top(pop(push(42,empty))) = ?

http://shemesh.larc.nasa.gov/fm/fm-humor.html

Algebraische Spezifikation

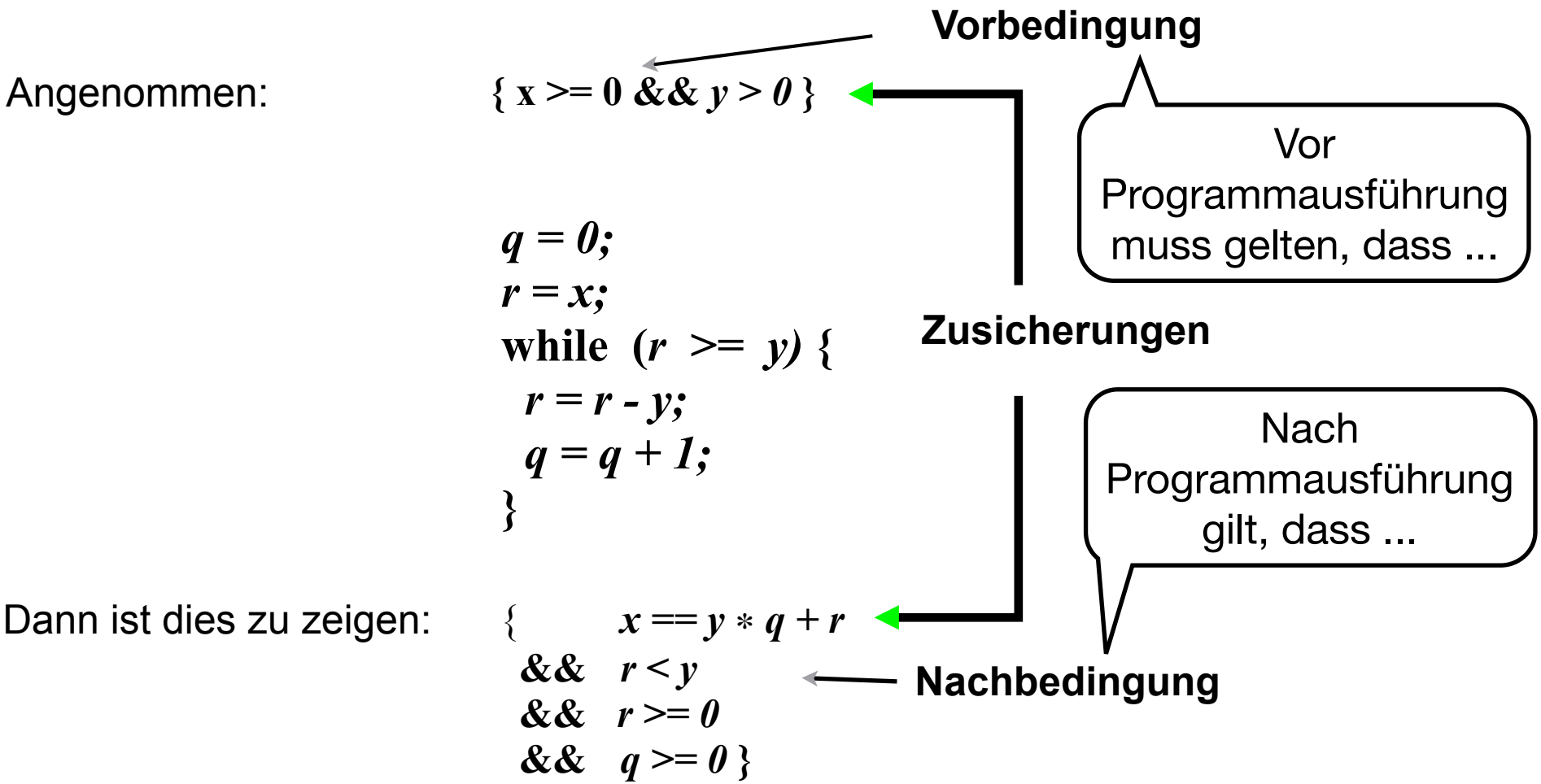
OOPM, Ralf Lämmel

Wie beschreiben wir das erwartete Verhalten von einem ADT?

```
public class IntStack {
   private IntListEntry top = null;
   public void push(int item) {
      IntListEntry e = new IntListEntry();
      e.item = item;
      e.next = top;
      top = e;
   }
   public boolean isEmpty() { return top == null; }
   public int top() { return top.item; }
   public void pop() { top = top.next; }
}
```

