Universität Stuttgart

PSE:
Schwächste Vorbedingungen und formale Beweise

Prof. Dr.-Ing. Steffen Becker

Vorlesung 24


Universität Stuttgart
Recap
PSE:
Streams
Prof. Dr.-Ing. Steffen Becker
Vorlesung 23

Javas Stream-API - worum geht es da?

Die Stream-API befindet sich im Paket `java.util.stream`

"Klassen, um **Operationen nach dem funktionalen Stil** auf Streams von Elementen zu unterstützen, wie bspw. Map-reduce **Transformationen auf Collections**.

Beispiel:

```
Collection<Widget> widgets = …;

int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Stream Operationen und Pipelines

- Streamoperationen warden in **Intermediär-** und **Terminaloperationen** unterteilt und zu **Stream-Pipelines** zusammengefasst.
- Eine Strompipeline besteht aus
  - einer **Quelle** (bspw. Collection, einer Generatorfunktion, oder einem I/O Channel)
  - gefolgt von null oder mehr **Intermediäroperationen** (z.B. `Stream.filter`, `Stream.map`)
  - und eine **Terminaloperation** (z.B. `Stream.forEach`, `Stream.reduce`)

Einen neuen Stream zurückgeben

Kann den Stream traversieren, um ein Ergebnis oder einen Nebeneffekt zu erzielen

Beispiel-Operationen

- Zwischenoperationen (zustandslos vs. zustandsbehaftet)

Stream<T>	distinct()
Stream<T>	filter(Predicate<? super T> predicate)
<R> Stream<R>	map(Function<? super T,? extends R> mapper)
Stream<T>	peek(Consumer<? super T> action)
Stream<T>	sorted()

- Terminal- (oder Reduzier-/Falt-) Operationen

boolean	allMatch(Predicate<? super T> predicate)
<R,A> R	collect(Collector<? super T,A,R> collector)
long	count()
void	forEach(Consumer<? super T> action)
int	sum()

Beispieloperation in einer bestimmten Klasse von Streams (hier: IntStream)

