Programmieren und Beweisen

Experimente mit dem Programmverifizierer NPPV

H. Peter Gumm

Fachbereich Mathematik und Informatik
Philipps Universität Marburg, 35032 Marburg
gumm@informatik.uni-marburg.de

Zusammenfassung Anhand von Fallbeispielen zeigen wir den Einsatz unseres Programmverifizierers NPPV für abstrakte Fragestellungen zwischen Mathematik und Informatik. Aus einem gegebenen und mit Invarianten annotierten Programmschema erzeugt das System automatisch logische Bedingungen für dessen Korrektheit.
Der Verifizierer wurde zum Einsatz im Informatikunterricht konzipiert. Er nimmt dem Benutzer die immer wiederkehrende Berechnung trivialer Verifikationsbedingungen ab und richtet die Aufmerksamkeit und die Kreativität auf das Wesentliche – aussagekräftige Invarianten, Axiome der verwendeten Datenstrukturen, Existenz von Funktionen mit bestimmten Eigenschaften.
Der Schwerpunkt liegt auf der Verifikation abstrakter Programme (Programmschemata), z.B. zur Transformation zwischen rekursiven und iterativen Programmen. Selbst die Äquivalenz zwischen einem abstrakten Induktionsbeweis und der Verifikation eines Programmes, welches ein Gegenbeispiel zur Behauptung sucht, läßt sich mit NPPV interaktiv erkunden.

1 Motivation

Die Gaußsche Formel $\sum_{i=0}^{N} i = N * (N+1)/2$ ist eines der einfachsten Beispiele für eine arithmetischen Aussage, die mit Induktion bewiesen werden kann. Zu ihrer Überprüfung hat man rasch ein kleines Programm – wir wollen es `GAUSS` nennen – erstellt, das diese Formel für beliebige eingegebene Zahlen N überprüfen kann:

```
begin
  i := 0 ; sum := 0 ;
  while i < N do
    begin
      i   := i+1;
      sum := sum + i
    end
end
```

Wenn dieses Programm - wir haben bewußt auf zusätzliche Ein-Ausgaberoutinen verzichtet - mit einer Reihe von Eingabewerten von N getestet worden ist und das Ergebnis in der Variablen *sum* immer identisch mit $N*(N+1)/2$ ist, so wird manch einer bereits so sehr von der Richtigkeit der Gaußschen Formel überzeugt sein, daß es schwerfällt, ihm die Notwendigkeit eines Beweises klarzumachen.

Informatiker sind es vielfach gewohnt, Programmierprobleme als gelöst zu betrachten, wenn das erstellte Programm *läuft* und auf einer genügend großen Anzahl von Eingabewerten getestet worden ist. Vielleicht hatte man anfänglich einmal die Schleifenbedingung in der Form

```
while i <= N do ...
```

ausprobiert, oder die beiden Zuweisungen im Schleifenkörper vertauscht,

```
sum := sum + i;
i   := i+1
```

aber nachdem die letzten Probeläufe immer ein richtiges Ergebnis geliefert haben sind solche anfänglichen Irrtümer schnell vergessen.

Die Frage ist angebracht, ob ein Programmierer überhaupt weiß was er da geschrieben hat. Kann er sich auf sein Werk verlassen? Kein Ingenieur würde eine mathematische Formel zur statischen Berechnung einer Brücke verwenden, wenn diese nur anhand von einigen Beispielen geprüft worden wäre. Schließlich läßt sich durch Testen nur *die Anwesenheit, nicht aber die Abwesenheit von Fehlern* zeigen (E. Dijkstra).

Die Analogie eines Programmes zu einer mathematischen Formel ist nicht zufällig. Programme sind mathematische Objekte. Die Behauptung, daß ein Programm eine vorgegebene Aufgabe löst, ist eine mathematische Aussage. Der Entwurf von Programmpaketen ist mit einer mathematischen Theoriebildung vergleichbar. C.A.R. Hoare hat diesen Zusammenhang auf den Punkt gebracht: "*Programming is a mathematical activity*".

Als mathematische Objekte können Programme allerdings sehr komplex werden. Ein formaler Beweis einer Programmeigenschaft führt zu einer Fülle von unscheinbar aussehenden logischen Aussagen, den sogenannten *Verifikationsbedingungen*. Nur wenn alle diese allgemeingültig sind, ist das vorgelegte Programm korrekt. Die Verifikation eines Programmes *von Hand* ist daher eine äußerst mühsame Angelegenheit. Jeder Beweisversuch, jede Modifikation des Programmtextes führt zu einer erneuten Ansammlung von Verifikationsbedingungen, die wiederum einzeln untersucht werden müssen. Die Konsequenz ist, daß in der Ausbildung nur einige wenige, dazu sehr kleine Programme beispielhaft analysiert werden. Für eigene Experimente der Lernenden erweisen sich bereits einfachste Programme als zu komplex und im Endeffekt als frustrierend.