Testfälle?
Vor-/Nachbedingungen?

So beschreiben wir das erwartete Verhalten von einem Algorithmus!



Optionen zur Verhaltensbeschreibung von ADTs

- **Testfälle**
  - Keine vollständige Beschreibung.
    - Siehe ADT-Vorlesung.
- **Vor- und Nachbedigungen**
  - Nur eingeschränkt möglich.
    - Wir benötigen Zugriff auf die ‘*Geschichte*’ von Kapseln.
- **Algebraische Spezifikationen**
  - Wie heute besprochen.

Algebraische Spezifikation von IntStack: **Konstruktoren**

```
empty    : IntStack
push     : int × IntStack → IntStack
```

Operationen
zur kanonischen Repräsentation

Repräsentation von Kellern:

```
empty,
push(42,empty),
push(88,push(42,empty)),
...
```

Algebraische Spezifikation von IntStack: **Operationen**

```
isEmpty  : IntStack → Boolean
top      : IntStack → int
pop      : IntStack → IntStack
```

Alle Operationen
außer den Konstruktoren.

***Die ultimative Frage*: was ergeben Anwendungen von Operationen auf kanonische Repräsentationen?**

```
top(push(88,push(42,empty))) = ?
```

Algebraische Spezifikation von IntStack: **Gleichungen**

```
isEmpty(empty)          = true

isEmpty(push(t,l))      = false

top(push(t,l))          = t

pop(push(t,l))          = l
```

Der leere Keller ist leer.

Ein durch “push” entstandener Keller ist nicht leer.

Der zuletzt eingekellerte Werte wird durch “top” zurückgegeben.

“pop” gibt den Keller zurück, auf welchen zuletzt eingekellert wurde.

Anwendung von Gleichungen zur Normalisierung

push(88,pop(push(42,empty))) | `pop(push(t,l)) = l`

⇒ push(88,**empty**)

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

Algebraische Spezifikation von IntStack

Konstruktoren

```
empty     : IntStack
push      : int × IntStack → IntStack
```

Operationen

```
isEmpty   : IntStack → Boolean
top       : IntStack → int
pop       : IntStack → IntStack
```

Gleichungen

```
isEmpty(empty)       = true
isEmpty(push(t,l))   = false
top(push(t,l))       = t
pop(push(t,l))       = l
```

Algebraische Spezifikation von Boolean

specification Boolean

sorts Boolean

constructors

true : → Boolean

false : → Boolean

operations

not : Boolean → Boolean

and : Boolean × Boolean → Boolean

or : Boolean × Boolean → Boolean

equations

...

Verwendung einer Spezifikationssprache; siehe die Schlüsselwörter.

Gleichungen zu Boolean

equations

not(true) = false
not(false) = true

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

...

Anwendung von Gleichungen zur Normalisierung

not(and(not(false),false)	`not(false) = true`
⇒ not(and(true,false))	`and(true,false) = false`
⇒ not(false)	`not(false) = true`
⇒ true	

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

Alternative 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.

