Beweis von Programmeigenschaften – Induktion

Übungsblatt 5

Eric Kunze — 12. Mai 2021

Vollständige Induktion auf $\mathbb{N}$

Für wollen eine Eigenschaft $P(n)$ für alle $n \in \mathbb{N}$ zeigen. Da die natürlichen Zahlen eine schöne Struktur besitzen, kennen wir das Prinzip der vollständigen Induktion für Beweise dieser Art. Dazu zeigen wir einen **Induktionsanfang** für das kleinste n, für das die Eigenschaft P gelten soll. Meist ist das $P(0)$ oder $P(1)$. Der entscheidende Prozess passiert im **Induktionsschritt**. Dafür nehmen wir an, die Eigenschaft gelte für *ein* $n \in \mathbb{N}$. Nun zeigen wir die Eigenschaft für $n+1$. Formal zeigt man also die Implikation $P(n) \implies P(n+1)$.

Induktion auf Listen

Das Prinzip

Dieses Prinzip wollen wir nun auf Listen anwenden, denn Listen haben eine ähnliche Struktur wie die natürlichen Zahlen. Wir wollen wieder eine Eigenschaft $P($`xs`$)$ für alle Listen `xs :: [a]` zeigen, wobei `a` ein beliebiger Typ ist. Im **Induktionsanfang** zeigen wir die Eigenschaft für die kleinste Form einer Liste: die leere Liste. Zu beweisen ist also $P($`[ ]`$)$. Im **Induktionsschritt** nehmen wir an, die Eigenschaft gelte für eine Liste `xs :: [a]` (Induktionsvoraussetzung) und zeigen davon ausgehend, dass für alle `x :: a` die Eigenschaft auch für `(x:xs)` gilt.

In der Praxis

Im Induktionsanfang macht es sich oftmals ganz gut, wenn man die linke Seite und die rechte Seite der Gleichung separat betrachtet. Im Induktionsschritt gehen wir von der linken Seite der Gleichung aus (mit der erweiterten Liste `(x:xs)`) und werten so lange Funktionen gemäß des gegebenen Codes aus bis wir nicht mehr weiter kommen. Dann ist in der Regel Zeit für die Induktionsvoraussetzung und schließlich wenden wir (evtl. nach ein paar Umformungen mit Kommutativ-, Assoziativ- oder Distributivgesetz für `+` und `*`) die Definitionen des Codes "rückwärts" an, d.h. wir ersetzen die rechte Seite einer "Code-Gleichung" durch die linke. Am Ende erhalten wir die rechte Seite der zu zeigenden Gleichung mit `(x:xs)`. Damit ist der Induktionsschritt gezeigt.

In der Lehrveranstaltung wird sehr großer Wert auf die korrekte Notation gelegt. Daher müssen wir auch sehr achtsam formulieren, insbesondere wenn wir die Induktionsvoraussetzung angeben. Daher sei folgendes Muster (analog zu den offiziellen Musterlösungen) empfohlen:

- **Induktionsanfang:** Sei `xs = []`. Dann gilt <Gleichung>
- **Induktionsvoraussetzung:** Sei `xs :: [a]`, sodass <Gleichung> gilt.
- **Induktionsschritt:** Sei `x :: a`. Dann gilt <Gleichungen>

Ein Beispiel: Aufgabe 1

Zu zeigen ist die Gleichung

$$\texttt{sum (foo xs) = 2 * sum xs - length xs} \quad \text{für alle } \texttt{xs :: Int}$$

mittels Induktion über Listen.

Induktionsanfang Sei `xs == []`.

$$\begin{aligned}
\text{linke Seite:} \quad & \texttt{sum (foo [])} \overset{(2)}{=} \texttt{sum []} \overset{(6)}{=} \texttt{0} \\
\text{rechte Seite:} \quad & \texttt{2 * sum [] - length []} \overset{(10)}{=} \texttt{2 * sum [] - 0} \overset{(6)}{=} \texttt{2 * 0 - 0 = 0}
\end{aligned}$$