Um dem abzuhelfen haben wir den Programmverifizierer "NPPV" (New Paltz Program Verifier) konstruiert, mit dem nicht nur die Verifikation von Programmen von dem geschilderten Ballast befreit wird, das System soll vor allem dazu dienen, die Zusammenhänge zwischen Programmen und mathematischen Beweisen interaktiv zu erkunden. Wenn auch später in der Praxis nicht jedes

Programm formal bewiesen werden kann, sollte z.B. die Erkenntnis, daß jede While-Schleife einer mathematischen Behauptung entspricht, den Programmierer dazu bewegen, Schleifen immer durch geeignete "Invarianten" zu dokumentieren. Diese entsprechen gerade den Induktionshypothesen. So wie in einem Induktionsbeweis der kreative Prozeß sich auf das Finden einer geeigneten Induktionshypothese konzentriert, so erwartet NPPV lediglich die Angabe einer Invarianten für jede Schleife und versucht, dem Benutzer den langweiligen aber umfangreichen Rest des Beweises abzunehmen. Übriggebliebene Beweisobligationen erweisen sich dabei entweder als zentrale Anforderungen an die verwendeten Datenstrukturen, oder sie zeigen, daß die Invariante nicht als Induktionshypothese taugt – weil sie entweder zu schwach oder zu stark ist, oder weil das Programm fehlerhaft ist.

Im Folgenden wollen wir eine Reihe von Beispielen vorstellen, anhand derer wir mit Hilfe von NPPV zeigen, wie sich Programmieren und Beweisen ergänzen und gegenseitig bedingen. Die Korrektheit eines Programmes erweist sich als gleichbedeutend mit der Gültigkeit einer algebraischen Eigenschaft und umgekehrt können wir jeden induktiven Beweis als Korrektheitsaussage für ein bestimmtes Programm formulieren.

2 Programme und Spezifikationen

In diesem Kapitel skizzieren wir kurz den theoretischen Hintergrund, soweit er für das Folgende benötigt wird. Eine ausführliche Erläuterung findet man in Lehrbüchern (z.B. [1] oder [3]).

2.1 While-Programme

Wir betrachten hier eine einfache Programmiersprache, welche außer Zuweisungen

$$v \texttt{ := } t$$

die Kontrollstrukturen *Sequentielle Komposition*, *Bedingte Anweisung*, und *While-Schleife* besitzt. Alle anderen Kontrollstrukturen können aus diesen primitiven Operatoren abgeleitet werden. Programme sind also induktiv definiert durch

- jede Zuweisung v `:=` t ist ein Programm
- ist B ein Boolescher Ausdruck und sind S_1, S_2 Programme, dann auch
  - S_1 `;` S_2
  - `if` B `then` S_1 `else` S_2
  - `while` B `do` S_1.

Wie üblich benutzen wir die Schlüsselworte `begin` und `end` als Klammern für Kontrollstrukturen. Natürlich erlaubt NPPV auch zusätzliche Kontrollstrukturen, wie z.B.

- `repeat` S `until` B
- `for` $i = A$ `to` B `do` S.

Da sie sich aber auf die vorherigen zurückführen lassen, werden sie hier nicht weiter betrachtet.

2.2 Korrektheitsformeln und Spezifikationen

Eigenschaften von Programmen beschreiben wir mit sogenannten *Korrektheitsformeln* der Form $\{P\}\,S\,\{Q\}$. Dabei ist S ein Programm und P, Q sind logische Ausdrücke die sich auf den Zustand vor bzw. nach der Ausführung von S beziehen. Die Formel $\{P\}\,S\,\{Q\}$ soll ausdrücken:

> Wenn das Programm S in einem Zustand startet, in dem P erfüllt ist, dann wird, nachdem S terminiert hat, Q erfüllt sein.

Eine Korrektheitsformel $\{P\}\,S\,\{Q\}$ verlangt nicht, daß S terminieren soll. Insbesondere gilt genau dann $\{P\}\,S\,\{\texttt{false}\}$, wenn das Programm S, gestartet in einem Zustand, in dem P wahr ist, nicht terminiert.

Eine Programmieraufgabe beginnt mit einer *Spezifikation* $\{P\}\,X\,\{Q\}$, wobei X als Platzhalter für das unbekannte Programm dient. In der Vorbedingung P und der Nachbedingung Q können neben den Programmvariablen von X auch sogenannte *logische Variablen* auftreten. Dies sind Variablen, welche von dem gesuchten Programm X weder gelesen noch geschrieben werden dürfen. Wir wollen als Konvention vereinbaren, daß Programmvariablen mit Kleinbuchstaben und logische Variablen mit Großbuchstaben beginnen sollen.