Definition versus Folgerung

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


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

Unterscheidung von Definition und Folgerung in der Spezifikationssprache

```
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 beweisen, dass die Folgerungen tatsächlich aus den anderen Gleichungen folgen.

Unterscheidung von Definition und Folgerung in der Spezifikationssprache

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

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

Alternative

Wir sollten beweisen, dass die Folgerungen tatsächlich aus den anderen Gleichungen folgen.

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
```

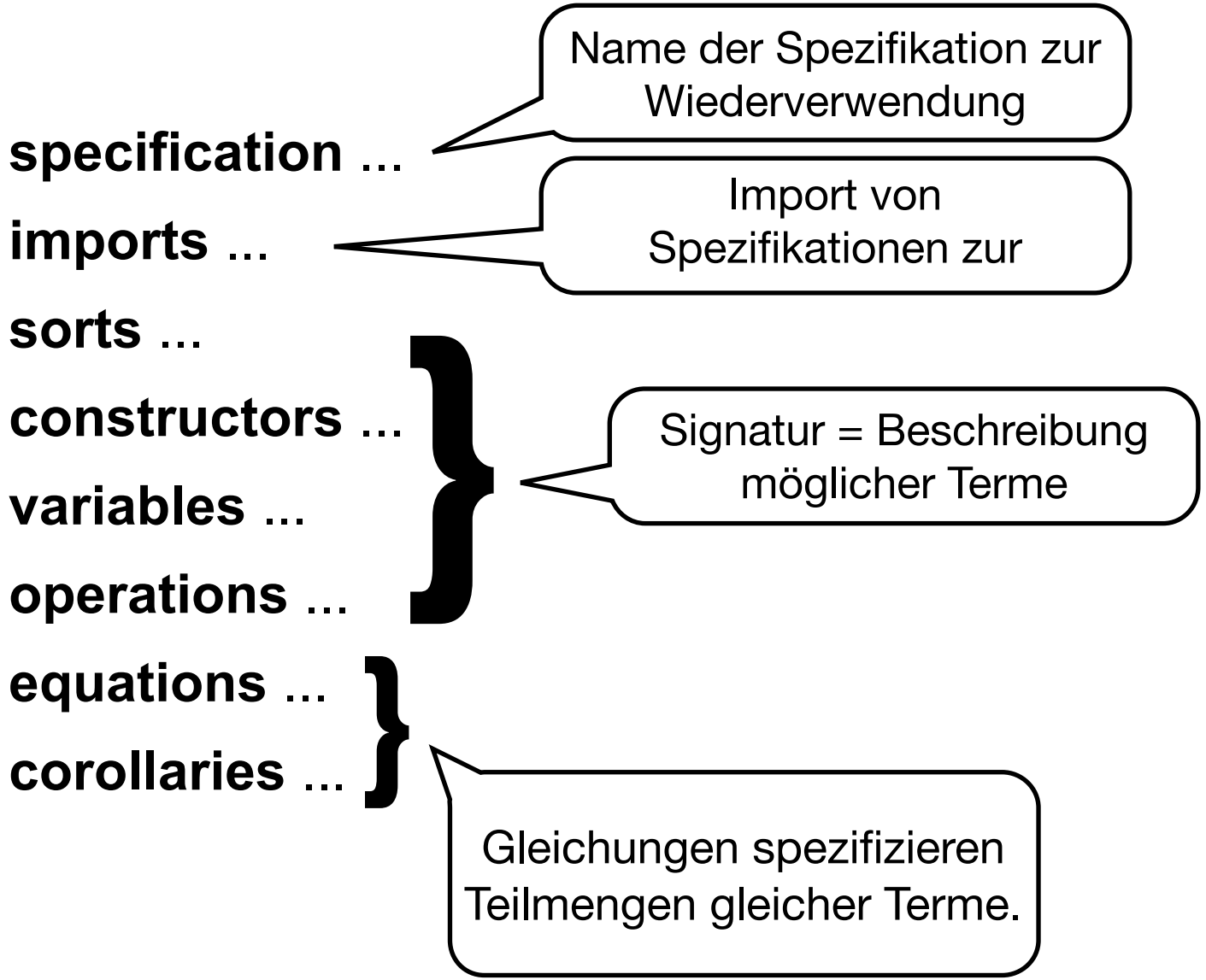
- Beweis per Fallunterscheidung
  - Fall 1: b1 = false
    - Fall 1a: b2 = false
      - LHS: and(false,false) = false wegen [1]
      - RHS: and(false,false) = false wegen [1]
    - Fall 1b: b2 = true
      - LHS: and(false,true) = false wegen [2]
      - RHS: and(true,false) = false wegen [3]
  - Fall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

Algebraische Spezifikation - Wozu?