Streams vs. Collections

Reduktions-Operationen

- Eine Reduktionsoperation (auch Faltung genannt) nimmt eine Folge von Eingabeelementen und kombiniert sie zu einem einzigen zusammenfassenden Ergebnis
- Beispiele:
  - die Summe oder das Maximum einer Menge von Zahlen zu finden
  - Elemente zu einer Liste zu akkumulieren
  - Allgemeine Reduktionsmaßnahmen von Stream
    - `reduce()`
    - `collect()`
  - Spezialisierte Reduktionsoperationen, die von IntStream usw. bereitgestellt werden
    - `sum()`
    - `max()`
    - `count()`

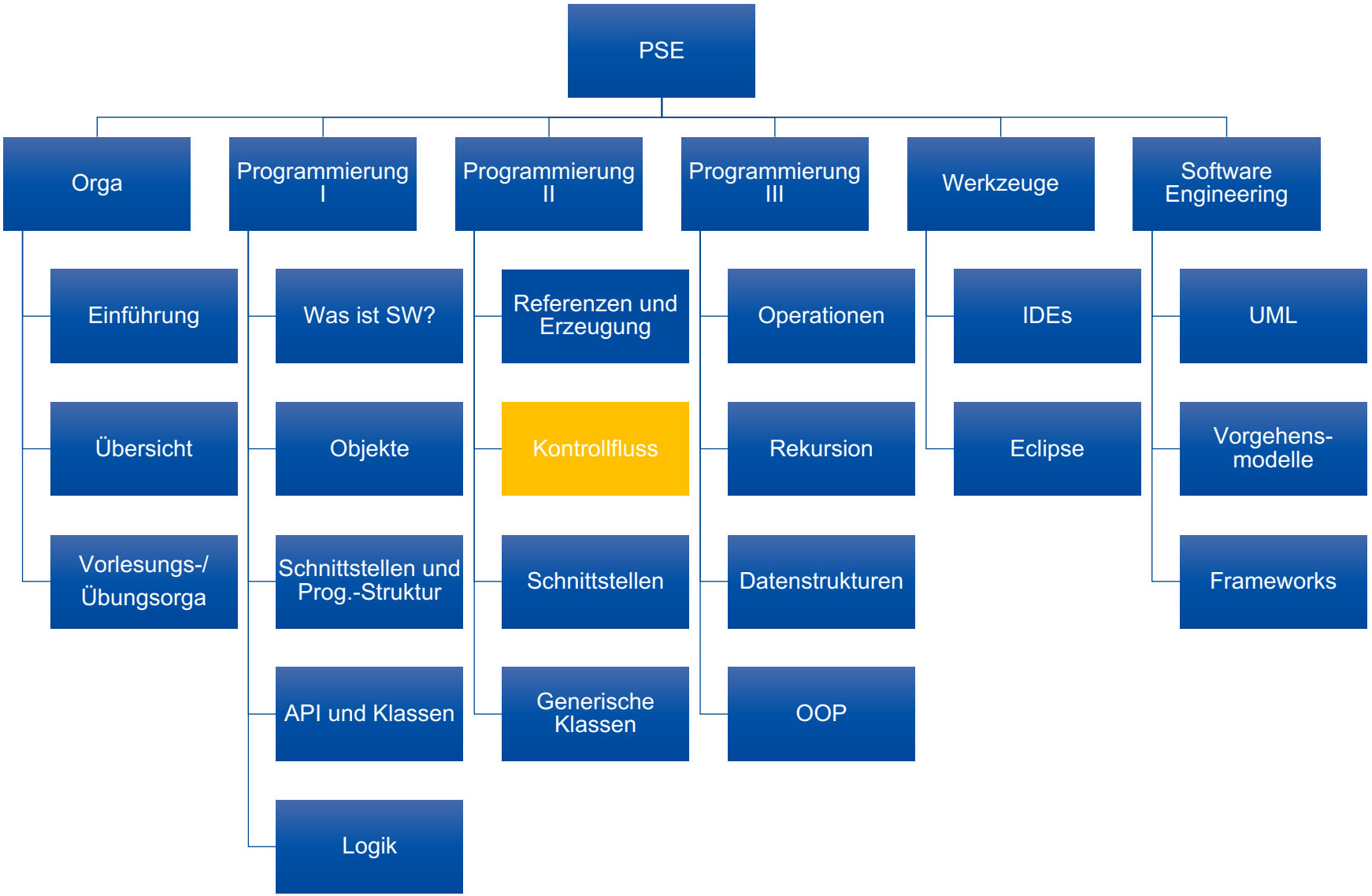
Universität Stuttgart

PSE:
Schwächste Vorbedingungen und formale Beweise

Prof. Dr.-Ing. Steffen Becker

Vorlesung 24

Vorlesungsübersicht



Lernziele

- Korrektheitsbeweise mittels WP-Calculus führen
- Schwächste Vorbedingungen verschiedener Kontrollflusskonstrukte

Vielen Dank

Einige der folgenden Folien oder ihrer Inhalte wurden freundlicherweise zur Verfügung gestellt durch Prof. Stefan Wagner.

Schwächste Vorbedingungen: Grundlagen

Motivation

In der Praxis bekommen wir Anforderungen von unseren Stakeholdern.

Anforderungen werden dann in **Spezifikationen** überführt, insbesondere Nachbedingungen.

Sehr einfaches Beispiel:

Anf.: Eine Funktion soll den doppelten Wert ihres Parameters berechnen.

Post.: `ensure \result = 2 * value`

Frage des Tages: Wie leiten wir unsere Vorbedingungen ab?

Motivation II

In der Praxis werden wir oft zuerst die Funktion implementieren!

```
public class Twice {

    public static void main(final String[] args) {
       System.out.println(twice(2));
    }

    public static int twice(final int value) {
        return value + value;
    }
}
```

Vorbedingung?