Die Notwendigkeit für logische Variablen kann man bereits an dem folgenden einfachen Beispiel erkennen: Angenommen, wir wollen ein Programm X spezifizieren, welches den Inhalt der Variablen `x` und `y` vertauschen soll. Mit logischen Variablen `A` und `B` spezifizieren wir nun:

```
{ x=A and y=B } X { x=B and y=A }.
```

Dürfte X die Variablen `A` und `B` schreiben oder lesen, so hätte man folgende unbeabsichtigte "Lösungen": `x := B ; y := A` oder gar `x := y ; B := A`.

2.3 Die Hoare-Regeln

Die traditionelle Aufgabe der Programmverifikation besteht darin, die Gültigkeit einer Korrektheitsformel $\{P\}\,S\,\{Q\}$ nachzuweisen, also zu zeigen, daß ein Programm S die durch P und Q gegebene Spezifikation erfüllt. Für die einfachsten Programme, also Zuweisungen der Form v `:=` t müssen wir uns überlegen, wann $\{P\}\,v$ `:=` $t\,\{Q\}$ gilt, und für zusammengesetzte Programme S der Bauart "S_1 `;` S_1", "`if` B `then` S_1 `else` S_2" oder "`while` B `do` S_1" müssen wir eine Methode finden, um die Gültigkeit einer Korrektheitsformel $\{P\}\,S\,\{Q\}$ auf die Gültigkeit entsprechender Formeln für S_1 und S_2 zurückzuführen.

Ein geeigneter Kalkül stammt von C.A.R. Hoare[4]. Er besteht aus zwei logischen Regeln, einem Axiom für die Zuweisung und je einer Regel für die verschiedenen Kontrollkonstrukte. Jede Regel hat die Form

$$\frac{Prämisse_1, \ldots, Prämisse_n}{\{P\}\,S\,\{Q\}.}$$

Ein Beweis erfolgt *rückwärts*. Um die Korrektheitsbehauptung (man nennt diese auch "Hoare-Tripel") im Nenner zu zeigen, genügt es, die Prämissen der entsprechenden Regel nachzuweisen.

Für den Fall einer Zuweisung lautet die Regel

$$\frac{P \Rightarrow Q[v/t]}{\{P\}\, v \texttt{ := } t\, \{Q\}}.$$

Hierbei steht $Q[v/t]$ für die Aussage, die aus Q entsteht, wenn überall die Variable v durch den Term t ersetzt wird. In diesem Falle erzeugt ein Rückwärtsbeweis aus einem Hoare-Tripel direkt eine logische Bedingung, eine Implikation.

Ist das Programm eine Komposition $S = S_1 \texttt{ ; } S_2$, so wird die folgende Regel benötigt:

$$\frac{\{P\}\, S_1\, \{R\},\ \{R\}\, S_2\, \{Q\}}{\{P\}\, S_1 \texttt{ ; } S_2\, \{Q\}}.$$

Bei der Rückwärtsanwendung muß also eine Eigenschaft R, eine sogenannte *Zwischenbehauptung*, geraten werden, welche nach der Ausführung von S_1 und vor Beginn von S_2 erfüllt ist.

Nicht ganz so einfach ist die Situation bei der While-Schleife. Für einen Beweis von $\{P\}\,\texttt{while}\ B\ \texttt{do}\ S\,\{Q\}$ wird eine *Invariante* I benötigt. Dies ist eine Bedingung, welche in jedem Schleifendurchlauf erhalten wird. Zusätzlich muß sie zu Beginn der Schleife wahr sein, also aus der Vorbedingung P folgen und außerdem am Schleifenende (wenn $\neg B$ gilt) die Nachbedingung Q implizieren. Die While-Regel ist damit:

$$\frac{P \Rightarrow I,\ \{I \wedge B\}\, S\, \{I\},\ I \wedge \neg B \Rightarrow Q}{\{P\}\,\texttt{while}\ B\ \texttt{do}\ S\,\{Q\}.}$$

Um ein Programm X, genauer ein Hoare-Tripel $\{P\}\, X\, \{Q\}$, zu beweisen, muß bei jeder Anwendung der Kompositionsregel eine geeignete Zwischenbehauptung gefunden werden und bei jeder While-Schleife eine Invariante. In NPPV können wir Zwischenbehauptungen (optional) nach jedem Semikolon ";" und Invarianten (notwendig) nach jedem "`do`" als logische Formeln und in"{" und "}" eingeschlossen, in das Programm einfügen. Man sagt, daß man das Programm *annotiert*. Zum Glück lassen sich geeignete Zwischenbehauptungen auch mit einem einfachen Algorithmus automatisch bestimmen. Dazu berechnet NPPV zu einem annotierten Programm S und einer beliebigen Nachbedingung Q zunächst eine geeignete Vorbedingung $pre(S, Q)$, [1] von der man zeigen kann, daß sie immer eine geeignete Zwischenbehauptung für eine Komposition $\{P\}\, S_1 \texttt{ ; } S_2\, \{Q\}$ ist, d.h. es gilt:

$$\{P\}\, S_1 \texttt{ ; } S_2\, \{Q\} \Leftrightarrow (\{P\}\, S_1\, \{pre(S_2, Q)\} \ \wedge\ \{pre(S_2, Q)\}\, S_2\, \{Q\}). \qquad (1)$$

