252-0027
Einführung in die Programmierung

6. Logisches Schliessen

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

6. Logisches Schliessen

6.1 Motivation und logische Aussagen

6.2 Hoare-Logik und Hoare-Tripel

6.3 Regel für Zuweisungen

6.4 Regel für Programmsegmente

6.5 Regel für Verzweigungen

6.6 Invarianten und Regel für Schleifen

Vogelperspektive Softwareentwicklung

- Grosse Programme werden aus einzelnen *Modulen* (Methoden, Klassen, Bibliotheken, …) zusammen-gesetzt ➜ Modulare Entwicklung

- Voraussetzung: Es muss klar *spezifiziert* werden, was ein Modul leisten soll
  - Für Modulentwickler/innen: Wann ist das Modul vollständig bzw. was fehlt noch?
  - Für Modulnutzer/innen: Wie ist das Modul zu benutzen? Was kann ich erwarten?

Bild mit Handy-Komponenten: ifixit.com

Beispielspezifikationen

```
// PRE:  x ≥ 0
// POST: result² == x
double sqrt(double x) {
  ...
  // viel und komplexer Code
  ...
}
```

```
// PRE:  data != null
// POST: ∀ i. data[i] ≤ data[i+1]
void sort(int[] data) {
  ...
  // viel und komplexer Code
  ...
}
```

Spezifikationen in «mathematischer Sprache» (Logik → *Diskrete Mathematik*), damit eindeutig und beweisbar

- **Vorbedingung** («**precondition**»): Was muss vor dem Aufruf gelten?
- **Nachbedingung** («**postcondition**»): Was gilt nach dem Aufruf?
- Hier als spezielle Java-Kommentare ins Programm geschrieben

result bezieht sich auf das Ergebnis der Berechnung (den Rückgabewert der Methode)

Vor- und Nachbedingungen: Zwei Perspektiven

```
// PRE:  x ≥ 0
// POST: result² == x
double sqrt(double x)
```

Methodenentwickler

```
double sqrt(double x) {
  // Kann x ≥ 0 annehmen
  // ... Code ...
  // Muss result² == x garantieren
}
```

Methodennutzer

```
// Muss x ≥ 0 garantieren
y = sqrt(x);
// Kann y² == x annehmen
```

Eine Spezifikation aus Vor- und Nachbedingung etabliert quasi einen *Vertrag*:

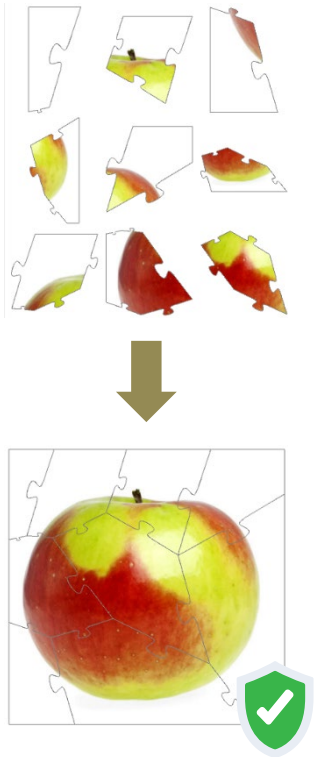
- Entwickler/in muss nicht alle Nutzungen kennen; verlässt sich auf die Vorbedingung
- Nutzer/in muss die Implementation nicht kennen; verlässt sich auf die Nachbedingung

result bezieht sich auf das Ergebnis der Berechnung (den Rückgabewert der Methode)

Nutzen spezifizierter Software

- **Vision**
  - Jedes Modul erhält eine formale Spezifikation
  - Programm entsteht durch Kombination passender Module
  - *Mathematischer Beweis*, dass das gesamte Programm das gewünschte Ergebnis liefert/sich wie gewünscht verhält

- **Für uns nützlich: Denken in Spezifikationen**
  - Kann die Entwicklung lenken → *systematisch* Programmieren
  - Hilft bei der Definition von *Schnittstellen* (z.B. Methoden)
  - Nutzen die Ideen und Konzepte teilweise informell/intuitiv; volle formale Theorie folgt dann in *Formal Methods* (4. Sem.)



Apfel-Puzzle: mathematica.stackexchange.com

Einstieg: Aussagen über Programmsegmente

```
int u = ...; // u hat irgendeinen Wert
int x = 17;
int y = 42;
int z = u + x + y;
```

- Wir wollen eine *logische Aussage* über obige Berechnung tätigen
  - Muss Variable (Endergebnis) `z` involvieren
  - Beziehen sich immer auf einen *bestimmten Programmpunkt* (Zeile)
- Diese Aussage ist die *Nachbedingung* des obigen Codes
  - Beispiel: `z > 0` (gilt dies?)

(Logische) Aussagen

- **Aussage** («**assertion**»): Behauptung, die entweder wahr oder falsch ist
  - Wir fragen dann oft «Ist die Aussage wahr?» oder «Gilt sie?»
  - «Logisch» heisst hier «im Sinne der mathematischen Logik»

- Beispiele:
  - 2 und 3 sind Primzahlen — wahr
  - 13 ist eine gerade Zahl — falsch
  - $x \geq 0$ — hängt von x ab
  - x geteilt durch y ergibt 8 — hängt von x und y ab

(Logische) Aussagen

- Aussage: Behauptung, die entweder wahr oder falsch ist
  - Nicht alle Aussagen sind wahr
  - Sind evtl. nur unter bestimmten Bedingungen wahr (z.B. «x % 5 ist 3»)

- Wichtig ist, dass es *Sinn* macht zu fragen, ob die Aussage wahr oder falsch ist
  - Für uns: Im Kontext der Entwicklung eines Programms

```
int u = ...;
int x = 17;
int y = 42;
int z = u + x + y;
// Gilt «z > 0»?
```

Aussagen in EProg

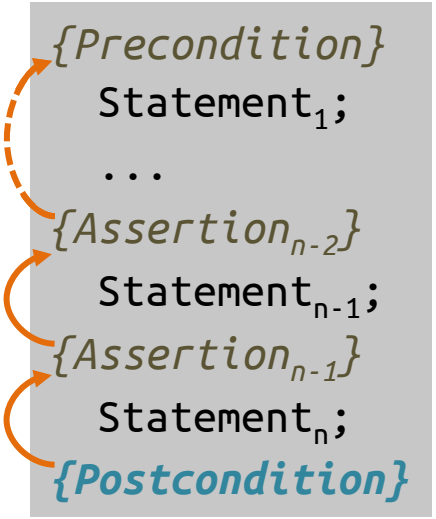
- Wir schreiben Aussagen als *Kommentare* in den Quelltext
  - Kommentare ➜ keine Bedeutung für Java, für uns aber schon
  - Geschweifte Klammern «{…}» sind übliche Notation (Lehrbücher, *Formal Methods*)
  - Die Kommentarzeichen «//» lassen wir aus Platzgründen i.d.R. weg

```
// {x > 0 und x < y}
  z = x * y;
// {z > 0}
```

```
{x > 0 und x < y}
...
```

- Aussagen sind seiteneffektfrei (d.h. ist `i++` keine Aussage)
- Wir nehmen *mathematische Zahlen* an: $\mathbb{Z}$ statt `int`, d.h. keine Überläufe (aber weiterhin ganzzahlige Division, d.h. `5 / 2` ist `2`)

Rückwärtsschliessen: Vorgehen



- **Start:** Wählen einer sinnvollen *Nachbedingung*
- **Schrittweise:** Herleiten der vorherigen Aussage (Assertion$_{i-1}$) durch Einbeziehen des *Effekts* der vorherigen Anweisung (Statement$_i$)
- **Ziel:** Herleiten einer *notwendigen und hinreichenden Vorbedingung*

- Rückwärts = «Welche Vorbedingung braucht mein Code, damit er die gewählten Garantien (Nachbedingung) geben kann?»

Rückwärtsschliessen: Beispiel

```
{u + 17 + 42 > 0}
  x = 17;
{u + x + 42 > 0}
  y = 42;
{u + x + y > 0}
  z = u + x + y;
{z > 0}
```

- Wir wählen als *Nachbedingung* $z > 0$
- Pro Schritt: Wenn nach Statement S_i Aussage A_i gelten soll, welche Aussage A_{i-1} muss dann vor der Ausführung von S_i gelten?
- Dann muss die *Vorbedingung* $u > -59$ gelten (warum?)

