Beweis von gleichungsbasierten Eigenschaften

OOPM, Ralf Lämmel

Distributivgesetz der natürlichen Zahlen

Axiom 1:

x * (y + z) = x * y + x * z

Kann man diese Gleichung aus einer Definition der Operationen ableiten?
Wie wendet man diese Gleichungen an?

Eine ähnliche Gleichung

Axiom 2:

2 * (y + z) = 2 * y + 2 * z

Dies ist auch “ein” Distributivgesetz.
Axiom 2 ist “weniger allgemein” als Axiom 1.

Anwendung von Gleichungen

Kann man Axiom 1
auf den folgenden Ausdruck anwenden?

5 * (6 + 7)

Was ist mit
Axiom 2?

Antwort: Ja!
Siehe das folgende Resultat.

5 * 6 + 5 * 7

Formale Notation

$$5 * (6 + 7) \rightarrow_{[A1]} 5 * 6 + 5 * 7$$

Der Term 5 * (6 + 7) kann durch Anwendung des Axioms 1 “von links nach rechts” in den Term 5 * 6 + 5 * 7 umgeschrieben werden.

Anwendung von Gleichungen

Kann man Axiom 1
auf den folgenden Ausdruck anwenden?

5 * 6 + 5 * 7

Antwort: Ja!
Siehe das folgende Resultat.

5 * (6 + 7)

Formale Notation

$$5 * 6 + 5 * 7 \leftarrow_{[A1]} 5 * (6 + 7)$$

Der Term 5 * 6 + 5 * 7 kann durch Anwendung des Axioms 1 “von rechts nach links” in den Term 5 * (6 + 7) umgeschrieben werden.

Anwendung von Gleichungen

Kann man Axiom 1
auf den folgenden Ausdruck anwenden?

5 * 6 + 5 * 7 + 1

Antwort: Ja!
Siehe das folgende Resultat.

5 * (6 + 7) + 1

Formale Notation

$$\underline{5 * 6 + 5 * 7} + 1 \leftarrow_{[A1]} \underline{5 * (6 + 7)} + 1$$

Der ***Teilterm*** 5 * 6 + 5 * 7 kann durch Anwendung des Axioms 1 “von rechts nach links” in den ***Teilterm*** 5 * (6 + 7) umgeschrieben werden.

Gleichungsbasiertes Schließen

... ist die Formalisierung von Ideen zur Anwendung von Gleichungen (Axiomen) auf Terme (z.B. Terme die arithmetische Ausdrücke repräsentieren).

Algebraische Spezifikation vom Booleschen Datentyp

```
specification bool
sort Bool
constructors
  true : -> Bool
  false : -> Bool
functions
  not : Bool -> Bool
  and : Bool x Bool -> Bool
  or : Bool x Bool -> Bool
equations
  not(true) = false
  not(false) = true
  and(false, false) = false
  and(false, true) = false
  and(true, false) = false
  and(true, true) = true
  or(false, false) = false
  or(false, true) = true
  or(true, false) = true
  or(true, true) = true
```

Anwendung von Gleichungen zur Normalisierung

not(and(<u>not(false)</u>,false)	`not(false) = true`
→ not(<u>and(true,false)</u>)	`and(true,false) = false`
→ <u>not(false)</u>	`not(false) = true`
→ true	

Wir wenden Gleichungen solange an bis nichts mehr geht (oder für immmer).

Beweis von Eigenschaften

Ausgewählte Eigenschaften natürlicher Zahlen und des Booleschen Datentyps

"Zu zeigen"*!*

- and(x,y) = and(y,x) [and-comm]
- or(x,x) = x [or-same]
- add(x,y) = add(y,x) [add-comm]
- add(x,add(y,z)) = add(add(x,y),z) [add-assoc]
- add(succ(x),y) = succ(add(x,y)) [succ-left]
- add(x,succ(y)) = succ(add(x,y)) [succ-right]

Bedeutung von Gleichungen

- **Für alle natürlichen Zahlen x und y, gilt dass**
  - add(x,y) = add(y,x) [add-comm]
- Beachte: Wir nehmen an, dass alle natürlichen Zahlen durch zero und succ repräsentiert werden. Die Unterscheidung von Konstruktoren und anderen Funktionen ist also essentiell.

Beweistechniken

- Anwendung von Gleichungen
- Erschöpfende Fallunterscheidung
- Vollständige Induktion
- Strukturelle Induktion

Nicht immer ausreichend

Beschränkung auf endliche Wertebereiche

Beschränkung auf natürliche Zahlen

"trivial"

- Zu zeigen:
  - add(succ(x),y) = succ(add(x,y)) [succ-left]
- Beweis:

```
        add(succ(x),y)
→[2]    succ(add(x,y))
```

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
```

Q.E.D.

- Zu zeigen:
  - add(x,succ(y)) = succ(add(x,y)) [succ-right]
- Annahme:
  - add(x,y) = add(y,x) [add-comm]
- Beweis:

	add(x,succ(y))
→[add-comm]	add(succ(y),x)
→[2]	succ(add(y,x))
→[add-comm]	succ(add(x,y))

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
```

Q.E.D.