- Algebraische Spezifikation als ***Modellierungsform***
  - Spezifikation mit mathematischer Interpretation
  - Ausführbare Spezifikationen
  - Grundlage korrekter Implementation
  - Anwendbarkeit mathematischer Beweismethoden
  - Deklarative Programmierung
    - Beschreibung des “Was”; nicht des “Wie”
    - Abstraktion von operationaler Semantik
    - Fokussierung auf Programmeigenschaften

Eine Sprache für (konstruktive) algebraische Spezifikationen



Weitere Beispiele

Eine Spezifikation für die natürlichen Zahlen

Repräsentation von natürlichen Zahlen:

zero,
succ(zero),
succ(succ(zero)),
succ(succ(succ(zero))),
...

```
sorts Nat

constructors
  zero : → Nat              // "0"
  succ : Nat → Nat          // Successor

operations
  add  : Nat × Nat → Nat    // Addition
  mult : Nat × Nat → Nat    // Multiplication
  factorial : Nat → Nat     // Factorial function

equations
  ...
```

Eine Spezifikation für die natürlichen Zahlen

```
specification Nat

sorts Nat

variables
  n, m : Nat

constructors
  zero : → Nat              // "0"
  succ : Nat → Nat          // Successor

operations
  add  : Nat × Nat → Nat    // Addition
  mult : Nat × Nat → Nat    // Multiplication
  factorial : Nat → Nat     // Factorial function

equations
 add(zero,n) = n
 add(succ(n),m) = succ(add(n,m))
 mult(zero,n) = zero
 mult(succ(n),m) = add(m,mult(n,m))
 factorial(zero) = succ(zero)
 factorial(succ(n)) = mult(succ(n),factorial(n))
```

Benennung der Spezifikation

Vereinbarung von Variablen

zero,
succ(zero),
succ(succ(zero)),
...

Variablen sind universell quantifiziert in Gleichungen. Man kann also jeden Term der Sorte “einsetzen”.

Definition versus Folgerung

```
equations
 add(zero,n) = n
 add(succ(n),m) = succ(add(n,m))
 mult(zero,n) = zero
 mult(succ(n),m) = add(m,mult(n,m))
 factorial(zero) = succ(zero)
 factorial(succ(n)) = mult(succ(n),factorial(n))
```

Beachte die Argumentmuster basierend auf Konstruktoren (und Variablen).

```
corollaries
  add(m,n) = add(n,m)
  add(m,add(n,k)) = add(add(m,n),k)
  ...
```

Wir mögen uns weitere, nicht-konstruktive Gleichungen gestatten, aber diese müssen wir “markieren” und nicht bei der Normalisierung verwenden sondern vielmehr als implizierte Eigenschaften annehmen oder herleiten.

Spezifikation von Listen über Nat

Repräsentation von Listen:

nil,
cons(42,nil),
cons(88,cons(42,nil)),
...

Statt: {}, {42}, {88,42}

```
sorts NatList

constructors
  nil    : → NatList
  cons   : Nat × NatList → NatList

operations
  append : NatList × NatList → NatList
  snoc   : NatList × Nat → NatList
  ...

equations
 ...
```

Spezifikation von Listen über Nat

```
specification NatList

imports Nat

sorts NatList

variables
  l, l1, l2 : NatList

constructors
  nil    : → NatList
  cons   : Nat × NatList → NatList

operations
  append : NatList × NatList → NatList
  snoc   : NatList × Nat → NatList
  ...
equations
 append(nil,l) = l
 append(cons(n,l1),l2) = cons(n,append(l1,l2))
 snoc(l,n) = append(l,cons(n,nil))
 ...
```

Modularisierung basiert auf Importierung von Spezifikationen.