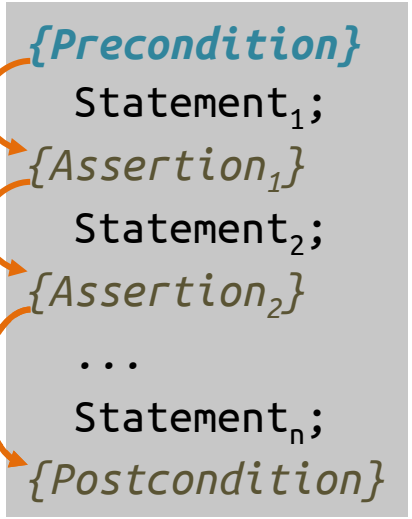
Logisches Schliessen: Vorwärts vs. Rückwärts

- Neben dem vorgestellten *Rückwärtsschliessen* ist auch *Vorwärtsschliessen* möglich. Beide Varianten haben Vor- und Nachteile.

- Wir nutzen in EProg nur das *Rückwärtsschliessen*, da
  - es einfacher für Selbstzuweisungen (x = e, wobei Ausdruck e Variable x selbst enthält, z.B. x = x + 1) ist, was essenziell für Schleifen ist
  - wir dem Thema in EProg nicht zu viel Raum geben möchten
  - und Formal Methods im 4. Semester ebenfalls Rückwärtsschliessen nutzt

- Die nächsten Folien sind daher rein optional

Vorwärtsschliessen: Vorgehen



- **Start:** Wählen (wissen, raten) einer sinnvollen *Vorbedingung* (`Precondition` = $\texttt{Assertion}_0$)
- **Schrittweise:** Herleiten der nächsten Aussage ($\texttt{Assertion}_i$) durch Einbeziehen des *Effekts* der nächsten Anweisung ($\texttt{Statement}_i$)
- **Ziel:** Herleiten einer *gültigen Nachbedingung* (`Postcondition` = $\texttt{Assertion}_n$)

- Vorwärts = «Welche Garantien (Nachbedingung) kann mein Code, unter der gewählten Vorbedingung, geben?»

Vorwärtsschliessen: Beispiel

In Aussagen nutzen wir «==» für (mathematische) Gleichheit

```
{u > 0}
  x = 17;
{u > 0 ∧ x == 17}
  y = 42;
{u > 0 ∧ x == 17 ∧ y == 42}
  z = u + x + y;
{u > 0 ∧ x == 17 ∧ y == 42 ∧ z == u + 17 + 42}
```

- Wir wählen als *Vorbedingung* $u > 0$
- Pro Schritt: Wenn vorher Aussage A_{i-1} gilt, welche Aussage A_i gilt dann nach Ausführung des nächsten Statements S_i?
- Dann gilt die *Nachbedingung* $z > 59$ (warum?)

Vor-/Rückwärts: Gemeinsamkeit

Vorw.

```
{u > 0}
  x = 17;
{u > 0 ∧ x == 17}
  y = 42;
{u > 0 ∧ x == 17 ∧ y == 42}
  z = u + x + y;
{u > 0 ∧ x == 17 ∧ y == 42 ∧ z == u + 17 + 42}
```

Rückw.

```
{u > -59}
  x = 17;
{u + x + 42 > 0}
  y = 42;
{u + x + y > 0}
  z = u + x + y;
{z > 0}
```

Beide Paare aus Vor- und Nachbedingung («Verträge») sind *gültig*:

Wenn vor der Programmausführung die Vorbedingung gilt,
dann gilt nach der Ausführung die Nachbedingung.

Vor-/Rückwärts: Unterschiede

Vorw.

```
{u > 0}
  x = 17;
{u > 0 ∧ x == 17}
  y = 42;
{u > 0 ∧ x == 17 ∧ y == 42}
  z = u + x + y;
{u > 0 ∧ x == 17 ∧ y == 42 ∧ z == u + 17 + 42}
```

Rückw.

```
{u > -59}
  x = 17;
{u + x + 42 > 0}
  y = 42;
{u + x + y > 0}
  z = u + x + y;
{z > 0}
```

Vorwärts

- Erscheint anfangs «natürlicher», da es dem (linearen) Kontrollfluss folgt
- Hält viele Details in den Aussagen fest, die letztendlich irrelevant sind

Rückwärts

- Code «rückwärtslesen» erfordert Übung, aber führt zu einer neuen Sicht auf das Programm
- Grosser praktischer Nutzen: Sie (müssen) verstehen, was jede Anweisung zum Erreichen des gewünschten Endzustands beiträgt

6. Logisches Schliessen

6.1 Motivation und logische Aussagen

6.2 Hoare-Logik und Hoare-Tripel

6.3 Regel für Zuweisungen

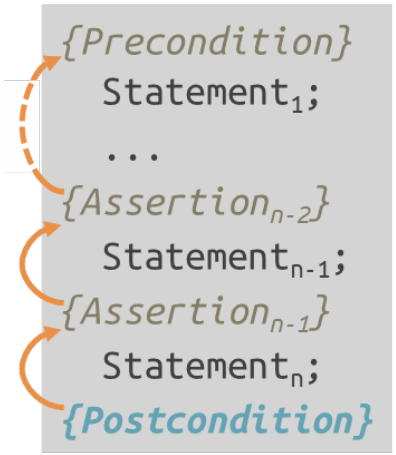
6.4 Regel für Programmsegmente

6.5 Regel für Verzweigungen

6.6 Invarianten und Regel für Schleifen

Hoare-Logik: Hintergrund

- **Hoare-Logik** («**Hoare logic**»): Formales System, um Aussagen über Programme zu beweisen
  - Genutzt von z.B. Amazon, Microsoft, Airbus;
    aber auch von kleineren Firmen in CH, ZH
  - Formale Beweise, plus Tools, in *Formal Methods* (4. Semester)
- Im (Arbeits-)Alltag genügt es oft, weniger detailliert als mit Hoare-Logik zu argumentieren
- Unsere Ziele: Grundlagen einführen, so dass
  1. Sie sehen, wie Programmkorrektheit (relativ) genau argumentiert werden kann
  2. Was die Invarianten aus *Algorithmen & Datenstrukturen* mit *EProg* zu tun haben
  3. Ihnen der Einstieg in *Formal Methods* leichter fällt



Hoare-Tripel: Syntax

- Ein **Hoare-Tripel** («**Hoare triple**»), auch *3-Tupel* genannt, besteht aus zwei Aussagen und einem Programmsegment:

```
{P} S {Q}
```

- Bestandteile eines Tripels
  - P ist die Vorbedingung (Precondition, z.B. *u > -59*)
  - S ist eine Anweisung (Statement; bzw. mehrere Anweisungen)
  - Q ist die Nachbedingung (Postcondition, z.B. *z > -59*)

Hoare-Tripel: Bedeutung

- Ein Hoare-Tripel {P} S {Q} ist **gültig** («**valid**») genau dann wenn:
  - Für *jeden* Zustand, für den P gültig ist, ergibt die Ausführung von S immer einen Zustand, für den Q gültig ist.
  - Informell: Wenn P wahr ist vor der Ausführung von S, dann muss Q nachher wahr sein
- Andernfalls ist das Hoare-Tripel **ungültig** («**invalid**»)

Hoare-Tripel etablieren

- Wie *etablieren* (= *beweisen*) wir ein Hoare-Tripel `{P} S {Q}`?
- Für jede Java-Anweisung gibt es *genaue Regeln*, die eine Vor- und Nachbedingung (für diese Anweisung) in Beziehung setzen
  - Regel für Zuweisungen
  - Regel für aufeinander folgende Anweisungen
  - Regel für Verzweigungen, Schleifen
  - …
- Regeln existieren als Vorwärts- und Rückwärts-Version
  - Wir lernen nur die Rückwärts-Regeln kennen, da relevanter

6. Logisches Schliessen

Hoare-Logik-Regel für Zuweisungen

- **Ziel:** Wir wollen ein Tripel `{P} X = E {Q}` etablieren, also zeigen, dass es **gültig** ist
  - P und Q sind Platzhalter für *beliebige* Aussagen
  - X ist ein Platzhalter für eine *beliebige* Java-Variable
  - E ist ein Platzhalter für einen *beliebigen** Java-Ausdruck

- **Regel** (**Vorgehen**, um **Gültigkeit** zu zeigen)
  1. Erhalte `Q'` durch: Ersetzen aller Vorkommen von X in Q durch E
  2. Zeige, dass P ⇒ `Q'` gilt (Implikation, d.h. dass `Q'` gilt wenn P gilt)

* Ohne Seiteneffekte (also z.B. nicht `++i`)

Erklärung

`{P} X = E {Q}`

1. Bilden von `Q'` durch Ersetzen aller `X` in `Q` durch `E`
2. Zeigen, dass `P` ⇒ `Q'` gilt

Intuition – Warum ergibt diese Regel Sinn?