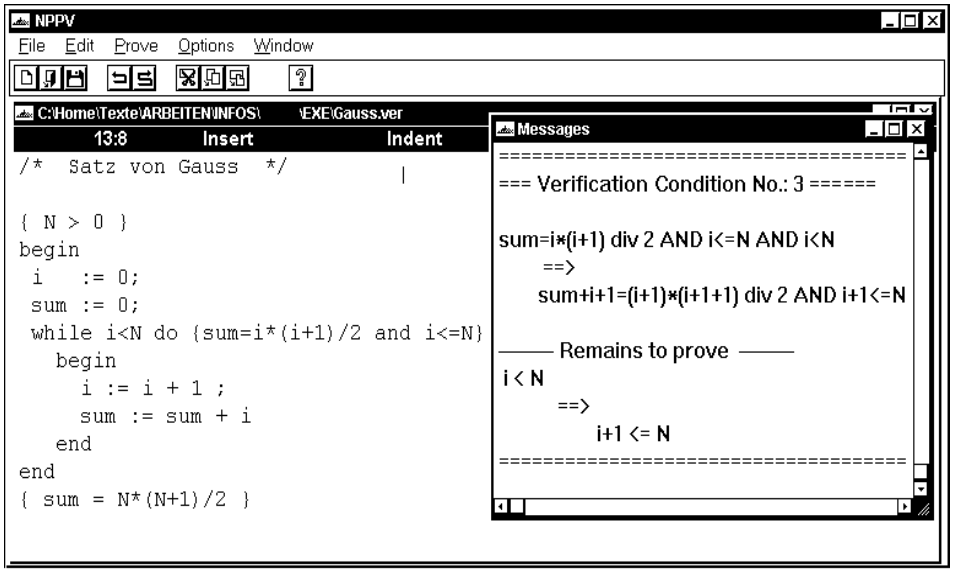
Anders als die Zwischenbehauptungen kann man eine geeignete Invariante I für die While-Regel nicht automatisch ermitteln. In der Tat reduziert sich die Verifikationsaufgabe in der Hauptsache auf das Finden geeigneter Invarianten. Alles andere sind vergleichsweise einfache Details, die ein Verifizierer weitgehend automatisch erledigen kann. Wie bereits angedeutet, steht dies in völliger Analogie

[1] Nicht zu verwechseln mit der *weakest precondition*, welche im Allgemeinen weder berechenbar noch ausdrückbar (*expressible*) ist

mit einer Beweisaufgabe, wo das Finden einer geeigneten Induktionshypothese die essentielle Aufgabe darstellt, der Rest des Beweises ist meist vergleichsweise einfach.

3 Der Programmverifizierer NPPV

Der Programmverifizierer NPPV ist ein Werkzeug, mit dem man interaktiv Programme erstellen, spezifizieren und beweisen kann. Seine Oberfläche ist den kommerziellen Sprachentwicklungsumgebungen weitgehend nachempfunden, so daß der Umgang mit dem System denkbar einfach ist. In der Menüleiste fällt aber die Auswahl "Prove", anstelle des üblichen "Compile" auf.



Die Figur zeigt NPPV bei der Bearbeitung des diskutierten Programmes "`GAUSS`". In dem Editorfenster wird ein Programm im Quelltext bearbeitet und mit Vor- und Nachbedingung versehen. Zusätzlich muß jede While-Schleife mit einer Invarianten *annotiert* werden. Zwischenbehauptungen können optional hinter jedem Semikolon " `;` " eingefügt werden, ansonsten werden sie vom System automatisch berechnet und intern verwendet. Selbstverständlich werden Syntaxfehler oder Auslassungen im Quelltext sofort an der fraglichen Stelle angemahnt.

Wählt man den Menüpunkt "Prove", so berechnet NPPV aus dem annotierten Programm eine Folge von Verifikationsbedingungen, die der Reihe nach in einem Fenster erscheinen. Wenn möglich werden sie auch von dem System selbständig bewiesen, ansonsten nur vereinfacht und mit der Überschrift "`Remains to prove :`" dem Benutzer übergeben. Sind alle Verifikationsbedingungen wahre Aussagen, so ist das Programm korrekt, d.h. es erfüllt seine Spezifikation.

