wp-Kalkül

1. Vor- und Nachbedingungen

Wie beeinflussen Algorithmen in Computer Programmen die Welt? Primär verändern die Anweisungen, die während des Programmlaufs ausgeführt werden, den Inhalt des Computer-Speichers. Gelegentlich haben diese Veränderungen dann weitergehende Wirkung, wie das Versenden von Daten über ein Netzwerk oder eine visuelle Darstellung auf einem Display.

Konzentriert man sich auf die Essenz – also den Zustand des Computer-Speichers vor und nach der Ausführung eines Algorithmus – wird der Algorithmus zu einem *Zustandstransformator*.

Wir betrachten gemeinhin den Computer-Speicher abstrahiert durch *Programm-Variablen* – bzw. den Speicher-Inhalt durch deren *Werte*. Das von einem Algorithmus erwartete Ergebnis lässt sich über die erwarteten Werte entsprechender Variablen charakterisieren. Normalerweise kennt man nicht die genauen Werte der Variablen im Voraus, sondern stellt *Erwartungen an Zusammenhänge* mehrerer Variablen-Werte.

Betrachten wir als einfaches Beispiel eine *Sequenzielle Suche* in einer Liste: [1]

```
i, lst_len = 0, len(lst)
while i != lst_len and lst[i] != val:
   i += 1
```

Nach Ablauf dieses Programms erwarten wir, dass folgende Beziehungen zwischen den Werten der Variablen `i`, `lst` und `val` gelten:

- Der Wert von `i` liegt zwischen 0 und der Länge der Liste `lst` (beide Grenzen eingeschlossen),
- alle Elemente von `lst` mit einem Index kleiner als `i` (wenn es welche gibt) haben einen Wert, der nicht äquivalent zum Wert der Variablen `val` ist,
- der Wert von `i` ist entweder gleich der Länge der Liste `lst` oder das Element von `lst` mit Index `i` ist äquivalent zum Wert von `val`.

Die Formulierungen solch mathematischer Zusammenhänge in deutscher Sprache wirken vielleicht unübersichtlich. Es drängt sich dann eine formale Darstellung mittels *Prädikatenlogik* auf:

$$0 \le i \le \text{len(lst)} \ \wedge \ (\forall\, j : 0 \le j < i : \text{lst}[j] \neq \text{val}) \ \wedge \ (i = \text{len(lst)} \vee \text{lst}[i] = \text{val})$$

Nach Ausführung des Programms sollen die Variablen also diese Bedingung erfüllen. Man bezeichnet eine solche logische Aussage deswegen auch als *Nachbedingung*.

Dafür, dass *nach* dem Programmablauf tatsächlich die Nachbedingung gilt, müssen oft *vor* dem Start des Programms Voraussetzungen geschaffen sein. Auch das lässt sich als Bedingung an bestimmte Variablen formulieren. Man nennt eine solche Aussage dann *Vorbedingung*.

Für das obige Beispiel der sequenziellen Suche ist die Vorbedingung, dass

- die Werte der Variablen `lst` und `val` überhaupt definiert sind, also den Variablen zuvor einmal ein Wert zugewiesen wurde, und
- der Wert von `lst` eine Liste ist oder eine andere indexierbare Datenstruktur mit fixer Länge.

Ob die Liste `lst` leer ist und ob sie den Wert `val` enthält oder nicht, spielt hingegen hier keine Rolle. Das Programm wird die Nachbedingung in allen diesen Fällen etablieren.

Entgegen landläufiger Meinung braucht beim Start eines Programms nicht geprüft zu werden, ob die *Vorbedingung* erfüllt ist. In einzelnen Fällen kann eine solche Prüfung sinnvoll sein, um das Weiterarbeiten mit falschen Daten zu verhindern, in anderen Fällen kann es schlicht zu aufwändig sein. Bei unserem Beispiel der sequenziellen Suche ist es unnötig, denn wenn die *Vorbedingung nicht erfüllt* ist, kommt es ohnehin im Laufe des Programms zu einer *Exception*.