- **Schritt 1:** Syntaktische Ersetzung von `X` durch `E`
  1. Nach der Zuweisung soll `Q` gelten
  2. Die Zuweisung ändert den Wert von `X` auf `E` – sonst ändert sich nichts

  → Wenn wir die Aussage `Q` mit `E` statt `X` formulieren (= `Q'`), muss diese Aussage bereits vor der Zuweisung gegolten haben

- **Schritt 2:** Implikation
  1. Gerade argumentiert: Wenn `Q'` vor der Zuweisung gilt, gilt danach `Q`
  2. `P` ⇒ `Q'` bedeutet, dass `P` eine stärkere Vorbedingung als `Q'` ist

  → `P` ist daher eine hinreichende Vorbedingung (damit am Ende `Q` gilt)

Beispiel 1

```
{b > 5}
  a = b + 25;
{a > 30}
```

`{P} X = E {Q}`

1. Bilden von `Q'` durch Ersetzen aller `X` in `Q` durch `E`
2. Zeigen, dass `P` ⇒ `Q'` gilt

- Tripel ist **gültig**
  - X ist *a*, E ist *b + 25*
  - Q ist *a > 30*, P ist *b > 5*
  - Q' ist *b + 25 > 30*
  - *(b > 5) ⟹ (b + 25 > 30)* (Beweis: einfache Formelumstellung)

Beispiel 2

```
{b > 0}
  a = b + 25;
{a > 30}
```

```
{P} X = E {Q}
```

1. Bilden von Q' durch Ersetzen aller X in Q durch E
2. Zeigen, dass P ⇒ Q' gilt

- Tripel ist **nicht gültig**
  - X ist *a*, E ist *b + 25*
  - Q ist *a > 30*, P ist *b > 0*
  - Q' ist *b + 25 > 30*
  - P impliziert Q' *nicht* (Gegenbeispiel: wenn b den Wert 2 hat)

Fragen 1

Welche der folgenden Tripel sind (un)gültig?

Bsp4

```
{x > 0}
  y = x + 1;
{y > 1}
```

Bsp5

```
{x > 100}
  y = x + 1;
{y > 1}
```

Bsp6

```
{v ≠ 1}
  w = v * v;
{w ≠ v}
```

Einschub: Welche Aussagen wollen wir?

- Gleicher Code, unterschiedliche (gültige) Tripel

A1
```
{a > 0 ∧ b > 0}
  x = a + b;
{x > 0}
```
vs.
A2
```
{a + b > 0}
  x = a + b;
{x > 0}
```

B1
```
{true}
  x = y + 1;
{x > y}
```
vs.
B2
```
{true}
  x = y + 1;
{x == y + 1}
```

- **Frage:** Gibt es «bessere» und «beste» Aussagen bzw. Tripel?
  - Welches A-Tripel ist «besser»? Welches B-Tripel? Warum?
  - Was könnte eine allgemeine Definition von «besser» sein?
  - Besser für … wen eigentlich?

Stärkere und schwächere Aussagen

- Wir können *stärkere* und *schwächere* Aussagen unterscheiden
  - Wenn $P_1 \Longrightarrow P_2$ gilt, dann ist P_1 stärker als P_2 (und P_2 schwächer als P_1)
- Beispiele

A1
```
{a > 0 ∧ b > 0}
  x = a / b;
{x > 0}
```

vs.

A2
```
{a + b > 0}
  x = a + b;
{x > 0}
```

- $(a > 0 \land b > 0) \Longrightarrow (a + b > 0)$, d.h. A1 hat die stärkere Vorbedingung

B1
```
{true}
  x = y + 1;
{x > y}
```

vs.

B2
```
{true}
  x = y + 1;
{x == y + 1}
```

- $(x == y + 1) \Longrightarrow (x > y)$, d.h. B2 hat die stärkere Nachbedingung

Beste Tripel? Zwei Perspektiven

Perspektive 1: Sie möchten *existierenden Code nutzen*
(d.h. wir nehmen an, dass der Code «fixiert» ist)

```
{P?}
  code
{Q?}
```

- Je *schwächer die Vorbedingung*, desto öfter nutzbar (breiter einsetzbar) ist der Code
  - Beispiel: $\{a + b > 0\}$ erlaubt den Einsatz in mehr Situationen (ist weniger «anspruchsvoll») als $\{a > 0 \wedge b > 0\}$
- Je *stärker die Nachbedingung*, desto mehr Garantien bekommen Sie (desto mehr wissen Sie) als Code-Nutzer/in
  - Beispiel: Mit $\{x == y + 1\}$ wissen Sie, dass der Abstand zwischen x und y genau eins ist, anders als bei $\{x > y\}$

Beste Tripel? Zwei Perspektiven

Perspektive 2: Sie müssen Code *für jemanden* schreiben
(wir nehmen an, dass die Anforderungen/das Tripel «fixiert» sind)

```
{P}
  code?
{Q}
```

- *Stärkere Vorbedingung* ➜ mehr Garantien für Sie, d.h. Ihre Aufgabe wird einfacher
  - Intuitives Beispiel:
    «Berechnen Sie die Quadratwurzel für $x \geq 0$» vs. «… für x»
- *Schwächere Nachbedingung* ➜ Potenziell mehr Wahlfreiheit, daher eventuell einfacher, für Sie
  - Intuitives Beispiel:
    «Endergebnis ist grösser als *n*» vs. «… ist die nächste Primzahl grösser als *n*»

Automatisierung

- Es gibt Algorithmen, um
  1. Die *schwächste Vorbedingung*, für gegebenen Code und Nachbedingung, zu berechnen
  2. Die *stärkste Nachbedingung*, für gegebenen Code und Vorbedingung, zu berechnen

`{P?} code {Q}`

`{P} code {Q?}`

- Die in EProg vorgestellten Regeln ergeben immer die *schwächste Vorbedingung* (da Rückwärtsschliessen)
  - Wir wenden sie händisch/auf Papier an
  - In der Realität braucht es Tool-Support — nicht unser Thema

* Für relative einfache Programme; Problem letztendlich unentscheidbar (→ *Theoretische Informatik*), daher braucht es menschliche Hilfe. Details für EProg nicht relevant.

Fragen 2

Welche der folgenden Tripel sind (un)gültig?

```
{age > 94}
  age = age + 1;
{age >= 95}
```

Bsp7

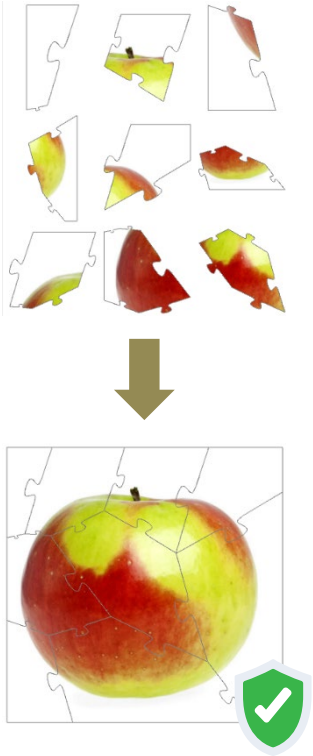
```
{z != 0 && y % z == 0}
  x = y / z;
{y == x * z}
```

Bsp8

6. Logisches Schliessen

Aussagen/Tripel kombinieren

- Beobachtung Programmentwicklung
  - Gegeben zwei Programmsegmente S_1 und S_2, ergibt die *Kombination* S_1`;` S_2 ein grösseres Programm
  - D.h. wir entwickeln grössere Programme aus kleineren (Modularität)
- Analog für Tripel
  - Gegeben $\{P_1\}$ S_1 $\{Q_1\}$ und {P2} S2 {Q2}
  - Wir möchten diese zu {P1} S1`;` S2 {Q2} kombinieren



Hoare-Logik: Aussagen kombinieren

- Wenn zwei Aussagen aufeinander treffen, muss die vorherige Aussage die darauf folgende *implizieren*

```
{P1}
  S1;
{Q1}
{P2}
  S2;
{Q2}
```

Die Kombination erfordert, dass

1. Tripel $\{P_1\}\ S_1\ \{Q_1\}$ gültig ist
2. Tripel $\{P_2\}\ S_2\ \{Q_2\}$ gültig ist
3. $Q_1 \Longrightarrow P_2$ gilt

- Analog
  - `{P1}{P2} S; {Q}` falls $\{P_2\}\ S\ \{Q\}$ und $P_1 \Longrightarrow P_2$
  - `{P} S; {Q1}{Q2}` falls $\{P\}\ S\ \{Q_1\}$ und $Q_1 \Longrightarrow Q_2$