```
requires
0 <= abs(value) <= Integer.MAX_INT / 2
```

Eigenschaft unserer Vorbedingung?

Wie sind wir auf diese Vorbedingung gekommen?

Ist es eine „gute“? Was charakterisiert gute Vorbedingungen?

Warum lautet sie nicht
`0 <= abs(value) <= Integer.MAX_INT / 4`?
Wäre diese Vorbedingung ungültig?

Antwort: Eine „gute“ Vorbedingung erlaubt es, die Operation in so vielen Situationen wie möglich zu verwenden, sodass wir die größtmögliche Wiederverwendbarkeit unseres Codes erreichen. Das nennt man **schwächste Vorbedingung** (beschrieben durch eine Funktion wp).

Hoare Tripel

Um mit Programmen und Vor-/Nachbedingungen zu arbeiten, haben wir bereits eine Notation in der Vorlesung verwendet:

{P} S {Q}

mit
- P als Vorbedingung
- S als Stück Code
- Q als Nachbedingung

Diese Notation nennt man ein **Hoare Tripel**. Das Hoare Tripel ist **gültig,** falls, gegeben P, Q durch S garantiert wird.

Anwendungsfälle des Hoare Tripels

Definition der axiomatischen Semantik einzelner Anweisungen
(falls S diese einzelne Anweisung ist) (**DEF**)

Beschreibung eines existierenden Programms (**CODE**)
(Beispiel: unsere `twice` Funktion inkl. `requires` und `ensures`).

Spezifikation eines zu entwickelnden Programms (**SPEC**)
(Beispiel, die Konvertierung unserer Anforderungen für die `twice` Funktion in ihre Nachbedingung. Es ist zu beachten, dass wir P absichtlich ausgelassen haben, um es später zu spezifizieren)

Korrekte Programme

Erinnerung: Verifikation und Validierung aus der SE Vorlesung.

Verifikation garantiert korrekte Programme, d.h., sie beantwortet die Frage, ob der Code mit seiner Spezifikation konform ist.

Unter Verwendung des eingeführten Hoare Tripels, ist ein Programm korrekt, falls:

Gegeben: $\{P_{SPEC}\}$ und $\{Q_{SPEC}\}$ seien eine Spezifikation

Und $\{P_{CODE}\}$ S $\{Q_{CODE}\}$ sei ein gültiges Hoare Tripel.

Dann: S ist korrekt, falls (und nur falls) $P_{SPEC} \Rightarrow P_{CODE}$ und $Q_{CODE} \Rightarrow Q_{SPEC}$

Schwächste Vorbedingung

Wie bereits erwähnt, ist P_{SPEC} häufig nicht gegeben, genauso wenig P_{CODE}.

Stattdessen wollen wir ein maximal wiederverwendbares S.

→ Identifiziere P_{CODE} mit $\{P_{CODE}\}$ S $\{Q_{CODE}\}$ als gültiges Hoare Tripel, sodass für alle anderen $P_{CODE'}$ mit $\{P_{CODE'}\}$ S $\{Q_{CODE}\}$ als gültiges Hoare Tripel, $P_{CODE'} \Rightarrow P_{CODE}$ gilt.

Die Funktion **wp(S, Q)** gibt das Prädikat P_{CODE} zurück
(welches, ausgenommen logischer Äquivalenz, eindeutig definiert ist)

Beispiel

In einer Liste aus Integern `l`, welche Elemente an den Indizes von 0 bis 1000 besitzt, finden wir Werte von {0..20}. Jeder Wert muss mehrmals vorhanden sein.

Wir wollen den kleinsten Index ermitteln, an dem ein bestimmter, gegebener Wert aus einer dem Bereich {0..20} gefunden wird (d.h. Auffinden des ersten Auftretens)

Anmerkung:
JML ist hier Pseudo-JML
...

Q_{SPEC}:

```
ensures \result = min{i | l.get(i) == a};
```

P_{SPEC}:

```
requires \notnull l and l.size() == 1001;
requires (\forall i:Integer; {0..1000}; {0..20}.contains(l.get(i)));
requires (\forall a:Integer; {0..20};
            \exists i,j:Integer; {0..1000};
                  i != j && a == l.get(i) == l.get(j));
```

Beispiel (Fortsetzung)

Typische Implementierung:

```
private static <G> int findMinIndex(final List<G> l, final G a) {
     for (int i = 0; i < l.size(); i++) {
         if (l.get(i) == a) {
             return i;
         }
     }
     throw new IllegalArgumentException();
 }
```

Q_{CODE}:

```
ensures l.get(\result) == a && (\forall j:Integer; {0..\result};
                                                    l.get(\result) != a);
```

P_{CODE}:

```
requires \notnull l;
requires (\exists i:Integer; {0..l.size()}; l.get(i) == a);
```

Problem

Wir kennen oft Q_{CODE} für den Code
(oder verwenden hierfür das gleiche Prädikat wie Q_{SPEC}).

Wir kennen auch P_{SPEC} aus unseren Anforderungen.

Wir wollen ein P_{CODE} definieren, welches so restriktiv wie möglich ist,
um Code wiederverwendbar zu machen → Setze $P_{CODE} = wp(S, Q_{CODE})$.

Wir benötigen eine Vorgehen, um $wp(S, Q_{CODE})$ für beliebigen Code abzuleiten!

Schwächste Vorbedingung Regeln

Eingeschränkte Programmiersprache

Das Feststellen von wp(S, Q) für Code in Java 1.8 ist sehr komplex.

Im Folgenden betrachten wir eingeschränkte Programme, bestehend aus

- Leeren Anweisungen
- Zuweisungen
- Sequenzen
- Bedingungen
- While-Schleifen

Leere Anweisungen

Semantik:

$wp(\varepsilon;, Q) \equiv Q$

Informell:

Die Bedingung nach ε (da sie nichts ändert) muss bereits vor ε gelten.

Zuweisungen

Semantik:

$wp(x = E;, Q(x)) \equiv \exists\, E \in TYP(x) \wedge Q(E)$

Informell:

E muss definiert und typkompatibel zu x sein. Dann gleicht die Vorbedingung der Nachbedingung mit allen Instanzen von x ersetzt durch den Ausdruck E. E kann weiterhin x enthalten (kein rekursives Ersetzen).

Sequenzen

Semantik:

$wp(S1; S2;, Q) \equiv wp(S1, wp(S2,Q))$

Informell:

Die schwächste Vorbedingung einer Sequenz von zwei Anweisungen ist die schwächste Vorbedingung der ersten Anweisung unter Verwendung der schwächste Vorbedingung der zweiten Anweisung als Nachbedingung.

Beispiel

Mit x, y: Integer

{ y == -1}

{ y >= -1 && y < 0}

{ y + 1 >= 0 && y * 2 < 0} - wp(S1, wp(S2,Q))

x = y + 1; - S1 (Zuweisung)

{ x >= 0 && y * 2 < 0} - wp(S2, Q)

y = y * 2; - S2 (Zuweisung)

{ x >= 0 && y < 0} - Q

2. Beispiel

Gegeben ist das Codefragment S mit S als

