Kapitel V:
Automatisierung der vollständigen Induktion

(Dieter Hutter)

1 Einführung

Vollständige Induktion ist in der Mathematik die grundlegende Beweistechnik, um Eigenschaften rekursiv definierter Objekte zu zeigen. Daher spielen Beweise durch Induktion in vielen Teilgebieten der Mathematik eine bedeutende Rolle. Die Beispiele reichen von der Analysis oder der Algebra bis zur Formalen Logik. In der Informatik ist die vollständige Induktion bei der Programmverifikation für die Verifikation von Schleifen und rekursiv definierten Funktionen unentbehrlich.

Die Bedeutung der Induktion liegt darin, daß mit ihrer Hilfe unendlich viele Objekte endlich repräsentiert werden können. Am bekanntesten ist dabei die vollständige Induktion über die natürlichen Zahlen. Das Induktionsprinzip ist dabei eng mit der Definition der natürlichen Zahlen verknüpft. Gibt man eine konstruktive Vorschrift für die Erzeugung der natürlichen Zahlen an, so könnten diese folgendermaßen lauten:

a) 0 ist eine natürliche Zahl,

b) ist n eine natürliche Zahl, so ist auch succ(n) eine natürliche Zahl und

c) es gibt keine anderen natürlichen Zahlen als die durch a) und b) konstruierbaren.

Die natürlichen Zahlen werden also dadurch charakterisiert, daß ein einzelnes Objekt - die 0 - explizit definiert wird und daß dann - ausgehend von diesem explizit definierten Objekt - neue Objekte durch Anwendung einer Konstruktionsvorschrift (wende succ auf eine natürliche Zahl an) aus den bereits bekannten Objekten erzeugt werden. Initial kennt man nur die 0 als natürliche Zahl, im ersten Schritt kommt succ(0) hinzu, im zweiten succ(succ(0)),... usw. So wird iterativ irgendwann jede natürliche Zahl erzeugt. Gleichzeitig ist aber auch sichergestellt, daß nur die natürlichen Zahlen erzeugt werden.

Dieselbe Konstruktionsvorschrift begegnet uns wieder, wenn wir Funktionen über den natürlichen Zahlen definieren wollen. Der Wert einer Funktion wird nur für wenige Argumente explizit definiert. Alle übrigen Werte werden *rekursiv* aus den bereits bekannten Werten berechnet. Definiert man zum Beispiel die Addition der natürlichen Zahlen rekursiv durch die Gleichungen:

$$\forall\ y{:}nat\ \ 0 + y = y \text{ und } \forall\ x,y{:}nat\ \ succ(x) + y = succ(x + y),$$

so sind nur die Werte eines Funktionsaufrufs 0 + y explizit vorgegeben. Die übrigen Werte werden rekursiv mit Hilfe der zweiten Gleichung aus dem explizit vorgegebenen Wert berechnet. Dabei ermittelt sich der Wert von succ(succ(0)) + y aus dem Wert von succ(0) + y Dieser basiert wiederum auf dem explizit gegebenen Wert von 0 + y. Ein solches

Definitionsprinzip ist deshalb korrekt, weil man aus der Konstruktionsvorschrift für die natürlichen Zahlen weiß, daß man jede natürliche Zahl dadurch erhält, daß man die Funktion succ endlich oft auf 0 anwendet. Umgekehrt erreicht man von jeder natürlichen Zahl aus die 0, wenn man lang genug in umgekehrter Richtung der obigen Konstruktionsvorschrift von succ(x) auf x zurückgeht.

Zum dritten Mal begegnet uns die Konstruktionsvorschrift bei *induktiven* Beweisen von Eigenschaften natürlicher Zahlen. Eine Eigenschaft E kann demnach folgendermaßen für alle natürlichen Zahlen nachgewiesen werden:

a) Die Eigenschaft gilt für n = 0 (Induktionsanfang);

b) wenn die Eigenschaft für eine Zahl n gilt, so gilt sie auch für ihren Nachfolger succ(n) (Induktionsschritt).

Der Nachweis einer Eigenschaft E erfolgt also explizit nur für die 0, ansonsten wird "nur" bewiesen, daß die Eigenschaft E bezüglich der Konstruktionsvorschrift für natürliche Zahlen invariant bleibt. Hat man also im Induktionsanfang die Eigenschaft E für die 0 bewiesen, so kann man in einem ersten Schritt mit Hilfe des Induktionsschrittes die Eigenschaft E für succ(0) folgern, im zweiten für succ(succ(0)),... usw.. Analog zur Konstruktionsvorschrift für die Erzeugung der natürlichen Zahlen ergibt sich somit eine Konstruktionsvorschrift für den Beweis der Eigenschaft E für jede beliebige natürliche Zahl.

Dieses Kapitel beschäftigt sich nun mit der Umsetzung dieser konstruktiven Sichtweise von Datenstrukturen, Funktionen und Beweisen auf die Ebene der Logik und des automatischen Beweisens. Der nächste Abschnitt beschäftigt sich mit den logischen Grundlagen der Induktion. Im Abschnitt 3 wird die Definition von Datenstrukturen behandelt und anschließend werden Funktionen, die auf diesen Datenstrukturen operieren, eingeführt. Der Abschnitt 4 widmet sich den verschiedenen Techniken für das Finden von Induktionsbeweisen.

2 Grundlagen

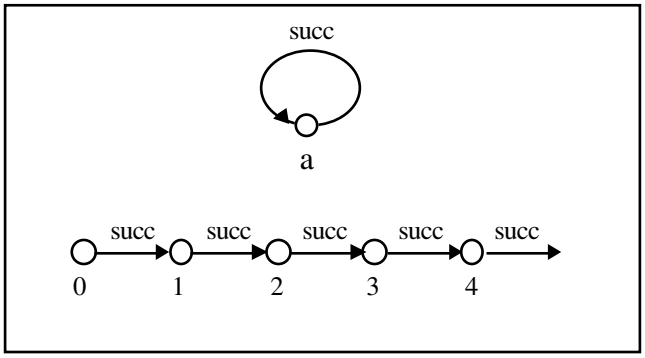
In der Einleitung wurden die natürlichen Zahlen durch Angabe einer Konstruktionsvorschrift beschrieben. Für das Beweisen von Eigenschaften ist diese Darstellung ungeeignet, man benötigt hier eine logische Axiomatisierung der natürlichen Zahlen, d.h. man sucht eine geeignete Formelmenge, die die natürlichen Zahlen hinreichend genau beschreibt. Eine solche Axiomatisierung, die die natürlichen Zahlen sogar bis auf Isomorphie genau beschreibt, wurde bereits im letzten Jahrhundert von Richard Dedekind (1887) und Guiseppe Peano (1889) angeben:

a) Es gibt eine natürliche Zahl 0 (*Basiskonstante*) und eine injektive Funktion succ : nat $\rightarrow$ nat (*Konstruktorfunktion*). Der Nachfolger succ(x) einer natürlichen Zahl x ist eine von 0 verschiedene natürliche Zahl.

b) Hat 0 die Eigenschaft E und hat, wenn eine natürliche Zahl x die Eigenschaft E hat, ihr Nachfolger succ(x) ebenfalls die Eigenschaft E, so haben alle natürlichen Zahlen die

Eigenschaft E, d.h. $E(0) \wedge (\forall\ y{:}nat\ \ E(y) \Rightarrow E(succ(y))) \ \Rightarrow\ \forall\ x{:}nat\ E(x)$ (*Induktionsaxiom*).

Teil a) umfaßt gerade die Axiomatisierung des ersten Teils unserer Konstruktionsvorschrift. Aus 0 und succ aufgebaute Terme sollen natürliche Zahlen darstellen. Schwierig wird dagegen die Axiomatisierung des zweiten Teils unserer Konstruktionsvorschrift, daß es neben solchermaßen aufgebauten Zahlen keine andere mehr gibt. Um dies sicherzustellen, benötigt man den Teil b) der Peano-Axiome, das sogenannte Induktionsaxiom. Im Gegensatz zu Teil a) ist aber das Induktionsaxiom in der Prädikatenlogik erster Stufe nicht formulierbar, da über Eigenschaften (d.h. z.B. Prädikate) quantifiziert werden müßte. Verzichtet man dagegen bei der logischen Beschreibung auf das in erster Stufe nicht darstellbare Induktionsaxiom, so werden die natürlichen Zahlen nur "ungenau" charakterisiert. Die Beschreibung trifft außer auf die natürlichen Zahlen auch auf andere Mengen, sogenannte Nichtstandardmodelle zu. Selbst einfache Eigenschaften der natürlichen Zahlen können nicht mehr bewiesen werden, da diese Eigenschaften nicht auf alle Nichtstandardmodelle zutreffen.



Fügt man zum Beispiel zu den natürlichen Zahlen noch ein neues Element a hinzu und erweitert die Definition von succ um succ(a) = a, so erfüllt auch dieses Modell (vgl. obige Abb.) den ersten Teil der Peano-Axiome. Verzichtet man nun in einem automatischen Beweiser völlig auf die Induktion, so ist zum Beispiel nicht mehr möglich, die Aussage $\forall\ x{:}nat\ x \neq succ(x)$ zu beweisen, da diese Aussage in dem oben genannten "größeren" Modell wegen a = succ(a) nicht gilt. In unserem Beispiel benötigt man zum Nachweis der Aussage eine Instantiierung des Induktionsaxioms mit der konkreten Eigenschaft $E \equiv x \neq succ(x)$, d.h. ein auf erster Stufe formulierbares Axiom:

$$0 \neq succ(0) \wedge (\forall\ y{:}nat\ \ y \neq succ(y)\ \Rightarrow succ(y) \neq succ(succ(y))) \Rightarrow \forall\ x{:}nat\ \ x \neq succ(x)$$

mit dessen Hilfe sich leicht die gewünschte Aussage beweisen läßt.

Wie das Beispiel der natürlichen Zahlen zeigt, lassen sich induktive Datenstrukturen in der Regel nicht in der Prädikatenlogik erster Stufe hinreichend genau beschreiben, denn für den Nachweis vieler Eigenschaften benötigt man das auf erster Stufe nicht formulierbare Induktionsaxiom. Bisher vorgestellte automatische Beweiser basieren jedoch gerade auf der Prädikatenlogik erster Stufe, so daß diese selbst einfache Eigenschaften der natürlichen Zahlen wegen deren unvollständigen Charakterisierung nicht beweisen können.

Die zentrale Idee ist nun, daß auf erster Stufe nicht darstellbare Induktionsaxiom in beliebig vielen Instanzen (Formeln erster Stufe) in die Axiomenmenge aufzunehmen. In diesen

Instanzen ist die all-quantifizierte Eigenschaft E durch eine prädikatenlogische Formel mit einer freien Variable x ersetzt. Für die natürlichen Zahlen kann dann zum Beispiel aus den beiden Formeln

$$P(0) \text{ und } \forall\, x{:}nat\;\; P(x) \Rightarrow P(succ(x))$$

mit Hilfe der Instanz des Induktionsaxioms:

$$(P(0) \wedge \forall\, x{:}nat\;\; P(x) \Rightarrow P(succ(x))) \Rightarrow \forall\, x{:}nat\;\; P(x)$$

die Formel $\forall\, x{:}nat\;\; P(x)$ mittels Modus Ponens für eine beliebige Formel P abgeleitet werden. Für den Beweis der bereits angesprochenen Formel $\forall\, x{:}nat\;\; x \neq succ(x)$ reicht es also aus, die beiden Formeln

$$0 \neq succ(0) \text{ und } \forall\, y{:}nat\;\; y \neq succ(y) \;\Rightarrow succ(y) \neq succ(succ(y))$$

abzuleiten, um mit Hilfe einer Instanz des Induktionsaxioms die gewünschte Formel zu erhalten.

Es stellt sich die Frage, ob wir mit einem solchen sogenannten *Induktionsschema*, das bei Bedarf entsprechend instantiiert wird, die natürlichen Zahlen eindeutig beschreiben können. Wie bereits Thoralf Skolem in den zwanziger Jahren dieses Jahrhunderts bewiesen hat, ist dies aber leider nicht der Fall. Der Beweis sprengt den Rahmen dieses Buches, doch die folgenden Überlegungen sollen das Resultat verdeutlichen:

Eine Eigenschaft E kann durch eine Menge M von natürlichen Zahlen charakterisiert werden, indem eine Zahl x genau dann die Eigenschaft E hat, wenn x in M enthalten ist. Die Menge aller Teilmengen der natürlichen Zahlen ist überabzählbar. Das heißt, es gibt keine bijektive Funktion f, die die natürlichen Zahlen in die Teilmengen der natürlichen Zahlen abbildet (gäbe es eine solche Funktion, so könnte eine Menge $M := \{x \mid x \notin f(x)\}$ nicht f-Bild einer Zahl a sein, denn $a \in M$ ist nach Definition gleichbedeutend mit $a \notin f(a)$. M ist aber sicherlich eine Teilmenge der natürlichen Zahlen !). Andererseits ist die Menge aller prädikatenlogischer Formeln erster Stufe abzählbar. Hier gibt es eine bijektive Funktion f, die jeder natürlichen Zahl eine prädikatenlogische Formel zuordnet. Somit gibt es also Eigenschaften bzw. Teilmengen von natürlichen Zahlen, die nicht durch eine prädikatenlogische Formel beschrieben werden können und für die dann auch keine Instanz des Induktionsaxioms gebildet werden kann.

Mit Hilfe des Induktionsschemas können die natürlichen Zahlen nicht eindeutig charakterisiert werden, da es immer auch sogenannte Nichtstandardmodelle gibt, die diese Axiome erfüllen. Die nächste Frage ist, ob man mit diesem Induktionsschema, wenn man Addition und Multiplikation rekursiv definiert, zumindestens alle in der Arithmetik gültigen Formeln ableiten kann. Aber auch dieses ist nicht der Fall, wie Kurt Gödel im Jahre 1931 gezeigt hat. Unsere Axiome für die natürlichen Zahlen sind mit dem Induktionsschema unvollständig. Es gibt in der Arithmetik wahre Sätze, die wir aus unserer Axiomatisierung nicht ableiten können. Der Hintergrund für dieses Resultat liegt in der Ausdrucksmächtigkeit der Arithmetik. Wir können in der Arithmetik Formeln und die Ableitbarkeit von Formeln aus unserem Axiomensystem kodieren. Es läßt sich dann eine Formel G konstruieren, die von sich selbst sagt, daß sie nicht ableitbar ist. G ist in der Arithmetik gültig, wäre sie aber ableitbar, wäre unser Axiomensystem

inkonsistent.

Die Unvollständigkeit liegt aber nicht daran, daß wir ein “schlechtes” Induktionsschema gewählt haben und mit einem anderen Induktionsschema diesen Mangel hätten beseitigen können. Jede vollständige (und konsistente) Axiomatisierung der Arithmetik ist auf erster Stufe unentscheidbar. Wir können bei einer vollständigen Axiomatisierung in der Regel nicht feststellen, ob eine Formel ein Axiom ist oder nicht. Somit gibt es also kein Induktionsschema, mit dessen Hilfe wir die Arithmetik eindeutig beschreiben könnten.