Hoare-Logik: Aussagen kombinieren

$\{P_1\}$
  S_1;
$\{Q_1\}$
$\{P_2\}$
  S_2;
$\{Q_2\}$

Die Kombination links erfordert, dass

1. Tripel $\{P_1\}\ S_1\ \{Q_1\}$ und $\{P_2\}\ S_2\ \{Q_2\}$ gültig sind
2. $Q_1 \Longrightarrow P_2$ gilt

Intuition

- Wenn P_2 ausreicht, um S_2 so auszuführen, dass nachher Q_2 gilt …
- … und wenn Q_1 stärker ist als P_2 …
- … dann reicht auch Q_1 aus, um S_2 erfolgreich auszuführen

Hoare-Logik-Regel für Anweisungsfolgen

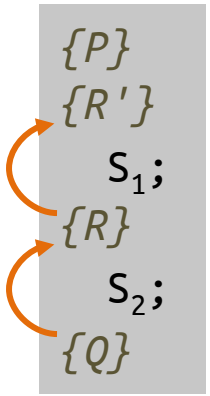
- Das Tripel $\{P\}\ S_1; S_2\ \{Q\}$ ist **gültig**, genau dann wenn
  - Aussage R existiert, so dass
  1. $\{P\}\ S_1\ \{R\}$ gültig ist
  2. $\{R\}\ S_2\ \{Q\}$ gültig ist

Vorgehen für Anweisungsfolgen

- **Situation:** Entscheiden, ob $\{P\}\ S_1; S_2\ \{Q\}$ gültig ist

- **Empfohlenes Vorgehen:**

1. Wende bekannte Regeln an, um Folgendes zu erhalten:

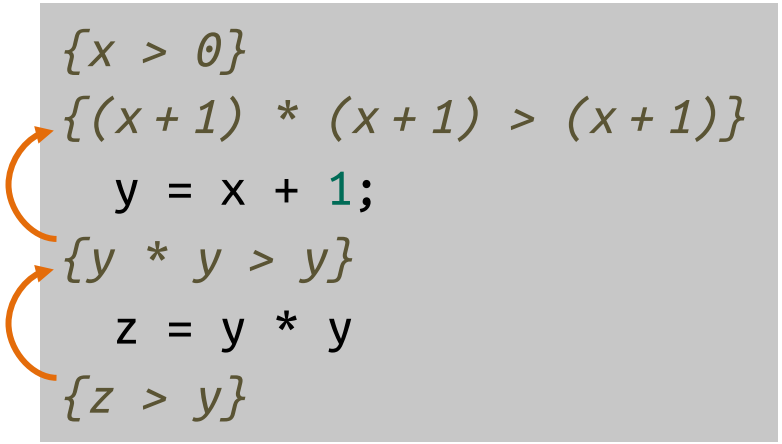


2. Zeige Implikation $P \Longrightarrow R'$

Beispiel

- **Frage:** Ist `{x > 0} y = x + 1; z = y * y {z > y}` gültig?

1. Regeln anwenden:



2. Implikation zeigen:

$(x > 0)$

$\Longrightarrow$

$(x+1) * (x+1) > (x+1)$

hält, denn für $x \geq 2$ ist $x^2 > x$

- **Antwort** daher: Ja

Frage

Welche der folgenden Tripel sind (un)gültig?

```
{true}
  x = y;
  z = x;
{y == z}
```

`true` als Vorbedingung bedeutet «keinerlei Anforderung» (Anfangswerte von x, y und z sind irrelevant)

```
{x == 7 ∧ y == 5}
  t = x;
  y = x;
  x = t;
{y == 7 ∧ x == 5}
```

6. Logisches Schliessen

If-Anweisungen und Aussagen

- **Ziel:** Regel für Tripel {P} `if (b)` S_1 `else` S_2 {Q}

- **Beobachtungen**
  - Ausführung von S_1 wenn b hält
  - Ausführung von S_2 wenn ¬b hält
  - P hält in beiden Fällen vor der Ausführung
  - Q muss in beiden Fällen nachher gelten

```
{P}
  if (b) {
    S1;
  } else {
    S2;
  }
{Q}
```

Hoare-Logik-Regel für if-Anweisungen

- Das Tripel {P} if (b) S_1 else S_2 {Q} ist gültig, genau dann wenn
  1. {P ∧ b} S_1 {Q} gültig ist
  2. {P ∧ ¬b} S_2 {Q} gültig ist

Vorgehen für if-Anweisungen

- **Situation:** Entscheide, ob $\{P\}$ `if (b)` S_1 `else` S_2 $\{Q\}$ gültig ist

- **Empfohlenes Vorgehen:**
  1. Wende bekannte Regeln wie rechts gezeigt an (auf S_1 und S_2)
  2. Zeige notwendige Implikationen
     1. $P \wedge b \implies R_1$
     2. $P \wedge \neg b \implies R_2$

```
if (b) {
  {P ∧ b}
  {R1}
    S1;
  {Q}
} else {
  {P ∧ ¬b}
  {R2}
    S2;
  {Q}
}
```

Beispiel 1: Gültiges Tripel

Zu zeigen: Gültigkeit von $\{true\}$ `if (x < 0) { y = -x; } else { y = x;}` $\{y \geq 0\}$

1. Regeln anwenden:

```
if (x < 0) {
  {true ∧ x < 0}
  {-x ≥ 0}
    y = -x;
  {y ≥ 0}
} else {
  {true ∧ ¬(x < 0)}
  {x ≥ 0}
    y = x;
  {y ≥ 0}
}
```

2. Implikationen zeigen:
   1. $(x < 0) \implies (-x \geq 0)$ ✓
   2. $(x \geq 0) \implies (x \geq 0)$ ✓

Beispiel 2: Ungültiges Tripel

Geändert

Zu zeigen: Gültigkeit von $\{true\}$ `if (x < 0) { y = -x; } else { y = x;}` $\{y \geq 0\}$

1. Regeln anwenden:

```
if (x < 0) {
  {true ∧ x < 0}
  {-x ≥ 0}
    y = -x;
  {y ≥ 0}
} else {
  {true ∧ ¬(x < 0)}
  {x ≥ 0}
    y = x;
  {y ≥ 0}
}
```

Entsprechende Unterschiede

2. Implikationen zeigen:
   1. $(x < 0) \Longrightarrow (-x \geq 0)$ ✓
   2. $(x \geq 0) \Longrightarrow (x \geq 0)$ ✗

Hält nicht mehr (Gegenbeispiel: $x == 0$)

Beispiel 3: Ohne else-Zweig

Zu zeigen: Gültigkeit von $\{x == y \land x \mathrel{!}= 0\}$ `if (a > 1) { a = a * x; }` $\{a \mathrel{!}= y\}$

1. Regeln anwenden:

```
if (a > 1) {
  {x == y ∧ x != 0 ∧ a > 1}
  {a * x != y}
    a = a * x;
  {a != y}
} else {
  {x == y ∧ x != 0 ∧ ¬(a > 1)}
  {a != y}
}
```

Leerer else-Block entspricht fehlendem else-Block

2. Implikationen zeigen:

1. ✓ $(x == y \land x \mathrel{!}= 0 \land a > 1) \Longrightarrow (a * x \mathrel{!}= y)$
2. ✗ $(x == y \land x \mathrel{!}= 0 \land a \leq 1) \Longrightarrow (a \mathrel{!}= y)$

Frage

Welche Tripel sind gültig?

```
{x > 0}
  if (x % 2 == 1) { y = x + 1; }
  else { y = x - 1;}
{y % 2 == 0}
```

```
{x ≠ 0}
  if (0 < y) { x = x + y; }
{x ≠ 0}
```

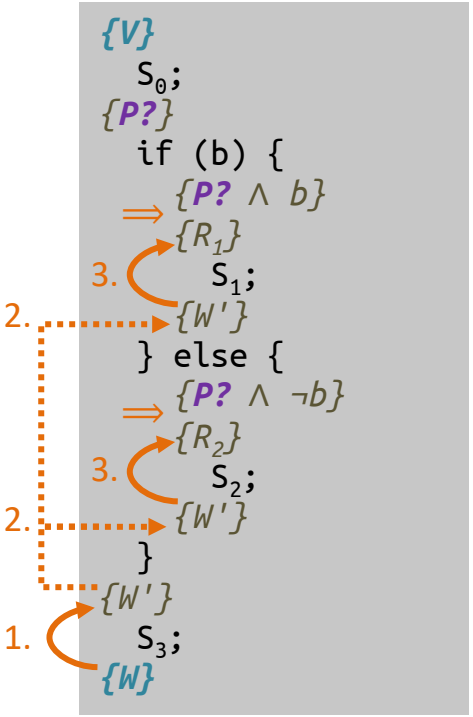
If-Anweisungen im Kontext

Zu zeigen:
Gültigkeit des Tripels

```
{V}
  S0;
  if (b) {
    S1;
  } else {
    S2;
  }
  S3;
{W}
```

Neu: Code vor und nach `if-else`

Bekanntes Vorgehen:



Offene Frage:
Wie bestimmen wir ***P*** (die Vorbedingung der if-Anweisung)?