3.1 Ein schleifenloses Programm

Wir beginnen mit dem Standard-Programm zur Vertauschung des Inhaltes zweier Variablen:

```
{x=A and y=B} temp:=x ; x:=y ; y:=temp {x=B and y=A}.
```

NPPV erzeugt daraus eine einzige Verifikationsbedingung und beweist diese sofort. Das Programm ist also korrekt. Ebenso problemlos verifiziert NPPV das folgende Trickprogramm `SWITCH`, das die Vertauschung ohne Verwendung einer Hilfsvariablen bewerkstelligt:

```
{x=A and y=B} x:=x-y; y:=x+y; x:=y-x {x=B and y=A}
```

Der Anfängerfehler

```
{x=A and y=B} x:=y ; y:=x {x=B and x=A}.
```

führt zur Meldung

```
Remains to prove:  A=B.
```

Die Aussage $\forall A.\forall B.A = B$ ist aber offensichtlich keine Tautologie, somit erfüllt das Programm nicht die Spezifikation.

3.2 Die Ermittlung der Invarianten

Zwischenbehauptungen werden automatisch ermittelt, dagegen müssen alle Invarianten vom Benutzer angegeben werden. Spezifizieren wir das Programm `Gauss` mit Vorbedingung {`N > 0`} und Nachbedingung {`sum = N*(N+1)/2`}, so bietet sich als Invariante die Aussage {`sum = i*(i+1)/2`} an. NPPV zeigt aber sofort eine Verifikationsbedingung als unbewiesen an:

```
N <= i  ==>  i*(i+1)/2 = N*(N+1)/2 .
```

Offensichtlich ist dies keine Tautologie, andererseits ist klar, daß in dem Programm i nie größer als N werden kann. Verstärken wir die Invariante mit dieser Information, also

```
{ i <= N and sum = i*(i+1)/2 }
```

so bleibt von den drei erzeugten Verifikationsbedingungen genau eine als unbewiesen zurück:

```
x < N  ==>  (x+1) <= N.
```

Diese Aussage (für beliebige x und N) ist natürlich keine Tautologie. Setzt man aber voraus, daß x und N ganzzahlig sind, so ist die Eigenschaft tatsächlich gültig. Folglich gilt: Das Programm `GAUSS` mit der gefundenen Annotierung ist genau dann korrekt, wenn die Variablen x und N einer diskreten Ordnung, z.B. **N** oder **Z** angehören.

3.3 Axiome für Datentypen

NPPV kann nicht nur mit konkreten Datentypen – Zahlen, Booleschen Werten etc. umgehen, sondern erlaubt auch die Benutzung abstrakt gegebener Funktionen und Daten. Wir wollen dies am Beispiel des erwähnten Trickprogrammes `SWITCH` klarmachen. Wir fragen uns, welche Eigenschaften der natürlichen Zahlen und ihrer Operationen für dessen Korrektheit verantwortlich sind. Mit abstrakten Funktionszeichen p, q, und r erstellen wir daher ein *abstraktes Programm* und spezifizieren es genau wie vorher:

```
{ x = A and y = B }
    x := p(x,y) ;
    y := q(x,y) ;
    x := r(x,y)
{ x = B and y = A }.
```

NPPV erzeugt in diesem Falle die Verifikationsbedingungen

```
q(p(A,B),B) = A
r(p(A,B),A) = B.
```

Diese lassen sich nicht nur mit den bereits gesehenen Operationen " + " und " - " im Bereich der Zahlen, sondern schon alleine mit der Operation *xor* ($p = q = r = xor$) im Bereich der Booleschen Werte oder der Bitfolgen erfüllen, daher vertauscht auch das folgende Programm den Inhalt der Variablen `x` und `y` :

```
x := x xor y ; y := x xor y ; x := x xor y .
```

3.4 Von iterativen zu funktionalen Programme

Wir betrachten noch einmal unser Programm `GAUSS`. Genau genommen haben wir in unserem ersten Beweis die Korrektheit der Gaußschen Formel bewiesen. Wollten wir lediglich zeigen, daß das Programm korrekt die Zahlen von 1 bis N summiert, so könnte die Nachbedingung lauten:

```
{sum = summe(N)}.
```

Hierbei soll `summe` die mathematische Funktion $summe(n) = \sum_{i=0}^{n} i$ beschreiben. Als Invariante bietet sich wieder an:

```
{i <= N and sum = summe(i)}.
```

Die von NPPV generierten und vereinfachten Verifikationsbedingungen sind:

```
i < N   ==>  i+1 <= N
summe(0) = 0
summe(i+1) = (i+1) + summe(i).
```

Die erste Bedingung schränkt, wie bereits gewohnt, den Typ der Variablen `i` und `N` ein, während die restlichen Verifikationsbedingungen als rekursive Definition der Summenfunktion aufgefaßt werden können.

