.f]\!]_{\mathcal{L}}\ \Gamma &= \{(\sigma, \mathit{fld}(l, f)) \mid (\sigma, l) \in [\![lexp]\!]_{\mathcal{L}}\ \Gamma\} \\
[\![*e]\!]_{\mathcal{L}}\ \Gamma &= [\![e]\!]_{\mathcal{A}}\ \Gamma
\end{aligned}$$

Erweiterung der Semantik: Aexp(1)

$$[\![-]\!]_{\mathcal{A}} : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
[\![n]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbb{N} \\
[\![e]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, \mathit{read}(\sigma, l)) \mid (\sigma, l) \in [\![e]\!]_{\mathcal{L}}\ \Gamma\} \\
&\qquad e \in \mathbf{Lexp} \text{ und } e \text{ kein Array-Typ} \\
[\![e]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in [\![e]\!]_{\mathcal{L}}\ \Gamma\} \\
&\qquad e \in \mathbf{Lexp} \text{ und } e \text{ ist Array-Typ} \\
[\![\&e]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in [\![e]\!]_{\mathcal{L}}\ \Gamma\}
\end{aligned}$$

Erweiterung der Semantik: Aexp(2)

$$[\![-]\!]_{\mathcal{A}} : \mathbf{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
[\![a_0 + a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 + n_1 \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![a_0 - a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![a_0 * a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma\} \\
[\![a_0 / a_1]\!]_{\mathcal{A}}\ \Gamma &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in [\![a_0]\!]_{\mathcal{A}}\ \Gamma\ \wedge (\sigma, n_1) \in [\![a_1]\!]_{\mathcal{A}}\ \Gamma \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ n_1 \neq 0\}
\end{aligned}$$

Erweiterung der Semantik: Stmt

$$[\![x = e]\!]_{\mathcal{C}}\ \Gamma = \{(\sigma, upd(\sigma, l, a)) \mid (\sigma, l) \in [\![x]\!]_{\mathcal{L}}\ \Gamma \\ \wedge (\sigma, a) \in [\![e]\!]_{\mathcal{A}}\ \Gamma\}$$

$$[\![x = f(t_1, \ldots, t_n)]\!]_{\mathcal{C}}\Gamma = \{(\sigma, upd(\sigma', l, v)) \mid (\sigma, (\sigma', v)) \in \Gamma(f)(v_1, \ldots, v_n)\ \} \\ \wedge (\sigma, v_i) \in [\![t_i]\!]_{\mathcal{A}}\Gamma \\ \wedge (\sigma, l) \in [\![x]\!]_{\mathcal{L}}\Gamma\}$$

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}

Arbeitsblatt 13.4: Kurze Semantik

Gegeben folgende Deklarationen:

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

mit folgender Umgebung

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

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

- Zusicherungenen (**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 die Zustandsoperationen *read* und *upd* **explizit**

Explizite Zustandsprädikate

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

$$\begin{aligned}
\mathbf{Lexp}_s\ & l ::= \ldots \mid *a \\
\mathbf{Assn}_s\ & b ::= \ldots \\
\mathbf{Aexp}_s\ & a ::= \mathit{read}(S, l) \mid \mathbf{Z} \mid \mathbf{C} \mid l \mid \&l \mid \ldots \mid e\ \mathbf{@pre} \mid \ldots \\
\mathbf{State}\ & S ::= \mathit{StateVar} \mid \mathit{upd}(S, l, e)
\end{aligned}$$

- Zustandsvariablen *StateVar*:
  - Aktueller Zustand σ, Vorzustand ρ_{old}, Zwischenzustände ρ_0, ρ_1, ρ_2, ...
- Explizite Zustandsprädikate enthalten kein $*$ oder &
- Im Gegensatz zur Semantik rechnen wir mit **symbolischen Namen**
- Damit Semantik:

$$[\![.]\!]_{\mathcal{B}sp} : \mathbf{Env} \to \mathbf{Assn}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbb{B}$$
$$[\![.]\!]_{\mathcal{A}sp} : \mathbf{Env} \to \mathbf{Aexp}_s \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}$$

Hoare-Triple

$$\Gamma \models \{P\}\, c\, \{Q \mid R\}$$

- $P, Q, R \in \mathbf{Assn}_s$ 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

$$\overline{\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, x, e)/\sigma]\ \{x = e \mid Q\}}$$

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

[1] Außer l ist ein Array-Typ.

Formal: Konversion in Zustandsprädikate

$$
\begin{aligned}
(-)^{\dagger} &: \mathbf{Lexp} \to \mathbf{Lexp}_s \\
i^{\dagger} &= i \quad (i \in \mathbf{Idt}) \\
I.id^{\dagger} &= I^{\dagger}.id \\
I[e]^{\dagger} &= I^{\dagger}[e^{\#}] \\
*I^{\dagger} &= I^{\#}
\end{aligned}
$$

$$
\begin{aligned}
(-)^{\#} &: \mathbf{Aexp} \to \mathbf{Aexp}_s \\
e^{\#} &= \mathit{read}(\sigma, e^{\dagger}) \quad (e \in \mathbf{Lexp}) \\
n^{\#} &= n \\
v^{\#} &= v \quad (v \text{ logische Variable}) \\
(\&e)^{\#} &= e^{\dagger} \\
(e_1 + e_2)^{\#} &= {e_1}^{\#} + {e_2}^{\#} \\
\backslash\mathbf{result}^{\#} &= \backslash\mathbf{result} \\
e\,\mathbf{@pre}^{\#} &= e\,\mathbf{@pre}
\end{aligned}
$$

Angepasste Regeln des Hoare-Kalküls

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

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

Angepasste Regeln des Hoare-Kalküls

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

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

Insbesondere gilt (**int** x, y, ∗z;):

- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, x, read(\sigma, y))/\sigma]\ \{x = y \mid Q\}$

Angepasste Regeln des Hoare-Kalküls

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

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

Insbesondere gilt (**int** x, y, ∗z;):

- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, x, read(\sigma, y))/\sigma]\ \{x = y \mid Q\}$
- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, y, x)/\sigma]\ \{y = \&x \mid Q\}$

Angepasste Regeln des Hoare-Kalküls

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

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

Insbesondere gilt (**int** x, y, ∗z;):

- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, x, read(\sigma, y))/\sigma]\ \{x = y \mid Q\}$
- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, y, x)/\sigma]\ \{y = \&x \mid Q\}$
- $\Gamma \vdash \{\Gamma\}\ Q[upd(\sigma, read(\sigma, z), read(\sigma, x))/\sigma]\ \{*z = x \mid Q\}$

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.

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) ∧ 0 < a_len; */
  /** post ∀i. 0 ≤ i < a_len ⟶ a[i] ≤ \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /** inv (∀i. 0 ≤ i < j ⟶ a[i] ≤ x) ∧ j ≤ 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
  - Derefenzierung 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 \valid(x)

$\backslash\mathbf{valid}(x) \iff \mathit{read}(\sigma, x^{\dagger})$ ist definiert

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

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

- Gültigkeit kann abgeleitet werden:

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

Was noch fehlt...

- **Vorwärtsrechung** mit expliziten Zustandsprädikaten.
  - Statt Existenzquantoren über Variablenwerte **unbestimmte Zwischenzustände** $\rho_1, \rho_2, \ldots$:

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

  - 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 agressive **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
- Modellierung
- Spezifikation von Funktionen
- Referenzen und Speichermodelle
- Ausblick und Rückblick