Korrekte Software: Grundlagen und Methoden
Vorlesung 12 vom 26.06.17: Programmsicherheit und Frame Conditions

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Korrektheit des Hoare-Kalküls
- Vorwärts und Rückwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen und Speichermodelle
- Verifikationsbedingungen Revisited
- Vorwärtsrechnung Revisited
- Programmsicherheit und Frame Conditions
- Ausblick und Rückblick

Heute im Angebot

- Programmsicherheit Revisited:
  - Deferenzierung von Pointer und Arrays
  - Division durch 0
- Frame Conditions
  - Was ist das und wozu braucht man das?
  - Wie könnte man das in unserer Sprache behandeln?
  - Modification sets

Zur Erinnerung: Totale Korrektheit

- Partielle Korrektheit: wenn das Programm terminiert, erfüllt es die Nachbedingung.

Wie sinnvoll ist diese Aussage?

Mein Programm wäre richtig gewesen, wenn es nicht vorher abgestürtzt wäre.

- Wir wollen mindestens ausschließen, dass Laufzeitfehler ("undefined behaviour" *C99 Standard*, §3.4.3) auftreten.
- Problem: wenn Pointer als Parameter übergeben werden, müssen sie dereferenzierbar sein.
- Dazu neue Annotationen: **valid**() und **array**()

Spezifikation von Zeigern und Feldern

Die Prädikate \valid(x) und \array(a, n)

\valid(x) für x Pointer-Typ $\Longleftrightarrow$ $*$x ist definiert.
\array(a, n) für a Pointer-Typ $\Longleftrightarrow$ a ist ein Feld der Länge n.

- Abhängig vom Zustand (warum?)
- Felder als Parameter werden Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger "in Wirklichkeit" ein Feld ist.
- Formale Definition:

$$\textbf{\valid}(x, S) \stackrel{def}{=} \exists v.\, read(S, read(S, x)) = v$$

$$\textbf{\array}(a, n, S) \stackrel{def}{=} \forall i.\, 0 \leq i < n \Longrightarrow \textbf{\valid}(a[i], S)$$

Wie beweisen wir Validität?

- Konvention: $\backslash\textbf{valid}(x) \stackrel{def}{=} \backslash\textbf{valid}(x, \sigma)$
- Herleitung von Validität:

$$\frac{\text{x= \&e}}{\backslash\textbf{valid}(x, S)} \qquad \frac{S1 = upd(S, x, y) \quad \text{mit } y \in \textbf{Loc}, y \neq read(S, y')}{\backslash\textbf{valid}(x, S1)}$$

$$\frac{\backslash\textbf{valid}(x, S) \;\&\&\; S1 = upd(S, y, l) \;\&\&\; y \neq x}{\backslash\textbf{valid}(x, S1)}$$

$$\frac{\backslash\textbf{valid}(y, S) \;\&\&\; S1 = upd(S, x, read(S, y))}{\backslash\textbf{valid}(x, S1)}$$

$$\frac{\backslash\textbf{array}(a, n, S) \quad 0 \leq i \quad i < n}{\backslash\textbf{valid}(a[i], S)}$$

Program Safety

- Hier: Dereferenzierungen definiert, keine Division durch 0
- Formal als Verifikationsbedingungen

$$
\begin{aligned}
\mathit{safe}(x, S) &= \emptyset \\
\mathit{safe}(*x, S) &= \{\backslash\mathbf{valid}(x, S)\} \\
\mathit{safe}(a[i]) &= \{\backslash\mathbf{array}(a, n)\ \&\&\ 0 <= i\ \&\&\ i < n\} \\
\mathit{safe}(x + y, S) &= \mathit{safe}(x, S) \cup \mathit{safe}(y, S) \\
\mathit{safe}(x/y, S) &= \{y\ !\ = 0\} \\
&\ldots
\end{aligned}
$$