Grundsätzlich gilt: Ist die Vorbedingung dafür, dass ein Programm eine bestimmte Nachbedingung etabliert, nicht erfüllt, ist das Verhalten des Programms nicht definiert. Oder kurz: Das Programm kann dann irgendwas tun.

---

[1] In *Python* führt die Formulierung mit einer *while*-Anweisung zu knapp der doppelten Laufzeit verglichen mit einem Programm mit einer *for*-Schleife und einer Aussprung-Anweisung (*break*). Die Gründe dafür liegen in der Art, wie *Python*-Programme ausgeführt werden. In vielen anderen Programmiersprachen gibt es diesen Laufzeit-Unterschied nicht. – Für die folgenden Betrachtungen sind in Blöcken strukturierte Programme zugänglicher als solche mit *Strukturbrüchen*, wie eben die *break*-Anweisung. Deswegen geben wir hier der Formulierung mit *while* den Vorzug.

Beispiel: Vor- und Nachbedingung von Binary Search

a) Für die *Nachbedingung* dieses Programms findet Ihr darunter ein Prädikat als «Lückentext». Füllt die Lücken aus.

```
lo, hi = -1, len(lst)
while lo + 1 != hi:
   m = (lo + hi) // 2
   if lst[m] < val:
      lo = m
   else:
      hi = m
```

____-1____ ≤ lo ∧ lo __+1 =__ hi ∧ hi ≤ ___len(lst)___

∧ (∀ i : 0 ≤ i < hi : ____lst[i] < val____) ∧ (∀ i : hi ≤ i < len(lst) : ____lst[i] >= val____)

b) Als *Vorbedingung* genügt in diesem Fall nicht, dass die Variablen `lst` und `val` definierte Werte haben und der Wert von `lst` eine Liste ist. Was muss darüber hinaus als Vorbedingung für die binäre Suche gelten? Formuliert diese Vorbedingung zunächst auf Deutsch. Könnt Ihr das gleiche auch mit einem Prädikat ausdrücken?

Da es sich um Binary Search handelt, müssen die Elemente der Liste geordnet sein

2. Vorbedingungen von Zuweisungen berechnen

Vielleicht ist es Euch beim Programmieren auch schon passiert, dass Variablen auf einmal Werte enthielten, die Ihr eigentlich nicht erwartet hattet. In einer solchen Situation habt Ihr Euch wohl auch die Frage gestellt, welche Zustände in Eurem Programm dazu geführt haben könnten.

a) Betrachten wir als ein erstes Beispiel einen einfachen solchen Fall mit einer Variablen `x`. Ihr stellt fest, dass diese den Wert *123* angenommen hat. Die letzte Programmanweisung, die den Wert von `x` verändert hat, lautet:

```
x = x * 40 + 3
```

Welchen Wert muss `x` vor dieser Zuweisung gehabt haben?

3

b) In einem anderen Programm scheint ein Problem zu existieren, das durch einen `AssertionError` angezeigt wird:

```
x = x + 2 * y
assert x >= y
```

Wie kommt es dazu? Welche Bedingung muss für `x` und `y` vor der Zuweisung an `x` gegolten haben?

Damit Assertion Error eintritt muss gelten:
x < y
Somit muss vor Zuweisung zu X auch gelten:
x + 2y < y
x + y < 0
x < -y

Die beiden betrachteten Beispiele zeigen, wie sich zu einer Zuweisung und ihrer Nachbedingung rein formal die Vorbedingung konstruieren lässt: Damit der Wert der Variablen, an die zugewiesen wird, die von der Nachbedingung geforderten Eigenschaften hat, muss der zugewiesene Ausdruck vorher genau diese Eigenschaften gehabt haben.

Die so ermittelte Vorbedingung ist die minimale Anforderung, also die schwächste mögliche Vorbedingung (engl. *weakest precondition*), um die Nachbedingung zu garantieren.