3 Aufbau einer Datenbasis

3.1 Definition von Datenstrukturen

In der Einführung wurden die natürlichen Zahlen mit Hilfe einer Konstruktionsvorschrift beschrieben, die nacheinander alle Zahlen erzeugt. Im vorigen Abschnitt wurde die logische Axiomatisierung solcher Mengen beleuchtet. In diesem Abschnitt beschäftigen wir uns damit, wie man allgemein Datenstrukturen, die mit Hilfe einer (endlichen) Konstruktionsvorschrift erzeugbar sind, axiomatisiert und damit einem automatischen Beweiser zugänglich macht.

Zunächst müssen wir konkretisieren, was wir unter einer Konstruktionsvorschrift verstehen. Die Definition einer Datenstruktur erfolgt durch Angabe einer Signatur Σ, das heißt man gibt eine Menge von Konstanten (sogenannte Basiskonstante) und Funktionen (sogenannte Konstruktorfunktionen) an, aus denen jedes Objekt der Datenstruktur aufgebaut sein muß (vgl. constructor-constants [Aub79], shell-principle bei [BM79], structure-definition bei [BHHW86]). Im Fall der natürlichen Zahlen ist dies gerade die Signatur $\Sigma_{nat} = \{0, succ\}$.

Der Aufbau der intendierten Datenstruktur erfolgt dann analog zu der Konstruktionsvorschrift in der Einleitung. Ausgehend von den Basiskonstanten wird die Datenstruktur iterativ um diejenigen Objekte erweitert, die durch Anwendung einer Konstruktorfunktion auf bereits vorhandene Objekte dieser Datenstruktur entstehen. Bei den natürlichen Zahlen erhält man nacheinander die Objekte 0, succ(0), succ(succ(0)) .... Die in diesem Beispiel entstandene Menge entspricht intuitiv den natürlichen Zahlen.

Aus bereits definierten Datenstrukturen können wir weitere komplexere Datenstrukturen zusammensetzen. So können Listen von natürlichen Zahlen durch die Basiskonstante nil und die Konstruktorfunktion cons: nat $\times$ list $\rightarrow$ list definiert werden, wobei nat die bereits bekannten natürlichen Zahlen bezeichnen soll. Man erhält dann die Objekte nil, cons(0 nil), cons(succ(0) nil),...., cons(0 cons(0 nil)),...

Die Aufgabe eines Beweissystems ist es, für diese intuitiv definierte Datenstruktur einen Satz von Axiomen zu erzeugen, der diese Datenstruktur hinreichend genau beschreibt. Hierfür nehmen wir im weiteren an, der Benutzer habe die Symbole $c_1 \ldots c_n$ als Basiskonstante und $f_1 \ldots f_m$ als Konstruktorfunktionen für die zu axiomatisierende Datenstruktur s ausgezeichnet.[1]

---

[1] Wir werden im Folgenden mit * stets eine Sequenz von Objekten, Sorten etc. bezeichnen. s^* bezeichnet also eine Sequenz $s_1 \ldots s_n$ von Sorten. $f(t)^*$ steht als Abkürzung für eine Sequenz $f_1(t_1) \ldots f_n(t_n)$ von Termen.

Im weiteren bezeichne s^*_i den Definitionsbereich einer Konstruktorfunktion f_i. Ist s in s^*_i enthalten, so nennt man f_i *reflexiv*. Hat s mindestens einen reflexiven Konstruktor, so nennt man s *induktiv* definiert.

Eine Forderung ist, daß sich jedes Objekt mit Hilfe der Konstruktorfunktionen konstruktiv aus den Basiskonstanten erzeugen läßt. Wie wir in der Einleitung gesehen haben, ist gerade diese Eigenschaft nicht auf erster Stufe formulierbar, da man hierfür ein Induktionsaxiom benötigt. Analog zu dem einleitenden Beispiel der natürlichen Zahlen wird statt eines Induktionsaxioms ein Induktionsschema für die Datenstruktur s erzeugt, aus dem während eines Beweises die benötigten Instanzen des Induktionsaxioms gebildet werden können. Dieses Induktionsschema hat allgemein die folgende Gestalt:

$$
\begin{aligned}
\{ \quad & P(c_1) \wedge \ldots \wedge P(c_n) \wedge \\
& \wedge \forall\, y^*_1 : s^*_1 \;\; (P(y_{1,1}) \wedge \ldots \wedge P(y_{1,n(1)})) \Rightarrow P(f_1(y^*_1)) \\
& \ldots \\
& \wedge \forall\, y^*_m : s^*_m \;\; (P(y_{m,1}) \wedge \ldots \wedge P(y_{m,n(m)})) \Rightarrow P(f_m(y^*_m)) \} \\
\Rightarrow\ & \forall\, x{:}s \;\; P(x) \;\; ,
\end{aligned}
$$

wobei $y_{i,1} \ldots y_{i,n(i)}$ die Variablen mit der Sorte s aus den jeweiligen Variablenlisten y^*_i sind.

Man beachte, daß auch im allgemeinen das Induktionsschema “schwächer” als das eigentliche Induktionsaxiom ist, da das Induktionsaxiom nur für Eigenschaften verwendet werden kann, die mit Hilfe einer prädikatenlogischen Formel ausdrückbar sind (vgl. Abschnitt 2). Die intendierte Datenstruktur kann also damit in der Regel nicht eindeutig charakterisiert werden.

Das Induktionsaxiom (bzw. das schwächere Induktionsschema) beschreibt die sogenannte *Termerzeugtheit* der zu definierenden Datenstruktur. Jedes Objekt der Datenstruktur ist aus den vorgegebenen Basiskonstanten und Konstruktorfunktionen sowie aus Objekten der zugrundeliegenden Datenstrukturen aufgebaut. Nicht geregelt wird durch das Induktionsschema, ob ein Objekt der Datenstruktur verschiedene Namen haben kann, d.h. ob zum Beispiel zwei Basiskonstante dasselbe Objekt bezeichnen. In den bisherigen Beispielen der natürlichen Zahlen und den Listen natürlicher Zahlen möchte man, daß zwei verschiedene Namen auch verschiedene Objekte bezeichnen, denn 0 soll ein anderes Objekt bezeichnen als succ(succ(0)) und cons(0 nil) ein anderes als cons(succ(0) nil).

Hat jedes Objekt einer Datenstruktur s genau einen Namen - d.h. es gibt nur einen aus Basiskonstanten und Konstruktorfunktionen zusammengesetzten Term, der ein Objekt beschreibt -, so nennt man s eine *initiale* Datenstruktur. Unsere Beispiele der natürlichen Zahlen und der Listen von natürlichen Zahlen sind initiale Datenstrukturen.

Ein Beispiel für nicht-initiale Datenstrukturen sind Mengen von natürlichen Zahlen. Gibt man als Konstruktionsvorschrift die Signatur Σ_{set} = {empty, ins} an, so möchte man verschiedene Namen demselben Objekt zuordnen: ins(0 ins(0 empty)) soll genauso die einelementige Menge {0} wie ins(0 empty) denotieren. Vergleicht man das erzeugte Induktionsschema für Listen und Mengen, so unterscheiden sich beide nicht. Das obige Induktionsschema ist in beiden Datenstrukturen (bis auf Signaturwechsel) dasselbe. Die Unterscheidung erfolgt erst durch

weitere Axiome, die die Beziehungen zwischen den einzelnen Namen regeln.

Wir werden uns im folgenden auf die Axiomatisierung von initialen Datenstrukturen beschränken, da andere Datenstrukturen von den bereits genannten Beweissystemen zur Zeit nicht unterstützt werden.

Für die Axiomatisierung initialer Datenstrukturen benötigen wir zusätzliche Formeln, die ausdrücken, daß je zwei verschiedene Namen auch zwei verschiedene Objekte bezeichnen. Das heißt:

- daß verschiedene Basiskonstanten auch verschiedene Objekte denotieren,
- daß zwei Terme mit verschiedenen führenden Konstruktorfunktionen verschiedene Objekte bezeichnen und
- daß eine Basiskonstante nicht mit Hilfe einer Konstruktorfunktion beschrieben werden kann,
- daß die Konstruktorfunktionen injektiv sein müssen.

Allgemein lauten damit die sogenannten *Eindeutigkeitsaxiome* wie folgt:

$c_i \neq c_j$ für alle $i,j \in \{1,...,n\}$ und $i \neq j$.

(alle Basiskonstanten sind verschieden)

$\forall\, y^*_i : s^*_i \;\forall\, y_j : s^*_j \;\; f_i\,(y^*_i) \neq f_j\,(y^*_j)$ für alle $i,j \in \{1,...,m\}$ und $i \neq j$

(je zwei Terme mit unterschiedlichen , führenden Konstruktorfunktionen sind verschieden)

$\forall\, y^*_i : s^*_i \;\; c_j \neq f_i\,(y^*_i)$ für alle $j \in \{1,...,n\}$ und für alle $i \in \{1,...,m\}$

(jede Basiskonstante ist verschieden zu einem zusammengesetzten Term)

$\forall\, x^*_i : s^*_i \;\forall\, y^*_i : s^*_i \;\; f_i\,(x^*_i) = f_i\,(y^*_i) \Rightarrow x^*_i = y^*_i$ für alle $i \in \{1,...,m\}$

(Konstruktorfunktionen sind injektiv)

Für die Datenstruktur nat erhält man beispielsweise:

$\forall\, x : nat \;\; 0 \neq succ(x)$ und $\forall x,y \; succ(x) = succ(y) \Rightarrow x = y$.

Für die Datenstruktur list lauten die beiden Eindeutigkeitsaxiome :

$\forall\, u : nat \;\forall\, v : list \;\; nil \neq cons(u\; v)$ und

$\forall\, y,v : list \;\forall\, x, u : nat \; cons(x\; y) = cons(u\; v) \;\Rightarrow\; x = u \wedge y = v$.

Zusätzlich zu den die Datenstruktur festlegenden Mengen der Basiskonstanten und Konstruktorfunktionen wird bei initialen Datenstrukturen zu jeder Argumentstelle i einer Konstruktorfunktion f eine sogenannte *Selektorfunktion* g_i eingeführt, mit deren Hilfe auf die entsprechende Argumentstelle x_i eines Terms $f(x_1 \ldots x_i \ldots x_n)$ zugegriffen werden kann. Das heißt, es gilt $g_i(f(x_1 \ldots x_i \ldots x_n)) = x_i$. Im Fall der natürlichen Zahlen erhält man eine Selektorfunktion pred : nat → nat mit der Eigenschaft pred(succ(x)) = x. Das heißt, die Funktion pred liefert intuitiv gerade den strukturellen Vorgänger einer natürlichen Zahl.

Für die Konstruktorfunktion cons erhält man zum einen eine Selektorfunktion g_1: list → nat, die intuitiv gerade das erste Element einer Liste liefert, zum anderen die Selektorfunktion g_2: list → list, die intuitiv die Restliste ohne das erste Element liefert.

Das Induktionsschema für nat läßt sich dann mit Hilfe des Selektors pred folgendermaßen formulieren:

$$\{ \forall x : nat \ [x = succ(pred(x)) \Rightarrow P(pred(x))] \Rightarrow P(x) \}$$
$$\Rightarrow \forall x{:}nat \ P(x) .$$

Allgemein können wir mit Hilfe der Selektoren das Induktionsschema für eine Datenstruktur s folgendermaßen umformulieren.

$$\{ \forall x : s \ (x = f_1(g_1(x)^*) \Rightarrow P(g_{1,1}(x)) \wedge .. \wedge P(g_{1,n(1)}(x)))$$
$$...$$
$$\wedge (x = f_m(g_m(x)^*) \Rightarrow P(g_{m,1}(x)) \wedge ... \wedge P(g_{m,n(m)}(x)))$$
$$\Rightarrow P(x) \}$$
$$\Rightarrow \forall x{:}s \ P(x)$$

$g_{i,1}$... $g_{i,n(i)}$ bezeichnet diejenigen Selektorfunktionen aus g_i^*, deren Bildbereich gerade wieder s ist.

3.2 Definition von Funktionen

Aufbauend auf den bereits definierten Datenstrukturen können Funktionen axiomatisch beschrieben werden, um anschließend Beweise über deren Eigenschaften zu führen. Im allgemeinen werden Funktionen durch Mengen von prädikatenlogischen Formeln denotiert, die die typischen Eigenschaften einer Funktion beschreiben. Man spricht in diesem Zusammenhang von einer *deklarativen* Beschreibung einer Funktion. Dabei ist in der Regel diese Axiomatisierung nicht eindeutig: es gibt nicht die *eine* Funktion, die diese Axiome erfüllt, sondern meistens mehrere (oder im ungünstigsten Fall gar keine). Wollen wir z.B. Eigenschaften einer Gruppe mit einer Funktion f nachweisen, so sind von dieser Funktion nur die Gruppeneigenschaften (z.B. Assoziativität) bekannt. Ob f die Multiplikation oder die Addition denotiert, spielt dabei keine Rolle, etwaige nachgewiesene Eigenschaften gelten für beide Funktionen.

Dagegen werden im Induktionsbeweisen Funktionen meistens *konstruktiv* definiert. Dies erfolgt durch Angabe eines Algorithmus, der auf den bereits vorhandenen Datenstrukturen arbeitet und mit Hilfe von Fallunterscheidung, Rekursion und Komposition definiert ist. Der Hintergrund für diese Vorgehensweise liegt darin, daß - wie in der Einleitung bereits erläutert wurde - in einer rekursiven Funktionsdefinition eine Konstruktionsvorschrift für die zugrunde liegende Datenstruktur versteckt ist. Eine solche Konstruktionsvorschrift kann nämlich als Induktionsschema interpretiert werden, mit dessen Hilfe Eigenschaften der definierten Funktion bewiesen werden können. Induktion und Rekursion sind zwei verschiedene Ansichten einer gemeinsamen Konstruktionsvorschrift. Wir werden später auf diese Beziehung noch zurückkommen.

In der Einleitung wurde nach der Einführung der Datenstruktur nat eine Menge bedingter Gleichungen zur Definition der Addition zweier natürlicher Zahlen angegeben, die wir mit Hilfe von Selektoren folgendermaßen notieren können:

$\forall$ x, y : nat $\quad x = 0 \Rightarrow (x + y) = y$

$\forall$ x, y : nat $\quad x = succ(pred(x)) \Rightarrow (x + y) = succ(pred(x) + y)$

Diese Menge spezifiziert einen Algorithmus, mit dessen Hilfe man zum Beispiel den Term succ(0) + succ(0) "ausrechnen" kann:

succ(0) + succ(0) = succ(0 + succ(0)) = succ(succ(0))

Im allgemeinen erfolgt die Beschreibung eines Algorithmus für eine Funktion f durch eine Menge bedingter Gleichungen, den sogenannten *Definitionsformeln* von f. Diese haben die Form $\forall\ x_1,\ldots,x_n\ C \Rightarrow f(x_1 \ldots x_n) = t$, wobei man C die *Bedingung* und t den *Ergebnisterm* der Definitionsformel nennt. In einer Definitionsformel von f treten ausschließlich die *formalen Parameter* $x_1,\ldots,x_n$, Basiskonstante, Konstruktor- und Selektorfunktionen, bereits definierte Funktionen und (bei Rekursion) die zu definierende Funktion f selbst auf. f tritt selbst aber nicht in den Bedingungen C auf. Eine Funktion f nennt man dann *rekursiv*, wenn es mindestens eine Definitionsformel von f gibt, in der f im Ergebnisterm auftritt.

