Kapitel 5

Verifikation nach Hoare und Dijkstra

Für funktionale Programme haben wir als Beweismethode bereits die Induktion über die Anzahl der Funktionsaufrufe kennengelernt.

Für imperative Programme mit Zuweisungen wie Java gibt es Verifikationsmethoden, die auf die Programmkonstrukte wie Zuweisung, If-then-else und while zugeschnitten sind. Insbesondere wird der globalen Speicher gesondert modelliert.

Verifikation bedeutet hier den formalen Nachweis der Korrektheit des Programms. Teilweise trennt man den Nachweis der Terminierung vom Beweis des Ein- Ausgabeverhaltens.

Für den Nachweis der Korrektheit muß man zunächst einmal unabhängig vom Programmcode sagen, was das Programm leisten soll: Man braucht Bedingungen (d.h. eine Spezifikation), die i.a. in Prädikatenlogik erster Stufe formuliert sind.

Damit kann man zwei formale System miteinander vergleichen: die Spezifikation und das Programm.

Hierzu benötigt man die Semantik des Programms bzw. der Programmkonstrukt als Axiome bzw. in der Form von Ableitungsregeln.

5.1 Kalkül von Hoare

Zur Vereinfachung nehmen wir an, daß es nur die globale Umgebung gibt, und keine geschachtelten Umgebungen.

Formalisierung:

$$\{P\}\ S\ \{Q\}$$

(wenn P gilt und man S ausführt und S terminiert, dann gilt Q)
P, Q sind Ausdrücke in Prädikatenlogik erster Stufe mit Gleicheit. Diese Ausdräucke (auch Prädikate genannt) haben die globalen Variablen als freie Parameter. P nennt man auch *Vorbedingung*, und Q *Nachbedingung*.

S ist ein Programm(fragment).

Beispiel 1. $\{x = 3\}(x = 1 + x)\{x = 4\}$ ist eine richtige Aussage.

Problematik der Verifikation:

- i.a. sehr aufwendig, auch für automatische Verfahren.
- es gibt kein hinreichend allgemeines Rezept, wie man Aussagen $\{P\}\ S\ \{Q\}$ beweist.
- Die Übersetzung von Ausdrücken in Prädikatenlogik muß exakt sein. Z.B. $x+y$ hat als Übersetzung nicht $x+y$, wenn es einen Überlauf geben kann, sondern eher: $min \leq x + y \leq max; x + y$. D.h. $min \leq x + y \leq max$ sollte in der Umgebung gelten, d.h. die Formel sollte aus der Vorbedingung ableitbar sein.
- automatische Verfahren existieren, sind aber nur für kleinere Programme und Aussagen praktikabel, für größere Probleme (Programme) kann man solange verfeinern (d.h. Zusicherungen und Spezifikation einfügen, bis das Problem klein genug ist.
- Bei größeren Programmen werden handgemachte Beweise schnell unübersichtlich und unverständlich.
- Bei umfangreichen Spezifikationen kann die Spezifikation selbst falsch sein
- Wenn der Beweis nicht gelingt, kann das mehrere Gründe haben:
  1. Beweis zu schwer
  2. Spezifikation falsch
  3. Programm falsch
- die Methoden geben fast keine Hinweise auf mögliche Fehler

Sinnvoll und praktikabel ist eine solche Verifikation nur in kritischen Anwendungen, die nicht zu oft geändert werden.

5.1.1 Methode von Hoare

(für Programme in Java-Syntax)
Die Methode stellt Ableitungsregeln bereit für:

- seiteneffektfreier Ausdruck (leere Anweisung)
- Zuweisung
- Sequenz von Ausdrücken
- Fallunterscheidung
- Iteration
- Anpassung von Prädikaten

Wir nehmen an, daß Ausdrücke keine Seiteneffekte verursachen. Außerdem soll λ die logische Übersetzung von Ausdrücken sein. Zum Beispiel für $B = ((x == 1)\&\&(y < 0))$ ist $\lambda(B) = (x = 1 \wedge y < 0)$. Wir nehmen der Einfachheit halber an, daß keine Bedingungen für den Über- Unterlauf bei arithmetischen Operationen notwendig sind.