- Nicht ganz exakt: $y\ !\ = 0$ muss im Zustand S ausgewertet werden,
  - D.h. $P \Longrightarrow y\ !\ = 0$ mit P Vorbedingung
  - Dazu muss *safe* in die Definition von asp/awp eingebunden werden
- Valid-Analysen bleiben rein schematisch
- Division durch 0 und Arrayzugriffe benötigen Auswertung

Funktionsparameter und Frame Conditions

- Problem: Funktionen können beliebige Änderungen im Speicher vornehmen.

```
int x, y, z;

z= x+ y;
swap(&x, &y);
/** { z = \old(x)+ \old(y) } */
```

- Vor/Nach dem Funktionsaufruf (hier swap) muss die Nachbedingung/Vorbedingung noch gelten.

Frame Rule

- Konstanzregel (Rule of Constancy):

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P \wedge R\}\, c\, \{Q \wedge R\}}$$

- Nebenbedingung: keine in c veränderte Variable tritt in R auf

- Problem: gilt mit Pointern nur eingeschränkt, da c eventuell ohne direkte Zuweisung Teile des Zustands verändert, über den R Aussagen macht.

Modification Sets

- Idee: Spezifiziere, welcher Teil des Zustands verändert werden darf.
  - ... denn wir können nicht spezifizieren, was gleich bleibt.
- Syntax: modifies **Mexp**

$$\textbf{Mexp} ::= \textbf{Loc} \mid \textbf{Mexp}\ [*] \mid \textbf{Mexp}\ [i : j] \mid \textbf{Mexp}\ .\ \textbf{name}$$

  - Mexp sind Lexp, die auch Teile von Feldern bezeichnen.
- Semantik: $[\![-]\!] : \mathit{Env} \to \textbf{Mexp} \to \Sigma \to \mathbb{P}(\textbf{Loc})$
- Modification Sets werden in die Hoare-Tripel integriert.

Erweiterung der Hoare-Tripel

- Hoare-Tripel mit Modification Sets:

$$\Lambda \models \{P\}\, c\, \{Q\} \iff \forall \sigma.\, P(\sigma) \wedge \exists \sigma'.\, \sigma' = c(\sigma) \implies Q(\sigma') \wedge \sigma \cong_\Lambda \sigma'$$

- wobei $\sigma \cong_L \tau$: Zustände σ und τ sind gleich bis auf die Adressen in L

$$\sigma \cong_L \tau \iff \forall l \in dom(\sigma) \cup dom(\tau) \setminus L.\, \sigma(l) = \tau(l)$$

bzw.

$$\sigma \cong_L \tau \iff \forall l.\, \sigma(l) \neq \tau(l) \implies l \in L$$

Erweiterung der Regeln

- Regeln werden mit Modification Set annotiert:

$$\Gamma, \Lambda \vdash \{P\}\, c\, \{Q_1 | Q_2\}$$

- Modification Set wird durchgereicht
- Zuweisungsregel wird ergänzt (vorwärts/rückwärts):

$$\frac{S \notin FV(P)}{\Gamma, \Lambda \vdash \{P\}\, x = e\, \{\exists S.\, P[S/\sigma] \;\&\&\; x_S^\dagger \in \Lambda \;\&\&\; \sigma == upd(S, x_S^\dagger, e_S^\#)\}}$$

$$\frac{}{\Gamma, \Lambda \vdash \{Q[upd(\sigma, x_S^\dagger)/\sigma] \;\&\&\; x_\sigma^\dagger \in \Lambda\}\, x = e\, \{Q\}}$$

Zusammenfassung

- Programmsicherheit kann als zusätzliche Verifikationsbedingung formuliert werden
  - Nachteil: teilweise komplexe Verifikationsbedingungen
  - Vorteil: semantische Integrität
- Frame Rule: spezifiziert unveränderte Teile des Zustands
  - Essentiell für Skalierbarkeit
  - Bei Zeigern rein syntaktische Analyse (freie Variablen) nicht ausreichend, daher modification sets
  - Spezifizieren veränderlichen Teil des Zustandes
  - Werden bei Zuweisungen geprüft