Um sicherzustellen, daß durch eine Menge von bedingten Gleichungen tatsächlich ein auf allen Eingabewerten definierter Algorithmus beschrieben wird, müssen die folgenden sogenannten *Zulässigkeitsbedingungen* erfüllt sein:

(1) Der Algorithmus terminiert auf allen möglichen Eingabewerten. (*Terminierung*)

(2) Für jeden möglichen Eingabewert ist mindestens die Bedingung einer Definitionsformel erfüllt. (*Vollständigkeit*)

(3) Die Definition ist widerspruchsfrei, d.h. sind die Bedingungen mehrerer Definitionsformeln erfüllt, so spielt es keine Rolle, welche Definitionsformel zur Berechnung des Ergebnisses herangezogen wird. (*Eindeutigkeit*)

Im Falle der obigen Definition von '+' ist die Terminierung durch die Eigenschaft der Datenstruktur nat gesichert, daß jedes Objekt dieser Datenstruktur entweder gleich 0 ist oder nach endlich vielen Anwendungen der Vorgängerfunktion pred zu 0 wird. Das heißt, zur Berechnung einer beliebigen Summe (x + y) zweier Objekte x und y aus nat sind nur endlich viele rekursive Aufrufe nötig, um zum Basisfall zu gelangen. Die Vollständigkeit und Eindeutigkeit der Definition von '+' wird durch die Axiome von nat erzwungen.

Ein weiteres Beispiel für eine zulässige Definition ist die folgende Beschreibung einer Funktion '-', die die Differenz zweier natürlicher Zahlen berechnet:

$\forall$ x, y : nat $\quad y = 0 \Rightarrow x - y = x$

$\forall$ x, y : nat $\quad \neg\, y = 0 \wedge x = 0 \Rightarrow\ x - y = 0$

$\forall$ x, y : nat $\quad \neg\, y = 0 \wedge \neg\, x = 0 \Rightarrow x - y = pred(x) - pred(y)$

In den nachfolgenden Abschnitten wollen wir näher auf die Überprüfung der oben genannten

Zulässigkeitsbedingungen (1) - (3) eingehen. Dazu nehmen wir an, daß die Menge von bedingten Gleichungen, die f definiert, folgendermaßen aussieht:

$$\begin{aligned} &\forall\, x^* \quad C_1 \;\Rightarrow\; f(x^*) = t_1\,, \\ &\qquad \ldots \qquad\qquad\qquad\qquad\qquad\qquad (\text{❖}) \\ &\forall\, x^* \quad C_p \;\Rightarrow\; f(x^*) = t_p \end{aligned}$$

Aus Gründen der Übersichtlichkeit lassen wir hier und gelegentlich auch in den folgenden Abschnitten die Sorten der Variablen weg.

3.2.1 Eindeutigkeit und Vollständigkeit

Die *Eindeutigkeit* sichert je nach Blickwinkel die Widerspruchsfreiheit der Definition beziehungsweise die Persistenz der zugrunde liegenden Datenstruktur. Nehmen wir an, wir könnten zum Beispiel eine Funktion f über nat für eine Eingabe 0 sowohl zu 0 als auch succ(0) "auswerten". Das heißt, es gilt f(0) = 0 und f(0) = succ(0). Legt man die natürlichen Zahlen für nat zugrunde, ist f keine Funktion, da der Wert für 0 nicht eindeutig definiert ist. Umgekehrt läßt sich aus der Definition von f ableiten, daß 0 = succ(0) gilt. Die Definition von f identifiziert also verschiedene Objekte unserer Datenstruktur nat, so daß die Persistenz nicht mehr gewahrt ist (in unserer Axiomatisierung von nat wird die Axiomenmenge sogar widersprüchlich).

Die Eindeutigkeit kann dadurch erreicht werden, daß man sicherstellt, daß wann immer die Bedingungen zweier Definitionsformeln einer Funktion vom Typ (❖) gleichzeitig erfüllt sind, beide Ergebnisterme dieser Definitionsformeln dieselben Werte liefern. D.h. für alle $i,j \in \{1,...,p\}$ kann aus der Datenbasis die Formel $\forall\, x_1,...,x_n\ (C_i \wedge C_j) \Rightarrow t_i = t_j$ abgeleitet werden. Aus praktischen Gründen wird in vielen Systemen diese Forderung dahingehend verschärft, daß für alle möglichen Eingabewerte höchstens eine Bedingung erfüllt ist. D.h. für alle $i, j \in \{1, ...,p\}$ ist bereits die Formel $\forall\, x_1,...,x_n \neg (C_i \wedge C_j)$ ableitbar.

Um die Eindeutigkeit von + nachzuweisen, muß zum Beispiel die Formel:

$$\forall\, x : \text{nat}\ \neg (x = 0 \wedge x = \text{succ}(\text{pred}(x)))$$

bewiesen werden, die unmittelbar aus den Axiomen für die Datenstruktur nat folgt.

Die *Vollständigkeit* garantiert, daß für alle möglichen Eingabewerte mindestens eine Bedingung erfüllt ist. D.h. es muß die Formel $\forall\, x_1,...,x_n\ C_1 \vee ... \vee C_p$ aus der bisher aufgebauten Datenbasis ableitbar sein. Für die Funktion + lautet diese Formel folgendermaßen:

$$\forall\, x : \text{nat}\ x = 0 \vee x = \text{succ}(\text{pred}(x)).$$

Für die Überprüfung der Eindeutigkeit einer Funktion mit p Definitionsformeln sind somit $1/2\ (p^2 - p)$ Teilbeweise nötig. Um diese Anzahl zu verringern, wird entweder das Definitionsprinzip für Funktionen dahingehend eingeschränkt, daß bereits syntaktisch die Eindeutigkeit und Vollständigkeit garantiert wird (vgl. if-then-else-Konstrukt bei [BM79] und Definition by Cases [Aub79]), oder es werden effiziente Algorithmen eingesetzt, die sich gegenseitig ausschließende Bedingungen erkennen und so eine Klasseneinteilung der Definitionsfälle der zu untersuchenden Funktion vornehmen, so daß die anfallende

Überprüfung auf Vollständigkeit und Eindeutigkeit nur relativ zu den einzelnen Klassen erfolgen muß.

Ein Klassifizierungsmerkmal ist das Vorhandensein bestimmter Bedingungsliterale, deren Disjunktion tautologisch ist [Aub79]. Trivialerweise gilt $\forall$ x:nat $\neg$ P(x) $\vee$ P(x), so daß sich einerseits die Bedingungen der beiden Fälle gegenseitig ausschließen (Eindeutigkeit). Andererseits muß somit für jedes Objekt der Datenstruktur nat eines der beiden Literale erfüllt sein (Vollständigkeit). Ein anderes Beispiel sind bei initialen Datenstrukturen Literale, die den Aufbau eines Objektes beschreiben, wie z.B. x = 0 und x = succ(pred(x)).

Wir wollen dies kurz am Beispiel einer Definition des größten gemeinsamen Teiler (ggt) verdeutlichen:

$\forall$ x,y : nat x = 0 $\Rightarrow$ ggt(x y) = y (1)

$\forall$ x,y : nat $\neg$ x = 0 $\wedge$ y = 0 $\Rightarrow$ ggt(x y) = x (2)

$\forall$ x,y : nat $\neg$ x = 0 $\wedge$ $\neg$ y = 0 $\wedge$ (x - y) = 0 $\Rightarrow$ ggt(x y) = ggt(x (y - x)) (3)

$\forall$ x,y : nat $\neg$ x = 0 $\wedge$ $\neg$ y = 0 $\wedge$ $\neg$ (x - y) = 0 $\Rightarrow$ ggt(x y) = ggt((x - y) y) (4)

Die Bedingungen x = 0 und $\neg$ x = 0 schließen sich gegenseitig aus und gleichzeitig ist für beliebiges x einer der beiden Fälle erfüllt. Somit lassen sich die Definitionsformeln des ggt in zwei Klassen einteilen. Die erste Klasse bilden die Formeln, deren Bedingungen das Literal x = 0 enthalten: {(1)}. Die zweite Klasse bilden die Formeln, deren Bedingungen das Literal $\neg$ x = 0 enthalten: {(2), (3), (4)}. Analog verfährt man mit den Literalen y = 0 und $\neg$ y = 0, um die zweite Klasse in die Unterklassen {(2)} und {(3), (4)} zu unterteilen. Zuletzt unterteilt man die letzte verbliebene Unterklasse mit mehr als einem Definitionsfall mit Hilfe der Literale (x - y) = 0 und $\neg$ (x - y) = 0 in zwei einelementige Klassen.

Natürlich ist nicht immer eine Klassifizierung in einelementige Klassen möglich. In diesen Fällen muß die Vollständigkeit und die Eindeutigkeit innerhalb der Klassen durch das Induktionssystem bewiesen werden.

3.2.2 Terminierung

In der Einleitung zu diesem Abschnitt erwähnten wir, daß man statt an deklarativen an konstruktiven Definitionen mittels Rekursion für Funktionen interessiert ist. Der Hintergrund liegt - wie gesagt - darin, daß in einer solchen Definition eine Konstruktionsvorschrift zur Erzeugung der zugrunde liegenden Datenstruktur und somit ein Induktionsschema versteckt ist.

Betrachtet man zum Beispiel die obige Definition von +, so unterscheidet die Definition von + den Basisfall x = 0 und den Rekursionsfall x = succ(pred(x)), wobei die Rekursion die Berechnung des Funktionswertes für x auf die Berechnung des Funktionswertes für den Vorgänger pred(x) zurückführt. Hat man bewiesen, daß die Definition von + auf allen Eingabewerten terminiert, so weiß man, daß sich jedes Objekt aus nat durch eine endliche Anzahl von Anwendungen der Vorgängerfunktion pred auf 0 abbilden läßt. Umgekehrt rechtfertigt diese Beobachtung das Induktionsprinzip:

$\{ \forall x{:}nat\ [x = succ(pred(x)) \Rightarrow P(pred(x))\ ] \Rightarrow P(x)\ \} \Rightarrow \forall x{:}nat\ P(x)$.

Dieses, aus der rekursiven Definition von + abgeleitete Prinzip ist aber nur deshalb korrekt, weil der Algorithmus für + auf allen Eingabewerten terminiert. Betrachtet man zum Beispiel die Tautologie f(x) = f(x) als rekursive Definition für eine Funktion f über nat, so terminiert f auf keinem Eingabewert. Insbesondere kann mittels der in der Definition vorkommenden "Rekursion" kein Objekt auf einen irgendwie gearteten Basisfall zurückgeführt werden. Aus der Definition von f läßt sich kein Konstruktionsprinzip für nat ableiten. Der Versuch, ein Induktionsprinzip aus der Definition von f zu gewinnen, führt zu einem völlig falschen Induktionsschema:

$\{ \forall x{:}nat\ P(x) \Rightarrow P(x)\ \} \Rightarrow \forall x{:}nat\ P(x)$.

Wie die obigen Beispiele zeigen, ist es in Induktionsbeweisen wichtig, eine Menge bedingter Gleichungen als terminierenden Algorithmus interpretieren zu können, um daraus neue Induktionsschemata zu gewinnen. Leider ist aber im allgemeinen der Nachweis, daß ein Algorithmus terminiert, unentscheidbar. Dies bedeutet, daß es prinzipiell kein Verfahren gibt, das für alle Algorithmen entscheiden kann, ob diese terminieren oder nicht. Dieses als sogenanntes *Halteproblem* bekannt gewordene Phänomen wurde von Alan Turing im Jahre 1936 gezeigt. Der Nachweis der Terminierung ist sozusagen eine kreative Tätigkeit. Trotzdem werden wir in den nachfolgenden Abschnitten Verfahren kennenlernen, mit deren Hilfe zumindestens bestimmte Klassen von Algorithmen als terminierend erkannt werden können.

Fundierte Ordnungsrelationen

Das zentrale Hilfsmittel zur Untersuchung der Terminierung von Algorithmen ist der Begriff der *fundierten Ordnungsrelation*. Hierbei handelt es sich um eine irreflexive und transitive Relation $< \subset s \times s$, die die Eigenschaft hat, daß es keine unendliche Folge $x_0, x_1, x_2, ...$ von Elementen aus s gibt, für die $x_{n+1} < x_n$ für alle $n \geq 0$ gilt. D.h. es existiert bezüglich $<$ keine unendlich absteigende Kette von Objekten $...x_2 < x_1 < x_0$. Eine dazu äquivalente Forderung ist, daß jede nicht-leere Teilmenge von s bezüglich $<$ ein minimales Element besitzt.

Die übliche „Kleiner"-Relation auf natürlichen Zahlen erfüllt zum Beispiel diese Forderung, da es keine unendliche Folge von natürlichen Zahlen gibt, derart, daß jede nachfolgende Zahl in der Folge kleiner als ihr Vorgänger ist. Im Gegensatz dazu erfüllt eine „Kleiner"-Relation $<$ auf ganzen Zahlen diese Forderung nicht, denn es gilt z.B.

$$...-3 < -2 < -1 < 0.$$

Betrachten wir wieder die Konstruktionsvorschrift zur Erzeugung der Datenstruktur nat, wie wir sie in der Einleitung kennengelernt haben. Wir definieren für nat eine Kleiner-Relation $<_{nat}$ mit $a <_{nat} b$ genau dann, wenn b aus a mit Hilfe der Konstruktorfunktion succ erzeugt wurde. Es gilt dann zum Beispiel: $0 <_{nat} succ(succ(0))$ oder $x <_{nat} succ(x)$ für ein beliebiges x aus nat. Weil jedes Objekt aus nat durch eine endliche Anzahl von Anwendungen der Konstruktorfunktion succ auf 0 entstanden ist, hat auch jedes Objekt aus nat nur endlich viele Vorgänger bezüglich $<_{nat}$. Somit ist also $<_{nat}$ eine fundierte Ordnungsrelation auf nat. Eine solchermaßen konstruierte Ordnung $<_s$ läßt sich für alle induktiven Datenstrukturen s angeben, wobei $a <_s b$ dann aussagt, daß b aus a mittels Anwendung der Konstruktorfunktionen

entstanden ist. $<_s$ nennt man dabei auch die *strukturelle* Ordnung von s; sie läßt sich unmittelbar aus der Konstruktionsvorschrift für s ablesen.

