Formale Verifikation mittels Hoare-Logik

Sebastian Wild und Markus E. Nebel

13. März 2012

Inhaltsverzeichnis

1. Einführung

Hat man von einem Programm eine formale Spezifikation, d. h. eine mathematisch präzise Beschreibung der zu berechnenden Eingabe-Ausgabe-Relation, so kann man versuchen nachzuweisen, dass ein gegebenes Programm diese Spezifikation erfüllt, also *korrekt* ist.[1] Eine Möglichkeit, das recht systematisch zu tun, stellt der HOARE-Kalkül dar. Dieser gibt für Anweisungen Vor- und Nachbedingungen an, die wie folgt zu verstehen sind: *Wenn* der Programmzustand vor Ausführung der Anweisung die Vorbedingung erfüllt, *dann* erfüllt der Zustand nach Ausführung der Anweisung die Nachbedingung.

Der HOARE-Kalkül erlaubt damit Aussagen über die sogenannte *partielle Korrektheit*: Wenn das Programm in einem adäquaten – d. h. die Vorbedingung erfüllenden – Zustand gestartet wird und es *terminiert*, dann erfüllt der Endzustand die Nachbedingung.

Weil Beweise im HOARE-Kalkül rein auf dem Code arbeiten, nennt man sie *statisch*. Andere Verfahren versuchen Aussagen über mögliche/erlaubte *Läufe* eines Programms zu treffen. Damit kann man auch nicht-terminierende Programme sinnvoll beschreiben. Der große Vorteil der HOARE-Verifikation ist dagegen, dass sie direkt im Code festgehalten werden kann. Auch sind die dabei gefundenen Schleifeninvarianten als Dokumentation eine wertvolle Verständnishilfe, da sie oft die algorithmische Idee eines Programms offenbaren.

[1] Natürlich braucht man auch eine formale Semantik der Programmiersprache! Für unsere Beispiele aus dem Buch ist diese gegeben, für viele in der Praxis verwendete Sprachen sieht das anders aus: Die Semantik von Java und C ist beispielsweise in natürlicher Sprache formuliert – Uneindeutigkeiten vorprogrammiert. Überdies ist im C Standard etwa die Auswertungsreihenfolge von Teilausdrücken nicht vollständig spezifiziert.

2. Das Beispielprogramm

Das Beispiel ist ein (mit Absicht) mäßig sinnvolles Programm zur Berechnung von $\sum_{i=1}^{n} i \cdot \binom{n}{i}$. Es ist weder eine besonders elegante, noch gut verständlich Lösung und wohl auch nicht der effizienteste Weg, diese Größe zu berechnen.[2] Dafür werden im Code die beiden wesentlichen Konstrukte auftreten, welche die Anwendung des HOARE-Kalküls erschweren: geschachtelte Schleifen und aufeinander folgende Schleifen.

Wir verwenden die folgende Identität

$$\sum_{i=1}^{n} i \cdot \binom{n}{i} = \sum_{i=1}^{n} i \cdot \frac{n!}{i!(n-i)!} = \sum_{i=1}^{n} i \cdot \frac{n \cdot (n-1) \cdots (n-i+1)}{i!} = \sum_{i=1}^{n} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j}.$$

In der letzten Darstellung lässt sich die Berechnung straight-forward in der deklarativen Programmiersprache aus dem Buch implementieren. Eine äußere Schleife repräsentiert die Summe, zwei innere Schleifen berechnen nacheinander die Produkte. Dafür braucht man Variablen s bzw. b, die die bisher berechnete Summe bzw. den bisherigen Teil des Binomialkoeffizienten zwischenspeichern. Das Ergebnis steht nach Ausführen des Programms SUM-OF-BINOMIALS in Variable s.

SUM-OF-BINOMIALS(n)

```
 1  k := 1;
 2  s := 0;
 3  While (k < n+1) do
 4      b := 1;
 5      l := n-k+1;
 6      While (l < n+1) do
 7          b := b*l;
 8          l := l+1;
 9      end;
10      l := 2;
11      While (l < k+1) do
12          b := b/l;
13          l := l+1;
14      end;
15      s := s+(k*b);
16      k := k+1;
17  end;
```

Dass SUM-OF-BINOMIALS grob “das Richtige” tut, scheint offensichtlich. Man neigt aber schon bei solch kleinen Programmen dazu, Randfälle zu vergessen.

Die Verifikation mittels HOARE-Kalkül liefert uns ›automatisch‹ die allgemeinste Vorbedingung, damit SUM-OF-BINOMIALS sich korrekt verhält und zeigt damit mögliche Randfälle auf.

Können Sie erkennen, was wir für den Anfangszustand fordern müssen, damit nach Ausführung von SUM-OF-BINOMIALS in s die Summe $\sum_{i=1}^{n} i \cdot \binom{n}{i}$ steht?

[2] So kann man für $n \geq 0$ zeigen, dass $\sum_{i=1}^{n} i\binom{n}{i} = n \cdot 2^{n-1}$, was sofort eine effizientere Methode bietet, diese Summe zu bestimmen.

3. Nachweis der partiellen Korrektheit

Dieser Abschnitt beschreibt das »Handwerk«, Korrektheitsbeweise mit dem HOARE-Kalkül zu führen. Logik-Interessierte werden einige Lücken zwischen der formalen Definition des Kalküls und der Anwendung hier entdecken. Insbesondere sei gewarnt:

! Wir kümmern uns hier **nicht** um die Logik, in der die Vor- und Nachbedingungen formuliert sind, sondern erlauben uns einfach, »in der Mathematik übliche« Formulierungen zu verwenden. !

Einige der Lücken werden im Abschnitt »Details für Logiker« diskutiert.

3.1. Verfahren

Das folgende Verfahren beschreibt Schritt für Schritt, wie man von einer gewünschten Nachbedingung zur allgemeinsten Vorbedingung kommt, die sicherstellt, dass nach Ausführung des Programms die Nachbedingung gilt – falls eine solche existiert.