- Zu zeigen:
  - and(x,y) = and(y,x) [and-comm]
- Beweis:
  - **Erschöpfende Fallunterscheidung**
    - über **alle** Werte für x und y.

Satz: “add” ist assoziativ.

- Zu zeigen:
  - add(x , add(y , z)) = add(add(x , y) , z)
- Beweis:
  - **Induktion über x**

Details folgen etwas später.

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
```

Beweis durch erschöpfende Fallunterscheidung

and(b1,b2) = and(b2,b1) *?*

- Konventionen:
  - LHS (linke Seite; left-hand side)
  - RHS (rechte Seite; right-hand side)
- Zu zeigen dass LHS = RHS
  - für alle Belegungen von b1 und b2
  - vermöge Gleichungen der Wahrheitstabelle

```
[1] and(false,false) = false
[2] and(false,true) = false
[3] and(true,false) = false
[4] and(true,true) = true
```

and(b1,b2) = and(b2,b1) *?*

```
[1] and(false,false) = false
[2] and(false,true) = false
[3] and(true,false) = false
[4] and(true,true) = true
```

- Beweis per Fallunterscheidung
  - Fall 1: b1 = false
    - Fall 1a: b2 = false
      - LHS: and(false,false) = false wegen [1]
      - RHS: and(false,false) = false wegen [1]
    - Fall 1b: b2 = true
      - LHS: and(false,true) = false wegen [2]
      - RHS: and(true,false) = false wegen [3]
  - Fall 2: b1 = true
    - Fall 2a: b2 = false
      - LHS: and(true,false) = false wegen [3]
      - RHS: and(false,true) = false wegen [2]
    - Fall 2b: b2 = true
      - LHS: and(true,true) = true wegen [4]
      - RHS: and(true,true) = true wegen [4]

Q.E.D.

Verschiedene Axiomatisierungen von “and”

```
and(false,false)  = false
and(false,true)   = false
and(true,false)   = false
and(true,true)    = true
```

Definition basierend auf erschöpfender Fallunterscheidung von Booleschen Argumenten

```
and(true,b) = b
and(b,true) = b
and(false,b) = false
and(b,false) = false
```

Verwendung von Variablen “beim Matchen” (Die Gleichungen schließen einander immer noch aus!)

```
and(true,true) = true
and(b1,b2) = false
```

Wir haben eine speziellere und eine allgemeinere Gleichung.

Definition versus Folgerung

```
and(false,false)  = false
and(false,true)   = false
and(true,false)   = false
and(true,true)    = true
```

Wir können diese Gleichungen als die primäre “Definition” von “and” ansehen.

```
and(b1,b2) = and(b2,b1)
and(b1,and(b2,b3)) = and(and(b1,b2),b3)
```

Die Eigenschaften der Kommutativität und Assoziativität folgen aus der “primären” Definition.

Mögliche Unterscheidung von Definition und Folgerung

```
equations
    and(false,false)  = false
    and(false,true)   = false
    and(true,false)   = false
    and(true,true)    = true

corollaries
    and(b1,b2) = and(b2,b1)
    and(b1,and(b2,b3)) = and(and(b1,b2),b3)
```

Wir sollten immer beweisen, dass angenommene Folgerungen tatsächlich aus anderen Gleichungen folgen.

Beweis durch vollständige Induktion

Beweis mittels vollständiger Induktion

- Behauptung: $A(k)$ gilt für alle natürlichen Zahlen k
- *Beweisschema*
  - Induktionsanfang (IA)
    - Zu zeigen dass $A(0)$ gilt.
  - Induktionsschritt (IS)
    - Sei k eine beliebige natürliche Zahl.
    - Induktionsvoraussetzung (IV): $A(k)$ gilt.
    - Induktionsbehauptung (IB): $A(k+1)$ gilt.
    - Beweis:

      Zeige dass IB gilt unter Verwendung von IV.

im Beispiel: add(x,add(y,z)) = add(add(x,y),z) gilt für alle natürlichen Zahlen x.

Warum klappt das?

- k = 0: gezeigt durch IA.
- k = 1: Beweis von IS zeigt: A(0) ⇒ A(1)
- k = 2: Beweis von IS zeigt: A(1) ⇒ A(2)
- ...

- Zu zeigen
  - add(x,add(y,z)) = add(add(x,y),z) [add-assoc]
- Beweis: Induktion über x
  - **Induktionsanfang**: x = zero
    - Zu zeigen: add(zero,add(y,z)) = add(add(zero,y),z)
    - Beweis

	add(zero,add(y,z))
→[1]	add(y,z)
←[1]	add(add(zero,y),z)

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
```

Q.E.D.

