

Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 05.07.17: Rückblick & Ausblick

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**

Was gibt's heute?

- ▶ Rückblick
- ▶ Ausblick
- ▶ Feedback

Rückblick

Semantik

- ▶ Operational — Auswertungsrelation $\langle c, \sigma \rangle \rightarrow \sigma'$
- ▶ Denotational — Partielle Funktion $\llbracket c \rrbracket : \Sigma \rightarrow \Sigma$
- ▶ Axiomatisch — Floyd-Hoare-Logik
- ▶ Welche Semantik wofür?
- ▶ Beweis: Äquivalenz von operationaler und denotationaler Semantik

Floyd-Hoare-Logik

- ▶ Floyd-Hoare-Logik: partiell und total
- ▶ $\vdash \{P\} c \{Q\}$ vs. $\models \{P\} c \{Q\}$: Vollständigkeit, Korrektheit
- ▶ Die sechs Basisregeln
- ▶ Zuweisungsregel: vorwärts (Floyd) vs. rückwärts (Hoare)
- ▶ VCG: Schwächste Vorbedingung und stärkste Nachbedingung
- ▶ Beweis: Korrektheit und Vollständigkeit der Floyd-Hoare-Logik

Erweiterung der Programmiersprache

- ▶ Für jede Erweiterung:
 - ▶ Wie modellieren wir semantisch?
 - ▶ Wie ändern sich die Regeln der Logik?
- ▶ Prozeduren und Funktionen
 - ▶ Modellierung von **return**: Erweiterung zu $\Sigma \rightarrow \Sigma \times \mathbf{V}_U$
 - ▶ Spezifikation von Funktionen durch Vor-/Nachbedingungen
 - ▶ Spezifikation der Funktionen muss im Kontext stehen

Erweiterung der Programmiersprache

- ▶ Strukturen, Felder, Referenzen
 - ▶ Lokationen, **Lexp**, strukturierte Werte
 - ▶ Lokationen nicht mehr symbolisch (Variablennamen), sondern abstrakt
 $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}, \mathbf{V} = \mathbf{N} + \mathbf{C} + \mathbf{Loc}$
 - ▶ Zustand als **abstrakter Datentyp** mit Operationen *read* und *upd*
 - ▶ Zuweisung nicht mehr mit Substitution, sondern explizit durch *upd*
 - ▶ Spezifikationen sind **explizite Zustandsprädikate**
 - ▶ Konversionen $(-)^{\dagger}, (-)^{\#}$

Erweiterung der Programmiersprache

- ▶ **Programmsicherheit**
 - ▶ Keine Division durch 0
 - ▶ Keine illegale Dereferenzierung (einschließlich Felder)
 - ▶ Dazu: `\valid`
- ▶ Frame Conditions und Modification Sets
 - ▶ Frame Problem: welcher Teil des Zustands bleibt **gleich**?
 - ▶ Mit Zeigern: **modification sets** — Spezifikation des **veränderlichen** Teils

Ausblick

Die Sprache C: Was haben wir ausgelassen?

Semantik:

- ▶ Nichtdeterministische Semantik: Seiteneffekte, Sequence Points
→ Umständlich zu modellieren, Effekt zweitrangig
- ▶ Implementationsabhängiges, unspezifiziertes und undefiniertes Verhalten
→ Genauere Unterscheidung in der Semantik

Kontrollstrukturen:

- ▶ **switch** → Ist im allgemeinen Fall ein **goto**
- ▶ **goto**, `setjmp/longjmp`
→ Allgemeinfall: tiefe Änderung der Semantik (*continuations*)

Die Sprache C: Was haben wir ausgelassen?

Typen:

- ▶ Funktionszeiger → Für “saubere” Benutzung gut zu modellieren
- ▶ Weitere Typen: **short/long int**, **double/float**, `wchar_t`, und Typkonversionen → Fleißarbeit
- ▶ Fließkommazahlen → Spezifikation nicht einfach
- ▶ **union** → Kompliziert das Speichermodell
- ▶ **volatile** → Bricht read/update-Gleichungen
- ▶ **typedef** → Ärgernis für Lexer/Parser, sonst harmlos

Die Sprache C: Was haben wir ausgelassen?

Für **realistische C-Programme**:

- ▶ Compiler-Erweiterungen (gcc, clang)
- ▶ Büchereien (Standardbücherei, Posix, ...)
- ▶ Nebenläufigkeit

Wie modelliert man Java?

- ▶ Die **Kernsprache** ist ähnlich zu C0.
- ▶ Java hat erschwerend:
 - ▶ dynamische Bindung,
 - ▶ Klassen mit gekapseltem Zustand und Invarianten,
 - ▶ Nebenläufigkeit, und
 - ▶ Reflektion.
- ▶ Java hat dafür aber
 - ▶ ein einfacheres Speichermodell, und
 - ▶ eine wohldefinierte Ausführungsumgebung (die JVM).

Wie modelliert man C++?

- ▶ Sehr **vorsichtig** (konservativ)
- ▶ Viele Features, fehlende formale Semantik, ...
- ▶ Mehrfachvererbung theoretisch anspruchsvoll
- ▶ Es gibt **keine** Formalismen/Werkzeuge, die C++ voll unterstützen
- ▶ Ansätze: Übersetzung nach C/LLVM, Behandlung dort

Wie modelliert man PHP?

Gar nicht.

Korrekte Software in der Industrie

- ▶ Meist in speziellen Anwendungsgebieten: Luft-/Raumfahrt, Automotive, sicherheitskritische Systeme, Betriebssysteme
- ▶ Ansätze:
 1. Vollautomatisch: **statische Analyse** (Abstrakte Interpretation) für spezielle Aspekte: Freiheit von Ausnahmen und Unter/Überläufen, Programmsicherheit, Laufzeitverhalten (WCET) (nicht immer korrekt, meist vollständig)
 - ▶ Werkzeuge: absint
 2. Halbautomatisch: **Korrektheitsannotationen**, Überprüfung automatisch
 - ▶ Werkzeuge: Spark (ADA), Frama-C (C), JML (ESC/Java, Krakatao; Java), Boogie und Why (generisches VCG), VCC (C)
 3. Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
 - ▶ Beispiele: L4.verified, CompCert, SAMS

Feedback

Deine Meinung zählt

- ▶ Was war gut, was nicht?
- ▶ Arbeitsaufwand?
- ▶ Mehr Theorie oder mehr Praxis?
- ▶ Mehr oder weniger Scala?

Tschüß!