Die *Soundness* des HOARE-Kalküls garantiert uns dabei, dass wir niemals *false positives* herleiten.

(1) Gewünschte Nachbedingung formulieren – *ohne Spezifikation keine Verifikation*.
Diese Nachbedingung schreiben wir ganz ans Ende des Programms.

(2) Wir arbeiten uns im Code von **unten nach oben** vor. Sei A die aktuelle Zusicherung, dann unterscheiden wir nach dem Inhalt der Zeile über A:

- **Zuweisung**
  Die Vorbedingung ergibt sich aus der Assign-Regel: Alle Vorkommen der zugewiesenen Variable in A durch die rechte Seite ersetzen.
- **While-Schleife**
  Hier müssen wir eine passende **Invariante** finden, um die Schleifenregel anwenden zu können (siehe Abbildung 1).

Die While-Schleifenregel

Für *jede* Zusicherung INV, für die wir $\{C \wedge INV\}\text{Body}\{INV\}$ im HOARE-Kalkül herleiten können, ist auch $\{INV\}$**While**(C) **Do** Body **End**$\{\neg C \wedge INV\}$ herleitbar.

Beachte, dass die Invariante INV **vor** und **nach** der Schleife, sowie an **Beginn** und **Ende** des Rumpfes gelten muss.

(1) $\{A_1 \equiv INV\}$
1 **While** (C) **do**
(2) $\{A_2 \equiv C \wedge INV\}$
2 Body
(3) $\{A_3 \equiv INV\}$
3 **end**;
(4) $\{A_4 \equiv \neg C \wedge INV\}$

Abbildung 1: Die While-Schleifenregel.

Im Allgemeinen ist das sehr schwierig.[3] Für den häufigen Spezialfall einer Zählschleife hilft die For-Regel aus Abbildung 2.

Sei C die Schleifenbedingung und INV die gewählte Invariante. Dann müssen wir

(a) sicherstellen, dass aus $\neg C \wedge INV$ auch A folgt, sowie

(b) INV ans Ende des Schleifenrumpfes,

(c) $INV \wedge C$ an den Anfang des Rumpfes und schließlich

(d) INV vor das While kopieren.

- Das Ende einer **If-Then-Else-Anweisung**
  Kopiere A an das Ende des Then-Blocks und an das Ende des Else-Blocks. (siehe Abbildung 3)

- Der Kopf einer **If-Then-Else-Anweisung**
  Schreibe $(C \rightarrow VB_T) \wedge (\neg C \rightarrow VB_E)$ als Zusicherung über den Kopf der If-Anweisung, wobei VB_T die Vorbedingung des Then-Blocks und VB_E die des Else-Blocks ist. (siehe Abbildung 3)

[3] Auch in der ursprünglichen If-Then-Else-Regel kam eine unbekannte Bedingung VB vor. Der Trick, der in Abbildung 3 verwendet wird, um diese aus den Bedingungen der Then- und Else-Blöcke zu konstruieren, funktioniert für die Schleifen-Regel leider nicht. Zum Einen gilt nach der Schleife $NB := \neg C \wedge INV$ und nach dem Rumpf INV, sodass wir nicht einfach die Bedingung kopieren können. Zum Anderen würde zwar z. B. $INV := \neg C \rightarrow NB$ sicherstellen, dass $\neg C \wedge INV \equiv NB$, aber in der schwächsten möglichen Vorbedingung für den Rumpf resultieren: $C \wedge INV \equiv C \wedge (\neg C \rightarrow NB) \equiv C$. Das wird im Allgemeinen nicht ausreichen, um die Nachbedingung des Rumpfes zu garantieren.

Die For-Schleifen-Regel

Wir betrachten eine While-Schleife, die eine For-Schleife (mit der erwarteten Semantik) simuliert:

For $i := i_0$ **to** $i_{\max}$ **do** Body **end**;

Dann können wir eine stärkste Schleifen-Invariante direkt angeben:

$$INV = i_0 \le i \le \max\{i_0, i_{\max}+1\} \ \wedge \ P(i-1)$$

wobei P eine von i abhängige Aussage über den aktuellen Zustand mit

$$\{i_0 \le i \le i_{\max} \wedge P(i-1)\} \text{ Body } \{i_0 \le i \le i_{\max} \wedge P(i)\} \qquad \text{(step)}$$

ist, d. h. Body macht einen Schritt im Sinne P und erhält die Rahmenbedingung für i. (Achtung: Wir verbieten nicht per se, dass Body i verändert!)

Typischerweise enthält P Aussagen über weitere Variablen, die Body nutzt um eine iterative Berechnung zu realisieren, etwa teilweise berechnete Summen.

(1) $\{A_1 \equiv P(i_0-1)\}$

1 $i := i_0$;

(2) $\{A_2 \equiv \underbrace{i_0 \le i \le \max\{i_0, i_{\max}+1\} \ \wedge \ P(i-1)}_{=INV}\}$

2 **While** $(i < i_{\max}+1)$ **do**

(3) $\{A_3 \equiv i < i_{\max}+1 \ \wedge \ i_0 \le i \le \max\{i_0, i_{\max}+1\} \ \wedge \ P(i-1)\}$

(4) $\{A_4 \equiv i_0 \le i \le i_{\max} \ \wedge \ P(i-1)\}$

3 Body

(5) $\{A_5 \equiv i_0 \le i \le i_{\max} \ \wedge \ P(i)\}$

4 $i := i+1$;

(6) $\{A_6 \equiv i_0 \le i \le i_{\max}+1 \ \wedge \ P(i-1)\}$

(7) $\{A_7 \equiv i_0 \le i \le \max\{i_0, i_{\max}+1\} \ \wedge \ P(i-1)\}$

5 **end**;

(8) $\{A_8 \equiv i \not< i_{\max}+1 \ \wedge \ i_0 \le i \le \max\{i_0, i_{\max}+1\} \ \wedge \ P(i-1)\}$

(9) $\{A_9 \equiv P(\max\{i_0, i_{\max}\})\}$

Die Implikationen $A_3 \Rightarrow A_4$, $A_6 \Rightarrow A_7$ sowie $A_8 \Rightarrow A_9$ sind trivial.