Induktionsvoraussetzung Sei `xs :: [Int]`, sodass gilt

$$\texttt{sum (foo xs) = 2 * sum xs - length xs}$$

Induktionsschritt Sei `x :: Int`. Es gilt

$$\begin{aligned}
\texttt{sum (foo (x:xs))} &\overset{(3)}{=} \texttt{sum (x : x : (-1) : foo xs)} \\
&\overset{3\cdot(7)}{=} \texttt{x + x + (-1) + sum (foo xs)} \\
&\overset{(IV)}{=} \texttt{x + x + (-1) + 2 * sum xs - length xs} \\
&\overset{(Komm.)}{=} \texttt{2 * x + 2 * sum xs - 1 - length xs)} \\
&\overset{(Dist.)}{=} \texttt{2 * (x + sum xs) - (1 + length xs)} \\
&\overset{(7)}{=} \texttt{2 * sum (x:xs) - (1 + length xs)} \\
&\overset{(11)}{=} \texttt{2 * sum (x:xs) - length (x:xs)}
\end{aligned}$$

Strukturelle Induktion auf algebraischen Datentypen

Neben Listen spielten aber auch algebraische Datentypen eine wichtige Rolle in der funktionalen Programmierung. Können wir das Induktionsprinzip auch auf diese Strukturen übertragen? Ihr

könnt die Antwort schon vermuten: ja!

Ich versuche im Folgenden einen Zwischenweg zu finden, der zwar die Theorie hinter struktureller Induktion auf algebraischen Datentypen vermittelt, aber auf korrekte Formalitäten verzichtet. Für die formal korrekte Darstellung sei auf das Skript verwiesen.

Zu zeigen sei wieder eine Eigenschaft P für einen algebraischen Datentyp. Erinnern wir uns, wie diese konzipiert sind: wir haben Konstruktoren, die wiederum Typen als Argumente haben. Dabei unterscheiden wir nun zwei Fälle:

- ein Konstruktor ist rekursiv[1], d.h. er enthält den definierenden Datentyp wieder, z.B. `Branch x Tree Tree`, wenn `Branch` Teil der Definition des Datentyps `Tree` ist.
- ein Konstruktor ist ein Basiskonstruktor[2], wenn er nicht rekursiv ist, z.B. `Leaf x` oder `Nil` im Datentyp `Tree`.

Wir haben im Wesentlichen zwei Baumstrukturen kennengelernt.

(1) `data BinTree a = Branch a (BinTree a) (BinTree a) | Leaf a`
Hier ist also `Branch` ein rekursiver Konstruktor, weil er zwei Argumente `BinTree a` hat, die er selbst aber definieren soll. Dagegen hat der Konstruktor `Leaf` nur ein Argumenttyp `a`, nicht aber `BinTree a`. Dementsprechend ist `Leaf` mit unserer Sprechweise ein Basiskonstruktor.

(2) `data BinTree a = Branch a (BinTree a) (BinTree a) | Nil`
Auch hier ist `Branch` ein rekursiver Konstruktor, `Nil` dagegen ein Basiskonstruktor.

Damit können wir nun das deutlich allgemeinere Prinzip der strukturellen Induktion besprechen.

Der **Induktionsanfang** besteht wieder aus den kleinsten Einheiten des Datentyps, also den Basiskonstruktoren. Im Falle von Bäumen wäre also die Eigenschaft für Blätter zu zeigen. In der **Induktionsvoraussetzung** nehmen wir nun an, dass die Eigenschaft für die rekursiven Argumente aller rekursiven Konstruktoren gelte. Im Falle von Bäumen gelte die Eigenschaft also für die Teilbäume eines Knotens. Im **Induktionsschritt** zeigen wir die Eigenschaft für den rekursiven Konstruktor. Das heißt also wir setzen die Teile aus der Induktionsvoraussetzung mithilfe des Konstruktors zu einem neuen Objekt zusammen und zeigen dafür die Eigenschaft. Für Bäume entspricht dieser Fall dem eines Knotens.

Quantifizierung

Unter Quantifizierung versteht man die richtige und vor allem vollständige Typangabe mit den richtigen Quantoren (Existenz- bzw. Allquantor). Beispielsweise ist `xs :: [Int]` die Quantifizierung von `xs`. Im Laufe der Induktionen geben wir diese Typen immer an. Zumindest für die Variablen, über die induziert wird, ist das meistens auch sehr logisch und geläufig.

[1] die Bezeichnung führe ich so ein, um einfacher darüber reden zu können; es ist aber keine Bezeichnung im Sinne der Vorlesung

[2] auch hier wieder meine eigene Bezeichnung, nicht im Sinne der Vorlesung

Schwieriger wird es bei der sogenannten Quantifizierung freier Variablen (ich sage oft kurz freie Quantifizierung). Es können in manchen Induktionen Variablen auftreten, die zwar vorkommen (und damit quantifiziert werden müssen), aber für die Induktion nicht von Belang sind. Daher werden deren Typen oft vergessen! Typisches Beispiel für solche freie Variablen sind Knotenbeschriftungen in Bäumen, die in der Regel nicht wichtig für die Induktion sind (da es eher um die Baumstruktur geht), aber trotzdem richtig quantifiziert werden müssen. Die freien Variablen sind stets mit einem Allquantor versehen, d.h. mit Formulierungen der Form "für alle ..." verbunden.