- **Induktionsschritt**: x = k
  - IV: add(k,add(y,z)) = add(add(k,y),z)
  - IB: add(succ(k),add(y,z)) = add(add(succ(k),y),z)
  - Beweis:

	add(succ(k),add(y,z))
→[2]	succ(add(k,add(y,z))
→[IV]	succ(add(add(k,y),z)
←[2]	add(succ(add(k,y)),z)
←[2]	add(add(succ(k),y),z)

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
```

Q.E.D.

- Zu zeigen:
  - add(x,y) = add(y,x) [add-comm]
- Beweis: Induktion über x
  - **Induktionsanfang**: x = zero
    - Zu zeigen: add(zero,y) = add(y,zero)
    - Beweis: Induktion über y
      - Übungsaufgabe

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
```

* **Induktionsschritt**: x = k
  - IV: add(k,y) = add(y,k)
  - IB: add(succ(k),y) = add(y,succ(k))
  - Beweis:

	add(succ(k),y)
←[2]	succ(add(k,y))
→[IV]	succ(add(y,k))
→[succ-right]	add(y,succ(k))

```
[1] add(zero,n) = n
[2] add(succ(n),m) = succ(add(n,m))
    add(n,succ(m)) = succ(add(n,m))
```

Q.E.D.

Ein Münchhausen-Problem

- Beweisabhängigkeiten
  - Beweis von [succ-right] nahm [add-comm] an.
  - Und umgekehrt*!*
- Was tun?
  - Beweis von [succ-right] per Induktion.
    - Übungsaufgabe

Beweis durch strukturelle Induktion

Von Vollständiger Induktion zur Strukturellen Induktion

- Induktion über Repräsentation
- Induktionsanfang
  - Fälle für *nichtrekursive* Konstruktoren
- Induktionsschritt
  - Fälle für *rekursive* Konstruktoren

Primäre Definition von Listen

[append@nil] append(nil, l) = l

[append@cons] append(cons(t, l1), l2) = cons(t, append(l1, l2))

[length@nil] length(nil) = zero

[length@cons] length(cons(t, l)) = succ(length(l))

[reverse@nil] reverse(nil) = nil

[reverse@cons] reverse(cons(t, l)) = append(reverse(l), cons(t,nil))

Folgerungen (?)

length(append(l1,l2)) = add(length(l1),length(l2)) [app-len]

append(l1,append(l2,l3)) = append(append(l1,l2),l3) [app-assoc]

reverse(reverse(l)) = l [revrev]

- Zu zeigen:
  - length(append(l1,l2)) = add(length(l1),length(l2)) [app-len]
- Beweis: Induktion über l1
  - **Induktionsanfang**: l1 = nil
    - Zu zeigen:
      - length(append(nil,l2)) = add(length(nil),length(l2)
    - Beweis:

	length(append(nil,l2))
→[append@nil]	length(l2)
→[add@zero]	add(zero,length(l2))
←[length@nil]	add(length(nil),length(l2))

Q.E.D.

* **Induktionsschritt**: l1 = k

- IV: length(append(k,l2)) = add(length(k),length(l2))
- IB: length(append(cons(t,k),l2)) = add(length(cons(t,k)),length(l2))
  für beliebige t
- Beweis

	length(append(cons(t,k),l2))
→ [append@cons]	length(cons(t,append(k,l2)))
→ [length@cons]	succ(length(append(k,l2)))
→ [IV]	succ(add(length(k),length(l2)))
← [add@succ]	add(succ(length(k)),length(l2))
← [length@cons]	add(length(cons(t,k)),length(l2))

Die Rechtfertigung für diese Wiedereinfuhr von *t* sollte man genauer erörtern.

Q.E.D.

- **Zusammenfassung**
  - Spezifikationen sind “Modelle” von Programmen.
  - Spezifikationen können wie Programme ausgeführt werden.
  - Wir unterscheiden “Definitionen” versus “Schlussfolgerungen”.
  - Letztere müssen als korrekt (als Folgerungen) bewiesen werden.
  - Spezifikationen dienen auch zum Testen von Programmen.
- **Ausblick**
  - Vor- und Nachbedingungen von (imperativen) Programmen.
  - Komplexität von (imperativen) Programmen.
  - Ausnahmen, Generics, …
  - Modellierung von Verhalten mit Zustandsdiagrammen
  - …