Abbildung 2: Die For-Schleifenregel.

Die If-Then-Else-Regel

(1) $\left\{ A_1 \equiv \begin{array}{r} (C \to VB_T) \\ \wedge\ (\neg C \to VB_E) \end{array} \right\}$

1 **If** (C) **then**

(2) $\{A_2 \equiv VB_E\}$

2 T // Then-Block

(3) $\{A_3 \equiv NB\}$

3 **else**

(4) $\{A_4 \equiv VB_T\}$

4 E // Else-Block

(5) $\{A_5 \equiv NB\}$

5 **end**;

(6) $\{A_6 \equiv NB\}$

Wenn für C, VB, NB die zwei Tripel $\{C \wedge VB\}\mathsf{T}\{NB\}$ und $\{\neg C \wedge VB\}\mathsf{E}\{NB\}$ herleitbar sind, gilt auch $\{VB\}$**If**(C) **Then** T **Else** E **End**$\{NB\}$.

Für unser Schema setzen wir stets

$$VB_T := C \wedge VB$$
$$VB_E := \neg C \wedge VB$$
$$VB := (C \to VB_T) \wedge (\neg C \to VB_E)$$

und erhalten die äquivalente Formulierung in Codeform links.

Abbildung 3: Die If-Then-Else-Regel.

- **Bedingung/Zusicherung**
  Sei B die Bedingung oberhalb von A. In diesem Fall haben wir uns eine *Proof-Obligation* eingehandelt: Wir müssen prüfen, ob B tatsächlich A impliziert; falls ja, erlaubt uns die Weak-Regel die Fortsetzung unseres Beweises.

  Gilt $B \Rightarrow A$ dagegen tatsächlich **nicht**, ist unser Beweis *ungültig*. Das kann prinzipiell zwei Gründe haben: Zum einen könnte die Nachbedingung, die wir zu beweisen versuchen, zu stark, also schlicht *falsch* sein – in dem Fall *muss* das Verfahren natürlich scheitern.

  Die andere Möglichkeit ist eine ungeeignete Schleifeninvariante. Da wir nach Konstruktion immer Invarianten wählen, die stark genug sind, um aus ihnen die gewünschte Nachbedingung der Schleife herzuleiten, können die Invarianten also nur *zu stark* sein. Zu ambitionierte Invarianten lassen sich entweder vor der Schleife nicht herleiten, oder vom Rumpf nicht erhalten. Wir können also versuchen, die verwendeten Invarianten abzuschwächen, müssen aber sicherstellen, dass die Nachbedingung der Schleife immer noch folgt.

(3) Sind wir am Anfang des Programms angelangt, so ist die oberste Bedingung eine gültige Vorbedingung für unser Progamm.

Bemerkungen:

- Stellen wir bei der Wahl der Schleifeninvarianten sicher, dass wir eine *schwächste* Invariante wählen, die die Nachbedingung noch zeigen kann, so erhalten wir in (3) eine **allgemeinste Vorbedingung** für unser Programm.

  In der Tat ist die schwächste Invariante *eindeutig* bestimmt: Gäbe es mehrere, so wäre die Verundung aller schwächsten Zusicherungen *echt* schwächer und immer noch hinreichend für die Nachbedingung $\lightning$.

- Wenn wir schon eine Vorbedingung V gegeben haben – zur vollständigen Spezifikation einer Funktion gehört Vor- *und* Nachbedingung – so können wir mit dem obigen Verfahren zuerst die allgemeinste Vorbedingung V' ableiten und anschließend prüfen ob $V'\ V$ impliziert. Wir haben also einfach eine weitere Proof-Obligation wie im letzten Punkt von Schritt (2).

- Bei **geschachtelte Schleifen** ist es typisch, dass wir in den inneren Schleifen gewisse »Frame-Bedingungen« durchschleifen müssen, die sich aus der Invariante der äußeren Schleife ergeben. Oft beziehen sich Rahmenbedingungen auf Variablen, die in der äußere Schleife verwendet, in der inneren aber gar nicht verändert werden.

- Auch wenn das Programm in einem Zustand gestartet wird, der die Vorbedingung erfüllt, machen wir keinerlei Aussagen über Terminierung.

3.2. Anwendung auf das Beispielprogramm

Wir beginnen mit Schritt (1):

Da wir unser Beispielprogramm explizit entworfen haben um

$$\sum_{i=1}^{n} i \cdot \prod_{j=n-i+1}^{n} j \Bigg/ \prod_{j=2}^{i} j$$

zu berechnen und das Ergebnis in s landen soll, lautet die gewünschte Nachbedingung am Ende des Programms wie angegeben.

Nun folgt Schritt (2).