Konkrete Beispiele seht ihr in der Übung.

Ein Beispiel: Zusatzaufgabe 2 [3]

Wir haben folgende Code-Vorgabe:

```
1 data BinTree a = Branch a (BinTreee a) (BinTree a) | Leaf a
2
3 p :: BinTree a -> [a]
4 p (Leaf x) = [x]
5 p (Branch x s t) = [x] ++ (p s ++ p t)
6
7 d :: BinTree a -> BinTree a -> BinTree a
8 d (Leaf x) u = Branch x u u
9 d (Branch x s t) u = Branch x s (d t u)
```

Wir sollen folgende Aussage zeigen:

`p (d t u) = p t ++ (p u ++ p u)` für jeden Typ `a` und alle Bäume `t, u :: BinTree a`

Dabei darf die folgende Eigenschaft benutzt werden (Assoziativität von `++`):

$$\forall\ \texttt{xs, ys, zs :: [a]} \qquad \texttt{xs ++ (ys ++ zs) = (xs ++ ys) ++ zs} \tag{B}$$

Gemäß Hinweis reicht der Beweis mittels struktureller Induktion über die Struktur des Baumes `t`. Damit wird `u` immer freie Variable sein!

Induktionsanfang Sei `x :: a` und `u :: BinTree a`. Dann gilt

$$\begin{aligned}\texttt{p (d (Leaf x) u)} &\overset{(9)}{=} \texttt{p (Branch x u u)}\\ &\overset{(6)}{=} \texttt{[x] ++ p u ++ p u}\\ &\overset{(5)}{=} \texttt{p (Leaf x) ++ p u ++ p u}\end{aligned}$$

Induktionsvoraussetzung Sei `t :: BinTree a`, sodass für alle `u :: BinTree a` gilt:

$$\texttt{p (d t u) = p t ++ p u ++ p u} \tag{IV}$$

[3] ehemalige Klausuraufgabe

Induktionsschritt Sei `x :: a` und `u,s :: BinTree a`. Es gilt:

$$
\begin{aligned}
\texttt{p (d (Branch x s t) u)} &\overset{(10)}{=} \texttt{p (Branch x s (d t u))} \\
&\overset{(6)}{=} \texttt{[x] ++ (p s ++ p (d t u))} \\
&\overset{(IV)}{=} \texttt{[x] ++ p s ++ p t ++ p u ++ p u} \\
&\overset{(B)}{=} \texttt{[x] ++ p s ++ p t ++ p u ++ p u} \\
&\overset{(6)}{=} \texttt{p (Branch x s t) ++ p u ++ p u}
\end{aligned}
$$

Übersicht

Induktion auf Listen

Induktionsanfang Sei < freie Quantifizierung > und `xs = []`.

- linke Seite: ...
- rechte Seite: ...

Induktionsvoraussetzung Sei `xs :: [a]`, sodass < freie Quantifizierung > gilt:

- < Gleichung für `xs` >

Induktionsschritt Sei `x :: a` und < freie Quantifizierung >. Es gilt: < Gleichungsfolge >

Baumstruktur mit leeren Bäumen

```
data BinTree a = Branch a (BinTree a) (BinTree a) | Nil
```

Induktionsanfang Sei < freie Quantifizierung > und `t = Nil`.

- linke Seite: ...
- rechte Seite: ...

Induktionsvoraussetzung Seien `l,r :: BinTree a`, sodass < freie Quantifizierung > gilt:

- < Gleichung für `l` > , < Gleichung für `r` >

Induktionsschritt Sei `x :: a` und < freie Quantifizierung >. Es gilt: < Gleichungsfolge >

Baumstruktur mit expliziten Blättern

```
data BinTree a = Branch a (BinTree a) (BinTree a) | Leaf a
```

Induktionsanfang Sei `x :: a` und < freie Quantifizierung >.

- linke Seite: ...
- rechte Seite: ...

Induktionsvoraussetzung Seien `l,r :: BinTree a`, sodass < freie Quantifizierung > gilt:

- < Gleichung für `l` >, < Gleichung für `r` >

Induktionsschritt Sei `x :: a` und < freie Quantifizierung >. Es gilt: < Gleichungsfolge >