leere Anweisung: $\{P\}\ A\ \{P\}$

Zuweisung: $\{P[x \rightarrow \lambda(a)]\}(x = a)\{P\}$

Sequenz: $\dfrac{\{P\}a_1\{R\}, \{R\}a_2\{Q\}}{\{P\}(a_1; a_2)\{Q\}}$

Selektion: $\dfrac{\{P \wedge \lambda(B)\}a_1\{Q\}, \{P \wedge \neg\lambda(B)\}a_2\{Q\}}{\{P\}(\texttt{if}\ (B)\ a_1\texttt{else}\ a_2)\{Q\}}$

(wenn die Auswertung der Bedingung terminiert)

Iteration: $\dfrac{\{P \wedge \lambda(B)\}\ S\ \{P\}}{\{P\}(\texttt{while}(B)\ S)\{P \wedge \neg\lambda(B)\}}$

Wenn B terminiert.

Den Ausdruck P nennt man auch **Schleifeninvariante**

Beispiel einer Herleitungsregel ist

Anpassung von Prädikaten:

$$\frac{(P \Rightarrow Q, \{Q\}\ S\ \{P\}, P \Rightarrow T}{\{P\}\ S\ \{T\}}$$

Beispiel 2. Wir betrachten die Zuweisung $x = x + 1$. Dafür gilt die Regel

$$\{P(x + 1)\}(x = x + 1)\{P(x)\}$$

Konkreter: Das Prädikat sei $P(x) := x = 5$:

$$\{x + 1 = 5\}(x = x + 1)\{x = 5\}$$

Beispiel 3. Eine while-Schleife, die die Addition definiert.

```
while (x  /=   0)
  { x = x - 1;  y =  y + 1};
```

Sei T der obige while-Ausdruck.

Zu zeigen:

$$\{x = m, x \geq 0, y = n\}\ T\ \{x = 0, y = m + n\}$$

Die Schleifeninvariante ist $P = \{x + y = m + n, x \geq 0\}$

Um die Regel für `while` anwenden zu können, muß man die Vorraussetzung zeigen, d.h. die Formel über dem Bruchstrich:

$$\{x+y=m+n, x \geq 0, x \neq 0\}(x=x-1; y=y+1)\{x+y=m+n, x \geq 0\}$$

$$\{x+y=m+n, x \geq 0, x \neq 0\}(x=x-1; y=y+1;)\{x+y=m+n, x \geq 0\}$$

$$\begin{array}{c} \{x+y=m+n, x \geq 0, x \neq 0\} \\ x=x-1 \\ \{x+y=m+n-1, x \geq 0\} \\ y=y+1 \\ \{x+y=m+n, x \geq 0\} \end{array}$$

Dies gilt, da $x \geq 0, x \neq 0$ bedeutet, daß $x > 0$ gilt.

Die Regel für while ergibt jetzt: $\{x+y=m+n, x \geq 0\}T\{x+y=m+n, x \geq 0, x=0\}$ nach Anwendung der while-Regel.

Beachte, daß die Terminierung der while-Schleife noch als extra Beweisaufgabe zu erledigen ist, wenn man zeigen will, daß diese Schleife die Summe berechnet.

Anmerkungen zum Verfahren von Hoare

- Vorwärtsmethode
- Aus Vorbedingung wird Nachbedingung geschlossen
- man kann mehrere Nachbedingungen ableiten
- man muß evtl. überflüssige Aussagen mitschleppen
- liefert partielle Korrektheit, die Terminierung muß gesondert bewiesen werden.
- Das Berechnen (Erraten) von Schleifeninvarianten ist schwierig

5.2 Methode von Dijkstra

Dieses Verfahren baut auf der Methode von Hoare auf, aber verbessert das Verfahren durch zielgerichtete Berechnung von Vorbedingungen. Es ist eine Rückwärtsmethode:
Aus einer Nachbedingung Q und einem Programmfragment S wird die schwächste (d.h. die bestmögliche) Vorbedingung errechnet: $wp(S,Q)$ (weakest precondition)

Ziel: Gilt $wp(S,Q)$ vor Ausführung von S, so

- terminiert S
- gilt Q danach

Der Terminierungsnachweis wird durch die Dijkstra Methode nur in trivialen Fällen geliefert. Wie wir sehen werden, versagt die Methode des automatischen Berechnens der Vorbedingung bei Schleifen.