Offenbar drehen wir uns jetzt aber im Kreis: Um neue Induktionsprinzipien zu gewinnen, suchen wir terminierende Algorithmen für Funktionen. Um deren Terminierung aber nachzuweisen, benötigen wir eine fundierte Ordnungsrelation, die wir letztlich wiederum aus den vorhandenen Induktionsprinzipen ableiten. Prinzipiell Neues läßt sich also damit nicht erwarten. Die Situation ändert sich, wenn es uns gelingt, in diesen "Kreislauf" zusätzliches Wissen in Form von fundierten Ordnungsrelationen einzuschleusen. Mit diesem Wissen können mehr Funktionen als terminierend erkannt werden, so daß dann wiederum mehr Induktionsschemata zur Verfügung stehen.

Eine zentrale Rolle spielt dabei die sogenannte Anzahlordnung $<_{Count}$ („Count-order" bei [BM79]), die für zwei Elemente q und r einer Datenstruktur s die Anzahl Count(q) bzw. Count(r) der reflexiven Konstruktorfunktionen von s vergleicht. Damit gilt q $<_{Count}$ r genau dann, wenn Count(q) $<_{nat}$ Count(r) gilt. Für die Datenstruktur nat stimmt die Anzahlordnung gerade mit der strukturellen Ordnung überein. Bei linearen Listen vergleicht die Anzahlordnung die Länge der Listen. Dagegen vergleicht die strukturelle Ordnung $<_{list}$ die Listen selbst. a $<_{list}$ b bedeutet dann, daß a eine Restliste von b ist. Stellt man sich die Erzeugung aller Objekte einer Datenstruktur s mit Hilfe einer Konstruktionsvorschrift als einen iterativen Prozeß vor, so besagt q $<_{Count}$ r nichts anderes, als daß q in einem früheren Iterationsschritt als r erzeugt wurde. Da jedes Objekt nach endlich vielen Iterationen gewonnen wird, ist $<_{Count}$ offensichtlich fundiert.

Man beachte dabei aber, daß Count nur dann eine Funktion ist, wenn der Definitionsbereich eine initiale Datenstruktur s ist, das heißt wenn jedes Objekt aus s auch einen eindeutigen Namen hat. Betrachtet man zum Beispiel die nicht-initiale Datenstruktur set aus Abschnitt 3, so ist der Wert der Anzahlfunktion für {0} nicht eindeutig definiert, da {0} unendlich viele verschiedene Darstellungen hat: ins(0 empty), ins(0 ins(0 empty)), ... Als Ausweg käme hier in Frage, als Wert der Anzahlfunktion stets die minimale Anzahl der benötigten Konstruktoren zu nehmen. Im obigen Beispiel wäre dann der Wert von der Anzahlfunktion für {0} eins. Ungeachtet dessen wollen wir uns im weiteren aber stets auf initiale Datenstrukturen beschränken.

Mit Hilfe dieser Anzahlordnung können wir für viele Funktionen nachweisen, daß sie terminieren. Insbesondere ist sie eine Verallgemeinerung der strukturellen Ordnung. Allerdings gibt es auch Funktionen, die nicht nach der Anzahlordnung terminieren. Betrachten wir dazu die folgende Definition einer zweistelligen Funktion diff, die analog zur Funktion – die Differenz zweier natürlichen Zahlen berechnet :

$$\forall\ x,y : nat\ \ (y - x) = 0 \Rightarrow diff(x\ y) = 0$$

$$\forall\ x,y : nat\ \ \neg\ (y - x) = 0 \Rightarrow diff(x\ y) = succ(diff(succ(x)\ y))\ .$$

diff terminiert offensichtlich nicht nach der Anzahlordnung, da weder das erste noch das zweite Argument im rekursiven Aufruf kleiner wird. Das erste Argument wächst stetig an, während das zweite gleich bleibt. Trotzdem terminiert diff für beliebige Eingaben. Der Grund liegt darin,

daß die Differenz (y - x) zwischen beiden Parametern im rekursiven Aufruf bezüglich der Anzahlordnung kleiner wird. Dies führt zu der folgenden Überlegung:

Mit der Anzahlordnung können wir mit Hilfe einer sogenannten *Maßfunktion* m eine neue Ordnungsrelation $<_m$ folgendermaßen definieren: Es gilt $x^* <_m y^*$ genau dann, wenn $m(x)^* <_{nat} m(y)^*$ gilt. Man kann sich leicht davon überzeugen, daß, unabhängig von den Eigenschaften von m, $<_m$ eine fundierte Ordnungsrelation ist. Dabei werden an die Maßfunktionen keinerlei besondere Bedingungen gestellt. Die obige Count-Funktion ist ein Beispiel für eine Maßfunktion. Die obige Funktion diff terminiert gemäß der Ordnung $<_{_}$.

Eine weitere Möglichkeit, neue fundierte Ordnungsrelationen zu erhalten, ist die Berücksichtigung *lexikographischer* Ordnungen. Ein typisches Beispiel für eine Funktion, die nach einer lexikographischen Ordnung terminiert, ist die oben bereits erwähnte Definition des ggt:

$$\forall\, x,y : nat \;\; x = 0 \Rightarrow ggt(x\ y) = y \tag{1}$$

$$\forall\, x,y : nat \;\; \neg\, x = 0 \wedge y = 0 \Rightarrow ggt(x\ y) = x \tag{2}$$

$$\forall\, x,y : nat \;\; \neg\, x = 0 \wedge \neg\, y = 0 \wedge (x - y) = 0 \Rightarrow ggt(x\ y) = ggt(x\ (y - x)) \tag{3}$$

$$\forall\, x,y : nat \;\; \neg\, x = 0 \wedge \neg\, y = 0 \wedge \neg\, (x - y) = 0 \Rightarrow ggt(x\ y) = ggt((x - y)\ y) \tag{4}$$

In dem Definitionsfall (3) bleibt das erste Argument im rekursiven Aufruf unverändert, während das zweite Argument bezüglich $<_{nat}$ kleiner wird. Im Definitionsfall (4) dreht sich die Situation gerade um: Das erste Argument wird kleiner, das zweite bleibt unverändert.

Definiert man im allgemeinen eine Relation $<^*$ durch

$$(x_1\ x_2) <^* (y_1\ y_2) :\Leftrightarrow (x_1 < x_2) \vee ((x_1 = x_2) \wedge (y_1 < y_2)) ,$$

so ist mit $<$ auch $<^*$ eine fundierte Ordnungsrelation. In unserem Beispiel terminiert ggt also nach der fundierten Ordnungsrelation $<_{nat}^*$.

Das Finden einer geeigneten Ordnungsrelation - und damit einer geeigneten Maßfunktion m - für den Nachweis der Terminierung eines Algorithmus kann in manchen Fällen äußerst schwierig sein, wie das folgende Beispiel von McCarthy [Man74] zeigt:

$$\forall\, x : nat \qquad Lt(x\ 101) \Rightarrow f(x) = f(f(x + 11)) ,$$

$$\forall\, x : nat \qquad \neg Lt(x\ \ 101) \Rightarrow f(x) = (x - 10)$$

wobei aus Platzgründen 10, 11 und 101 Abkürzungen für die entsprechenden Objekte der Datenstruktur nat seien. Die Funktion f terminiert immer, und zwar mit den Werten

$$f(x) = \begin{cases} x - 10 & \text{für } x > 100 \\ 91 & \text{sonst.} \end{cases}$$

Wie man sich leicht an folgender Beispielsequenz f(99) = f(f(110)) = f(100) = f(f(111)) = f(101) = 91 überzeugen kann, helfen die gängigen Ordnungsrelationen beim Nachweis der Terminierung überhaupt nicht.

Automatisierung des Terminierungsbeweises

Mit Hilfe der Anzahlordnung und entsprechenden Maßfunktionen stehen uns verschiedene fundierte Ordnungsrelationen zur Verfügung, die wir zum Nachweis der Terminierung einer Funktion f verwenden können.

Nehmen wir an, unser Induktionsbeweissystem möchte die Terminierung einer Definition für eine Funktion f nachweisen, wobei diese aus einer Menge bedingter Gleichungen der Form $\forall\, x_1,\ldots,x_n\ C_i \Rightarrow f(x_1 \ldots x_n) = t_i$ (für $1 \le i \le p$) bestehe. Als erstes benötigen wir eine geeignete Maßfunktion m (und daraus resultierend eine fundierte Ordnungsrelation $<_m$), gemäß derer die Terminierung gezeigt werden soll. Als nächstes wählt man eine geeignete Teilmenge der Argumentpositionen $k_1 \ldots k_j$, die sogenannten *Rekursionspositionen* aus, die bei dem Nachweis der Terminierung überhaupt betrachtet werden soll. Bezüglich dieser Rekursionspositionen muß jeder rekursive Aufruf von f bezüglich $<_m$ kleiner werden. Formal muß also für jeden rekursiven Aufruf $f(q_1 \ldots q_n)$ in einem Ergebnisterm t_i gezeigt werden, daß dessen Parameter $q_{k1} \ldots q_{kj}$ in den Rekursionspositionen bezüglich $<_m$ kleiner als die korrespondierenden formalen Parameter $x_{k1} \ldots x_{kj}$ unter der entsprechenden Bedingung C_i sind. Die zu beweisende sogenannte *Terminierungshypothese* hat somit die Form:

$$\forall\, x_1,\ldots,x_n\ \ C_i \ \Rightarrow\ m(q_{k1} \ldots q_{kj}) <_{nat} m(x_{k1} \ldots x_{kj})$$

Wählt man für die Definition von + zum Beispiel die Anzahlfunktion als Maßfunktion und das erste Argument als einzige Rekursionsposition, so muß zum Nachweis der Terminierung die folgende Formel bewiesen werden:

$$\forall\, x : nat\ \ x = succ(pred(x)) \Rightarrow pred(x) <_{nat}\ x.$$[2]

Sie besagt, daß unter der Bedingung x = succ(pred(x)) des rekursiven Definitionsfalles der aktuelle Parameter pred(x) im rekursiven Aufruf bezüglich der Anzahlordnung kleiner als der formale Parameter x ist. Da aber definitionsgemäß $y <_{nat}\ succ(y)$ für alle y - also auch für pred(x) - gilt, kann diese Formel leicht bewiesen werden.

Für die Automatisierung dieses Nachweises treten allerdings die folgenden Probleme auf. Wie kann man erkennen, welche Maßfunktion für den Nachweis der Terminierung einer Funktion geeignet ist? Wie ermittelt man die geeigneten Rekursionspositionen? Wie zeigt man die Terminierungshypothesen, das heißt, daß jeder rekursive Aufruf bezüglich dieser Ordnung kleiner wird? Es gibt dafür verschiedene Ansätze, die hier kurz skizziert werden sollen.

Im ersten Ansatz [BM79] werden die Terminierungshypothesen in Prädikatenlogik erster Stufe formuliert und es wird versucht, diese mit Hilfe des Induktionsbeweissystems aus den vorhandenen Axiomen abzuleiten. Voraussetzung hierfür ist, daß die natürlichen Zahlen nat, die Ordnungsrelation $<_{nat}$ sowie die Anzahlfunktion hinreichend axiomatisiert sind. Dementsprechend sind bei [BM79] die Datenstruktur nat sowie die Relation $<_{nat}$ bereits vordefiniert. Die Axiome für die Anzahlfunktion werden für jeden neuen Datentyp automatisch erzeugt. Als

---

[2] Man beachte hierbei, daß Count auf der Datenstruktur nat die Identität darstellt und daher in dieser Terminierungshypothese nicht explizit auftritt.

Maßfunktion m kommt theoretisch - neben der Anzahlfunktion - jede bereits im System definierte Funktion in Frage. Praktisch erfolgt die Auswahl eines geeigneten m bei [BM79] an Hand des Vorwissens des Systems. Der Benutzer kann Lemmata der Form

$$\forall\, y_1,\ldots,y_n \;\; D \;\Rightarrow\; m(r_1 \ldots r_j) <_{nat} m(y_1 \ldots y_j),$$

wobei r_i beliebige Terme über den Variablen $y_1,\ldots,y_n$ sind, explizit als „Induktionslemma“ kennzeichnen und vom Induktionssystem beweisen lassen. Nur wenn eine Funktion m in einem solchen Induktionslemma auftritt, kann sie als Maßfunktion für den Nachweis der Terminierung herangezogen werden. Ein geeignetes Induktionslemma - und damit auch eine geeignete Maßfunktion m - wird für eine Funktion f folgendermaßen ausgewählt. Man sucht zum einen nach einer Maßfunktion m, für die ein oder mehrere Induktionslemmata in der Datenbasis vorhanden sind, und zum anderen nach einer Teilmenge $k_1 \ldots k_j$ der Argumentpositionen von f. Für jeden rekursiven Aufruf $f(q_1,\ldots,q_n)$ muß es dann ein Induktionslemma geben, das garantiert, daß die aktuellen Parameter in dem rekursiven Aufruf bezüglich den Rekursionspositionen kleiner sind als die formalen Parameter.

Kehren wir zu unserem Beispiel der Definition einer Funktion diff zurück.

$$\forall\, x,y : nat \;\; (y - x) = 0 \Rightarrow diff(x\ y) = 0$$

$$\forall\, x,y : nat \;\; \neg\, (y - x) = 0 \Rightarrow diff(x\ y) = succ(diff(succ(x)\ y))$$

Gibt es in der Datenbasis ein Induktionslemma der Form

$$\forall\, u,v : nat \;\; \neg\, (u - v) = 0 \;\Rightarrow (u - succ(v)) \;<_{nat}\; (u - v)\ ,$$

dann kann damit die Terminierung der Funktion diff mit den Rekursionspositionen (2, 1) bewiesen werden. Denn für u = y und v = x gilt unter der Bedingung ¬ (y - x), daß y - succ(x) bezüglich $<_{nat}$ kleiner als y - x ist.

Im zweiten Ansatz [Wal90] wird die Terminierung der Funktionen nicht im Induktionssystem, sondern in einem speziellen Kalkül nachgewiesen. Jede Definition einer n-stelligen Funktion f wird in einem Spezialverfahren daraufhin untersucht, welche Argumente i die Eigenschaft besitzen, daß stets $f(x_1 \ldots x_n) \leq_{Count} x_i$ gilt. Funktionen, die diese Eigenschaft für ein Argument i besitzen, nennt man im i-ten *Argument beschränkt*. Findet das Induktionssystem heraus, daß eine Funktion f im i-ten Argument beschränkt ist, wird implizit ein sogenanntes *Differenzprädikat* $\Delta_{f,i}$ konstruktiv definiert, das angibt, unter welchen Bedingungen sogar $f(x_1 \ldots x_n) <_{Count} x_i$ gilt. D.h. es wird implizit das Induktionslemma

$$\forall\, x_1,\ldots,x_n \;\; \Delta_{f,i}\,(x_1 \ldots x_n) \;\Rightarrow f(x_1 \ldots x_n) <_{Count} x_i\ .$$

erzeugt.

Die Funktion pred ist zum Beispiel auf dem ersten Argument beschränkt. Das zugehörige Prädikat $\Delta_{pred,1}$ ist als $\Delta_{pred,1}\,(x) \Leftrightarrow \neg\, x = 0$ definiert.