SUM-OF-BINOMIALS(n)

```
1   k := 1;
2   s := 0;
3   While (k < n+1) do
4       b := 1;
5       l := n-k+1;
6       While (l < n+1) do
7           b := b*l;
8           l := l+1;
9       end;
10      l := 2;
11      While (l < k+1) do
12          b := b/l;
13          l := l+1;
14      end;
15      s := s+(k*b);
16      k := k+1;
17  end;
```

(1) $\left\{ A_0 \equiv s = \sum_{i=1}^{n} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j} \right\}$

A_0 Über der neuesten (und einzigen ...) Zusicherung A_0 steht Zeile 17, d. h. eine While-Schleife. Dankenswerterweise ist diese eine Instanz der For-Schleifen-Regel aus Abbildung 2 mit Zählvariable k, Startwert 1 und Maximalwert n. Damit hat die Invariante die Form

$$INV := 1 \leq k \leq \max\{1, n+1\} \ \wedge \ P(k-1)$$

für ein unäres Prädikat P. Die Aufgabe des Rumpfes ist die Berechnung des neuen Binomialkoeffizienten und das Update von s in Zeile 15, d. h. s enthält die Teilsumme bis k und wir setzen

$$P(k) \equiv s = sum(k) \text{ mit}$$
$$sum(k) := \sum_{i=1}^{k} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j}.$$

Wir müssen jetzt prüfen, dass $\neg C \wedge INV$ die Nachbedingung A_0 impliziert (Schritt (a)). Das sieht man durch Einsetzen und elementare Vereinfachungen (konkret: Antisymmetrie von $\leq$), siehe A_1 bis A_4 unten. Danach müssen wir in (b) – (d) noch Zusicherungen entsprechend der Schleifenregel eintragen. Diese markieren wir als *›Schema‹-Zusicherungen*, da sie mechanisch aus den Regeln folgen.

...

(1) $\{A_7 \equiv INV\}$ Schema

3 **While** $(k < n+1)$ **do**

(2) $\{A_6 \equiv k < n+1 \ \wedge \ INV\}$ Schema

...

16 $k := k+1;$

(3) $\{A_5 \equiv INV\}$ Schema

17 **end**;

(4) $\{A_1 \equiv k \not< n+1 \ \wedge \ INV\}$ Schema

(5) $\{A_2 \equiv k \geq n+1 \ \wedge \ 1 \leq k \leq \max\{1, n+1\} \ \wedge \ s = sum(k-1)\}$

(6) $\{A_3 \equiv k = \max\{1, n+1\} \ \wedge \ s = sum(k-1)\}$

(7) $\{A_4 \equiv s = sum(\max\{0, n\})\}$

(8) $\left\{A_0 \equiv s = \sum_{i=1}^{n} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j}\right\}$

Damit ist der Schritt für A_0 abgeschlossen.

$\boxed{A_1 - A_4}$ sind schon abgearbeitet.

$\boxed{A_5}$ Über A_5 steht die Zuweisung $k := k+1$, d. h. wir müssen in A_5 alle Vorkommen von k durch $k+1$ ersetzen

...

15 $s := s + (k * b);$

(1) $\{A_9 \equiv 1 \leq k+1 \leq \max\{1, n+1\} \ \wedge \ s = sum(k+1-1)\}$ Schema

16 $k := k+1;$

(2) $\{A_8 \equiv 1 \leq k \leq \max\{1, n+1\} \ \wedge \ s = sum(k-1)\}$

(3) $\{A_5 \equiv INV\}$ Schema ...

$\boxed{\ldots}$ Die restlichen Schritte sind ähnlich ...
Gerade das Finden der schwächsten Invarianten erfordert Übung und gelingt meist erst nach ein paar Iterationen. Unter anderem deshalb lohnt es sich, eine Abkürzung für die Invarianten einzuführen.

Schritt (3) ergibt für unser Beispielprogramm eine Tautologie. Damit hat SUM-OF-BINOMIALS *gar keine* Voraussetzung an den Startzustand! (Hatten Sie erwartet/erkannt, dass auch für $n < 0$ alles wohl-definiert und korrekt bleibt? Siehe dazu auch Anhang C.)

3.3. Abkürzungen

Bevor wir uns ins Vergnügen stürzen, das komplette Beispielprogramm zu verifizieren, definieren wir uns syntaktische Abkürzungen für häufig verwendete Teilformeln oder Terme. *Syntaktische* Abkürzungen heißen sie, weil wir jedes Vorkommen *textuell* (ohne Interpretation) durch die – entsprechend der Variablen substituierte – rechte Seite ersetzen.[4] Solche Abkürzungen helfen, die Zusicherungen halbwegs übersichtlich zu halten.

$$
\begin{aligned}
\mathit{sum}(k) &:= \sum_{i=1}^{k} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j} \\
\mathit{binom}(n,k) &:= \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{k} j} \\
\mathit{prod}(k) &:= \prod_{j=n-k+1}^{n} j \\
\mathit{INV} &:= 1 \le k \le \max\{1, n+1\} \ \wedge \ s = \mathit{sum}(k-1) \\
\mathit{frame} &:= 1 \le k \le \max\{1, n\} \ \wedge \ s = \mathit{sum}(k-1) \\
\mathit{INV1} &:= \mathit{frame} \ \wedge \ 1 \le l \le n+1 \ \wedge \ b = \prod_{j=n-k+1}^{l-1} j \\
\mathit{INV2} &:= \mathit{frame} \ \wedge \ 1 \le l \le k+1 \ \wedge \ b = \mathit{prod}(k) \Big/ \prod_{j=2}^{l-1} j
\end{aligned}
$$

Direkt nach Definition ist

$$
\begin{aligned}
\mathit{sum}(k-1) + k \cdot \mathit{binom}(n,k) &= \sum_{i=1}^{k-1} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j} + k \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{k} j} \qquad (1) \\
&= \sum_{i=1}^{k} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j} \\
&= \mathit{sum}(k)
\end{aligned}
$$

[4] Wem es hilft, der mag den Vergleich zum C/C++ Präprozessor heranziehen. Die Abkürzungen sind Makros, die der Präprozessor stumpf im Text ersetzt, *bevor* der Compiler versucht, dem Quelltext eine Semantik (in Form von Assembler-Code) zuzuweisen. Insbesondere sind Abkürzungen erlaubt, die außerhalb eines passenden Kontexts nicht wohlgeformt sind, z. B. $\mathit{closing}(x) := x$). An einer konkreten Stelle $f(y, \mathit{closing}(x)$ kann das dann Sinn machen.