Rein formal lässt sich daraus folgende Regel ableiten:

> Gegeben sei eine Zuweisung der Form `x = E`, wobei `x` eine Variable und `E` ein auswertbarer Ausdruck (*Expression*) ist. Die Nachbedingung dieser Zuweisung sei gegeben als Prädikat *R*.
>
> Man erhält die schwächste Vorbedingung dafür, dass nach der Ausführung der Zuweisung die Nachbedingung *R* gilt, indem man in *R* alle Vorkommen der Variablen `x` durch `E` ersetzt. (Ggf. müssen dabei um `E` Klammern geschrieben werden.)
>
> Wir notieren dies so: $R_{x:=E}$ ist das Prädikat *R* mit allen Vorkommen der Variablen `x` durch `E` ersetzt.

3. Die *wp*-Funktion

Zur klareren Darstellung der folgenden Überlegungen verwenden wir die *wp-Notation* von *Edsger W. Dijkstra*: Wir schreiben für die *w*eakest *p*recondition (schwächste Vorbedingung) eines Programms *S* und einer Nachbedingung *R* kurz *wp(S, R)*. Mathematisch ist *wp(S, R)* eine Funktion, die ein Programmstück *S* und ein Prädikat *R* auf ein Prädikat abbildet – die schwächste Vorbedingung, die garantiert, dass die Ausführung des Programms *S* in einem Zustand endet, der die Bedingung *R* erfüllt.[2]

Für Zuweisungen gilt also: *wp*(`x = E`, *R*) ≡ $R_{x:=E}$. [3]

Beispiele:

a) Die schwächste Vorbedingung dafür, dass die Zuweisung `x = x * x` zur Nachbedingung x = 16 führt, ist:

wp(`x = x * x`, x = 16)

≡

$(x = 16)_{x:=x.x}$

≡

x = 4

b) Die Vorbedingung dafür, dass `x = y % 256 - 128` zur Nachbedingung x ≥ 0 ∧ x < 32 führt, ist:

wp(`x = y % 256 – 128`, x ≥ 0 ∧ x < 32)

≡

(x >= 0 /\ x < 32)x:=y%256-128

≡

y%256-128 >= 0 /\ y%256-128 < 32

y%256 >= /\ y y%256 < 160

c) Die Vorbedingung dafür, dass `i = i + 1` zur Nachbedingung (∀ j : 0 < j < i : a[j-1] ≤ a[j]) führt, ist:

≡

wp(x = x+1, j:0 < j < i : a[j-1] <= a[j])

?

---

[2] Etwas formaler ausgedrückt: wp: Program × Prädiakt → Prädikat, wp(S, R) → schwächste Vorbedingung für *S endet in R*. Die *wp*-Funktion erlaubt uns also, Programme auf die Prädikatenlogik abzubilden.

[3] Wir notieren die Äquivalenz von Prädikaten mit ≡ (und die Implikation entsprechend mit ⇛) als Abgrenzung zum Vergleich von Zahlen und anderen Werten. Letztlich liegt der Unterschied nur in der Bindungsstärke: = und ⇒ binden stärker als ≡ und ⇛.

4. Sequenzen von mehreren Zuweisungen

Das Prinzip, durch Ersetzung zu einer Nachbedingung die Vorbedingung einer Zuweisung zu ermitteln, lässt sich auf Sequenzen von mehreren Zuweisungen ausdehnen. So wie die Zuweisungen Schritt für Schritt „von oben nach unten" ausgeführt werden, lassen sich aus der Nachbedingung Schritt für Schritt die Vorbedingungen „von unten nach oben" berechnen.

Beispiel 1:

Vorbedingung: wp(x=x-1, x=2 v x=-2) =. x=3 v x=-1

```
x = x - 1
```

Hier muss gelten: wp(x=x*x, x=4 v x=-4) = x=2 v x=-2

```
x = x * x
```

Hier muss gelten *wp*(`x = x * x`, *x = 16)* ≡ x^2 = 16

```
x = x * x
```

Nachbedingung: *x* = 16

Das Beispiel lässt eine allgemeine Regel für *Sequenzen von Programm-Anweisungen S₁; S₂* erahnen. Wir benutzen hier das *Semikolon* (;) als Trennsymbol bzw. als *Sequenz-Operator*, um Zeilenumbrüche zu vermeiden. *Python* erlaubt eine solche Schreibweise auch. Sie ist aber unüblich.