~~{42, 88}~~
~~[42, 88]~~
cons(42,cons(88,nil))

Definitionen dürfen auch “Substitution” benutzen.

Implementation algebraischer Spezifikationen

- Zielstellung
  - Abbildung der Spezifikationen auf Java.
  - Prototypische Implementation ausreichend.

Es gibt Sprachen die keine Codierung verlangen: F#, Scala, Haskell, ...

- ***Eine*** Vorgehensweise
  - Sorte wird Schnittstelle.
  - Operationen sind Methoden der Schnittstelle.
  - **Konstruktoren werden Klassen.**
  - Die Klassen implementieren die Schnittstelle.
  - Die Klassen haben Attribute für Konstruktorenargumente.

Es gibt diverse Optionen und Spezialfälle.

Sorte versus Schnittstelle

```
sorts Nat
constructors
  zero : → Nat              // "0"
  succ : Nat → Nat          // Successor
operations
  add  : Nat × Nat → Nat    // Addition
  mult : Nat × Nat → Nat    // Multiplication
  factorial : Nat → Nat     // Factorial function
```

```
public interface Nat {
   Nat add(Nat m);
   Nat mult(Nat m);
   Nat factorial();
}
```

"this" steht für die erste Argumentposition.

Konstruktoren versus Klassen

```
sorts Nat
constructors
  zero : → Nat              // "0"
  succ : Nat → Nat          // Successor
```

```
public class Zero implements Nat { ... }

public class Succ implements Nat {

   private Nat n;
   public Succ(Nat n) { this.n = n; }
   ...
}
```

Gleichungen für den “zero”-Fall

```
equations
 add(zero,n) = n
 add(succ(n),m) = succ(add(n,m))
 mult(zero,n) = zero
 mult(succ(n),m) = add(m,mult(n,m))
 factorial(zero) = succ(zero)
 factorial(succ(n)) = mult(succ(n),factorial(n))
```

```
public class Zero implements Nat {
   public Zero() { }
   public Nat add(Nat n) { return n; }
   public Nat mult(Nat m) { return new Zero(); }
   public Nat factorial() {
      return new Succ(new Zero());
   }
}
```

Gleichungen für den “succ”-Fall

```
equations
 add(zero,n) = n
 add(succ(n),m) = succ(add(n,m))
 mult(zero,n) = zero
 mult(succ(n),m) = add(m,mult(n,m))
 factorial(zero) = succ(zero)
 factorial(succ(n)) = mult(succ(n),factorial(n))
```

```
public class Succ implements Nat {
  private Nat n;
  public Succ(Nat n) { this.n = n; }
  public Nat add(Nat m) { return new Succ(n.add(m)); }
  public Nat mult(Nat m) { return m.add(n.mult(m)); }
  public Nat factorial() { return mult(n.factorial()); }
}
```

Anteil für den Konstruktor

Diskussion ***dieser*** Implementationsstrategie

- *Seiteneffektfreiheit*
  - Die Methoden verändern keine bestehenden Objekte.
- *Ineffizienz*
  - Die Vermeidung von **int** erscheint potentiell unvernünftig.
- *Ad-hoc Zuordnung von Operationen zu Klassen*
  - Die Sorte des ersten Argumentes entscheidet.
- *Beschränkungen*
  - durch Substitution definierte Funktionen
  - ***verschachtelte Muster***
  - ***Matching in mehreren Argumenten***

Beweise von Eigenschaften

Ausgewählte Eigenschaften natürlicher Zahlen

- “succ” kann von links nach aussen gezogen werden.
  - add(succ(x),y) = succ(add(x,y))
- … auch von rechts.
  - add(x,succ(y)) = succ(add(x,y))
- “add” ist assoziativ.
  - add(x,add(y,z)) = add(add(x,y),z)
- “add” ist kommutativ.
  - add(x,y) = add(y,x)

Beweistechniken

- Anwendung von Gleichungen
- Erschöpfende Fallunterscheidung
- Strukturelle Induktion

Nicht immer ausreichend

Beschränkung auf endliche Wertebereiche

Anwendbarkeit auf Eigenschaften von Nat, List<T> usw.