3.5 Entrekursivierung

Populärer als die Umwandlung eines iterativen in ein rekursives Programm ist der umgekehrte Weg, den man als *Entrekursivierung* bezeichnet. Besonders einfach gestaltet sich dies im Falle einer *endrekursiven* (*tailrekursiven*) Funktion f. Eine solche läßt sich durch Gleichungen der folgenden Form beschreiben:

$$f(x) = \begin{cases} g(x), & \text{falls } P(x) \\ f(r(x)), & \text{sonst.} \end{cases}$$

Ein iteratives Programm zur Berechnung von $f(x)$ verändert das Argument x von f, bis dieses die Eigenschaft P erfüllt. Abschließend wird die Funktion g angewendet. Die Spezifikation ist offensichtlich: Zu einem beliebigen Eingabewert M in `x` soll bei Programmende in der Variablen `x` der Funktionswert $f(M)$ abgelegt sein. Die Invariante drückt aus, daß der Wert von $f(x)$ konstant bleibt:

```
{ x = M }
  WHILE not P(x) DO    { f(x) = f(M) }
      x := r(x);
  x := g(x)
{ x = f(M) }
```

Aus diesem annotierten Programm erzeugt NPPV die folgenden Verifikationsbedingungen, die deutlich zeigen, daß es genau die Forderung der Endrekursion ist, welche die Korrektheit der Programmtransformation bedingen:

```
P(x)      ==>   f(x) = g(x)
not P(x)  ==>   f(x) = f(r(x)).
```

3.6 Lineare Rekursion

Viele Funktionen, darunter die bereits untersuchte Funktion *summe* oder auch die Fakultätsfunktion, sind endrekursiv. Dennoch umfaßt erst der Begriff der *linearen Rekursion* weitgehend alle in der Praxis relevanten Funktionen. Linear rekursive Funktionen f lassen sich durch Gleichungen der folgenden Form beschreiben:

$$f(x) = \begin{cases} g(x), & \text{falls } P(x) \\ h(f(r(x)), x), & \text{sonst.} \end{cases}$$

Für die Entrekursivierung wird nun ein Stack benötigt:

```
{ x = M }
begin
    s := Empty ;
    while not P(x) do   { p(x,s) = f(M) }
        begin
            s := push(x,s) ;
            x := r(x)
        end;
```

```
    z := g(x) ;
    while s <> Empty do { q(z,s) = f(M) }
        begin
            z := h(z,top(s)) ;
            s := pop(s)
        end
end
{ z = f(M) }
```

Die vorläufigen Invarianten {`p(x,s) = f(M)`} und {`q(x,s) = f(M)`} drücken unsere Erwartung aus, daß trotz der in den Schleifen stattfindenden Modifikationen von `x`, `z` und `s` der Wert von `f(M)` aus den geänderten Daten (mittels irgendwelcher Funktionen p und q) rekonstruierbar bleiben muß. Ein erster Beweisversuch mit NPPV liefert unter anderen die Bedingungen:

```
p(M,Empty)=f(M)
q(z,Empty)=f(M) ==> z=f(M)
P(x)    ==> p(x,s) = q(f(x),s).
```

Diese legen die Vermutung nahe, daß der Beweis auch funktionieren könnte, wenn wir in der Invarianten der ersten Schleife `p(x,s)` durch `q(f(x),s)` ersetzen. In der Tat erhalten wir nun die folgenden Verifikationsbedingungen:

```
P(x)        ==>  q(f(x),s) = q(g(x),s)
not P(x)    ==>  q(f(x),s)= q(f(r(x)),push(x,s))
q(z,Empty) = z
s <> Empty  ==>  q(h(z,top(s)),pop(s))= q(z,s).
```

Es ist zu beachten, daß NPPV nichts über die implizit benutzte Datenstruktur *Stack* "weiß". Wir können dem abhelfen, indem wir die definierenden Gleichungen zwischen *push*, *pop*, *top* und *Empty* verwenden, insbesondere die Tatsache, daß `s <> Empty` genau dann gilt, wenn `s = push(x,u)` und `top(push(x,u)) = x` sowie `pop(push(x,u)) = u`. Aus den letzten beiden Bedingungen entsteht:

```
q(z,Empty)= z
q(z,push(x,u)) = q(h(z,x),u).
```

Hierdurch wird eindeutig eine Funktion q definiert. Setzen wir diese Definition in die ersten Verifikationsbedingungen ein, so erhalten wir exakt die Bedingungen für die lineare Rekursivität.

4 Programmieren und Beweisen

Die bisherigen Beispiele haben bereits die Verwandschaft zwischen dem Beweisen von Programmen und mathematischen Induktionsbeweisen nahegelegt. Wir wollen diese in unserem abschließenden Beispiel präzisieren. Sei dazu $E(n)$ eine beliebige Eigenschaft natürlicher Zahlen. Wir stellen uns die Frage, ob $E(n)$ für alle natürlichen Zahlen n gilt.