$$wp(S_1;\ S_2, R) \equiv wp(S_1, wp(S_2, R))$$

Beispiel 2:

```
i = i + 1; s = s + i
```

Nachbedingung: $s = \frac{i(i+1)}{2}$

Vorbedingung:

$$wp(\texttt{i = i + 1; s = s + i}, s = \tfrac{i(i+1)}{2})$$

$$\equiv$$

$$wp(\texttt{i = i + 1}, wp(\texttt{s = s + i}, s = \tfrac{i(i+1)}{2}))$$

$$\equiv$$

$$wp(\texttt{i = i + 1},\ s+i = i(i+1)/2)$$

$$\equiv$$

$$s + i + 1 = (i+1)(i+2)/2$$

5. Programme mit Fallunterscheidungen

Auch für Programme mit *Fallunterscheidungen* – also **`if`**-Anweisungen lässt sich die schwächste Vorbedingung angeben:

$$wp(\texttt{if } E_b\texttt{: } S_1 \texttt{ else: } S_2, R) \equiv (E_b \Rightarrow wp(S_1, R)) \wedge (\neg E_b \Rightarrow wp(S_2, R))$$

Vorsicht ist allerdings nötig, wenn man **`if`**-Anweisungen ohne einen **`else`**-Teil formuliert. Das bedeutet nicht, dass man den zweiten Teil der Vorbedingung einfach weglassen kann. Vielmehr muss dann, falls die Bedingung in der **`if`**-Anweisung nicht gilt, die Nachbedingung unmittelbar aus der Vorbedingung folgen. Man kann also **`else: pass`** implizit ergänzen, um zu zeigen, dass bei nicht gültiger **`if`**-Bedingung der Zustand der Maschine nicht verändert wird.

Anstelle von $\texttt{if } E_b\texttt{: } S_1$

müsste also eigentlich stehen $\texttt{if } E_b\texttt{: } S_1 \texttt{ else: pass}$

und damit gilt $wp(\texttt{if } E_b\texttt{: } S_1, R)$

$$\equiv wp(\texttt{if } E_b\texttt{: } S_1 \texttt{ else: pass}, R)$$

$$\equiv (E_b \Rightarrow wp(S_1, R)) \wedge (\neg E_b \Rightarrow wp(\texttt{pass}, R))$$

$$\equiv (E_b \Rightarrow wp(S_1, R)) \wedge (\neg E_b \Rightarrow R) \qquad \text{da } wp(\texttt{pass}, P) \equiv P$$

Beispiel 1 die Vorbedingung, dass `if x < y: r = x else: r = y` der Variablen r das Minimum von x und y zuweist, ist immer erfüllt (*true*):

wp(`if x < y: r = x else: r = y`, $r \leq x \wedge r \leq y$)

≡

Beispiel 2 die Vorbedingung, dass `if x < 0: x = -x` x einen nicht-negativen Wert zuweist, ist immer erfüllt (*true*) [4]:

wp(`if x < 0: x = -x`, $x \geq 0$)

≡

6. Programme mit Schleifen

Programme mit Schleifen wiederholen die Schritte innerhalb der Schleifenanweisung mehrere Male. Wir könnten anstelle einer Schleife, die *n*-mal ausgeführt wird, die Anweisungen, die innerhalb der Schleife stehen, *n*-mal untereinanderschreiben. Das ergäbe dann eine Sequenz von Anweisungen und dafür haben wir ja bereits ein Verfahren zur Berechnung der Vorbedingung. Dieser Weg ist aber praktisch nicht umsetzbar, schon gar nicht, wenn *n* eine Variable ist, von deren Wert die Anzahl der Wiederholungen abhängt.