Systematische Wahl für *P*

```
{P}
if (b) {
  {P ∧ b}
  {R1}
    S1;
  …
} else {
  {P ∧ ¬b}
  {R2}
    S2;
  …
}
…
```

1. **Beobachtung:**
   a. S_1 wird nur ausgeführt, falls b hält
   b. S_2, falls ¬b hält
2. **Konsequenz:**
   a. R_1 muss daher nur halten, wenn b hält
   b. R_2, falls ¬b hält
3. **Systematische Wahl für *P* daher:**

$$(b \implies R_1) \land (\neg b \implies R_2)$$

Systematische Wahl für *P* richtig?

```
{P}
if (b) {
  {P ∧ b}
⇒ {R1}
    S1;
  …
} else {
  {P ∧ ¬b}
⇒ {R2}
    S2;
  …
}
…
```

- Systematische Wahl für ***P***:

$$(b \Longrightarrow R_1) \ \wedge \ (\neg b \Longrightarrow R_2)$$

- **Konsequenz:** Es halten die notwendigen Implikationen

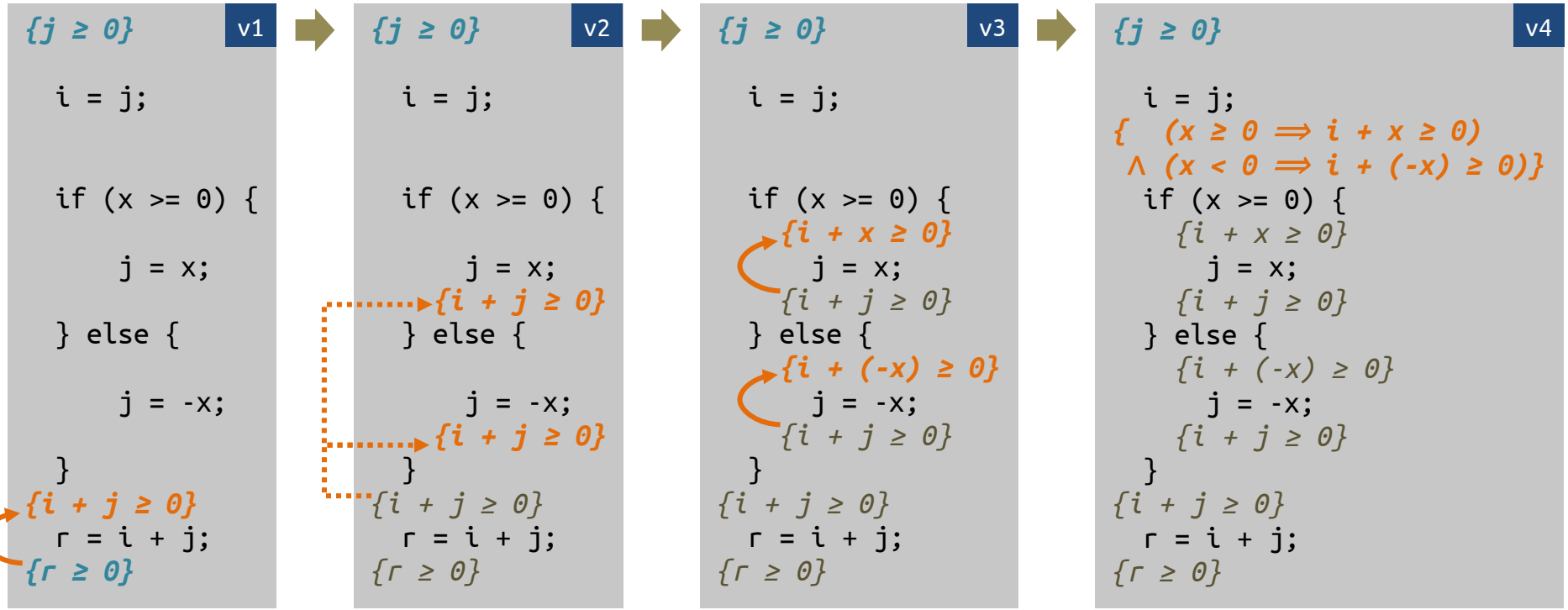
1. $P \wedge b \implies R_1$,
   also $((b \Longrightarrow R_1) \ \wedge \ (\neg b \Longrightarrow R_2)) \wedge b \implies R_1$
2. Analog für $P \wedge \neg b \implies R_2$

Andere Wahlmöglichkeiten für *P*?

```
{P}
if (b) {
  {P ∧ b}
⇒ {R1}
    S1;
  …
} else {
  {P ∧ ¬b}
⇒ {R2}
    S2;
  …
}
…
```

- Alle äquivalenten Aussagen sind nutzbar:
  - $(b \Longrightarrow R_1) \ \wedge \ (\neg b \Longrightarrow R_2)$
  - $(b \wedge R_1) \ \vee \ (\neg b \wedge R_2)$
  - $(R_1 \vee \neg b) \ \wedge \ (R_2 \vee b)$
  - …
- Unterschiedliche Personen finden unterschiedliche syntaktischen Formen intuitiv bzw. leichter verständlich

Beispiel Schritt für Schritt



v4 → v5

```
{j ≥ 0}
{  (x ≥ 0 ⟹ j + x ≥ 0)
 ∧ (x < 0 ⟹ j + (-x) ≥ 0) }
  i = j;
{  (x ≥ 0 ⟹ i + x ≥ 0)
 ∧ (x < 0 ⟹ i + (-x) ≥ 0) }
  if (x >= 0) {
    {i + x ≥ 0}
      j = x;
    {i + j ≥ 0}
  } else {
    {i + (-x) ≥ 0}
      j = -x;
    {i + j ≥ 0}
  }
{i + j ≥ 0}
  r = i + j;
{r ≥ 0}
```

Dann noch zu zeigen:

$$(j \geq 0)$$

$$\Longrightarrow$$

$$(\quad (x \geq 0 \Longrightarrow j + x \geq 0)$$
$$\wedge \ (x < 0 \Longrightarrow j + (-x) \geq 0))$$

Fallunterscheidung (zwei Konjunkte):

1. Wenn $j \geq 0$ und $x \geq 0$, dann $j + x \geq 0$ ✓
2. Wenn $j \geq 0$ und $x < 0$, dann $j + (-x) \geq 0$ ✓

6. Logisches Schliessen

6.1 Motivation und logische Aussagen

6.2 Hoare-Logik und Hoare-Tripel

6.3 Regel für Zuweisungen

6.4 Regel für Programmsegmente

6.5 Regel für Verzweigungen

6.6 Invarianten und Regel für Schleifen

Aussagen über Programm(segment)e

- Bisher: Regeln für die Gültigkeit von Hoare-Tripel für
  - Zuweisungen
  - if-Anweisungen
  - Anweisungssequenzen

- Jetzt: Regel für Schleifen
  - Wie bisher: Regel gesucht, wann Tripel `{P} while (b) S {Q}` gültig
  - *Neue Herausforderung:* Unbekannt, wie oft S genau ausgeführt wird

Vergleich

- Tripel abstrahiert den Effekt *einer* Ausführung

```
{P}
  x = e;
{Q}
```

```
{P}
  if (b) { S1; }
  else { S2; }
{Q}
```

- Tripel muss den Effekt *beliebig vieler* Ausführungen abstrahieren

```
{P}
  while (b) {
    S;
  }
{Q}
```

Beispiel und intuitive Anforderungen

```
{x >= 0}
  y = 0;
  i = 0;
  while (i != x) {
    {???}
      i = i+1;
      y = y+i;
    {???}
  }
{???}
{y == sum(1,x)}
```

Anforderungen:

1. Aussagen (über `i`, `y`) müssen gültig sein, unabhängig davon, wie oft der Schleifenrumpf ausgeführt wird bzw. wurde
   - Für *keine* Iteration (falls `i != x` direkt falsch)
   - Für *eine* Iteration
   - …
   - Für *k* Iterationen (aber *k* ist statisch unbekannt)
2. Aussagen müssen den Effekt beliebig vieler Ausführungen «zusammenfassen», so dass daraus *y == sum(1,x)* folgt

sum(a, b) ist die Summe der Zahlen im Integer-Interval *[a, b]*, d.h. $\sum_{i=a}^{b} i$