Offensichtlich gilt genau dann $\forall n \in \mathbf{N}.E(n)$, wenn ein Programm, das der Reihe nach alle Zahlen $n \in \mathbf{N}$ auf die Eigenschaft $E(n)$ hin untersucht, nie ein Gegenbeispiel findet. Nichtterminierung können wir aber als Korrektheitsformel mit der Nachbedingung `false` ausdrücken:

```
{true}
   begin
     n := 0 ;
     while E(n) do
         n := n+1
   end
{ false }
```

Bevor wir dieses abstrakte Programm an NPPV geben, müssen wir die While-Schleife mit einer Invarianten versehen. Dies muß irgendeine Eigenschaft des Zustandsraumes sein, also eine Eigenschaft I, die nur von n abhängen kann, denn dies ist die einzige Variable in dem Programm. Als Invariante wählen wir einfach einen abstrakten Ausdruck {`I(n)`}. NPPV liefert uns die folgenden Verifikationsbedingungen, deren Interpretation offensichtlich ist:

```
I(0)
I(n) ==> I(n+1)
I(n) ==> E(n).
```

5 Erfahrungen in der Lehre

NPPV wurde für den Einsatz in der Lehre konzipiert. Ursprünglich wurde es in der Ausbildung von Studenten der Informatik in einem Bachelor-Studiengang an einem amerikanischen College (State University of New York, New Paltz) eingesetzt. Unter den Lehrenden bestand Einigkeit, daß dem "Programmieren durch Experimentieren" entgegengewirkt werden sollte. Programmverifikation wurde als ein geeignetes Gebiet ausgewählt, in dem eine Beschäftigung mit Programmen als formalen mathematische Objekten erzwungen wird.

Erste Versuche, einen Kurs zu diesem Thema im dritten Studienjahr (junior year) durchzuführen, erzeugten Frustration sowohl bei den Lehrenden als auch bei den Studenten. Zum einen konnten nur extrem kleine Programme mit Bleistift und Papier diskutiert werden. Das Problem bestand in der schieren Anzahl von überwiegend trivialen Verifikationsbedingungen.

Außerdem mußten die zu analysierenden Programme von Beginn an "stehen". Wollte man mit dem Beweis eines unfertigen oder fehlerhaften Programmes beginnen, um zunächst am Beweis zu scheitern und aus dieser Situation eine Korrektur oder eine Verbesserung des ursprünglichen Programmes zu entwickeln, so erwies sich der Aufwand immer als zu groß. Jede Modifikation des Programmes führte zu einer erneuten Erzeugung von teilweise nur geringfügig veränderten Verifikationsbedingungen.

Die genannten Probleme bewirkten, daß in dem Kurs immer nur eine kleine Reihe von Standardbeispielen, behandelt werden konnten. An die Entwicklung und Verifikation von selbst entwickelten Programmen war nicht zu denken.

Auch der qualitative Unterschied in den Verifikationsbedingungen, beispielsweise zwischen `N>0 ==> 0=0 AND N>0` und `x<N ==> x+1<=N`, wie sie schon in dem Programm `GAUSS` entstehen, geht in der ermüdenden Detailfülle unter. Die erste Bedingung erfordert einen simplen aussagenlogischen Schluß, die zweite verlangt eine Eigenschaft der verwendeten Datenstruktur.

Eine Falle, in die Studenten immer wieder tappten, betraf die Trennung (oder Vermischung) von Verifikationsbedingungen und Programm. Für die Schleife des erwähnten Programms `GAUSS` ergibt ein typischer erster Versuch: `sum=x*(x+1)/2.` Leider entsteht daraus eine Verifikationsbedingung

```
N <= x   ==>  (x*x+x) div 2=(N*N+N) div 2.
```

An dieser Stelle wurde meist argumentiert, daß es "aus dem Programm klar sei", daß immer `N >= x` gilt, folglich die Prämisse zu `N = x` werde, womit dann die Verifikationsbedingung erfüllt wäre.

In NPPV kann man alle Verifikationsbedingungen ausdrucken lassen. Diese Liste ist dann losgelöst von dem Programm aus dem sie entstanden ist, zu diskutieren, d.h. alle enthaltenen Bedingungen müssen als Tautologien nachgewiesen werden. In Abwesenheit des Programms gelingt es leichter, implizite Annahmen, die sich aus der "Kenntnis der Verhältnisse" ergeben, abzuschütteln und als problematisch zu erkennen.

