

Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 04.07.24
Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2024

Fahrplan

- ▶ Einführung
- ▶ Operationale Semantik
- ▶ Denotationale Semantik
- ▶ Äquivalenz der Operationalen und Denotationalen Semantik
- ▶ Der Floyd-Hoare-Kalkül
- ▶ Invarianten im Floyd-Hoare-Kalkül
- ▶ Korrektheit des Floyd-Hoare-Kalküls
- ▶ Strukturierte Datentypen
- ▶ Verifikationsbedingungen
- ▶ Funktionen und Prozeduren I
- ▶ Funktionen und Prozeduren II
- ▶ Referenzen
- ▶ **Ausblick und Rückblick**

Was gibt's heute?

- ▶ Rückblick
- ▶ Ausblick
- ▶ Feedback
- ▶ Prüfungsvorbereitung

I. Rückblick

Prüfungsrelevanz

- ▶ Was waren die **zentralen** theoretischen Konzepte?
- ▶ Wie sind sie formal definiert, was bedeuten sie?
- ▶ Wie hängen sie zusammen?

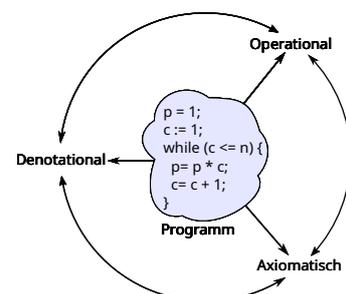
Semantik

- ▶ 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

Zusammenhang der Semantiken



Erweiterungen der Programmiersprache

- ▶ Für jede Erweiterung:
 - ▶ Wie modellieren wir sie semantisch?
 - ▶ Wie ändern sich die Regeln der Logik?

1. Erweiterung der Programmiersprache

- ▶ Strukturen und Felder
 - ▶ Locations und strukturierte Werte **Lexp**
 - ▶ Zustand: endliche Abbildung von Locations auf Werte
 - ▶ Erweiterte Substitution in Zuweisungsregel
 - ▶ Sonstige Regeln bleiben

2. Erweiterung der Programmiersprache

- ▶ Prozeduren und Funktionen
 - ▶ Modellierung von **return**: Erweiterung zu $\Sigma \rightarrow \Sigma + \Sigma \times \mathbf{V}_U$
 - ▶ Modellierung von lokalen Variablen
 - ▶ Spezifikation von Funktionen durch Vor-/Nachbedingungen
 - ▶ Spezifikation der Funktionen muss im Kontext stehen
 - ▶ Unterscheidung zwischen zwei Nachbedingungen
 - ▶ Regeln für den Funktionsaufruf

3. Erweiterung der Programmiersprache

- ▶ Referenzen
 - ▶ Grundproblem: Aliasing
 - ▶ Zustand enthält auch Locations im Wertebereich: $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}, \mathbf{V} = \mathbb{Z} + \mathbf{Loc}$
 - ▶ Spezifikationen sind **explizite Zustandsprädikate** mit read und upd, Konversion $(-)^{\dagger}, (-)^{\#}$
 - ▶ Zuweisung nicht mehr mit Substitution, sondern explizit durch upd

II. Ausblick

Was geht noch?

- ▶ Die Sprache C
- ▶ Andere Programmiersprachen
- ▶ Logik und Spezifikation
- ▶ Success Stories

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 (Standardbibliothek, Posix, ...)
- ▶ Nebenläufigkeit

Andere Sprachen: 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).

Andere Sprachen: 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

Andere Sprachen: Wie modelliert man PHP?

Gar nicht.

Logik und Spezifikation

- ▶ Wir **generieren** Verifikationsbedingungen, wie kann man sie **beweisen**?
- ▶ **Automatische Beweiser**:
 - ▶ **SAT-Checker** lösen Erfüllbarkeitsproblem der Aussagenlogik (MiniSAT, Chaff)
 - ▶ **SMT-Beweiser** beweisen Aussagen der Prädikatenlogik mit linearer Arithmetik, Funktionen und Induktion (Z3, Yices, CVC)
- ▶ **Interaktive Beweiser**:
 - ▶ Beweisführung durch Benutzer, **Überprüfung** durch Beweiser
 - ▶ Sehr **mächtige** Logiken, aber nicht vollautomatisch (Isabelle, Coq, Lean)

Beispiel: Z3

- ▶ SMT-Beweiser versuchen Gegenbeweis zu konstruieren
- ▶ Daher: um ϕ zu beweisen, versuchen wir $\neg\phi$ zu widerlegen

Beweis einer VC:
 $x \geq 0 \wedge y > 0 \implies x = 0 * y + x$

Unerfüllbare VC:
 $x \geq 0 \wedge y > 0 \implies x \geq y$

Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
  (not (=> (and (>= x 0) (> y 0))
          (= x (+ (* 0 y) x))))
)
(check-sat)
```

Antwort:

unsat

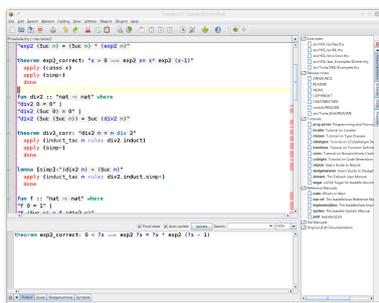
Input Z3:

```
(declare-const x Int)
(declare-const y Int)
(assert
  (not (=> (and (>= x 0) (> y 0))
          (>= x y)))
)
(check-sat)
```

Antwort:

sat

Beispiel: Isabelle



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, Krakatoa; Java), Boogie and Why (generisches VCG), VCC (C)
 - 3 Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
 - ▶ Beispiele: L4.verified, CompCert, SAMS

III. Prüfungsvorbereitung

Prüfungsvorbereitung

- ▶ Termine: Do 15.08.2024 / Fr 13.09.2024 (ab 10 Uhr) (Anmeldung über stud.ip)
- ▶ Mündliche Modulprüfung, 20–30 Minuten
- ▶ Schwerpunkte:
 - ▶ **Verständnis** des Stoffes, weniger Folien auswendig lernen
 - ▶ Zentrale theoretische Konzepte kennen und benennen
 - ▶ Stoff der Vorlesung und Übungsblätter, weniger eure Lösungen
- ▶ Bewertung
 - ▶ Sicherheit/Beherrschung des Stoffes
 - ▶ *covered ground*

IV. Feedback

Deine Meinung zählt

- ▶ Bitte die **Evaluation** auf stud.ip beantworten!
- ▶ Was war gut, was nicht?
- ▶ Arbeitsaufwand?
- ▶ Mehr **Theorie** oder mehr **Praxis**?
- ▶ Programmieraufgaben?
- ▶ Leichtgewichtiger Übungsbetrieb — mehr oder weniger?



Werbeblock

- ▶ **Studentische Tutoren** gesucht:
 - ▶ Praktische Informatik 3 — Einführung in die funktionale Programmierung
- ▶ Veranstaltung:
 - ▶ „Adventures in Programming Languages“
 - ▶ Praktische Einführung in Rust
 - ▶ 04.09.2024, 09:00 – 16:00
 - ▶ Details:
https://www.bremen-research.de/data-train/courses/course-details?event_id=70

Tschüß!