3.4. Endergebnis: Annotiertes Programm

SUM-OF-BINOMIALS(n)

(1) $\{A_{47} \equiv true\}$

(2) $\{A_{46} \equiv 1=1 \ \wedge \ 0=0\}$ nach Schema aus A_{45}

1 $k := 1;$

(3) $\{A_{45} \equiv k=1 \ \wedge \ 0=0\}$ nach Schema aus A_{44}

2 $s := 0;$

(4) $\{A_{44} \equiv k=1 \ \wedge \ s=0\}$

(5) $\{A_{43} \equiv k=1 \ \wedge \ s=sum(k-1)\}$

(6) $\{A_7 \equiv INV\}$ nach Schema aus A_1

3 **While** $(k<n+1)$ **do**

(7) $\{A_6 \equiv k<n+1 \ \wedge \ INV\}$ nach Schema aus A_1

(8) $\{A_{40} \equiv n \geq k \ \wedge \ 1 \leq k \leq \max\{1,n+1\} \ \wedge \ s=sum(k-1)\}$

(9) $\{A_{41} \equiv 1 \leq k \leq n \ \wedge \ s=sum(k-1)\}$

(10) $\{A_{42} \equiv frame \ \wedge \ \underbrace{k \leq n \leq n+k}_{\Downarrow -k+1} \ \wedge \ 1=1\}$

(11) $\left\{A_{39} \equiv frame \ \wedge \ 1 \leq n-k+1 \leq n+1 \ \wedge \ 1=\prod_{j=n-k+1}^{n-k+1-1} j\right\}$ nach Schema aus A_{38}

4 $b := 1;$

(12) $\left\{A_{38} \equiv frame \ \wedge \ 1 \leq n-k+1 \leq n+1 \ \wedge \ b=\prod_{j=n-k+1}^{n-k+1-1} j\right\}$ nach Schema aus A_{37}

5 $l := n-k+1;$

(13) $\left\{A_{37} \equiv frame \ \wedge \ 1 \leq l \leq n+1 \ \wedge \ b=\prod_{j=n-k+1}^{l-1} j\right\}$

(14) $\{A_{31} \equiv INV1\}$ nach Schema aus A_{25}

6 **While** $(l<n+1)$ **do**

(15) $\{A_{30} \equiv l<n+1 \ \wedge \ INV1\}$ nach Schema aus A_{25}

(16) $\left\{A_{35} \equiv l<n+1 \ \wedge \ frame \ \wedge \ 1 \leq l \leq n+1 \ \wedge \ b=\prod_{j=n-k+1}^{l-1} j\right\}$

(17) $\left\{A_{36} \equiv frame \ \wedge \ 1 \leq l \leq n \ \wedge \ b*l=l*\prod_{j=n-k+1}^{l-1} j\right\}$

(18) $\left\{A_{34} \equiv frame \ \wedge \ 1 \leq l+1 \leq n+1 \ \wedge \ b*l=\prod_{j=n-k+1}^{l+1-1} j\right\}$ nach Schema aus A_{33}

7 $b := b*l;$

(19) $\left\{A_{33} \equiv frame \ \wedge \ 1 \leq l+1 \leq n+1 \ \wedge \ b=\prod_{j=n-k+1}^{l+1-1} j\right\}$ nach Schema aus A_{32}

8 $l := l+1;$

(20) $\left\{A_{32} \equiv frame \ \wedge \ 1 \leq l \leq n+1 \ \wedge \ b=\prod_{j=n-k+1}^{l-1} j\right\}$

(21) $\{A_{29} \equiv INV1\}$ nach Schema aus A_{25}

9 **end**;

(22) $\{A_{25} \equiv l \not< n+1 \ \wedge \ INV1\}$ Schema

(23) $\left\{A_{26} \equiv l \geq n+1 \ \wedge \ frame \ \wedge \ 1 \leq l \leq n+1 \ \wedge \ b=\prod_{j=n-k+1}^{l-1} j\right\}$

9 **end**;

(22) $\{A_{25} \equiv l \not< n+1 \ \wedge \ INV1\}$ Schema

(23) $\left\{A_{26} \equiv l \geq n+1 \ \wedge \ frame \ \wedge \ 1 \leq l \leq n+1 \ \wedge \ b = \prod_{j=n-k+1}^{l-1} j\right\}$

(24) $\left\{A_{27} \equiv frame \ \wedge \ l = n+1 \ \wedge \ b = \prod_{j=n-k+1}^{l-1} j\right\}$

(25) $\{A_{28} \equiv frame \ \wedge \ 1 \leq k \ \wedge \ b = prod(k)\}$

(26) $\left\{A_{24} \equiv frame \ \wedge \ 1 \leq 2 \leq k+1 \ \wedge \ b = prod(k) / \prod_{j=2}^{2-1} j\right\}$ nach Schema aus A_{23}

10 $l := 2$;

(27) $\left\{A_{23} \equiv frame \ \wedge \ 1 \leq l \leq k+1 \ \wedge \ b = prod(k) / \prod_{j=2}^{l-1} j\right\}$

(28) $\{A_{17} \equiv INV2\}$ nach Schema aus A_{11}

11 **While** $(l < k+1)$ **do**

(29) $\{A_{16} \equiv l < k+1 \ \wedge \ INV2\}$ nach Schema aus A_{11}

(30) $\left\{A_{21} \equiv frame \ \wedge \ 1 \leq l \leq k \ \wedge \ b = prod(k) / \prod_{j=2}^{l-1} j\right\}$

(31) $\left\{A_{22} \equiv frame \ \wedge \ 0 \leq l \leq k \ \wedge \ b/l = prod(k) / l \cdot \prod_{j=2}^{l-1} j\right\}$

(32) $\left\{A_{20} \equiv frame \ \wedge \ 1 \leq l+1 \leq k+1 \ \wedge \ b/l = prod(k) / \prod_{j=2}^{l+1-1} j\right\}$ nach Schema aus A_{19}

12 $b := b/l$;

