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

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Was gibt's heute? ► Rückblick ► Ausblick ► Feedback Gorrekte Software 3 [20]

- $lackbox{ }$ Operational Auswertungsrelation $\langle c,\sigma
 angle
 ightarrow\sigma'$
- ▶ Denotational Partielle Funktion $\llbracket c \rrbracket : \Sigma \rightharpoonup \Sigma$
- Axiomatisch Floyd-Hoare-Logik
- ► Welche Semantik wofür?

Semantik

▶ Beweis: Äquivalenz von operationaler und denotationaler Semantik

Korrekte Software 5 [20]

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 $\Sigma \rightharpoonup \Sigma \times V_U$
 - ► Spezifikation von Funktionen durch Vor-/Nachbedingungen
 - ► Spezifikation der Funktionen muss im Kontext stehen

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

DK W

Rückblick

Software

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

Korrekte Software

6 [20]

Erweiterung der Programmiersprache

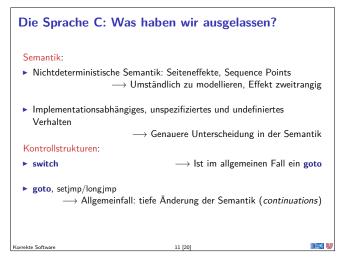
- ► Strukturen, Felder, Referenzen
 - ▶ Lokationen, **Lexp**, strukturierte Werte
 - Lokationen nicht mehr symbolisch (Variablennamen), sondern abstrakt $\Sigma=\text{Loc} \rightharpoonup \text{V}, \text{V}=\text{N}+\text{C}+\text{Loc}$
 - ► Zustand als abstrakter Datentyp mit Operationen read und upd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

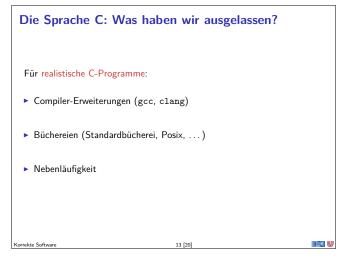
DK W

8 [2

re 7 [2

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





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

Ausblick Korrekte Software 10 [20]

Die Sprache C: W	as haben wir ausgelassen	?
Typen: ▶ Funktionszeiger -	ightarrow Für "saubere" Benutzung gut zu	modellieren
► Weitere Typen: short/l Typkonversionen	long int, double/float, wchar_t, und	→ Fleißarbeit
► Fließkommazahlen	$\longrightarrow Spezifikation$ ı	nicht einfach
► union	\longrightarrow Kompliziert das Sp	eichermodell
▶ volatile	\longrightarrow Bricht read/update-	Gleichungen
► typedef	\longrightarrow Ärgernis für Lexer/Parser, so	onst harmlos
Korrekte Software	12 [20]	DK (

Wie modelliert man Java? Die Kernsprache ist ähnlich zu CO. Java hat erschwerend: dynamische Bindung, Klassen mit gekapselten Zustand und Invarianten, Nebenläufigkeit, und Reflektion. Java hat dafür aber ein einfacheres Speichermodell, und eine wohldefinierte Ausführungsumgebung (die JVM).

Wie modelliert man	PHP?	
	Gar nicht.	
Correkte Software	16 [20]	

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

Korrekte Software

17 [20]

Feedback Korrekte Software 18 [20]

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

19 [20]