Satz: Für alle x,y : Nat gilt: add(succ(x),y) = succ(add(x,y))

"trivial"

- Zu zeigen
  - add(succ(x),y) = succ(add(x,y))
- Beweis ("Forme LHS in RHS um.")

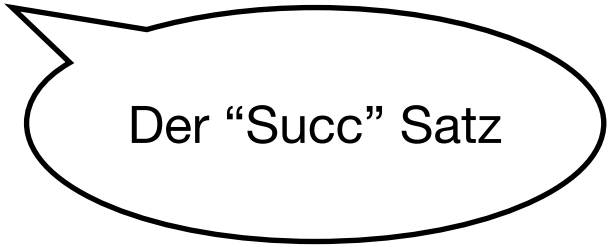
```
 add(succ(x),y)
=    succ(add(x,y))          { Wende [2] an.
                               n := x
                               m := y }
```

Substitutionen werden nachfolgend weggelassen.

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

Q.E.D.

Satz: Für alle x,y : Nat gilt: add(x,succ(y)) = succ(add(x,y))



- Annahme
  - add ist kommutativ, d.h.:
    - add(x,y) = add(y,x)
- Zu zeigen: add(x,succ(y)) = succ(add(x,y))
- Beweis

```
  add(x,succ(y))
=   add(succ(y),x)          { Kommutativität }
=   succ(add(y,x))          { [2] }
=   succ(add(x,y))          { Kommutativität }
```

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

Q.E.D.

Satz: “add” ist assoziativ.

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

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

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: $A(0) \Rightarrow A(1)$
- $k = 2$: Beweis von IS zeigt: $A(1) \Rightarrow A(2)$
- ...

Satz: “add” ist assoziativ.

- Zu zeigen
  - add(x,add(y,z)) = add(add(x,y),z)
- 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))
      = add(y,z) { →[1] }
      = add(add(zero,y),z) { ←[1] }

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

Q.E.D.

Satz: “add” ist assoziativ.

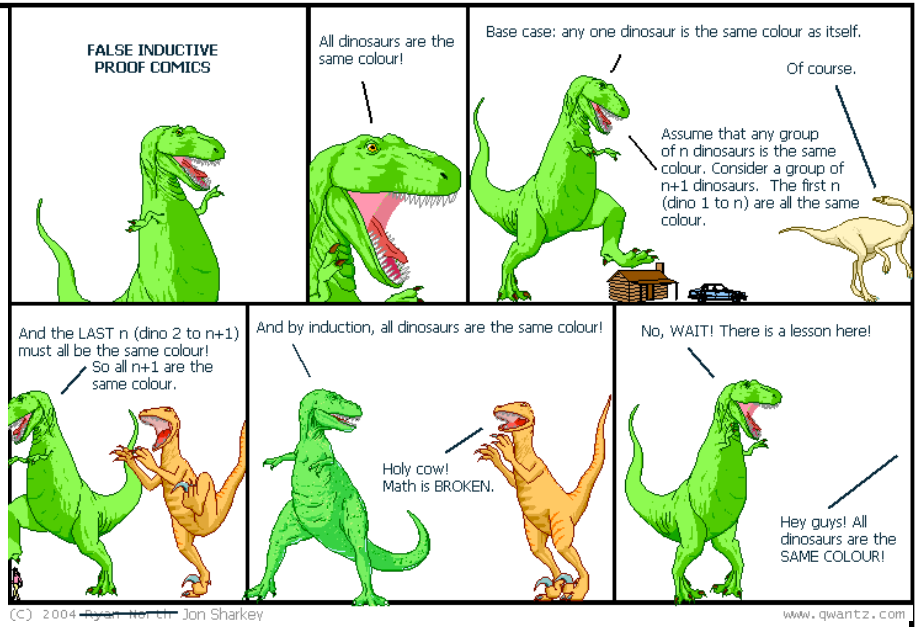
- **Induktionsschritt**: x = k
  - IV: add(k,add(y,z)) = add(add(k,y),z)
  - IB: add(succ(k),add(y,z)) = add(add(succ(k),y),z)
  - Beweis