Angenommen, man möchte nun nachweisen, daß $f(g(x)) <_{Count} x$ unter einer Bedingung C gilt. Weiß man, daß f und g auf dem ersten Argument beschränkt sind, so gilt nach Voraussetzung bereits $f(g(x)) \leq_{Count} g(x)$ und $g(x) \leq_{Count} x$. Um zu zeigen, daß f(g(x)) echt kleiner als x ist, muß entweder $f(g(x)) <_{Count} g(x)$ oder $g(x) <_{Count} x$ gelten. Mit Hilfe der

Differenzprädikate für f und g kann dafür eine hinreichende (und notwendige) Bedingung formuliert werden:

$$\forall\, x\;\; C \Rightarrow \Delta_{g,1}\,(x) \vee \Delta_{f,1}\,(g(x))$$

Der Nachweis der Formel sichert dann die Eigenschaft $f(g(x)) <_{Count} x$ unter der Bedingung C.

Mit Hilfe dieses Verfahrens läßt sich sehr einfach und schnell überprüfen, ob ein aktueller Parameter t_i in einem rekursiven Aufruf von f bezüglich der Anzahlordnung kleiner oder gleich seinem korrespondierenden formalen Parameter x_i ist. Aus dem Vergleich aller aktuellen Parameter in den rekursiven Aufrufen mit den entsprechenden formalen Parametern erhält man eine Auswahl geeigneter Rekursionsparameter, für die dann explizit untersucht wird, inwieweit unter den gegebenen Bedingungen die aktuellen Parameter in den rekursiven Aufrufen echt kleiner als die formalen sind.

4 Nachweis von Funktionseigenschaften (Lemmata)

Der vorangegangene Abschnitt beschäftigte sich mit dem Aufbau einer Datenbasis. Mit Hilfe eines Definitionsprinzips können Datenstrukturen eingefügt werden. Basierend auf diesen Datenstrukturen können mit Hilfe von bedingten Gleichungen Algorithmen spezifiziert werden, die bestimmte Funktionen berechnen. Dieser Abschnitt beschäftigt sich nun mit dem Beweis von Eigenschaften dieser Funktionen, die in Form sogenannter *Lemmata* formuliert werden. Einen breiten Raum nimmt dabei die Einbettung der Induktion in einen automatischen Beweiser ein.

4.1 Verwendung der Induktionsschemata

In der Einleitung zu diesem Kapitel haben wir festgestellt, daß die Induktionsaxiome, die wir zur eindeutigen Charakterisierung von bestimmten Datentypen benötigen, nicht in der Prädikatenlogik erster Stufe formulierbar sind. Als Ersatz führten wir das Induktionsschema ein, das bei Bedarf mit einer Formel instantiiert als zusätzliches Axiom in die Datenbasis aufgenommen wird. Für die Datenstruktur nat lautet dieses Induktionsschema (mit Selektoren) zum Beispiel:

$$\{\ \forall\, x{:}nat\ \ [\ x = succ(pred(x)) \Rightarrow P(pred(x))\ ] \Rightarrow P(x)\ \} \Rightarrow \forall\, x{:}nat\ \ P(x)\ ,$$

wobei P für eine beliebige Formel mit einer freien Variablen x steht.

Soll zum Beispiel die Assoziativität der Funktion + mit Hilfe dieses Induktionsschemas gezeigt werden, so setzt man $P(x) := \forall\, y,z{:}nat\ x + (y + z) = (x + y) + z$ und erhält damit die folgende Implikation als Instanz des obigen Induktionsschemas:

$$\{\forall\, x{:}nat\ \ [\ x = succ(pred(x)) \Rightarrow\ \forall\, y,z{:}nat\ \ pred(x) + (y + z) = (pred(x) + y) + z\ ]$$
$$\Rightarrow \forall\, y,z{:}nat\ \ x + (y + z) = (x + y) + z\ \}$$
$$\Rightarrow \forall\, x{:}nat\ \forall\, y,z{:}nat\ \ x + (y + z) = (x + y) + z$$

Um also die Assoziativität von + zu zeigen, reicht es somit, die Prämisse der instantiierten

Induktionsschemas zu zeigen. Wir spalten diesen Nachweis in zwei Fälle auf:

Für den Fall x = 0, den sogenannten Induktionsanfang, erhält man das Unterproblem

$$\forall\ y,z{:}nat\ \ 0 + (y + z) = (0 + y) + z\ ,$$

das leicht mit Hilfe der ersten Definitionsformel von + gezeigt werden kann.

Im Fall x = succ(pred(x)), dem sogenannten Induktionsschritt, steht uns zum Beweis des Induktionsschlusses

$$\forall\ y,z{:}nat\ \ x + (y + z) = (x + y) + z$$

eine Induktionshypothese

$$\forall\ y,z{:}nat\ \ pred(x) + (y + z) = (pred(x) + y) + z$$

zur Verfügung. Für den Beweis müssen wir die zwei Terme x + (y + z) und (x + y) + z mit Hilfe der vorhanden Axiome angleichen. Verwendet man die Definition von +, so lassen sich beide Terme unter Verwendung der Induktionshypothese identifizieren. Somit hat man die Prämisse des instantiierten Induktionsschemas gezeigt und folgert damit die Conclusio, d.h. die Assoziativität von +.

Ein etwas komplizierteres Beispiel ist der Nachweis der Kommutativität des ggt. Wenden wir hier das obige Induktionsschema an, so unterteilen wir den Nachweis der Prämisse wieder in mehrere Fälle.

Für den Fall x = 0 müssen wir die Formel

$$\forall\ y{:}nat\ ggt(0, y) = ggt(y, 0)$$

und im Fall x = succ(pred(x)) die Formel

$$\forall\ x{:}nat\ [\ x = succ(pred(x)) \Rightarrow ggt(pred(x), y) = ggt(y, pred(x))\ ]$$
$$\Rightarrow ggt(x, y) = ggt(y, x)$$

zeigen. Während der Fall x = 0 mit Hilfe der Definition des ggt einfach zu beweisen ist, kann der Fall x = succ(pred(x)) aus den vorhandenen Axiomen nicht abgeleitet werden (zumindest nicht ohne weitere Induktion). Die Ursache liegt darin, daß wir den Wert ggt(x, y) nicht in Termini des Werts ggt(pred(x), y) ausdrücken und damit die Induktionshypothese nicht anwenden können. Die Rekursion des ggt ist mit dem obigen Induktionsschema nicht verträglich. Dies ist nicht weiter verwunderlich, da wir die Möglichkeit, Funktionen zu definieren, erweitert haben. Funktionsdefinitionen, deren Rekursion nicht dem in der Einleitung vorgestellten Konstruktionsprinzip genügt, sind explizit zugelassen, um aus diesen Definitionen neue Induktionsschemata zu gewinnen.

4.1.1 Erzeugung von Induktionsschemata

Im folgenden werden wir uns also damit beschäftigen, wie man aus der rekursiven Definition einer Funktion ein Induktionsschema extrahiert. Zentrales Hilfsmittel ist auch diesmal wieder der Begriff der fundierten Ordnungsrelation.

Grundlage für unsere Vorgehensweise ist die in der Algebra bekannte *noethersche Induktion* [Coh65] :

Ist $\prec \subset s \times s$ eine fundierte Ordnungsrelation, so gilt:

(*) $\quad [\ \forall\ x{:}s\ \ (\ \forall\ y{:}s\ \ y \prec x \Rightarrow P(y)\ )\ \Rightarrow\ P(x)\ ]\ \Rightarrow \forall\ x{:}s\ P(x).$

Das heißt, um zu zeigen, daß eine Formel P(x) für alle x aus s gilt, reicht es aus zu zeigen, daß sie für ein beliebiges x aus s gilt, wenn vorausgesetzt wird, daß P(y) für alle bezüglich $\prec$ kleineren y aus s gilt (*Induktionsschritt)*. Da es für jedes minimale Element c aus s bezüglich $\prec$ kein kleineres Element gibt und somit $\forall\ y{:}s\ \ y \prec c \Rightarrow P(y)$ trivialerweise gilt, ist in diesen Fällen der Beweis von P(c) wirklich zu erbringen (*Induktionsanfang*).

Mit Hilfe dieses Induktionsprinzips ist es also möglich, anstatt der Conclusio der Formel (*) $\forall\ x{:}s\ P(x)$ ihre Hypothese $\forall\ x{:}s\ \ (\forall\ y{:}s\ \ \ y \prec x \Rightarrow P(y))\ \Rightarrow\ P(x)$ zu beweisen. Dabei ist die Wahl einer geeigneten fundierten Ordnungsrelation $\prec$ für das Finden eines Beweises per Induktion in gleicher Weise ausschlaggebend wie beim Nachweis der Terminierung eines Algorithmus. Dies liegt an der schon mehrmals erwähnten engen Beziehung zwischen Induktion und Rekursion. Betrachtet man einen typischen Beweisgang für eine Induktionsformel der Form:

$\forall\ x{:}s\ \ (\forall\ y{:}s\ \ \ y \prec x \Rightarrow P(y))\ \Rightarrow\ P(x),$

so wird mit Hilfe der Definitionen für die in P vorkommenden Funktionen und Prädikate die Eigenschaft, daß P für x gilt, auf die Eigenschaft zurückgeführt, daß P für ein y gilt, das bezüglich $\prec$ kleiner als x ist. Es liegt daher nahe, für die Auswahl einer geeigneten fundierten Ordnungsrelation $\prec$ zur Generierung einer Induktionsformel auf die Ordnungsrelationen zurückzugreifen, die bei Analyse der Terminierung der verschiedenen in der Formel vorkommenden Funktionen verwendet wurden. In den Fällen, in denen die in der Formel vorkommenden Funktionen nach derselben Ordnungsrelation $\prec$ terminieren, wird man daher diese Ordnungsrelation $\prec$ auch zur Formulierung der Induktionsformel verwenden.

Für den Beweis der Kommutativität des ggt greifen wir daher auf die lexikographische Ordnung $\prec_{nat}^{*}$ zurück, die wir für den Nachweis der Terminierung des ggt verwendet haben, und formulieren damit eine Instanz des Noetherschen Induktionsprinzips:

$$\begin{aligned}&\forall\ x,y{:}nat\ \ \{\ (\ \forall\ u,v{:}nat\ \ (u, v) \prec_{nat}^{*} (x, y) \Rightarrow ggt(u, v) = ggt(v, u)\ )\\ &\qquad\qquad \Rightarrow\ \ ggt(x, y) = ggt(y, x)\ \}\\ &\quad \Rightarrow \forall\ x,y{:}nat\ ggt(x, y) = ggt(y, x)\end{aligned}$$

Wir scheinen am Ziel angekommen zu sein: Mit Hilfe der fundierten Ordnungsrelation, nach der der ggt terminiert, erhielten wir ein neues Induktionsschema, das wir zum Beweis von Sätzen über den ggt verwenden können. Die Kommutativität des ggt läßt sich zum Beispiel mit Hilfe des obigen Induktionsschemas ohne weitere Induktion zeigen. Leider erweist sich aber dieses Schema aus mehreren Gründen in der Praxis als ungeeignet.

Ein Grund dafür ist, daß die fundierte Ordnungsrelation $\prec_{nat}^{*}$ explizit in der erzeugten Formel auftritt. Um die Induktionshypothese anwenden zu können, müssen wir die Eigenschaften von $\prec_{nat}^{*}$ kennen. Dies bedeutet, daß $\prec_{nat}^{*}$ hinreichend genau in unserer Datenbasis axiomatisiert

sein muß. Ob aber $<_{nat}^{*}$ in der Datenbasis definiert ist, hängt von dem verwendeten Verfahren zum Nachweis der Terminierung von Funktionen ab. Zum Beispiel benötigt das Verfahren von [Wal90] keine explizite Axiomatisierung der fundierten Ordnungsrelationen.

Aber angenommen, wir hätten wie in [BM79] die fundierte Ordnungsrelation explizit in unserer Datenbasis definiert und wollten die obige Aussage beweisen. Dann müßten wir den Term ggt(x, y) mit Hilfe der Datenbasis und der Induktionshypothese

$$\forall\ u,v{:}nat\ \ (u, v) <_{nat}^{*} (x, y) \Rightarrow ggt(u, v) = ggt(v, u)$$

zu dem Term ggt(y x) umformen. Analog zu der Definition des ggt unterscheiden wir vier verschiedene Fälle:

(1) $x = 0$,

(2) $\neg\ x = 0 \wedge\ y = 0$,

(3) $\neg\ x = 0 \wedge\ \neg\ y = 0 \wedge\ (x - y) = 0$ und

(4) $\neg\ x = 0 \wedge\ \neg\ y = 0 \wedge\ \neg\ (x - y) = 0$.

Betrachten wir zum Beispiel den dritten Fall: Nach Anwendung der Definition des ggt erhalten wir aus dem Term ggt(x, y) den Term ggt(x, (y - x)). Der nächste Schritt wäre jetzt die Anwendung der Induktionshypothese auf diesen Term. Dazu müßten wir aber nachweisen, daß unter den gegebenen Voraussetzungen $(x, (y - x)) <_{nat}^{*} (x, y)$ gilt. Implizit haben wir dies bereits bei dem Nachweis der Terminierung des ggt gezeigt; die fundierte Ordnungsrelation $<_{nat}^{*}$ wurde ja gerade für diesen Nachweis verwendet. Explizit müssen wir dies in unserem Kalkül aber nochmals (mit Hilfe der Induktionslemmata) aus den Axiomen ableiten, um die Induktionshypothese verwenden zu können.

Diese Gründe führen dazu, daß man für die Automatisierung der vollständigen Induktion das Noethersche Induktionsprinzip abwandelt. Man möchte, daß die dem Induktionsprinzip zugrunde liegende Ordnungsrelation nicht mehr explizit in dem Induktionsschema erscheint.

In dem obigen Beispiel möchte man im dritten Fall nicht die „abstrakte“ Induktionshypothese

$$\forall\ u,v{:}nat\ \ (u, v) <_{nat}^{*} (x, y) \Rightarrow ggt(u, v) = ggt(v, u)$$

sondern die benötigten „konkreten“ Instanzen verwenden. Im obigen Beispiel ist dies also insbesondere die Teilformel

$$ggt(x, (y - x)) = ggt((y - x), x)\ .$$

Betrachtet man sich also das noethersche Induktionsprinzip

$$[\ \forall\ x{:}s\ \ (\ \forall\ y{:}s\ \ y < x \Rightarrow P(y)\ )\ \Rightarrow\ P(x)\ ]\ \Rightarrow \forall\ x{:}s\ P(x)\ ,$$

dann soll zum Beweis von P(x) in der Prämisse nicht mehr verlangt werden, daß P(y) bereits für *alle* y gilt, die bezüglich der fundierten Ordnungsrelation < kleiner sind. Statt dessen setzten wir nur noch voraus, daß P(y) für *bestimmte* y gilt, die bezüglich < kleiner sind. Die Auswahl dieser „geeigneten“ y erfolgt mit Hilfe der Definitionsformeln, zu deren Terminierungsnachweis die Ordnung < verwendet wurde.