Herleitung der Hoare-Logik-Regel

```
{I}
  while (b) {
    {b ∧ I}
      S;
    {I}
  }
{¬b ∧ I}
```

- Schleife betreten → b gilt
- Nach der Schleife → ¬b gilt
- Wenn I vor dem Schleifenrumpf gelten soll
  - muss es bereits vor der Schleife gelten
  - und auch nach dem Schleifenrumpf
- Auch nach der Schleife gilt dann I

Beobachtungen: Anforderung 1 (vorherige Folie) erfüllt — I hält wie gewünscht

- Für keine Iteration (falls b direkt falsch)
- Für eine, zwei, drei, …, k Iteration

Zu Anforderung 2 bald mehr

Hoare-Logik-Regel für while-Schleifen

- Das Tripel `{I} while (b) S {I ∧ ¬b}` ist gültig, genau dann wenn
  1. `{I ∧ b} S {I}` gültig ist

- `I` ist die **(Schleifen-)Invariante** («**(loop) invariant**»)
  - *Invariant ≈ unveränderlich*
  - Hier: Eine Aussage (Eigenschaft), die vor und nach jeder Schleifeniteration gilt

Regel für while-Schleife, da allgemeiner als for-Schleife. Regel geht davon aus, dass `break` & `continue` nicht genutzt werden.

Invariante(n) für unser Beispiel?

{I} while (b) S {I ∧ ¬b}

1. {I ∧ b} S {I}

```
  ...
{i >= 0}
  while (i != x) {
    {i >= 0}
      i = i+1;
      y = y+i;
    {i >= 0}
  }
{i >= 0}
{y == sum(1,x)}
```

- Invariante *i >= 0* (oder einfach *true*) ergibt ein gültiges Tripel …
- … aber ist *zu schwach*, um die gewünschte Aussage *y == sum(1,x)* zu zeigen 😢

Nützliche Invarianten

```
{P}
  setupCode;
{I}
  while (b) {
    {I ∧ b}
      S;
    {I}
  }
{I ∧ ¬b}
{Q}
```

- Schleifenbasierter Code hat oft links gezeigte Struktur
- Eine Aussage `I` ist eine *nützliche* Invariante, wenn Folgendes gilt
  1. *Stark* genug, um die letztendlich gewünschte Nachbedingung `Q` zu zeigen (`I` ⟹ `Q`)*
  2. *Schwach* genug, um
     1. Vom `setupCode` mit Vorbedingung `P` etabliert zu werden
     2. Vom Schleifenrumpf erhalten zu werden (➔ Invariante)

* Falls es noch Code nach der Schleife gibt, kann `Q` rückwärts zu `Q'` transformiert werden und es muss `I` ⟹ `Q'` gelten

Beispiel Schritt für Schritt

{I} while (b) S {I ∧ ¬b}

1. {I ∧ b} S {I}

```
{x >= 0}
  y = 0;
  i = 0;
  while (i != x) {
      i = i+1;
      y = y+i;
  }
{y == sum(1,x)}
```

- Invariante I gleich *y == sum(1,i)*?
- Nun zu zeigen: I ist
  1. Eine Invariante (wird vom Rumpf erhalten)
  2. Eine nützliche Invariante
     1. Schwach genug, um vor der Schleife zu gelten
     2. Stark genug, um am Ende *y == sum(1,x)* zu zeigen

Beispiel Schritt für Schritt

{I} while (b) S {I ∧ ¬b}

1. {I ∧ b} S {I}

```
{x >= 0}
  y = 0;
  i = 0;
  while (i != x) {
⇒ {y == sum(1,i) ∧ i != x}
  {y+i+1 == sum(1,i+1)}
    i = i+1;
    y = y+i;
  {y == sum(1,i)}
  }
{y == sum(1,x)}
```

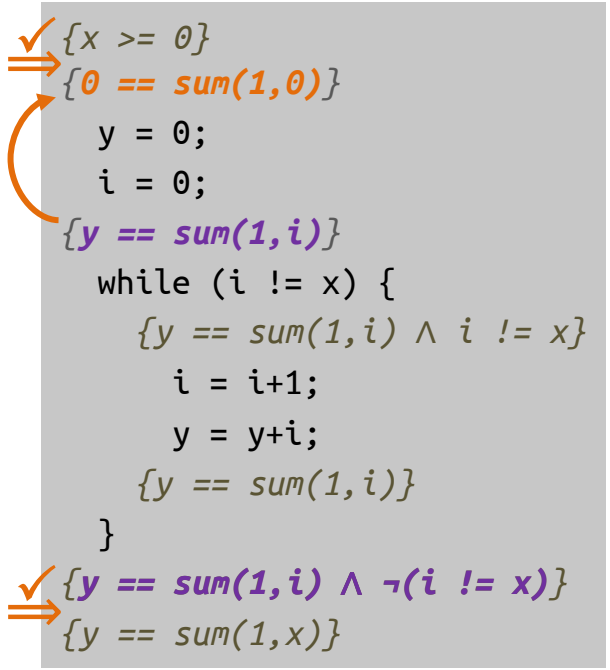
1. Ist $y == sum(1,i)$ eine Invariante?
   1. Aussagen der Regel entsprechend einfügen (rein automatisches Vorgehen)
   2. Zeigen, dass das Rumpf-Tripel gültig ist:

$$
\begin{aligned}
& y == sum(1,i) \wedge i != x \\
\Longrightarrow\ & \text{[Def. sum(); 2. Konjunkt weglassen]} \\
& y == 1 + 2 + \ldots + i \\
\Longrightarrow\ & \text{[Addiere «i+1» auf beiden Seiten]} \\
& y + i + 1 == 1 + 2 + \ldots + i + i + 1 \\
\Longrightarrow\ & \text{[Def. sum()]} \\
& y + i + 1 == sum(1, i + 1)
\end{aligned}
$$

Beispiel Schritt für Schritt

{I} while (b) S {I ∧ ¬b}

1. {I ∧ b} S {I}



2.1 Invariante vor Schleife etablierbar?

$$x >= 0$$
$$\Longrightarrow \quad [\text{da } sum(1,0) == 0]$$
$$0 == sum(1,0)$$

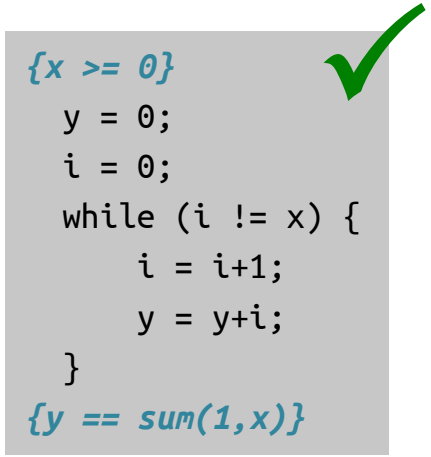
2.2 Invariante impliziert Nachbedingung?

$$y == sum(1,i) \wedge i == x)$$
$$\Longrightarrow y == sum(1,x)$$

Beispiel Schritt für Schritt

Abschluss:

- Mit der gewählten Invariante und den vorherigen Schritten konnten wir zeigen, dass das links stehende Tripel gültig.
- (Leicht) Andere Invarianten bzw. (leicht) anderes Vorgehen könnten ebenfalls zum Erfolg führen



Was denken Sie: Sind Invarianten nur ein Mittel zum Zweck, um die Nachbedingung zu zeigen?

Gleiches Tripel, anderer Code

v1

```
{x >= 0}
  y = 0;
  i = 0;
  while (i != x) {
      i = i+1;
      y = y+i;
  }
{y == sum(1,x)}
```

v2

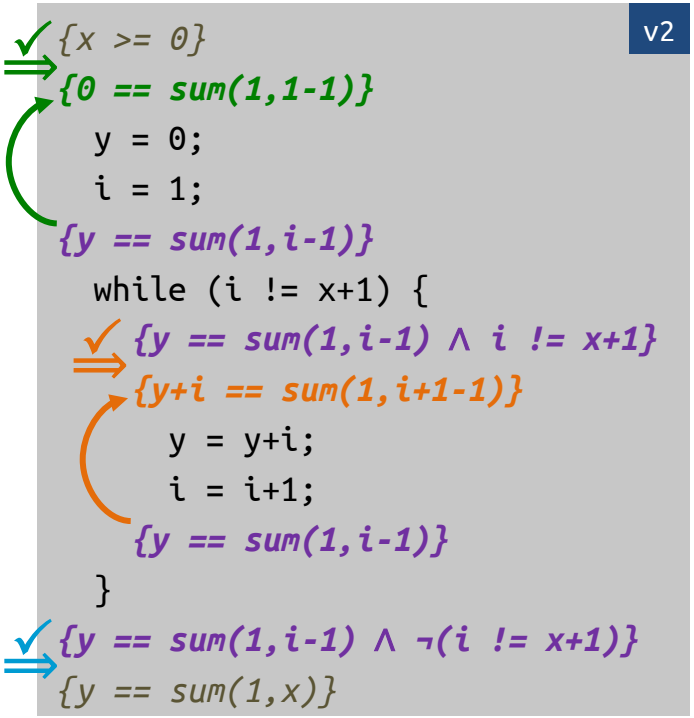
```
{x >= 0}
  y = 0;
  i = 1;
  while (i != x+1) {
    y = y+i;
    i = i+1;
  }
{y == sum(1,x)}
```

Invariante:

$y == sum(1,i)$

Invariante?

