Korrekte Software: Grundlagen und Methoden

Vorlesung 12 vom 25.06.19

Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2019

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
- Funktionsaufrufe und das Framing-Problem
- Ausblick und Rückblick

Motivation

- Warum Referenzen?
  - Nötig für *call by reference*
  - Funktion 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:
  - Alles ist eine Referenz
  - Schwach getypt (Subtyping und Typumwandlung)
- Haskell, SML, OCaml:
  - Stark getypt (typsicher)
- Scriptsprachen (Python, Ruby):
  - Ähnlich Java

Ausdrücke

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

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

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.
- Großes Problem für Semantik und Hoare-Kalkül.

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** (wie bisher) oder **axiomatisch** beschreiben.

Axiomatisches Zustandsmodell

- Der Zustand ist ein abstrakter Datentyp Σ (und **Loc**) 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*:

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

Zeigerarithmetik

- Ziegerarithmetik: Rechnen mit Zeigern
  - Implementiert Felder und Strukturen
  - Wir betrachten keine **Differenz** von Zeigern

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

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

Erweiterung der Semantik

- 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

Umgebung

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

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

- Diese muss erweitert werden für Variablen:

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

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

Erweiterung der Semantik: Lexp

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

$$\begin{aligned}
\mathcal{L}[\![x]\!]\ \Gamma &= \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\} \\
\mathcal{L}[\![lexp[a]]\!]\ \Gamma &= \{(\sigma, add(l, i \cdot sizeof(\tau))) \mid (\sigma, l) \in \mathcal{L}[\![lexp]\!]\ \Gamma, (\sigma, i) \in \mathcal{A}[\![a]\!]\ \Gamma\} \\
&\qquad type(\Gamma, lexp) = \tau \text{ ist der Basistyp des Feldes} \\
\mathcal{L}[\![lexp.f]\!]\ \Gamma &= \{(\sigma, l.f) \mid (\sigma, add(l, fld_off(\tau, f))) \in \mathcal{L}[\![lexp]\!]\ \Gamma\} \\
&\qquad type(\Gamma, lexp) = \tau \text{ ist der Typ der Struktur} \\
\mathcal{L}[\![*e]\!]\ \Gamma &= \mathcal{A}[\![e]\!]\ \Gamma
\end{aligned}$$

- $type(\Gamma, e)$ ist der **Typ** eines Ausdrucks
- $fld\_off(\tau, f)$ ist der **Offset** des Feldes f in der Struktur τ
- $sizeof(\tau)$ ist die **Größe** von Objekten des Typs τ

Erweiterung der Semantik: Aexp(1)

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

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

Erweiterung der Semantik: Aexp(2)

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

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

Und jetzt?

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

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*:

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

- Zustandsvariablen *StateVar*: Aktueller Zustand σ, Vorzustand ρ
- Explizite Zustandsprädikate enthalten kein $*$ oder &
- Damit Semantik:

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

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 mit expliziten Zustandsprädikaten

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

- 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

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

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

$$I.id^\dagger = I^\dagger.id$$

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

$$*I^\dagger = I^\#$$

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

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

$$n^\# = n$$

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

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

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

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

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

Angepasste Regeln des Hoare-Kalküls

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

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

Zwei kurze Beispiele

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

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

Ein problematisches Beispiel

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

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

Ein problematisches Beispiel

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

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

- Können **weder** beweisen, dass $*p = x$ **noch** $*p \neq x$
- 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 Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger "in Wirklichkeit" ein Feld ist.
- $\backslash\mathbf{array}(a, n)$ bedeutet: a ist ein Feld der Länge n, d.h.

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

- Validität kann abgeleitet werden:

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

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