Kehren wir zum Beispiel des ggt zurück. In den Definitionsformeln des ggt gibt es im dritten

und vierten Fall jeweils einen rekursiven Aufruf. Da wir bezüglich der fundierten Ordnungsrelation $<_{nat}^*$ nachgewiesen haben, daß der ggt terminiert, sind die aktuellen Parameter in den rekursiven Aufrufen unter den jeweiligen Bedingungen bezüglich $<_{nat}^*$ kleiner als die formalen Parameter. Es gilt also:

$$\forall\, x,y : nat \;\; \neg\, x = 0 \wedge \neg\, y = 0 \wedge (x - y) = 0 \Rightarrow \; (x, (y - x)) <_{nat}^* \; (x, y) \qquad \text{und}$$

$$\forall\, x,y : nat \;\; \neg\, x = 0 \wedge \neg\, y = 0 \wedge \neg\, (x - y) = 0 \Rightarrow ((x - y), y) <_{nat}^* \; (x, y)\,.$$

Für die beiden anderen Fälle der Definition des ggt kennen wir keine Vorgänger bezüglich $<_{nat}^*$.

Diese Informationen über $<_{nat}^*$ werden wir verwenden, um das obige Induktionsschema für den Beweis der Kommutativität des ggt abzuändern. Die Prämisse des neuen Induktionsschemas lautet somit:

$$\begin{aligned}\forall\, x,y{:}nat \quad & \{\, (\neg\, x = 0 \wedge \neg\, y = 0 \wedge (x - y) = 0 \Rightarrow ggt(x, (y - x)) = ggt((y - x), x)\,) \\ & \quad \wedge\, (\neg\, x = 0 \wedge \neg\, y = 0 \wedge \neg\, (x - y) = 0 \Rightarrow ggt((x - y), y) = ggt(y, (x - y))\,)\,\} \\ & \Rightarrow\; ggt(x, y) = ggt(y, x)\,\}\end{aligned}$$

Wie sieht nun unser konkretisiertes Induktionsschema im allgemeinen für eine Definition einer Funktion f aus ? Wenn wir die Terminierung von f gemäß dem vorherigen Abschnitt bewiesen haben, dann gibt es zu dieser Funktion eine fundierte Ordnungsrelation $<$ bezüglich derer die rekursiven Aufrufe von f kleiner werden. Hat f also die formalen Parameter $x_1 \ldots x_n$ und die Rekursionspositionen $k_1 \ldots k_j$, so wissen wir für jeden rekursiven Aufruf $f(q_1 \ldots q_n)$ in einem Definitionsfall mit einer Bedingung C, daß

$$\forall\, x_1,\ldots,x_n \;\; C \;\Rightarrow\; (q_{k1}, \ldots, q_{kj}) < (x_{k1}, \ldots, x_{kj})$$

gilt. Sammeln wir diese Information für alle rekursiven Aufrufe auf, so erhalten wir eine Menge von Paaren $(C, (q_{k1} \ldots q_{kj}))$, für die die obige Formel gilt. Diese Menge beschreibt einzelne Vorgänger von $x_{k1} \ldots x_{kj}$ bezüglich der Relation $<$. Man nennt sie daher eine *Vorgängermenge* von $<$ bezüglich des Variablentupels $x_{k1} \ldots x_{kj}$.

Haben wir eine solche Vorgängermenge $\{(C_1, q_1{*}) \ldots (C_n, q_n{*})\}$ bezüglich eines Variablentupels x^* gegeben, so können wir somit das Noethersche Induktionsprinzip

$$[\, \forall\, x^*{:}s^* \;\; (\, \forall\, y^*{:}s^* \;\; y^* < x^* \Rightarrow P(y^*)\,) \;\Rightarrow\; P(x^*)\,] \;\Rightarrow \forall\, x^*{:}s^*\; P(x^*)$$

zu dem konkreten Induktionsschema

$$[\, \forall\, x^*{:}s^* \;\; [\, \{\, C_1 \Rightarrow P(q_1{}^*) \wedge \;\ldots\; \wedge\, C_n \Rightarrow P(q_n{}^*)\,\} \;\Rightarrow\; P(x^*)\,] \;\Rightarrow \forall\, x^*{:}s^*\; P(x^*)\,,$$

umformen. In diesem Schema tritt jetzt die fundierte Ordnungsrelation $<$ nicht mehr explizit auf. Es zeigt sich, daß wir zur Erzeugung dieses Induktionsschemas die Ordnungsrelation $<$ nicht zu kennen brauchen. Es reicht aus, daß wir wissen, daß die verwendete Liste eine Vorgängermenge von *irgendeiner* fundierten Ordnungsrelation $<$ ist.

Betrachten wir als abschließendes Beispiel die bereits definierte Funktion diff. Aus ihrer Definition erhält man die einelementige Vorgängermenge $\{\, (\neg\, (y - x) = 0,\;\; (succ(x), y))\,\}$ bezüglich des Variablentupels (x, y). Diese aus der Definition von diff gewonnene Vorgängermenge erzeugt das folgende Induktionsschema:

$$\forall\ x,y : nat\ \ [\ \{\neg\ (y - x) = 0\ \ \Rightarrow P(succ(x), y)\}\ \ \Rightarrow\ \ P(x, y)\ ]\ \Rightarrow \forall\ x,y : nat\ P(x, y)\ .$$

4.1.2 Auswahl eines Induktionsschemas

Wir sind jetzt in der Lage, für jede als terminierend nachgewiesene Funktionsdefinition ein eigenes Induktionsschema zu erzeugen. Somit stehen uns für den induktiven Beweis eines Satzes in der Regel mehrere verschiedene Induktionsschemata zur Verfügung. Konnten wir bisher schon wählen, über welche Variablen wir induzieren möchten, so können wir jetzt zusätzlich auch noch das Induktionsschema bestimmen. Wie aber erkennt ein automatischer Beweiser, welche Variablen und welches Schema für eine Formel am geeignetsten sind?

Die Antwort liefert uns - wie so oft - der enge Zusammenhang zwischen der Rekursion und der Induktion. Die Idee ist, daß mit Hilfe der rekursiven Definitionsfälle der in der Formel auftretenden Funktionen der Induktionsschluß auf die Induktionshypothese zurückgeführt werden soll. Da dazu Rekursionsordnung und Induktionsordnung möglichst übereinstimmen sollten, bestimmt sich die Induktionsordnung nach den Rekursionsordnungen der in der Formel auftretenden Funktionen.

Nehmen wir an, in der Formel tritt ein Term der Form $f(t_1 \ldots t_n)$ auf, wobei f rekursiv definiert sei. Stehen in diesem Term an den Rekursionspositionen $k_1 \ldots k_j$ von f paarweise verschiedene Variablen, so können wir die zum Nachweis der Terminierung von f verwandte fundierte Ordnungsrelation $\prec$ bzw. die korrespondierende Vorgängermenge von $\prec$ bezüglich $(t_{k1}, \ldots, t_{kj})$ zur geeigneten Instantiierung eines Induktionsschemas verwenden. Da die Vorgängermenge sich gerade aus den rekursiven Aufrufen von f in den Definitionsfällen bestimmt, können wir den Term $f(t_1 \ldots t_n)$ unter Verwendung der Funktionsdefinition von f zu einem Term umformen, der in den Rekursionspositionen schon mit dem korrespondierenden Term in der Induktionshypothese übereinstimmt. Somit ist uns mit der Anwendung der Funktionsdefinition bereits ein wichtiger Schritt bei der Rückführung des Induktionsschlusses auf die Induktionshypothese gelungen.

Im allgemeinen können in einer Formel mehrere Terme auftreten, die die obige Bedingung erfüllen. Es gilt dann, aus den verschiedenen vorgeschlagenen Induktionsschemata ein geeignetes auszuwählen oder mit Hilfe bestimmter Verfahren aus den verschiedenen Induktionsschemata ein gemeinsames zu erzeugen. Beispiele sind hier die Subsumption oder das Verschmelzen verschiedener Induktionsschemata wie sie in [BM79] ausführlich beschrieben sind. Wir werden an dieser Stelle nicht näher auf diese Techniken eingehen und nur eine kurze Bemerkung dazu geben.

Die Erzeugung eines Induktionsschemas aus mehreren Induktionsschemata erfordert, daß die Vorgängermenge einer Relation $\prec$ möglichst einfach denotiert wird. Für ein Paar (C, q) aus der Vorgängermenge bedeutet dies zum Beispiel, daß die Bedingung C soweit wie möglich zu einer Bedingung C' abzuschwächen ist, ohne aber die zugrundeliegende fundierte Ordnungsrelation $\prec$ in Frage zu stellen.

4.1.3 Induktion über beliebige Terme

Betrachten wir das oben eingeführte Induktionsprinzip, so fällt auf, daß wir bisher - um ein Induktionsschema zu instantiieren - eine Vorgängermenge bezüglich eines Variablentupels x^* nur auf eine Formel der Form $\forall\ x^*\ P(x^*)$ anwenden können. Dies bedeutet, daß das Variablentupel mit einem Teil der allquantifizierten Variablen in der Formel übereinstimmen muß. Soweit beide Variablenlisten sich nur in den Variablennamen unterscheiden, können wir diese Forderung durch Umbenennung der Variablen in der Formel erfüllen. Schwieriger ist die Einschränkung, daß wir nur über paarweise verschiedene Variablen induzieren können

Was passiert aber, wenn die Elemente des Variablentupels nicht als Argumente rekursiver Funktionen auftreten? Diese Frage soll uns im Weiteren beschäftigen.

Ein Beispiel dafür ist die folgende Aussage:

$$\forall\ x,y : nat\ \ \neg\ ggt(succ(x), succ(y)) = 0$$

Induziert man zum Beispiel gemäß dem vom ggt vorgeschlagenen Induktionsschema auf x und y, so schlägt der Beweisversuch fehl. Der Grund liegt darin, daß man sich der Rekursion des ggt bedienen möchte, um den Induktionsschluß auf die Induktionshypothese zurückzuführen. Die Rekursion setzt aber in dieser Formel an einer anderen Stelle als die Induktion an.

Während die Rekursion die Argumente des ggt im Term ggt(succ(x) succ(y)) verändert und in einem Fall den Term

$$ggt((succ(x) - succ(y)), succ(y))$$

liefert, wird die Induktionshypothese auf Variablen, d.h. hier auf bestimmten Untertermen der Argumente des ggt, formuliert. Der entsprechende Term in der Induktionshypothese lautet in dem obigen Fall

$$ggt(succ(x-y), succ(y))$$

Beide Terme stimmen somit nicht überein, womit die Induktionshypothese für den Beweis nicht verwendet werden kann. Eine ähnliche Beobachtung macht man bei Verwendung des strukturellen Induktionsschemas.

Die Lösung wäre ein Induktionsschema, das analog zur Rekursion des ggt ebenfalls direkt an den Argumenten des ggt angreift. Das heißt aber, daß man nicht über den Wert von x und y sondern über den Wert von succ(x) und succ(y) induzieren muß. Um dies zu erreichen, bedienen wir uns desselben Tricks wie beim Nachweis der Terminierung von Funktionen. Ist $<$ eine fundierte Ordnungsrelation, so läßt sich mit Hilfe einer beliebigen Funktion m eine neue fundierte Ordnungsrelation $<_m$ mit $x^* <_m y^* :\Leftrightarrow m(x)^* < m(y)^*$ definieren. Formulieren wir das noethersche Induktionsprinzip für eine solche fundierte Ordnungsrelation $<_m$, und verwenden dabei die ursprüngliche Relation $<$ und die Maßfunktion m, so erhalten wir das folgende Schema:

$$[\ \forall\ x^*{:}s^*\ \ (\ \forall\ y^*{:}s^*\ m(y)^* < m(x)^* \Rightarrow P(y^*)\ )\ \Rightarrow\ P(x^*)\ ]\ \Rightarrow \forall\ x^*{:}s^*\ P(x^*)$$

Wie in dem vorangegangenen Abschnitt möchte man aber, daß die fundierte Ordnungsrelation $<$

in dem Induktionsschema nicht explizit auftritt. Mit Hilfe einer Vorgängermenge von $<$ werden sie aus dem Induktionsschema eliminiert.

Sei $\{(C_1, q_1^*) \ldots (C_n, q_n^*)\}$ eine Vorgängermenge von $<$ bezüglich eines Variablentupels u^*, so ist q_i für jedes $1 \leq i \leq n$ unter der Bedingung C_i ein Vorgänger von u^* bezüglich $<$. Instantiiert man u^* durch $m(x)^*$, so ist $q_i^* [u^* \leftarrow m(x)^*]$ unter der Bedingung $C_i [u^* \leftarrow m(x)^*]$ ein Vorgänger von $m(x)^*$ bezüglich $<$. Das heißt, unter der Voraussetzung daß $m(y)^*$ gerade den Wert $q_i^* [u^* \leftarrow m(x)^*]$ annimmt, ist y^* ein Vorgänger von x^* bezüglich der fundierten Ordnungsrelation $<_m$.

Wir können also die Induktionshypothese $\forall\, y^*{:}s^*\ m(y)^* < m(x)^* \Rightarrow P(y^*)$ in dem Noetherschen Induktionsprinzip mit Hilfe der Vorgängermenge folgendermaßen abändern:

$$\forall\, y^*{:}s^*\ \ C_1 [u^* \leftarrow m(x)^*] \wedge m(y^*) = q_1^* [u^* \leftarrow m(x)^*] \ \Rightarrow P(y^*)$$
$$\wedge\ \ldots\ \wedge$$
$$\forall\, y^*{:}s^*\ \ C_n [u^* \leftarrow m(x)^*] \wedge m(y^*) = q_n^* [u^* \leftarrow m(x)^*] \ \Rightarrow P(y^*).$$

Nach dieser etwas mühevollen Herleitung sind wir am Ziel. Mit Hilfe des obigen Induktionsprinzips können wir jetzt über den Wert eines „beliebigen“ Termes induzieren.