Das *Beispiel 2* aus *Abschnitt 4* oben zeigt einen anderen Weg, Programme mit Schleifen formal zu fassen. Betrachten wir dazu das folgende Programm, an dessen Ende als *Nachbedingung* gelten soll: $s = \frac{n(n+1)}{2}$.

```
i, s = 0, 0
while i != n:
   i = i + 1
   s = s + i
```

Im *Abschnitt 4* haben wir herausgefunden, dass $s = \frac{i(i+1)}{2} \Rrightarrow$ wp(`i = i + 1; s = s + i`, $s = \frac{i(i+1)}{2}$). Gilt also $s = \frac{i(i+1)}{2}$, wenn die Schleife in einen neuen Durchlauf startet, dann gilt dasselbe auch am Ende dieses Durchlaufs durch die Schleife. Die Anweisungen in der Schleife verändern zwar die *Werte* der Variablen `i` und `s`, aber nicht die Beziehung zwischen ihnen.

Gilt aber am Ende eines Schleifendurchlaufs $s = \frac{i(i+1)}{2}$, dann gilt dasselbe natürlich auch gerade wieder am Anfang des nächsten Schleifenlaufes, denn dazwischen werden die Werte der Variablen ja nicht verändert. Und damit ist bereits wieder die *Vorbedingung* dafür erfüllt, dass auch am Ende des *nächsten* Schleifendurchlaufs $s = \frac{i(i+1)}{2}$ gilt, usw.

Die Bedingung $s = \frac{i(i+1)}{2}$ gilt also vor und nach *jedem* Schleifendurchlauf. Sie gilt *invariant*. Eine Bedingung mit dieser Eigenschaft nennt man *Invariante der Schleife* oder kurz *Schleifeninvariante*.

Allerdings ist diese Überlegung nur relevant, wenn tatsächlich diese *Invariante* auch als *Vorbedingung* für die ganze Schleife erfüllt ist. Vor der ersten Ausführung der `while`-Anweisung muss also bereits $s = \frac{i(i+1)}{2}$ gelten.

Berechnet dazu zur Übung: wp(`i, s = 0, 0`, $s = \frac{i(i+1)}{2}$). [5]

[4] Die Einschränkungen der Ganzzahlarithmetik insbesondere für MIN_VALUE wollen wir diesmal ignorieren.

[5] Bei Mehrfach-Zuweisungen kann man einfach gleichzeitig alle Vorkommen aller Variablen, an die zugewiesen wird durch die entsprechenden Expressions ersetzen.

Ihr solltet das Prädikat *0 = 0* oder noch deutlicher *True* erhalten.

Ein Programm, das unter der Vorbedingung *True* immer seine Nachbedingung etabliert, funktioniert voraussetzungslos, denn *True* gilt ja immer.

Damit haben wir also in zwei Schritten gezeigt, dass am Ende unseres Beispielprogramms die Bedingung $s = \frac{i(i+1)}{2}$ gilt:

- Ein erster Beweis-Schritt zeigt, dass nach der Zuweisung `i, s = 0, 0` die Bedingung $s = \frac{i(i+1)}{2}$ gilt.
- Ein zweiter Schritt ist zu zeigen, dass es sich bei $s = \frac{i(i+1)}{2}$ um eine *Schleifeninvariante* handelt. Das heisst: Wenn diese Bedingung vor dem ersten Start der Schleife erfüllt ist, bleibt sie gültig, egal wie oft die Schleife ausgeführt wird.[6]

  Deswegen muss die Invariante nach Ende des letzten Schleifendurchlaufs immer noch gelten.
- Wenn die Schleife nicht mehr weiter ausgeführt wird, weil die Bedingung in der `while`-Anweisung nicht mehr gilt, muss das Gegenteil dieser Schleifen-Bedingung gelten. Zusätzlich zur *Schleifen-Invarianten* gilt daher in unserem Beispiel $i = n$.

  Haben zwei Variablen denselben Wert, kann man sie gegenseitig ersetzen. Damit ergibt sich – als dritten Schritt unseres Beweises – die gewünschte Nachbedingung $s = \frac{n(n+1)}{2}$ aus $(s = \frac{i(i+1)}{2})_{i:=n}$.