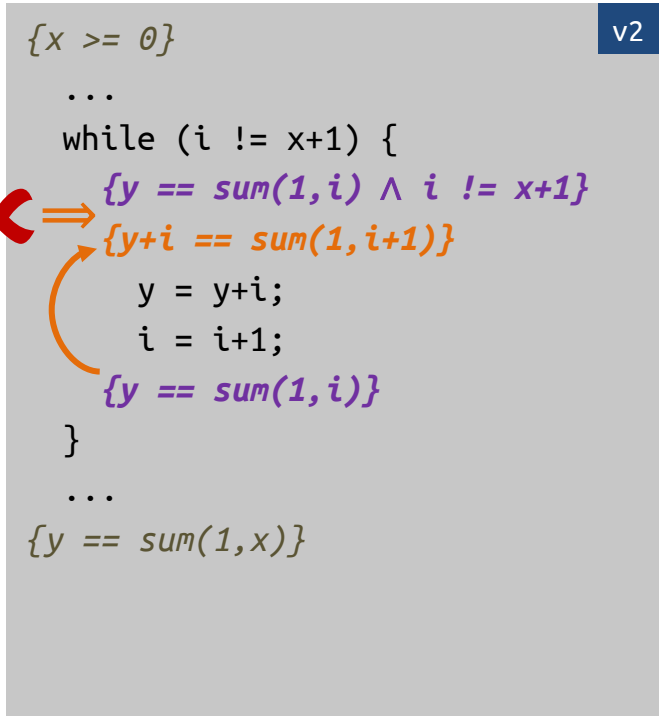
Gleiches Tripel, anderer Code, *andere* Invariante



- Neue Invariante für v2: `y == sum(1,i-1)`
- Aussagen der while-Regel folgend einfügen
- Zeigen: Schleifenrumpf erhält die Invariante (d.h. es ist wirklich eine Invariante)
- Zeigen: Invariante hält vor der Schleife
- Zeigen: Invariante garantiert die gewünschte Nachbedingung

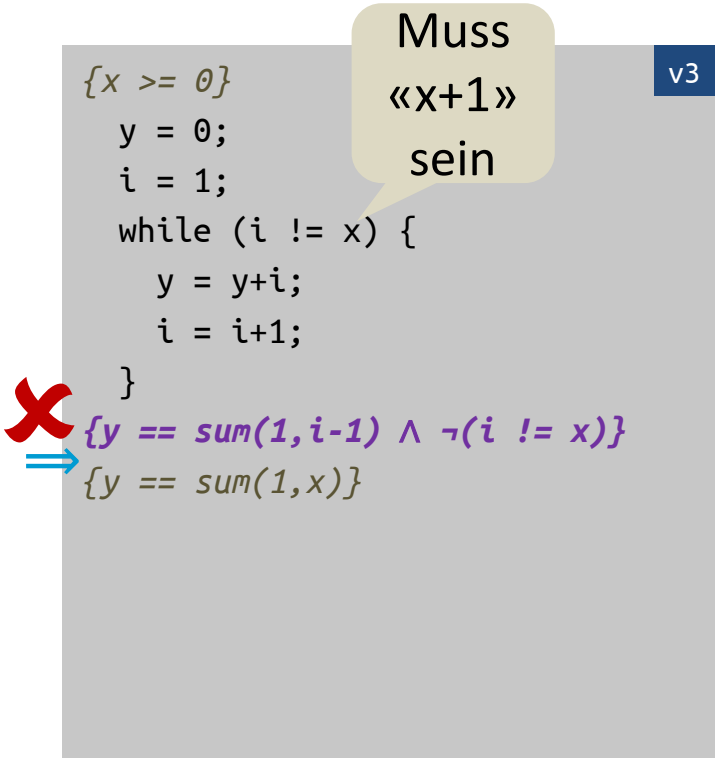
Die drei «Zeigen»-Schritte können in beliebiger Reihenfolge ausgeführt werden

Gleiches Tripel, anderer Code, *gleiche* Invariante



- Invariante für v1 — *y == sum(1,i)* — auch eine Invariante für diesen Code?
- Aussagen der while-Regel folgend einfügen
- Zeigen: Schleifenrumpf erhält die Invariante
  - Schlägt fehl ➜ Invariante für Code v1 ist *keine* Invariante für den geänderten Code v2
  - Wichtig: Das gewünschte Tripel ist weiterhin gültig, mit dem hier gewählten Invarianten-kandidaten können wir dies aber nicht zeigen

Invarianten helfen, Bugs zu finden



- Invariante für v2: $y == sum(1, i-1)$
  - Ist eine Invariante für diesen Code (vom Rumpf erhalten; hier nicht gezeigt)
- Zeigen: Invariante garantiert die gewünschte Nachbedingung → schlägt fehl

$$
\begin{aligned}
& y == sum(1, i-1) \wedge i == x \\
\Longrightarrow\ & y == sum(1, x-1) \\
\not\Longrightarrow\ & y == sum(1, x)
\end{aligned}
$$

  - Gibt einen *Hinweis* auf den Bug im Code: Schleife muss bis x + 1 laufen

Invarianten erfordern Nachdenken

- Invarianten können im Allgemeinen nicht automatisch gefunden werden (unentscheidbares Problem)
  - Intuitiver Grund: Balance zwischen «zu schwach» und «zu stark»
- Programmieren ist eine *kreative* Tätigkeit — das Finden von Invarianten ebenfalls
  - Beides erfordert *konstruktives Nachdenken* (und Erfahrung/Intuition/«Educated Guessing»)
  - Zweigleisiges Denken «in Code» und «in Aussagen» hilft dabei, Programme *systematisch* zu entwickeln
- Wenn es keinen Beweis für ein Tripel gibt, muss entweder der Code oder die Invarianten/Aussagen — oder beides — geändert werden
- Manchmal gibt es verschiedene Invarianten, die alle genügen — so wie es in der Mathematik verschiedene Beweise für dasselbe Theorem geben kann

Eine Strategie für Schleifen

- Es folgt eine *grobe Strategie*, wie wir Schleifen halbwegs systematisch entwickeln können
- *Kein* Algorithmus, *kein* vollständiges Rezept
  - Daher nicht stur zu befolgen
- Aber besser als der vermeintlich schnelle Weg, erst «blind» den Code zu entwickeln und dann die Invariante zu suchen
  - Denn sich Gedanken zur Invariante zu machen bedeutet, sich genau(er) zu überlegen, welche Aufgabe eine Schleife hat

Eine Strategie für Schleifen

1. **Grundsätzliche Idee** (grober Algorithmus) für die Berechnung überlegen
2. Bestimmen Sie dann eine **(Kern-)Invariante** und lassen Sie sich von ihr bei der Programmierung *leiten:* Wie bringt uns *jede* Iteration *näher* ans Ziel?
3. Schreiben Sie einen **Schleifenrumpf**, der die Invariante erhält
4. Bestimmen Sie die **Abbruchbedingung** der Schleife so, dass aus der Invariante und der Abbruchbedingung die Nachbedingung folgt
5. Schreiben Sie den **Initialisierungscode** vor der Schleife so, dass dieser Code die Invariante etabliert

Während der Entwicklung kann es nötig sein, die Invariante anzupassen.