Vor allem, um die Aufgabe der Verifikation von dem drückenden trivialen Ballast zu befreien, wurde NPPV entwickelt. Zweitens sollte auch die Entwicklungsumgebung so sein, wie die Studenten es damals von Turbo-Pascal gewohnt waren. Die Verwendung des Werkzeuges durfte keinen zusätzlichen Lernaufwand erfordern. Dennoch sollte das System auch nicht im "trial and error"-Modus einsetzbar sein. Beispielsweise erlaubt es der Syntaxprüfer nicht, Semikola ";" an Stellen zu benutzen, die zwar in Pascal erlaubt, begrifflich aber unsinnig sind, etwa vor einem "`end`".

Die oben diskutierten Probleme haben sich durch den Einsatz von NPPV weitgehend abgemildert. Fehler und Lücken in ihrer Argumentation werden dem Lernenden sofort – und nicht erst nach Rückgabe der Hausaufgaben – bewußt. Einfache aussagenlogische und viele triviale algebraische Schlüsse für die Standard-Datenstrukturen fallen weg, es gelingt eine Konzentration auf das wesentliche. Hausaufgaben können mit weit mehr Freiraum gestellt werden.

Das System wird aus der gleichen Motivation heraus und mit den gleichen Lernzielen auch im Grundstudium Informatik der Philipps-Universität Marburg, oder auch in den ersten Stunden eines Kurses zur Programmverifikation eingesetzt. Dabei werden die Studenten aufgefordert, sich ein möglichst "erhellendes" und möglichst abstraktes Programm auszudenken und zu verifizieren. Ein kleines Beispiel eines Studenten, das einen einfachen aber interessanten mathematischen Zusammenhang demonstriert, wollen wir hier kurz vorstellen.

Der Algorithmus repräsentiert ein Spiel, in dem ein Spieler einen gewissen Betrag einsetzt. Dann wird gewürfelt. Fällt der Würfel zu seinen Gunsten, so

erhält er das doppelte des Einsatzes, ansonsten verliert er diesen. Das Spiel endet, wenn der Spieler zum erstenmal gewinnt.

Eine Strategie für den Spieler besteht darin, in jedem neuen Versuch seinen Einsatz zu verdoppeln. Es läßt sich zeigen, daß sein Nettogewinn dann gerade dem allerersten Einsatz entspricht. Das folgende Programm implementiert das Spiel mit der genannten Strategie. Man beachte, daß `Wuerfel` durch eine unbekannte (abstrakte) Funktion dargestellt wird. Die Natur dieser Funktion darf natürlich keine Rolle spielen, genauso wie das Ergebnis eines Wurfes unbekannt ist. Es wird behauptet, daß der Gewinn des Spielers gerade seinem ersten Einsatz entspricht. Mit Vorbedingung {`einsatz = A`} und Nachbedingung {`gewinn - verlust = A`} können wir das Spiel und die Strategie des Spielers durch das folgende Programm modellieren:

```
i := 0; verlust := 0;
repeat
    i := i+1;
    verlust := verlust+einsatz;
    If Wuerfel(i)
        then gewinn := einsatz + einsatz
        else einsatz := einsatz + einsatz
until Wuerfel(i)
```

Das Finden einer geeigneten Invariante (vor dem `until` einzusetzen) ist das einzige interessante Problem – dessen Lösung wir hier offenlassen wollen – den Rest erledigt NPPV.

6 Zusammenfassung

Programmieren ist eine mathematische Tätigkeit. Die Erstellung eines Programmes sollte den gleichen Qualitätskriterien genügen wie die Abfassung eines mathematischen Beweises. Die Essenz zum Verständnis eines Programmes steckt in den Schleifeninvarianten. Sie entsprechen genau den Induktionshypothesen in mathematischen Beweisen. Wenn auch in der Praxis viele Programme nicht formal verifiziert werden können, so sollten zumindest Schleifen durch aussagekräftige Invarianten dokumentiert werden.

Mit Hilfe des Programmverifizierers NPPV, der auch mit abstrakten Programmschemata umgehen kann, haben wir den Zusammenhang zwischen Programmieren und Beweisen ausgelotet. Das System dient nicht nur zur Verifikation von konkreten Programmen, sondern liefert auch Bedingungen für die Korrektheit von Programmschemata welche sich in den algebraischen Axiomen der Datenstrukturen wiederspiegeln. NPPV ist für den Einsatz in der Lehre geeignet, da es den Lernenden von trivialen Überlegungen befreit und zum Kern der Sache hinführt.

Auf weitere *features* von NPPV, wie z.B. die Terminierungsanalyse, kann hier nicht eingegangen werden. Der theoretische Hintergrund und eine Reihe weiterer Fallbeispiele finden sich in [2].

Literatur

1. J.W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall, London, 1980.
2. H.P. Gumm. Generating algebraic laws from imperative programs. *Theoretical Computer Science*, 217(2):385–405, 1999.
3. H.P. Gumm and M. Sommer. *Einführung in die Informatik.* Oldenbourg Verlag, München, 4. Auflage, 1999.
4. C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:567–580, 1972.