(33) $\left\{A_{19} \equiv frame \ \wedge \ 1 \leq l+1 \leq k+1 \ \wedge \ b = prod(k) / \prod_{j=2}^{l+1-1} j\right\}$ nach Schema aus A_{18}

13 $l := l+1$;

(34) $\left\{A_{18} \equiv frame \ \wedge \ 1 \leq l \leq k+1 \ \wedge \ b = prod(k) / \prod_{j=2}^{l-1} j\right\}$

(35) $\{A_{15} \equiv INV2\}$ nach Schema aus A_{11}

14 **end**;

(36) $\{A_{11} \equiv l \not< k+1 \ \wedge \ INV2\}$ Schema

(37) $\left\{A_{12} \equiv l \geq k+1 \ \wedge \ frame \ \wedge \ 1 \leq l \leq k+1 \ \wedge \ b = prod(k) / \prod_{j=2}^{l-1} j\right\}$

(38) $\left\{A_{13} \equiv frame \ \wedge \ l = k+1 \ \wedge \ b = prod(k) / \prod_{j=2}^{k} j\right\}$

(39) $\{A_{14} \equiv frame \ \wedge \ b = binom(n,k)\}$

$\Downarrow$ verwendet (1).

(40) $\{A_{10} \equiv 1 \leq k+1 \leq \max\{1, n+1\} \ \wedge \ s + (k * b) = sum(k)\}$ nach Schema aus A_9

15 $s := s + (k * b)$;

(41) $\{A_9 \equiv 1 \leq k+1 \leq \max\{1, n+1\} \ \wedge \ s = sum(k)\}$ nach Schema aus A_8

16 $k := k+1$;

(42) $\{A_8 \equiv 1 \leq k \leq \max\{1, n+1\} \ \wedge \ s = sum(k-1)\}$

(43) $\{A_5 \equiv INV\}$ nach Schema aus A_1

17 **end**;

(44) $\{A_1 \equiv k \not< n+1 \ \wedge \ INV\}$ Schema

(45) $\{A_2 \equiv k \geq n+1 \ \wedge \ 1 \leq k \leq \max\{1, n+1\} \ \wedge \ s = sum(k-1)\}$

(46) $\{A_3 \equiv k = \max\{1, n+1\} \ \wedge \ s = sum(k-1)\}$

(47) $\{A_4 \equiv s = sum(\max\{0, n\})\}$

(48) $\left\{A_0 \equiv s = \sum_{i=1}^{n} i \cdot \frac{\prod_{j=n-i+1}^{n} j}{\prod_{j=2}^{i} j}\right\}$

Anhang

A. Nachweis der Terminierung

Auch wenn es hier hauptsächlich um den HOARE-Kalkül geht, sollte ein kleiner Exkurs in Terminierungsbeweise nicht schaden. Die einfachste Möglichkeit dazu bieten wohl-fundierte Ordnungen (auch NOETHERsche Ordnungen genannt):

Definition (Wohl-fundierte Ordnung)**:** Eine (partielle) Ordnung $(\mathcal{U}, \leq)$ heißt **wohl-fundiert**, wenn es keine unendliche Sequenz $u_1, u_2, \ldots$ mit $u_1 \gneq u_2 \gneq \cdots$, d. h. keine strikt fallende Kette gibt. (Äquivalent dazu kann man fordern, dass jede in $\leq$ fallende Folge schließlich stationär wird.)

Für Terminierungsbeweise wählen wir $\mathcal{U} =$ Menge von Programmzuständen, formal also Variablenbelegungen $\sigma : \mathit{Vars} \to \mathbb{Z}$. Können wir dann zeigen, dass es eine wohl-fundierte Ordnung $(\mathcal{U}, \leq)$ gibt, so dass jeder Zustand σ *nach* dem Schleifenrumpf **strikt kleiner** als der Zustand τ *vor* dem Rumpf ist, so kann die Schleife nur endlich oft ausgeführt werden.

Zur Demonstration weisen wir nach, dass unser Beispielprogramm terminiert. Definiere $q(\sigma) = \big(|\sigma(n) - \sigma(k)|, |\sigma(n) - \sigma(l)|\big) \in \mathbb{N}^2$ und wähle die lexikographische Ordnung $\leq_{\text{lex}}$ auf $\mathbb{N}^2$, welche bekanntermaßen wohl-fundiert ist. Auf Zuständen erhalten wir die Ordnung

$$\sigma \leq \tau \quad \text{gdw} \quad q(\sigma) \leq_{\text{lex}} q(\tau),$$

die ebenfalls wohl-fundiert ist, da jede unendliche fallende Kette $\sigma_1 > \sigma_2 > \cdots$ eine unendliche Kette $q(\sigma_1) >_{\text{lex}} q(\sigma_2) >_{\text{lex}} \cdots$ in $\mathbb{N}^2$ implizierte.

In beiden inneren Schleifen bleiben n und k unverändert und l wird inkrementiert. Da sie nur betreten werden, wenn $l \leq n$, nähert sich l also echt an n an, d. h. der Zustand nach dem Rumpf ist echt kleiner als der Zustand vor dem Rumpf. Die äußere Schleife wiederum erhöht k und lässt n unverändert und wird nur ausgeführt für $k \leq n$, sodass die erste Komponente von $q(\sigma)$ die strikte Ordnung liefert.

Folglich terminiert SUM-OF-BINOMIALS für alle Startzustände.

B. Details für Logiker

Lesen auf eigene Gefahr! Eltern haften für ihre Kinder.

! Dieser Abschnitt soll ein paar formale Stolpersteine diskutieren, die dem einen oder anderen aufgefallen sein mögen. Für die Anwendung des HOARE-Kalküls sind diese nicht notwendig. !