Man kann mit dem Dijkstra- Verfahren den Nachweis von $\{P\}\ S\ \{Q\}$ schrittweise durchführen:

1. Berechne $wp(S, Q)$
2. Zeige $P \Rightarrow wp(S, Q)$

5.3 Regeln zur Berechnung von $wp(.)$

5.3.1 Leere Anweisung

Sei A die leere Anweisung. Bzw. eine Anweisung, die die Umgebung nicht ändert und terminiert. Dann ist

$$wp(A, Q) = Q$$

5.3.2 Zuweisung

$x = E$, wobei E seiteneffektfreier, terminierender Ausdruck ist. Dann ist

$$wp((x = E), Q(x)) = Q(\lambda(E))$$

z.B. sei $E = (x + 10)$. Dann ist $\lambda(E) = (x + 10)$

$$wp((x = x + 10), \{x \geq 0\}) = x + 10 \geq 0$$

5.3.3 Sequenz von Anweisungen:

$$wp((S_1; S_2), Q) = wp(S_1, wp(S_2, Q))$$

Begründung: Damit Q nach S_2 gilt, muß $wp(S_2, Q)$ vor S_2 gelten. Damit $wp(S_2, Q)$ nach S_1 gilt, muß $wp(S_1, wp(S_2, Q))$ vor S_1 gelten.

Verallgemeinerung:

$$wp((S_1; \ldots; S_n), Q) = wp(S_1, wp(S_2, wp(\ldots, wp(S_n, Q) \ldots)))$$

Beispiel 4. $wp((x = x + 1; x = x - 1), \{x > 1\}$

1. Berechne $wp((x = x - 1), x > 1) = \{x - 1 > 1\}$
2. Berechne $wp(x = x + 1, \{x - 1 > 1\} = \{((x + 1) - 1) > 1\} = \{x > 1\}$

5.3.4 If-then-else; Fallunterscheidung

Wir betrachten den Programmausdruck (`if` (B) S_1 `else` S_2). Voraussetzung sei, daß B seiteneffektfrei und terminierend ist.

$$wp((\texttt{if}\ (B)\ S_1\ \texttt{else}\ S_2), Q) = (\lambda(B) \Rightarrow wp(S_1, Q)) \wedge (\neg\lambda(B) \Rightarrow wp(S_2, Q))$$

Logisch äquivalent dazu ist die leichter zu verwendende Aussage:

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

Beispiel 5. Wir berechnen für das Fragment $S = (\texttt{if}\ (x > 0); y = x\ \texttt{else}\ y = -x$ und drei verschiedene Nachbedingungen die Vorbedingung.

$$\begin{array}{cl} & wp((\texttt{if}(x > 0); y = x\ \texttt{else}\ y = -x, \{y \geq 2\}) \\ = & (x > 0 \Rightarrow wp(y = x, \{y \geq 2\}) \wedge \neg(x > 0) \Rightarrow wp(y = -x, \{y \geq 2\}) \\ = & (x > 0 \Rightarrow x \geq 2 \wedge \neg(x > 0) \Rightarrow (-x \geq 2) \\ & \text{logische und arithmetische Umformung ergibt:} \\ = & (x \geq 2 \vee x \leq -2) \end{array}$$

Die zweite Variante der wp-berechnung für if-then-else hätte die logische Umformung erspart.

$$\begin{array}{cl} & wp((\texttt{if}(x > 0); y = x\ \texttt{else}\ y = -x, \{y \geq 0\}) \\ = & (x > 0 \Rightarrow wp(y = x, \{y \geq 0\}) \wedge \neg(x > 0) \Rightarrow wp(y = -x, \{y \geq 0\}) \\ = & (x > 0 \Rightarrow x \geq 0 \wedge \neg(x > 0) \Rightarrow (-x \geq 0) \\ & \text{logische und arithmetische Umformung ergibt:} \\ = & (\texttt{True}) \end{array}$$

d.h. Die Nachbedingung gilt immer, unabhängig von der Vorbedingung.

$$\begin{array}{cl} & wp((\texttt{if}(x > 0); y = x\ \texttt{else}\ y = -x, \{y < 0\}) \\ = & (x > 0 \Rightarrow wp(y = x, \{y < 0\}) \wedge \neg(x > 0) \Rightarrow wp(y = -x, \{y < 0\}) \\ = & (x > 0 \Rightarrow x < 0 \wedge \neg(x > 0) \Rightarrow (-x < 0) \\ & \text{logische und arithmetische Umformung ergibt:} \\ = & (\texttt{False}) \end{array}$$

Da $\{P\}\ S\ \{Q\}$ den Status einer Implikation hat, bedeutet das, daß die Nachbedingung im letzten Fall nicht erfüllt werden kann.