Beispiel

Gesucht: Ganzzahligen Quotienten `q` für positive `x` und `y`,
also `q == x / y` (`x`, `y` und `q` sind `int`-getypt; `x > 0`, `y > 0`)

1. **Idee** für einen Algorithmus
   - Zählen, wie oft `y` in `x` enthalten ist
   - Dazu wiederholt `y` von `x` subtrahieren
2. Eine mögliche **Invariante**
   - Wenn `q` zählt, wie oft `y` bereits subtrahiert wurde …
   - … dann sollte gelten `q * y + r == x`, für einen Rest `r`

Beispiel: Ganzzahliger Quotient

3. Schreiben Sie einen **Schleifenrumpf**, der die Invariante erhält

- Mögliche Invariante (vorherige Folie):
  $q * y + r == x$, für einen Rest r
  - q zählt, wie oft y bereits subtrahiert wurde
- Daraus ableitbar
  - Allgemein: Wenn q hochzählt, muss r kleiner werden
  - Schleife: Wenn q in Einerschritten hochzählt, muss r jeweils um y kleiner werden

```
{x > 0 ∧ y > 0}
  ???
{inv: q * y + r == x}
  while (???) {
    r = r - y;
    q = q + 1;
  }

{q == x / y}
```

Beispiel: Ganzzahliger Quotient

4. **Abbruchbedingung** bestimmen …

- Wie oft y subtrahieren?
- So lange, wie y noch abgezogen werden kann
- D.h. solange `r >= y` gilt

```
{x > 0 ∧ y > 0}
  ???
{inv: q * y + r == x}
  while (r >= y) {
    r = r - y;
    q = q + 1;
  }

{q == x / y}
```

Beispiel: Ganzzahliger Quotient

4. Abbruchbedingung bestimmen, so dass die **Nachbedingung** hält

Gilt $(q * y + r == x \land r < y) \implies (q == x / y)$?

Ja, denn

1. `int`-Division `x/y` «zählt» wie oft ganz `y` in `x` passt; da $r < y$ geht kein ganzes `y` «verloren»
2. Aber: Implikation hält nur, falls $0 \leq r$!

   ➜ Wir müssen die Invariante entsprechend erweitern ...

   .... und zeigen, dass die nun stärkere Invariante weiterhin für den Rumpf gilt

```
{x > 0 ∧ y > 0}
  ???
{inv: q * y + r == x ∧ 0 ≤ r}
  while (r >= y) {
    r = r - y;
    q = q + 1;
  }
{q * y + r == x ∧ 0 ≤ r ∧ r < y}
⟹ {q == x / y}
```

Beispiel: Ganzzahliger Quotient

5. **Initialisierungscode**, der die Invariante etabliert

Zusammenfassung und Abschluss

- Sind von der *Kernidee* auf die *Kerninvariante* gekommen
- Haben uns bei der Programmentwicklung von der Invariante *leiten* lassen
- Haben dabei die Invariante um «technische Details» *erweitern* müssen

```
{x > 0 ∧ y > 0}
  r = x;
  q = 0;
{inv: q * y + r == x ∧ 0 ≤ r}
  while (r >= y) {
    r = r - y;
    q = q + 1;
  }
{q == x / y}
```

Einschub: Partielle Korrektheit

- **Erinnerung:** Hoare-Tripel {P} S {Q} gültig, genau dann wenn
  - *Nach* der Ausführung von Anweisung S Aussage Q gilt
  - Vorausgesetzt dass vor der Ausführung P galt
- Was, wenn wir den Punkt nach S *nicht* erreichen weil S nicht terminiert (z.B. Endlosschleife)?
- Demfalls treffen unsere Hoare-Tripel keine Aussage!
  - Sie drücken *partielle Korrektheit* («*partial correctness*») aus
  - *Totale Korrektheit* («*total correctness*») erfordert Terminierungsbeweis

Ausblick: Terminierung und totale Korrektheit

```
{x > 0 ∧ y ≥ 0}
  r = x;
  q = 0;
{inv: q * y + r == x ∧ 0 ≤ r}
  while (r >= y) {
    r = r - y;
    q = q + 1;
  }
{q == x / y}
```

- Schleife *terminiert nicht*, falls $y == 0$
  - Tripel $\{x > 0 \land y \geq 0\} \ldots \{q == x / y\}$ ist daher partiell korrekt
- Vorbedingung $y > 0$ würde dies verhindern — und uns erlauben Terminierung (und somit totale Korrektheit) zu zeigen
- Idee: Zeigen, dass es nur endlich viele Schleifeniterationen geben kann
  - Ausdruck finden (hier: r), dessen Wert
    1. Eine untere Schranke hat
    2. In jeder Iteration kleiner wird
  - Wird in *Formal Methods* behandelt

Letztes Beispiel: Maximum eines Arrays

- **Gesucht:** Maximaler Wert eines *nicht leeren* `int`-Arrays

- **Vorgehen:** Invariante nutzen, um systematisch zu entwickeln
  - Wir nutzen eine semi-formelle Invariante
  - Nicht ausreichend für formale Beweise …
  - … aber ein pragmatischer Kompromiss

Beispiel: Maximum eines Arrays

Gesucht: Maximaler Wert `max` eines *nicht leeren* Arrays `int[] items`

1. **Idee für einen Algorithmus**
   - In einer Schleife mittels Index `i` über das gesamte Array iterieren
   - Wenn `items[i] > max`, dann wird `items[i]` das neue bisherige Maximum
2. **Eine mögliche Invariante (zwei Konjunkte)**
   1. *max* speichert grössten Wert aus *items[0], …, items[i-1]*
   2. *0 < i ≤ length* (wir kürzen *items.length* mit *length* ab)

Mögliche formale Invariante:
$(\forall\ 1 \leq k < i\ .\ \text{max} \geq \text{item}[k]) \wedge (\exists\ 0 \leq k < i\ .\ \text{max} = \text{item}[k]) \wedge (0 \leq i \leq \text{items.length})$

Beispiel: Maximum eines Arrays

3. Schreiben Sie einen **Schleifenrumpf**, der die Invariante erhält

```
{inv: (max grösster Wert items[0], …, items[i-1]) ∧ (0 < i ≤ length)}
  while (???) {
    if (max < items[i]) {
      max = items[i]; // Invariante nun verletzt
    } else { /* Nichts zu tun */ }
    // max grösster Wert items[0], …, items[i]
    i = i+1; // Invariante wieder hergestellt
  }
```

Mit einer formalen Invariante könnten wir nun mittels Regelanwendungen prüfen, ob der Rumpf sie wirklich erhält

Beispiel: Maximum eines Arrays

4. **Abbruchbedingung** bestimmen, so dass die **Nachbedingung** hält

```
{inv: (max grösster Wert items[0], …, items[i-1]) ∧ (0 < i ≤ length)}
  while (i != items.length) {
    if (max < items[i]) {
      max = items[i];
    }
    i = i+1;
  }
{(max grösster Wert items[0], …, items[i-1]) ∧ (0 < i ≤ length) ∧ (i == length)}
⇒ {(max grösster Wert items[0], …, items[length-1])}
```

Beispiel: Maximum eines Arrays

5. **Initialisierungscode**, der die Invariante etabliert

```
{length > 0}
  int max = items[0];
  int i = 1;
{inv: (max grösster Wert items[0], …, items[i-1]) ∧ (0 < i ≤ length)}
  while (i != items.length) {
    if (max < items[i]) {
      max = items[i];
    }
    i = i+1;
  }
{(max grösster Wert items[0], …, items[length-1])}
```

Beispiel: Maximum eines Arrays

Abschluss

- Haben *systematisch* ein *korrektes* Programm entwickelt, dass das Maximum eines nicht leeren Arrays berechnet

```
{length > 0}
  ...
{(max grösster items-Wert}
```

- Alternative zur Vorbedingung?
  - Schwächere Nachbedingung, z.B.: *{(length > 0) ⟹ (max grösster items-Wert)}*
  - Wert von `max` wäre dann *unspezifiziert*, falls das Array leer sein sollte
  - Code müsste angepasst werden

- Haben ignoriert, dass Referenz `items` auch `null` sein könnte
  - Invariante muss entsprechend gestärkt werden (*items != null*)
  - Vorbedingung ebenfalls

Abschluss «Logisches Schliessen»

- Haben verschiedene Bausteine kennengelernt: logische Aussagen, Hoare-Logik-Regeln, Anwendungsstrategien
- Ziele: Sehen, dass korrekte Programme systematisch entwickelt werden können
  - Idee «Spezifikationen als Verträge zwischen Implementoren und Nutzern»
  - Durch Vor-/Nachbedingungen und Invarianten leiten lassen
  - Sinnvolle (schwächere/stärkere) Aussagen identifizieren
- Spannende Erweiterungen — aber nicht mehr in EProg
  - Referenzsemantik und Framing (Beweise im Kontext von Aliasing und Concurrency)
  - Vererbung und Spezialisierung
  - Terminierung
  - Hyper-Properties, z.B. Determinismus, Security-Eigenschaften
  - …