B.1. Formale Definition

Die **Syntax** der HOARE-Logik $\mathcal{H}$ über einer Programmiersprache $\mathcal{P}$ und einer Logik $\mathcal{L}$ besteht aus allen Tripeln

$$\mathcal{H} = \big\{\{V\}P\{N\} : P \in \mathcal{P} \text{ und } V,N \in \mathcal{L}\big\}$$

für Programme $P \in \mathcal{P}$ und Formeln $V,N \in \mathcal{L}$. V heißt **Vorbedingung** und N **Nachbedingung** für P. Dabei müssen die Zustände von Programmen aus $\mathcal{P}$ Strukturen von $\mathcal{L}$ sein, also Objekte, die Formeln aus $\mathcal{L}$ erfüllen können.

Wir beschränken uns hier auf die (namenlose) Programmiersprache aus dem Buch und verwenden als Semantik eines Programms P $\mathsf{Eff}(P)$, den *Effekt* von P, also die Relation auf Variablenbelegungen induziert von P (vgl. Definition 3.12 im Buch).[5] Unsere Zustände sind damit Variablenbelegungen $\sigma : \mathit{Vars} \to \mathbb{Z}$ und wir nehmen an, dass Formeln aus $\mathcal{L}$ ›Aussagen‹ über solche Zustände treffen. Formal nehmen wir für $\mathcal{L}$ eine »Erfüllt-Relation« $\models$ an und schreiben $\sigma \models F$, wenn F in σ gilt, $\models F$, wenn F Tautologie ist, $\sigma \not\models F'$ falls F' in σ *nicht* gilt, usw.

Damit können wir die **Semantik** von $\mathcal{H}$ sehr einfach definieren:

$$\models \{V\}P\{N\} \quad \text{gdw} \quad \forall\sigma : \sigma \models V \to \big(\forall\tau : \tau \in (\mathsf{Eff}\,P)(\sigma) \to \tau \models N\big).$$

Beachte nochmals: Gibt es für einen Startzustand σ gar keinen Endzustand, d. h. $(\mathsf{Eff}P)\sigma = \emptyset$, so ist die Aussage trivialerweise wahr.

B.2. Hoare-Kalkül

Der im Buch angegebene Kalkül (Definition 3.23) erlaubt uns, von der deklarativen Definition von $\models$ in $\mathcal{H}$ zu einer *Operationalisierung* überzugehen: Da der HOARE-Kalkül nach Satz 3.7

[5] Im Buch ist Eff als *partielle Funktion* definiert. Für deterministische Programme reicht das aus. Allgemeiner kann es für einen Startzustand und ein potentiell *nicht-deterministisches* Programm natürlich gar keinen, einen oder *mehrere mögliche* Endzustände geben.

für jede Logik $\mathcal{L}$ *sound* ist, können wir mechanisch gültige Tripel finden – wie wir es im Korrektheitsbeweis für unser Beispielprogramm getan haben.

B.3. Wahl der Logik $\mathcal{L}$

Die Definition des HOARE-Kalküls hängt von der gewählten Logik $\mathcal{L}$ ab. Im Anwendungsabschnitt oben haben wir uns darum aber nicht gekümmert – heißt das, unser Beweis ist womöglich doch nicht formal sauber?

B.3.1. Modelle

Kümmern wir uns zuerst um die *Modelle*: $\mathcal{L}$ muss Aussagen über Variablenbelegungen treffen. Es bietet sich also an, eine Logik zu verwenden die selbst Variablen hat; dann können wir Variablen im Programm und Variablen in den Zusicherungen einfach identifizieren. Alle diese Variablen werden mit Werten aus *einem* Universum $\mathcal{U}$ interpretiert, wie in den Prädikatenlogiken … also versuchen wir es mal mit PL-1.

B.3.2. Signatur

Um die Aussagen über Zustände aus Abschnitt 3 ausdrücken zu können, brauchen wir einige Funktionen und Prädikate. Die meisten wirken (mindestens auf den ersten Blick) harmlos:

- Addition, Multiplikation, Subtraktion, Division auf *ganzen Zahlen*[6]
- Kleiner-Ordnung
- Maximum (von zwei Zahlen)

Aber dann hatten wir da auch noch $\sum$ und $\prod$ mit variablen Grenzen! Hätten wir die gar nicht so einfach schreiben dürfen?
Doch, denn wir können die Summe $\sum_{i=i_0}^{i_{\max}} t$ auffassen als das 3-stellige Funktionssymbol $\sum_{i=\bullet}^{\bullet}\bullet$ mit den Parameter-Termen i_0, $i_{\max}$ und t. Für geschachtelte Summen müssen wir das Symbol für

[6] Tatsächlich sind alle Vorkommen dieser Operationen in unserem Programm und Beweis die entsprechenden Integer-Operationen. Hätten Sie dran gedacht? Warum geht die Division in Zeile 12 immer ohne Rest auf?

verschiedene Zähl-Variablen i in die Signatur aufnehmen. Mit der Formel

$$\forall i_0, i_{\max}, t \quad i_{\max} < i_0 \to \sum_{i=i_0}^{i_{\max}} t = 0 \qquad \wedge$$
$$\forall i_0, i_{\max}, t \quad i_{\max} \geq i_0 \to \sum_{i=i_0}^{i_{\max}} t = t\big[i_{\max}/i\big] + \sum_{i=i_0}^{i_{\max}-1} t$$

ist die Semantik von $\sum_{i=\bullet}^{\bullet} \bullet$ auf den ganzen Zahlen eindeutig festgelegt (und entspricht der erwarteten!).

B.3.3. Universum

Als Letztes wollen wir uns nochmal genauer dem Universum $\mathcal{U}$ zuwenden. Die verwendeten Funktionen und Prädikate lassen sich problemlos über ihre Eigenschaften auf den ganzen Zahlen charakterisieren – wie wir es eben für $\sum$ getan haben. Allerdings garantiert uns die Prädikatenlogik nicht, dass $\mathcal{U} = \mathbb{Z}$ – tatsächlich könnte das Universum endlich oder überabzählbar sein, womit wir den Korrektheitsbeweis implizit in einer *falschen Semantik* für unsere Programmiersprache geführt hätten!