$$
\begin{aligned}
& \underline{\mathrm{add}(\mathrm{succ}(k),\mathrm{add}(y,z))} \\
=\ & \mathrm{succ}(\underline{\mathrm{add}(k,\mathrm{add}(y,z))} && \{\ \rightarrow [2]\ \} \\
=\ & \underline{\mathrm{succ}(\mathrm{add}(\mathrm{add}(k,y),z)} && \{\ \mathrm{IV}\ \} \\
=\ & \mathrm{add}(\underline{\mathrm{succ}(\mathrm{add}(k,y))},z) && \{\ \leftarrow [2]\ \} \\
=\ & \mathrm{add}(\mathrm{add}(\mathrm{succ}(k),y),z) && \{\ \leftarrow [2]\ \}
\end{aligned}
$$

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

Q.E.D.

- **Zusammenfassung**
  - ADTs können (“algebraisch”) spezifiziert werden.
  - Eigenschaften von algebraischen Spezifikation
    - Präzise (verglichen zu imperativen/OO-Programmen)
    - Formal (verwendbar für mathematische Beweise)
    - Ausführbar (im funktionalen oder OO-Paradigma)
    - Implementiertierbar
- **Ausblick**
  - Komplexitätsanalyse
  - Objektorientierung



http://www.qwantz.com/fanart/dino-comic-proof.png

Weitere induktive Beweise

Fakultatives Material

Satz: “add” ist kommutativ.

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

Fakultatives Material

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

Satz: “add” ist kommutativ.

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

	add(succ(k),y)	
=	succ(add(k,y))	{ ←[2] }
=	succ(add(y,k))	{ IV }
=	add(y,succ(k))	{ Der “Succ” Satz }

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

Q.E.D.

Fakultatives Material

Satz: “add” ist kommutativ.

- Oops!
  - Beweis von “Succ” Satz benutzt Kommutativität.
  - Beweis von Kommutativität benutzt “Succ” Satz.
- Was tun?
  - Beweise “Succ” Satz per Induktion.
    - Übungsaufgabe

Fakultatives Material

Von Vollständiger Induktion zur Strukturellen Induktion

- Induktion immer über kanonische Repräsentation
- Induktionsanfang
  - Fälle für nichtrekursive Konstruktoren
- Induktionsschritt
  - Fälle für rekursive Konstruktoren

Fakultatives Material

Eigenschaften von Listen

```
append(nil,l) = l
append(cons(t,l1),l2) = cons(t,append(l1,l2))

length(nil) = zero
length(cons(t,l)) = succ(length(l))

reverse(nil) = nil
reverse(cons(t,l)) = append(reverse(l),cons(t,nil))
```

- “append” bewahrt Länge.
  - length(append(l1,l2)) = add(length(l1),length(l2))
- “append” ist assoziativ.
  - append(l1,append(l2,l3)) = append(append(l1,l2),l3)
- Zweimal “reverse” ist Identität.
  - reverse(reverse(l)) = l

Fakultatives Material

Satz: “append” bewahrt die Länge.

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

```
    length(append(nil,l2))
=   length(l2)                          { [append@nil] }
=   add(zero,length(l2))                { [add@zero] }
=   add(length(nil),length(l2))         { [length@nil]
```

Q.E.D.

Fakultatives Material

Satz: “append” bewahrt die Länge.

- **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))	
=	length(cons(t,append(k,l2)))	{ → append@cons }
=	succ(length(append(k,l2)))	{ → length@cons }
=	succ(add(length(k),length(l2)))	{ IV }
=	add(succ(length(k)),length(l2))	{ ←add@succ }
=	add(length(cons(t,k)),length(l2))	{ ←length@cons }

Q.E.D.

Fakultatives Material