```
t = t * x; i = i + 1;
```

$i,n \in int$, $Q = \{ i <= n \;\&\&\; t = x^i \}$

Aufgabe: Berechne wp(S,Q)

Zeit: 10min

Ergebnis: Online Diskussion

Verzweigungen

Semantik:

wp(if (B) { S_1; } else { S_2; }, Q)

$\equiv (B \Rightarrow wp(S1, Q)) \wedge (\neg B \Rightarrow wp(S2, Q))$

$\equiv (B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q))$

Anwendung der Definition der Implikation. Nur korrekt, falls alle Prädikate definiert sind (siehe Vorlesung 5)

Informell:

Falls B wahr ist, ist die SV der Bedingung gleich der SV von S1.
Falls B falsch ist, ist die SV der Bedingung gleich der SV von S2.

Verzweigte Anweisungen

Semantik:

wp(if (B) { S; }, Q)

≡ (B ⇒ wp(S, Q)) ∧ (¬B ⇒ Q)

≡ (B ∧ wp(S, Q)) ∨ (¬B ∧ Q)

Beweis durch Einsetzen der leeren Anweisung in die Bedingung.

Informell:

Falls B wahr ist, ist die SV der bedingten Anweisung gleich der SV von S. Falls B falsch ist, ist die SV der bedingten Anweisung gleich der Nachbedingung (Regel der leeren Anweisung).

Beispiel

S seien die Anweisungen `if (x < 0) { x = -x } else { x = x-1 }`
x sei ein INT und Q = { x >= 0}

```
if (x < 0) {

      x = -x;

} else {

      x = x - 1;

}
```

While-Schleifen

Beispiel:

```
// invariant t = x^i; variant n − i;
{ n >= 0 } => { ( 0 < n && 1 == x^0 && n − 0 >= 0) or (0 >= n && 1 == x^n) }
t = 1; i = 0;
{ (i < n && t == x^i && n − i > 0) or (i >= n && t = x^n) }
// (B && inv && var > 0) || (!B && Q)
while (i < n) { // while (B) { ... }
    // inv && B && 0 < var <= v
    { t == x^i && i < n &&  0 < n − i <= v } => { t ∗ x == x^(i+1) && 0 < n − i <= v}
    t = t ∗ x;
    { t == x^(i+1) && 0 <= n − (i + 1) < v } = { t == x^(i+1) && 0 < n − i <= v }
    i = i + 1;
    { t == x^i && 0 <= n − i < v } // inv && 0 <= var < v (v: Int, fixed arbitrary value)
}
{ t == x^i && i >= n && n − i == 0 } // inv & not B && var = 0
Q = { t == x^n }
```

While-Schleifen

Semantik:

Gegeben seien die schwächste Invariante inv und die Variante var mit

{inv and B and 0<var<=v} S {inv and 0<=var<v} für ein beliebiges v:Integer ist ein gültiges Hoare Tripel und

{inv and not B and var = 0} ==> Q

dann

wp(while (B) { S; }, Q)

≡ (B && inv && var > 0) || (!B && Q)

Garantiert die schwächste Vorbedingung, falls (und nur falls) inv und var korrekt gewählt sind (d.h. sind die schwächstmöglichen).

Vollständige Induktion für While-Schleifen

Für den Fall, dass wir keine guten Invariante und Variante kennen oder finden, müssen wir uns auf vollständige Induktion verlassen. Dies basiert auf der folgenden Tatsache:

$wp_0(S, Q) = \{ i > n \;\&\&\; Q \}$

$wp_1(S, Q) = \{ i <= n \;\&\&\; wp(S, i > n \;\&\&\; Q) \}$

$wp_2(S, Q) = \{ i <= n \;\&\&\; wp(S, i <= n \;\&\&\; wp(S, i > n \;\&\&\; Q)) \}$

...

$wp(S,Q) = wp_0(S,Q) \;||\; wp_1(S,Q) \;||\; wp_2(S,Q) \;||\; ...$

→ Semantik durch Zerteilen aller möglichen Schleifen und Anwenden der Regeln für Bedingungen

Zusammenfassung

Gewonnene Erkenntnisse

Code sollte eine Vorbedingung haben, die maximale Wiederverwendbarkeit erlaubt → schwächste Vorbedingung.

Wir haben schwächste Vorbedingungen für verschiedene Anweisungen und Kontrollflusskonstrukte kennengelernt:

- Leere Anweisungen
- Zuweisungen
- Sequenzen
- Bedingungen
- While-Schleifen mit inv und var

Universität Stuttgart

PSE:
Streams

Prof. Dr.-Ing. Steffen Becker

Vorlesung 25