5.3.5 Iteration

Der Ausdruck sei (`while` (B) S), wobei B seiteneffektfrei und terminierend sei.

Sei Q die Nachbedingung.

Man kann folgende "unendliche VerOderung" herleiten:

	B ist falsch
oder	B ist richtig, und B falsch nach einmal S
oder	B ist richtig, B richtig S, und B falsch nach $S; S$
oder	...
...	

Dies ergibt eine “unendliche “ Formel für $wp((\texttt{while}\ (B)\ S), Q)$

$$\begin{array}{ll} = & (\neg\lambda(B) \wedge Q) \\ \vee & (\lambda(B) \wedge wp(S, \neg\lambda(B)) \wedge Q)) \\ \vee & (\lambda(B) \wedge wp(S, \lambda(B)) \wedge wp((S;S), \neg\lambda(B)) \wedge Q)) \\ \vee & \ldots \end{array}$$

Folgendes Verfahren kann verwendet werden: $wp(.) = H_0 \vee H_1 \vee H_2 \vee \ldots$
mit $H_0 = (\neg\lambda(B) \wedge Q)$
$H_i = \lambda(B) \wedge wp(S, H_{i-1})$ für ≥ 1

Für vorgegebenes S ist somit folgendes zu tun :

- Berechnen von H_i,
- Erkennen von Regelmäßigkeiten in H_i,
- Raten einer “äquivalenten“ Formel zu $H_0 \vee H_1 \vee H_2 \vee \ldots$

Beachte, daß es keine unendlichen Disjunktionen in der Prädikatenlogik gibt. Die logischen Regeln für solche Formeln sind anders!.

Möglichkeit einer Lösung:

Sei $\vec{x}$ der Vektor aller durch S geänderten Variablen. Um

$$wp((\texttt{while}\ (B(\vec{x}))\ S), Q(\vec{x}))$$

zu berechnen, ermittelt man eine allgemeine Formel $\vec{F}(\vec{x}, j)$, die den Zustand nach j Schleifendurchläufen (von S) beschreibt.

Dann gilt:

$$wp((\texttt{while}\ B(\vec{x})\ S), Q(\vec{x})) = Q(\vec{F}(\vec{x}, k)$$

wobei k ein expliziter Ausdruck ist, der die Anzahl der Schleifendurchläufe beschreibt.

Begründung. Es gilt:
$wp(S, Q(\vec{x})) = Q(\vec{F}(\vec{x}, 1))$
$wp((\underbrace{S;\ S; \ldots; S}_{k}), Q(\vec{x})) = Q(F(\vec{x}, k)$

Probleme: Die Formel für $\vec{F}(\cdot,\cdot)$ ist normalerweise schwer zu raten, häufig findet man keine keine geschlossene (logische) Darstellung. Ein automatische Erzeugung ist i.a. nicht möglich.

Beispiel 6. while-Schleife, die die Addition definiert.

```
while (x  /=   0)
  { x = x - 1;  y =  y + 1};
```

Die Nachbedingung sei $Q(x,y) = \{x + y = m\}$. Raten einer Funktion $\vec{F}$ ergibt: $F((x_0, y_0), j) = (x_0 - j, y_0 + j)$

Die Anzahl der Durchläufe ist x_0

$wp = Q(\vec{F}((x_0, y_0), x_0)) = Q(0, x_0 + y_0) = \{x_0 + y_0 = m\}$

Man sieht auch, daß das man beim Raten der Funktion $\vec{F}$ die while-Schleife zunächst in eine funktionale Form überführt.

5.3.6 Anmerkungen zum Verfahren von Dijkstra

- ist Rückwärtsmethode von der Nachbedingung auf Vorbedingung
- Beweis der totalen Korrektheit ist implizit
- Problem: Vereinfachung der unendlichen Formel für while-Ausdrücke.
- Für while-Ausdrücke muß man i.a. genau wissen, was berechnet wird. Man muß auch angeben können, wie oft die Schleife durchlaufen wird.
- wenn Programm oder Spezifikation falsch ist, und man dies nicht weiß, so liefert diese Methode wenig Hinweise