Die Vorbedingung für eine Schleife ergibt sich also daraus, dass eine Invariante *Inv* gilt, diese Invariante bei jedem Schleifendurchlauf erhalten bleibt und stark genug ist, dass die Nachbedingung daraus folgt:

$$\text{wp}(\texttt{while } E_b\texttt{: } S, R) \equiv \text{Inv} \quad \wedge \quad (E_b \wedge \text{Inv} \Rightarrow \text{wp}(S, \text{Inv})) \quad \wedge \quad (\neg E_b \wedge \text{Inv} \Rightarrow R)$$ [7]

Zusammenfassung (Schleifen)

Allgemein können wir Schleifen als Programme der folgenden Form betrachten:

```
while Eb:
   S
```

Dabei ist E_b ein logischer Ausdruck, der vor jedem Schleifendurchlauf evaluiert wird und darüber entscheidet, ob die Schleife noch einmal läuft. `S` steht für eine Folge von Anweisungen («Statements») innerhalb der Schleife.

Das betrachtete Beispiel illustriert, wie man formal zu einer Vorbedingung eines solchen Programms mit Hilfe einer Invarianten *Inv* kommen kann. Zu zeigen sind dazu 3 Dinge:

1. *Inv* gilt vor dem ersten Start der Schleife.
2. Aus $E_b \wedge$ *Inv* folgt die Vorbedingung dafür, dass nach Ausführung von `S` wiederum *Inv* gilt.
3. Aus $\neg E_b \wedge$ *Inv* folgt die Nachbedingung.

Schleifeninvarianten können auf verschiedene Arten als Werkzeug zum Nachdenken über Programme verwendet werden. Eine Möglichkeit ist, für gegebene Programme und Nachbedingungen eine ausreichende Vorbedingung zu ermitteln. Dies kann als formaler Beweis der Korrektheit verwendet werden.

Eine andere Verwendung von Schleifeninvarianten ist die gezielte Konstruktion von Programmen. Wir haben Invarianten verwendet, um bei der binären Suche Initialisierungswerte, Abbruchkriterium und Indexverschiebungen zu konstruieren. Manchmal lohnt es sich also, über eine Invariante nachzudenken, um überhaupt einen guten Lösungsansatz zu finden.

---

[6] Präziser formuliert: Die Invariante gilt *immer* am Ende eines Schleifendurchlaufs und damit vor dem Beginn des nächsten Durchlaufs. Während die Anweisungen *innerhalb* der Schleife ausgeführt werden, braucht die Invariante nicht immer gültig zu sein. In unserem Beispiel gilt sie nach der Ausführung von `i = i + 1` zunächst nicht mehr, sondern erst wieder, wenn `s = s + i` danach ebenfalls ausgeführt wurde.

[7] Korrekt müsste es hier *wlp* heissen, für *weakest least precondition*, denn die Aussage gilt nur unter der Bedingung, dass die Schleife überhaupt terminiert. Dass sie das tut, muss separat bewiesen werden.

7. Beispiel: Division mit Rest

Im folgenden Programm seien x und y zwei ganzzahlige positive Werte.

Vorbedingung: $x > 0 \wedge y > 0$

```
r, q = x, 0
while r >= y:
   r, q = r - y, q + 1
```

Nachbedingung: $x = q \cdot y + r \wedge 0 \leq r \wedge r < y$

a) Das zentrale Element dieses Programmes ist eine Schleife. Ein Beweis, dass das Programm unter der angegebenen *Vorbedingung* immer die angegebene *Nachbedingung* erzeugt, also dass gilt:

$$x > 0 \wedge y > 0 \Rrightarrow wp(\texttt{r, q = x, 0; while r >= y: r, q = r - y, q + 1,}\ x = q \cdot y + r \wedge 0 \leq r \wedge r < y)$$

setzt sich aus drei Schritten zusammen.

Schreibt diese drei Schritte konkret (also nicht als allgemeines Muster) als Prädikate bzw. *wp-Funktionen* auf.

Eine hier nützliche *Invariante* lässt sich finden, wenn man die Schleifen-Bedingung `r >= y` mit der Nachbedingung vergleicht und überlegt, dass gelten muss: *Invariante* $\wedge\ r < y \Rrightarrow$ *Nachbedingung*.

b) Führt diese einzelnen Beweis-Schritte aus.