Kommen wir also wieder auf unser Beispiel des ggt zurück. Damit Induktion und Rekursion an der gleichen Stelle in der Formel ansetzen, möchten wir über den Wert der Argumente succ(x) und succ(y) induzieren. $m(x)^*$ ist dann das Tupel (succ(x), succ(y)) und man erhält die folgende Formel als Prämisse des instantiierten Induktionsschemas.

$$\begin{aligned}
&\forall\, x,y{:}nat \\
&\quad \{\forall\, u,v{:}nat\ \ (\neg\, succ(x) = 0 \wedge \neg\, succ(y) = 0 \wedge succ(x) - succ(y) = 0 \\
&\qquad\qquad \wedge\, succ(u) = succ(x) \wedge\ succ(v) = succ(y) - succ(x)\ ) \\
&\qquad\qquad \Rightarrow \neg\, ggt(succ(u), succ(v)) = 0 \\
&\quad \wedge\ \forall\, u,v{:}nat\ \ (\neg\, succ(x) = 0 \wedge \neg\, succ(y) = 0 \wedge \neg\, succ(x) - succ(y) = 0 \\
&\qquad\qquad \wedge\, succ(u) = succ(x) - succ(y) \wedge\ succ(v) = succ(y)) \\
&\qquad\qquad \Rightarrow \neg\, ggt(succ(u), succ(v)) = 0\ \} \\
&\Rightarrow \neg\, ggt(succ(x), succ(y)) = 0
\end{aligned}$$

Diese nur für automatische Beweiser „lesbare“ Formel läßt sich unter Verwendung der Axiome für die Datenstruktur nat und der Funktion „–“ zu der folgenden - auch für menschliche Leser -verständlichen Formel vereinfachen:

$$\begin{aligned}
&\forall\, x,y{:}nat \\
&\quad \{\ \forall\, v{:}nat\ \ (succ(x) - succ(y) = 0 \wedge\ \ succ(v) = succ(y) - succ(x)\ ) \\
&\qquad\qquad \Rightarrow \neg\, ggt(succ(x), succ(v)) = 0 \\
&\quad \wedge\ \forall\, u{:}nat\ \ (succ(u) = succ(x) - succ(y)\ \Rightarrow \neg\, ggt(succ(u), succ(y)) = 0\ \} \\
&\Rightarrow \neg\, ggt(succ(x), succ(y)) = 0
\end{aligned}$$

Wir erhalten je eine Induktionshypothese für den Fall, daß succ(x) größer als succ(y) ist, sowie für den Fall, daß succ(y) größer als succ(x) ist. Im Fall, daß beide gleich sind, steht uns keine

Induktionshypothese zur Verfügung. Wir benötigen in diesem Fall auch keine, da nach Anwendung der Definition des ggt ein Argument gleich 0 wird, so daß mit Hilfe eines nicht-rekursiven Definitionsfalls das Theorem direkt bewiesen werden kann.

Als ein weiteres Beispiel wollten wir zeigen, daß die oben definierte Funktion diff eine Inverse zu der Addition + ist. Als Formel denotiert hat diese Aussage die folgende Form:

$$\forall\ x,y : \text{nat}\ \ \text{diff}(x, (x + y)) = y$$

Um das von diff vorgeschlagene Induktionsschema verwenden zu können, wollen wir über die Werte von x und (x + y) induzieren. Man erhält daher die folgende Prämisse für das instantiierte Induktionsschema

$$\begin{aligned}&\forall\ x,y{:}\text{nat}\\ &\quad \{\ \forall\ u,v{:}\text{nat}\ \ (\ \neg\ (x + y) - x = 0\ \wedge\ \ u = \text{succ}(x)\ \wedge\ (u + v) = (x + y)\ )\\ &\qquad\qquad \Rightarrow \text{diff}(u, (u + v)) = v\ \}\\ &\quad \Rightarrow\ \ \text{diff}(x, (x + y)) = y\ .\end{aligned}$$

Angenommen, man weiß bereits das Theorem (x + y) - x = y, so kann die obige Formel mit Hilfe der bestehender Definitionen zu der folgenden Formel simplifiziert werden:

$$\begin{aligned}&\forall\ x,y{:}\text{nat}\\ &\quad \{\ \forall\ v{:}\text{nat}\ \ (\ \neg\ y = 0\ \wedge\ \ (\text{succ}(x) + v) = (x + y)\ )\\ &\qquad\qquad \Rightarrow \text{diff}(\text{succ}(x), (\text{succ}(x) + v)) = v\ \}\\ &\quad \Rightarrow\ \ \text{diff}(x, (x + y)) = y\ .\end{aligned}$$

Der Beweis dieser Formel ist dann eine einfache Konsequenz aus der Definition von diff und einem benötigten Lemma der Form $\forall\ x,y{:}\text{nat}\ \ (x + \text{succ}(y)) = (\text{succ}(x) + y)$. Auch hier würde eine Induktion mit dem von diff vorgeschlagenen Induktionsschema auf x und y zu einem Fehlschlagen des Beweises führen.

4.2 Spezielle Strategien für Induktionsbeweise

4.2.1 Symbolische Auswertung

Eine der ersten Strategien für das Führen von Induktionsbeweisen war die sogenannte *symbolische Auswertung* von Formeln. Darunter versteht man die Verwendung der Funktionsdefinitionen zur Auswertung von Termen. Terme denotieren in der Regel Mengen von Objekten eines Datentyps, da in ihnen Variable vorkommen dürfen. Da die Verwendung der Definitionen nicht auf variablenfreie Terme begrenzt ist, spricht man von einer symbolischen Auswertung [BM79], [Hut86].

Mit Hilfe der Definition von + kann man zum Beispiel Terme der Form 0 + a zu a bzw. Terme der Form succ(a) + b zu succ(a + b) auswerten. Hinreichendes Kriterium dafür ist der Nachweis der instantiierten Bedingungen 0=0 bzw. succ(a)=succ(pred(succ(a))) in den Definitionsformeln von +.

In vorigen Abschnitt hatten wir gesehen, daß sich die Auswahl eines geeigneten Induktionsschemas an der Rekursion der auftretenden Funktionen orientiert. Stimmen Rekursionsordnung und Induktionsordnung überein, so erhält man durch Anwendung der Funktionsdefinition insbesondere einen Vorgänger bezüglich der Induktionsordnung.

Das nächste Beispiel soll dies verdeutlichen:

Für den Nachweis der Assoziativität von + erhielten wir im letzten Abschnitt die folgende Formel als Prämisse des instantiierten Induktionsschemas:

$$\forall\, x{:}nat \quad \{x = succ(pred(x)) \Rightarrow pred(x) + (y + z) = (pred(x) + y) + z\}$$
$$\Rightarrow x + (y + z) = (x + y) + z$$

Betrachten wir den Fall x = succ(pred(x)). Zum Beweis dieser Formel müssen wir die beiden Terme

$$x + (y + z) \qquad \text{und} \qquad (x + y) + z$$

unter Verwendung der Induktionshypothese einander angleichen. Wir können den Rekursionsfall von + für die symbolische Auswertung in beiden Termen verwenden und vereinfachen sie zu:

$$succ(pred(x) + (y + z)) \qquad \text{und} \qquad succ(pred(x) + y) + z\ .$$

Bei dem rechten Term können wir nochmals die Definition von + anwenden und erhalten

$$succ(pred(x) + (y + z)) \qquad \text{und} \qquad succ((pred(x) + y) + z),$$

so daß wir mit Hilfe der Induktionshypothese beide Terme identifizieren können.

In dem obigen Beispiel können wir die Formel nur mit Hilfe von Induktion und den Definitionen von nat und + beweisen. Schwieriger wird es, wenn zum Beweis eines Satzes noch zusätzliche Lemmata benötigt werden, mit deren Hilfe der Induktionsschluß geschickt umgeformt werden muß. Die Verwendung dieser Lemmata kann nämlich durch die symbolische Auswertung nicht gesteuert werden.

Betrachten wir dazu die Distributivität der Addition über der Multiplikation:

$$\forall\, x,y,z : nat\ x * (y + z) = (x * y) + (x * z)\ .$$

Wir definieren zunächst die Multiplikation * durch die folgenden Formeln:

$$\forall\, x,y : nat\ \ x = 0 \Rightarrow x * y = 0$$

$$\forall\, x,y : nat\ \ x = succ(pred(x)) \ \Rightarrow x * y = y + (pred(x) * y)$$

Für den Beweis setzen wir voraus, daß uns bereits die Kommutativität und Assoziativität von + als Hilfssätze zur Verfügung stehen.

Verwenden wir für den Beweis der Distributivität das aus der Rekursion von * gewonnene Induktionsschema, so erhalten wir folgende zu beweisende Formel:

$$\forall\, x{:}nat \quad [\ x = succ(pred(x)) \Rightarrow$$
$$\forall\, y, z : nat\ \ pred(x) * (y + z) = (pred(x) * y) + (pred(x) * z)\ ]$$

$$\Rightarrow \forall\, y, z : nat\ x * (y + z) = (x * y) + (x * z)$$

Betrachten wir wiederum den Fall x = succ(pred(x)), so müssen wir mit Hilfe der Induktionshypothese die beiden Terme

$$x * (y + z) \qquad \text{und} \qquad (x * y) + (x * z)$$

angleichen. Wir können diesmal den Rekursionsfall von * für die symbolische Auswertung verwenden und werten beide Terme folgendermaßen aus:

$$(y + z) + (pred(x) * (y + z)) \quad \text{und} \quad (y + (pred(x) * y)) + (z + (pred(x) * z))$$

Auf den linken Term ist die Induktionshypothese unmittelbar anwendbar, um beide Terme aber anzugleichen, müßte der rechte Term mit der Assoziativität und der Kommutativität weiter umgeformt werden. Die symbolische Auswertung hilft uns an dieser Stelle nicht mehr weiter.

Diese Beschränkung der symbolischen Auswertung führte in den letzten Jahren zu der Entwicklung des sogenannten *Ripplings*, das im nächsten Abschnitt erläutert werden soll.

4.2.2 Rippling

Der Grundgedanke des Ripplings [Bun88], [Bun90], [Hut90] besteht darin, gezielt auf die Verwendung der Induktionshypothese hinzuarbeiten, indem man sukzessive die syntaktischen Differenzen zwischen Induktionsschluß und Induktionshypothese verringert („difference reduction"). Wir stehen dabei vor der Schwierigkeit, messen zu müssen, „wie stark" sich zwei Terme unterscheiden.

Betrachten wir dazu den Induktionsbeweis für die Distributivität der Addition über der Multiplikation. Um die Induktionshypothese verwenden zu können, müssen wir die linke bzw. die rechte Seite des Induktionsschlusses so umformen, daß die jeweils korrespondierende Seite der Induktionshypothese darin als Unterterm auftritt. Mit Hilfe der symbolischen Auswertung konnten wir die linke Seite so umformen, daß wir die Induktionshypothese verwenden könnten. Auf der rechten Seite dagegen scheiterte diese Umformung, da wir neben der symbolischen Auswertung auch verschiedene Lemmata verwenden müßten, um die Rückführung auf die Induktionshypothese durchzuführen.

Um die Induktionshypothese auf der rechten Seite des Induktionsschlusses anwenden zu können, muß die (instantiierte) rechte Seite der Induktionshypothese als Unterterm in der rechten Seite des Induktionsschlusses auftreten. Zum Vergleich der beiden Terme schraffieren wir die Gemeinsamkeiten der rechten Seiten von Induktionshypothese und -schluß:

$$((pred(x) * y) + (pred(x) * z)) \quad \text{und} \quad ((x * y) + (x * z))$$

Die nicht-schraffierten Bereiche kennzeichnen die Termteile, die kein entsprechendes Pendant im anderen Term besitzen. Da wir aber die Induktionshypothese verwenden wollen, muß die rechte Seite der Induktionshypothese vollständig als Unterterm im Induktionsschluß auftreten. Wir müssen also den Induktionsschluß durch geeignete Umformungen soweit „aufblähen", daß auch die nicht-schraffierten Bereiche ein Pendant im Induktionsschluß haben, ohne dabei die

schraffierten Bereiche im Induktionsschluß zu verändern. Die schraffierten Bereiche bleiben invariant bezüglich der geplanten Umformung und bilden ein Muster, wie das Ergebnis jedes Umformungsschrittes auszusehen hat.

Um dies sicherzustellen, klassifizieren wir die verschiedenen (bedingten) Gleichungen in unserer Datenbasis danach, welche syntaktischen Gemeinsamkeiten linke und rechte Seiten haben. Termteile einer Seite, die ein Pendant auf der anderen Seite haben, werden schraffiert. Ein Beispiel hierfür ist die folgende Schraffur für die rekursive Definitionsgleichung von *:

$$x = succ(pred(x)) \Rightarrow (x * y) = (y + (pred(x) * y))$$

Hier hat jedes Termteil auf der linken Seite ein entsprechendes Pendant auf der rechten. Das heißt, die linke Seite ist vollständig schraffiert, während die rechte Seite nicht-schraffierte Bereiche aufzeigt. Wendet man diese Gleichung also von links nach rechts an, so wird der umzuformende Term durch die instantiierten weißen Bereiche der rechten Seite aufgebläht. Diese Information nutzt man aus, um gezielt Gleichungen auszuwählen, die bei ihrer Anwendung einen Term um passende Termfragmente erweitern. In unserem Beispiel eignet sich der obige Definitionsfall von *, da ihre Anwendung gerade die gewünschten weißen Bereiche erzeugt; wendet man ihn zweimal auf eine Seite des Induktionsschlusses an, so erhält man den Term

$$((y + (pred(x) * y)) + (z + (pred(x) * z)))$$

Die nicht-schraffierten Bereiche der Induktionshypothese haben jetzt ebenfalls Pendants im Induktionsschluß. Markieren wir diese Bereiche zusätzlich als Gemeinsamkeiten von Induktionshypothese und Induktionsschluß, so erhalten wir die folgende Schraffur für die rechte Seite des Induktionsschlusses:

$$((y + (pred(x) * y)) + (z + (pred(x) * z)))$$,

während die rechte Seite der Induktionshypothese keine nicht-schraffierten Bereiche mehr hat. Die rechte Seite der Induktionshypothese tritt vollständig - in einzelne schraffierte Bereiche aufgespalten - im Induktionsschluß auf. Für die Anwendung der Induktionshypothese benötigen wir aber einen zusammenhängenden schraffierten Bereich; die rechte Seite der Induktionshypothese muß als Term im Induktionsschluß auftreten.

Bildlich gesprochen müssen wir die schraffierten Bereiche durch geeignete Umformungen zusammen bzw. die nicht-schraffierten Bereiche nach außen „schieben“. Für diese Zwecke bedienen wir uns wieder der obigen Klassifikation der Gleichungen nach den Gemeinsamkeiten ihrer linken und rechten Seiten.

Betrachten wir dazu zwei mögliche Schraffuren für das Assoziativgesetz der Addition:

$$((x + y) + z) = (x + (y + z))$$

$$(x + (y + z)) = ((x + y) + z)$$

Während auf den beiden linken Seiten die Schraffur unterbrochen ist, ist auf den rechten Seiten die Schraffur durchgängig. Wendet man diese Gleichungen so an, daß schraffierte Bereiche des

umzuformenden Terms stets nur mit schraffierten Bereichen der linken Seite zusammenfallen, so werden - analog zu der angewandten Gleichung - die nicht-schraffierten Bereiche nach außen verlagert und die schraffierten Bereiche einander angenähert. Eine entsprechende Anwendung der Assoziativität löst also unser Problem, die schraffierten Bereiche zusammen zu schieben. Die Umformung können wir also anhand der Schraffuren folgendermaßen vornehmen:

((y + (pred(x) * y)) + (z + (pred(x) * z)))

(y + ((pred(x) * y) + (z + (pred(x) * z))))

(y + ((pred(x) * y) + ((pred(x) * z) + z)))

(y + (((pred(x) * y) + (pred(x) * z)) + z))

Im zweiten Schritt haben wir das Kommutativgesetz angewendet, daß folgendermaßen schraffiert sei:

(x + y) = (y + x)

Die Anwendung dieser Gleichung verschiebt keine nicht-schraffierten Bereiche nach außen, sondern verlagert sie nur von einem Argument von + in das andere. Im unserem Beispiel benötigten wir sie, um ein zweites Mal die Assoziativität anzuwenden. Danach haben wir unser Ziel erreicht, es gibt nur noch einen zusammenhängenden schraffierten Bereich, die Induktionshypothese kann also angewendet werden.