Folglich möchten wir erzwingen, dass das einzige Modell unserer Formeln die ganzen Zahlen als Universum verwendet. Dazu brauchen wir eine **Axiomatisierung** unseres Universums, d. h. eine große Formel, die wir an jede unserer Formeln »dran-∧-en« und die $\mathcal{U} = \mathbb{Z}$ verlangt. Die gute Nachricht ist nun: Es gibt solche Axiomatisierungen. Eigentlich führt die genaue Betrachtung zu weit, aber das trifft wohl auf den ganzen Abschnitt zu, also was solls …

Typischerweise baut man sich schrittweise zuerst die natürlichen Zahlen, z. B. mit den Peano-Axiomen aus der Konstanten 0 und der einstelligen Funktion s (successor):

(1) $0 \in \mathbb{N}$

(2) $\forall x : s(x) \in \mathbb{N}$

(3) $\forall x : s(x) \neq 0$

(4) $\forall x, y : s(x) = s(y) \to x = y$

(5) $\forall X : \big(X(0) \wedge \forall x : X(x) \to X(sx)\big) \to \forall x : X(x)$ [Induktionsprinzip]

Leider brauchen wir den second-order Quantifier in (5), wenn wir eindeutig sein wollen.[7] Unseren Versuch, mit PL-1 auszukommen – bisher erfolgreich – müssen wir hier also als gescheitert ansehen.

[7] Das folgt aus dem LÖWENHEIM-SKOLEM-Theorem: Jede Axiomatisierung der natürlichen Zahlen durch PL-1 Formeln hat mehrere nicht-isomorphe Modelle.

Als nächstes kann man Rechenoperationen auf $\mathbb{N}$ durch ihre Axiome angeben, z. B. die Addition durch

$$\begin{aligned} \forall x \quad & x+0=x \\ \forall x,y \quad & x+s(y)=s(x+y) \end{aligned}$$

Zu guter Letzt können wir $\mathbb{Z}$ aufbauend auf $\mathbb{N}$ definieren: Sei $\sim$ die Äquivalenzrelation auf $\mathbb{N}^2$ definiert durch

$$(x_1,x_2) \sim (y_1,y_2) \quad \text{gdw} \quad x_1+y_2=y_1+x_2\ .$$

Dann ist $\mathbb{Z}$ isomorph zu den $\sim$-Äquivalenzklassen – wähle die Differenz x_1-x_2 der beiden Elemente des Paars (x_1,x_2) – und wir können charakterisierende Formeln für die Operationen auf $\mathbb{Z}$ angeben. Es gilt für die Addition und die Multiplikation:

$$\forall x_1,x_2,y_1,y_2 \in \mathbb{N} \quad (x_1,x_2)+(y_1,y_2)=(x_1+y_1,x_2+y_2),$$

$$\forall x_1,x_2,y_1,y_2 \in \mathbb{N} \quad (x_1,x_2)\cdot(y_1,y_2)=(x_1y_1+x_2y_2,x_1y_2+x_2y_1).$$

Fazit:
Auch der genaueren Betrachtung durch die LogikerInnen-Brille hält unser Beweis der partiellen Korrektheit für das Beispielprogramm stand: Die dort verwendeten Konstrukte aus dem mathematischen Alltag lassen sich herunterbrechen auf Formeln der Prädikatenlogik erster Stufe. Der Wertebereich $\mathbb{Z}$ für Variablen lässt sich durch PL-2-Formeln erzwingen.

C. Leere Summen und Produkte

Beim Erstellen des Beispielprogramms und auch in der Analyse habe ich implizit angenommen, dass **leere Summen und Produkte** passend definiert sind. Hier die entsprechenden Definitionen formal.

Sei $(\mathcal{U}, \circ)$ ein Monoid, d. h. ein Universum $\mathcal{U}$ mit der zweistelligen assoziativen Operation $\circ$ und einem neutralen Element $e \in \mathcal{U}$. Dann definieren wir die *Â»Big-Operator«-Schreibweise* für $n \in \mathbb{Z}$

$$\bigcirc_{i=1}^{n} u_i := \begin{cases} u_1 \circ u_2 \circ \cdots \circ u_n & \text{für } n \geq 1 \\ e & \text{sonst} \end{cases}.$$

Ist $(\mathcal{U}, \circ)$ kommutativ kann man auf die Ordnung auch verzichten und einfach $\bigcirc_{i \in I} u_i$ für eine endliche Indexmenge I schreiben; ist $I = \emptyset$ so ist $\bigcirc_{i \in I} u_i = e$.

Diese Definition ist sinnvoll, da sie verträglich mit der Verwendung von Big-Operator-Ausdrücken in größeren $\circ$-Produkten ist:

$$\bigcirc_{i=1}^{n} u_i \circ v = \begin{cases} u_1 \circ \cdots \circ u_n \circ v & \text{für } n \geq 1 \\ e \circ v = v & \text{sonst} \end{cases}$$

Für die Spezialfälle $(\mathbb{Z}, +)$ und $(\mathbb{Z}, \cdot)$ ergibt sich, dass leere Summen 0 und leere Produkte 1 sein müssen. Häufig verwendet wird auch $\big(\mathbb{Z} \cup \{+\infty\}, \min\big)$ und $\big(\mathbb{Z} \cup \{-\infty\}, \max\big)$ oder $(\mathbb{N}, \max)$ mit den neutralen Elementen $+\infty$, $-\infty$ und 0.

Im Code finden sich die neutralen Elemente bei der Initialisierung der Variablen wieder: Zu Beginn wird s auf 0 und b auf 1 gesetzt. Ist die Indexmenge leer, wird die entsprechende Schleife nie betreten und s bzw. b behalten ihren Wert. Das erklärt, warum SUM-OF-BINOMIALS ohne Vorbedingungen auskommt.