4.3 Generalisierung

Für Induktionsbeweise ist es oft notwendig, statt der ursprünglichen Formel eine allgemeinere, stärkere Formel für die Instantiierung des Induktionsschemas zu verwenden. Der Grund liegt in der Struktur von Induktionsbeweisen. Verschärfen wir eine zu beweisende Aussage, dann müssen wir zwar einerseits einen stärkeren Induktionsschluß beweisen, doch können wir andererseits auch eine stärkere Induktionshypothese voraussetzen. Die Gefahr besteht allerdings, daß die nach der Generalisierung zu beweisende Aussage falsch wird und damit nicht mehr aus der bestehenden Datenbasis ableitbar ist.

An Hand des nächsten Beispiel wollen wir einige Techniken der Generalisierung verdeutlichen:.

Wir wählen als zu beweisende Aussage eine Instanz der Assoziativität von +:

$\forall$ x:nat x + (x + x) = (x + x) + x (*)

Beweist man diese Formel mit Induktion, so erhält man als Prämisse des instantiierten Induktionsschemas die folgende Formel:

$\forall$ x:nat [x = succ(pred(x)) $\Rightarrow$

pred(x) + (pred(x) + pred(x)) = (pred(x) + pred(x)) + pred(x)]

$\Rightarrow$ x + (x + x) = (x + x) + x .

Im Fall x = succ(pred(x)) können wir die beiden Terme

$$x + (x + x) \qquad \text{und} \quad (x + x) + x$$

symbolisch auswerten und erhalten so die beiden zu identifizierenden Terme:

$$succ(pred(x) + succ(pred(x) + x)) \qquad \text{und} \quad succ((pred(x) + x) + x)$$

Um die Induktionshypothese verwenden zu können, müßten wir beide Terme mit Hilfe geeigneter Lemmata über + weiter umformen. Kennen wir keine weiteren Eigenschaften von +, scheitert unser Induktionsbeweis. Die Induktionshypothese pred(x) + (pred(x) + pred(x)) = (pred(x) + pred(x)) + pred(x) erweist sich als zu schwach, um im Induktionsschluß verwendet zu werden, da sie für jedes Argument von + den Term pred(x) explizit vorschreibt. Die Definition von + erlaubt es uns aber nur für die Rekursionsposition von +, x auf pred(x) zurückzuführen.

Eine mögliche Lösung dieses Problems besteht darin, daß man bestimmte Unterterme, die in der Formel mehrfach auftreten, durch eine neugenerierte Variable ersetzt, um so die zu beweisende Formel weitgehend zu verallgemeinern [BM79]. Wendet man diese Technik in der obigen Formel (*) an und ersetzt man den Term x + x, der in der Formel zweimal auftritt, durch eine neu eingeführte Variable y, dann erhält man als Verallgemeinerung des obigen Satzes die Kommutativität der Addition:

$$\forall\ x,y : nat \;\; x + y = y + x\ .$$

Bei der Ersetzung beliebiger Unterterme, die mehrmals in der zu beweisenden Formel auftreten, kann es leicht passieren, daß die so erhaltene, verallgemeinerte Formel nicht mehr gültig und damit nicht mehr aus der bestehenden Datenbasis ableitbar ist. Ist zum Beispiel sort eine Funktion, die lineare Listen nach einer Ordnung sortiert, so führt die Generalisierung der Formel

$$\forall\ x{:}list\ sort(sort(x)) = sort(x)$$

zu einem falschen Satz

$$\forall\ y{:}list\ sort(y) = y,$$

da die Information, daß y bereits sortiert ist, verloren geht. In [BM79] werden daher diese Art der Generalisierung vom Benutzer explizit durch Eingabe ausgezeichneter Generalisierungslemmata angesteuert.

Eine weitere Lösung des obigen Problems ist die Ersetzung von bestimmten Auftritten einzelner Variablen durch neue Variable. Man “entkoppelt” dabei die Variablen in den Rekursionspositionen von den Variablen in den Nicht-Rekursionspositionen, indem man erstere durch neue Variablen ersetzt, und damit die Formel *generalisiert*.

Im obigen Beispiel führt dies zu dem folgendem Satz:

$$\forall\ y, x : nat \;\; y + (x + x) = (y + x) + x$$

Verwendet man das strukturelle Induktionsschema und induziert über y, dann erhält man die folgende Formel

$$\forall\, y : nat \quad [\, y = succ(pred(y)) \Rightarrow \forall\, x' : nat \;\; pred(y) + (x' + x') \;=\; (pred(y) + x') + x' \,]$$
$$\Rightarrow \forall\, x : nat \;\; y + (x + x)) = (y + x) + x$$

In dem Fall y = succ(pred(y)) können wir jetzt die Terme

$$y + (x + x) \qquad \text{und} \quad (y + x) + x$$

mit Hilfe der Definition von + zu den folgenden Termen umformen:

$$succ(pred(y) + (x + x)) \qquad \text{und} \quad succ((pred(y) + x) + x)$$

Die Induktionshypothese ist auf beide Terme anwendbar .

Im allgemeinen erweist es sich als sinnvoll, die Variablen in den Rekursionspositionen von denen in den Nicht-Rekursionspositionen zu trennen. Man benennt dazu die Auftritte einer Variablen in den Rekursionspositionen um und entkoppelt sie somit von den Auftritten in den Nicht-Rekursionspositionen. Ansonsten sind die bei einer Induktion erzeugten Induktionshypothesen oft zu schwach, um im Induktionsschritt verwendet zu werden, da mit Hilfe der Definitionsaxiome in der Regel nur die Auftritte der Variablen in den Rekursionspositionen auf ihre Vorgänger zurückgeführt werden können.

4.4 Existenzbeweise

Einen Schwerpunkt im automatischen Beweisen mit vollständiger Induktion bildet der Beweis von existenzquantifizierten Formeln der Form $\forall\, x^*\ \exists\, y\ P(x^*\ y)$. Eine solche Formel sagt gerade aus, daß es zu jedem x^* (mindestens) ein y gibt, so daß $P(x^*\ y)$ gilt. Führt man diesen Beweis konstruktiv, so erhält man als Ergebnis einen Term t^*, der die Eigenschadt hat, daß $\forall\, x^*\ P(x^*\ t^*)$ gilt. In t* treten höchstens die Variablene x^* selbst wieder auf, so daß wir t* als eine Funktion f über x^* betrachten können. Interpretiert man die obige Formel $\forall\, x^*\ \exists\, y\ P(x^*\ y)$ als eine formale Spezifikation für die Funktion f, so liefert der Beweis dieser Aussage gerade eine mögliche *konstruktive* Definition der gesuchten Funktion. Das heißt, mit Hilfe von Induktionsbeweisen lassen sich rekursive Funktionsdefinitionen (bzw. Programme) *synthetisieren*. Man nennt daher dieses Spezialgebiet auch *deduktive Programmsynthese* [MW80] [Biu92] [BH84].

Es würde den Rahmen dieses Buches bei weitem sprengen, wenn man die verschiedenen Ansätze und Ergebnisse in diesem Bereich vorstellen wollte. Daher werden wir uns nur kurz an einem Beispiel die prinzipielle Vorgehensweise klarmachen.

Zu diesem Zweck definieren wir uns eine Funktion half, die eine natürliche Zahl durch zwei teilt und das Ergebnis abrundet:

$$\forall\, x : nat \;\; x = 0 \Rightarrow half(x) = 0$$

$$\forall\, x : nat \;\; x = succ(0) \Rightarrow half(x) = 0$$

$$\forall\, x : nat \;\; x = succ(succ(pred(pred(x)))) \Rightarrow half(x) = succ(half(pred(pred(x))))$$

Nehmen wir nun an, wir wollten die Formel

$$\forall\, x : nat\ \exists\, y : nat \;\; half(y) = x$$

beweisen. Betrachtet man sich die Formel genauer, so spezifiziert sie gerade die Existenz der Inversen zu half. Aus dem Beweis für diesen Satz wollen wir eine Definition für diese Funktion, die wir im folgenden „double“ nennen, erhalten.

Mit Hilfe des Induktionsschemas

$$\{ \forall\, x{:}nat\ [\ x = succ(pred(x)) \Rightarrow P(pred(x))\ ] \Rightarrow P(x)\ \} \Rightarrow \forall\, x{:}nat\ P(x)\ .$$

müssen wir zum Beweis des Satzes die folgende Prämisse des instantiierten Induktionsschemas beweisen:

$$\forall\, x{:}nat\ \ (\ x = succ(pred(x)) \Rightarrow \exists\, y' : nat\ \ half(y') = pred(x)\ )$$
$$\Rightarrow\ \exists\, y : nat\ \ half(y) = x$$

Wir unterscheiden zum Beweis des Satzes die beiden Fälle x = 0 und x = succ(pred(x)).

Im ersten Fall müssen wir zeigen, daß $\exists\, y : nat\ \ half(y) = 0$ gilt. Instantiieren wir y durch 0 (oder alternativ durch succ(0)), so können wir mit Hilfe des nicht-rekursiven Teils der Definition von half die Formel zeigen. Die benötigte Instantiierung erhalten wir dabei automatisch durch die für die Anwendung der jeweiligen Definitionsfälle von half notwendigen Unifikationen. Für den Fall x = 0 wissen wir durch die Instantiierung der Variable y, welchen Wert unsere Funktion double annehmen darf: 0 oder succ(0). Somit können wir einen ersten Definitionsfall für die Funktion double formulieren:

$$\forall\, x{:}nat\ \ x = 0 \Rightarrow double(x) = 0 \quad \text{(oder } double(x) = succ(0)\ \text{)}$$

Im zweiten Fall x = succ(pred(x)) steht uns eine Induktionshypothese zur Verfügung. Wir müssen den umformulierten Induktionsschluß

$$\exists\, y : nat\ \ half(y) = succ(pred(x))$$

auf die Induktionshypothese zurückführen. Analog zu dem Abschnitt über Rippling versucht man die Unterschiede zwischen beiden Termen sukzessive zu verringern. Da der rechte Term succ als führendes Funktionssymbol besitzt, möchte man auch half(y) in terminis von succ ausdrücken. Mit Hilfe des rekursiven Falls von half können wir dies erreichen, wenn wir y durch einen Term succ(succ(z)) ersetzen. Wir erhalten nach Anwendung der Definition von half die zu beweisende Formel

$$\exists\, z : nat\ \ succ(half(z)) = succ(pred(x))\ ,$$

die eine einfache Folgerung aus der Induktionshypothese ist (wobei wir z und y’ identifizieren).

Betrachten wir nun, wie sich der Wert der existenzquantifizierten Variable y im Verlauf des Beweises verändert hat: Im ersten Schritt wurde y zu succ(succ(z)) instantiiert. Später haben wir z mit y’ identifiziert, so daß letztlich y mit succ(succ(y’)) identifiziert wurde. Für unsere zu synthetisierende Funktion double bedeutet dies folgendes: angenommen man kennt den Wert von double für pred(x), so bestimmt sich der Wert von double für x aus succ(succ(double(pred(x)))). Nehmen wir noch die Bedingung der Fallunterscheidung hinzu, so können wir folgenden Definitionsfall für double angeben:

$$\forall\, x{:}nat\ \ x = succ(pred(x)) \Rightarrow double(x) = succ(succ(double(pred(x))))\ .$$

Fügen wir die Ergebnisse aus beiden Fällen zusammen, so erhalten wir je nach Beweisverlauf zwei alternative Definitionen für double, die beide den obigen Satz erfüllen:

$\forall$ x:nat x = 0 $\Rightarrow$ double(x) = 0 (oder double(x) = succ(0))

$\forall$ x:nat x = succ(pred(x)) $\Rightarrow$ double(x) = succ(succ(double(pred(x))))

Ausgehend von einer Spezifikation synthetisierten wir durch induktives Beweisen eine konstruktive Definition der gewünschten Funktion double.

5 Schlußbemerkungen

Das Gebiet des automatischen Beweisens mit Induktion erfuhr in den letzten Jahren ein wachsendes Interesse. Ein Hauptgrund dafür liegt in der zunehmenden Bedeutung der Programmverifikation beziehungsweise der Programmsynthese für die Entwicklung „sicherer", das heißt fehlerfreier Software. Sowohl Programmverifikation als auch Programmsynthese lassen sich wegen ihrer inhärenten Komplexität nicht ohne maschinelle Unterstützung durchführen, so daß hier automatische Beweiser mit Induktion zum Einsatz kommen. Zum Beispiel wurde mit Hilfe des Systems von [BM79] der komplette Aufbau eines Mikroprozessors 8501 [Hun85] verifiziert.

Literatur

Aub79 R. Aubin: *Mechanizing Structural Induction. Part I und II*. Theoretical Computer Science 9, North Holland Publishing Company, 1979

BH84 Bibel, W. Hörnig, K.M. *LOPS - A System Based on a Strategical Approach to Programm Synthesis*. in: Automatic Programm Construction Techniques. A. Biermann, G. Guiho u. Y. Kodratoff (Hrsg.), Macmillan Publishing Comp., 1984

BHHW86 S. Biundo, B. Hummel, D. Hutter, C. Walther: *The Karlsruhe Induction Theorem Proving System*. Proc. of the 8th CADE, Springer LNCS Vol. 230, 1986

Biu92 S. Biundo: *Automatische Synthese rekursiver Programme als Beweisverfahren*. Informatik Fachberichte, Springer-Verlag, 1992

Bun88 A. Bundy et al.: *The Use of Explicit Plans to Guide Inductive Proofs.* Proc. of the 9th International Conference on Automated Deduction. Springer-Verlag LNCS 310, 1988

Bun90 A. Bundy et al.: *Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs.* Proc. of the 10th International Conference on Automated Deduction. Springer-Verlag LNAI 449, 1990

BM79 R.S. Boyer, J.S. Moore: *A Computational Logic*. Academic Press, 1979

Bur69 R.M. Burstall: *Proving Properties of Programs by Structural Induction*. The Computer Journal 12, 1969

Coh65 P.M. Cohn: *Universal Algebra*. D.Reidel Publishing Company, Holland, 1965

Hun85 W.A. Hunt, Jr: *FM8501: A Verified Microprocessor*. Technical Report 47 Institute of Computer Science, University of Texas at Austin, 1985

Hut86 D. Hutter: *Using Resolution and Paramodulation for Inductive Proofs*. Proc. of the 10th GWAI, Springer FB 124, 1986

Hut90 D. Hutter: *Guiding Induction Proofs*. Proc. of the 10th International Conference on Automated Deduction. Springer-Verlag LNAI 449, 1990

Man74 Z. Manna: *Mathematical Theory of Computation*. Mc Graw-Hill Book Company, 1974

MW80 Z. Manna, R. Waldinger: *A Deductive Approach to Program Synthesis*. ACM Transactions on Programming Languages and Systems Vol. 2, No.1, 1980

Wal90 C. Walther: *Automatisierung von Terminierungsbeweisen.*. Vieweg Verlag, 1991