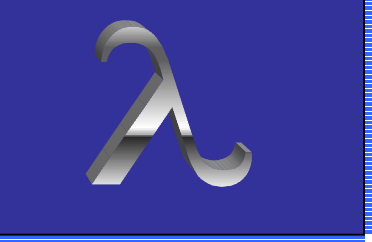
Einführung in die Funktionale Programmiersprache Haskell

Zahlen in Haskell

Kapitel 3

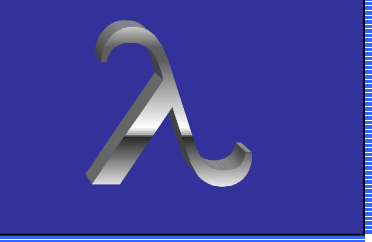
FH Wedel
IT-Seminar: WS 2003/04
Dozent: Prof. Dr. Schmidt
Autor: Timo Wlecke (wi3309)
Vortrag am: 04.11.2003



Einleitung - Kapitel 3

- Bereits bekannt: *Integer*, *Int* und *Float*
- Heute: Rekursive Datentypen und Definitionen
- „Primitive“ Datentypen werden deklariert
- Beweis durch (vollständige) Induktion
- *Fold*-Funktion
- Rationale und komplexe Zahlen

$\Rightarrow$ Es geht um verschiedene Zahlenmengen und dazugehörige Operationen



Natürliche Zahlen

- 0, 1, 2...
- *Succ* = Nachfolger

```
data Nat = Zero | Succ Nat
```

- Bsp.: Succ(Succ(Succ Zero))) → 3
- Einfache arithmetische Funktionen für *Nat, z.B.* hier die Addition:

```
(+) :: Nat → Nat → Nat

m + Zero = m

m + Succ n = Succ(m + n)
```

Natürliche Zahlen

- Instanz-Deklaration:

```
instance Eq Nat where
Zero   == Zero   = True
Zero   == Succ n = False
Succ m == Zero   = False
Succ m == Succ n = (m == n)
```

Natürliche Zahlen

- Alternative Deklaration:

```
data Nat = Zero | Succ Nat
     deriving (Eq, Ord, Show)
```

- Subtraktion ist eine partielle Funktion

  → Einführung der Integer

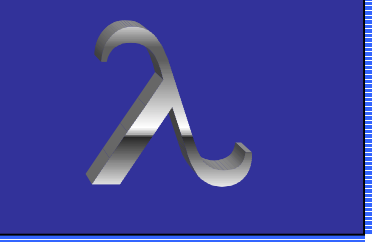
```
(-) :: Nat → Nat → Nat
 m - Zero = m
 Succ m - Succ n = m - n
```

Partielle Zahlen

- Spezialfälle natürlicher und anderer Zahlen:
  - Undefinierte Werte und
  - unendliche Werte
- Undefinierte Zahlen:

```
undefined :: Nat
undefined = undefined
```

- dargestellt als ⊥, Succ ⊥, Succ(Succ ⊥),...



Partielle Zahlen

- Wert *infinity*

```
infinity :: Nat
infinity = Succ infinity
```

  - Größtes Element aus *Nat*
  - Einzige Zahl x, für die immer *Succ m* < *x* = true
  - Für alle Zahlen n gilt

```
infinity + n = infinity
```

  - aber nur für endliche Zahlen

```
n + infinity = infinity
```

Grundlagen der Induktion

- Logisches Verfahren, um vom Besonderen auf das Allgemeine zu schließen
- Prüfung der Gültigkeit von Eigenschaften durch Pattern-Matching
- Um zu zeigen, dass eine Eigenschaft *P(n)* für jede endliche Zahl *n* aus *Nat* gilt, ist es ausreichend, dass:
  - *P(Zero)* gilt und wenn
  - *P(n)* gilt, dass in dem Fall auch *P(Succ n)* gültig ist.

  (Prinzip der strukturellen Induktion)

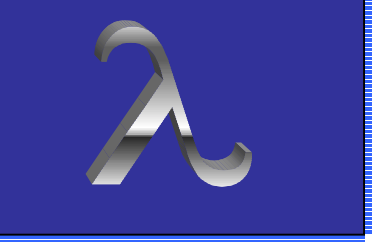
Grundlagen der Induktion

- Beispiel: *Zero* + *n* = *n* (Induktionshypothese)
- Ersetzen von *n* erst durch *Zero*, dann durch *Succ n*
- **Case (Zero):** *Zero* + *Zero* = *Zero*.
- **Case (Succ n):** *Zero* + *Succ n* = *Succ n*

```
Zero + Succ n
= {2. Gleichung für (+)}
Succ(Zero + n)
= {Induktionshypothese}
Succ n
```

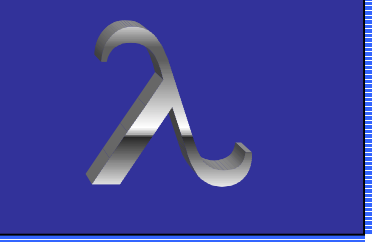
Grundlagen der Induktion

- Fast automatischer Prozess:
  - Beide Seiten substituieren und dadurch vereinfachen
  - Vereinfachungsschritt wird durch die Form des Ausdrucks bestimmt
- Entscheidend ist Wahl der richtigen Variablen



Vollständige Induktion

- $\perp$ wird jetzt mit einbezogen
- Um zu zeigen, dass eine Eigenschaft *P(n)* gilt, muss bewiesen werden, dass:
  - P($\perp$) gilt,
  - P(Zero) gilt und wenn
  - *P(n)* gilt, dass in dem Fall auch *P(Succ n)* gültig ist.



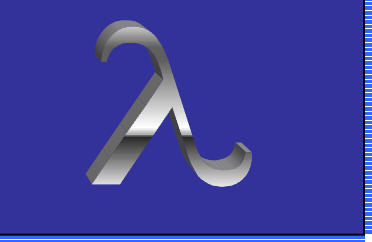
Programmsynthese

- Umkehrung der Induktion nutzen, um Funktionsdefinitionen zu synthetisieren
- Beispiel Subtraktion:

```
(m + n) - n = m   ∀ n,m
```

- Ersetzen von n durch Zero:

```
(m + Zero) - Zero = m
≡ {1. Gleichung für (+)}
m - Zero = m
```



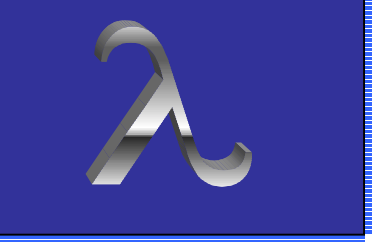
Programmsynthese

- Ersetzen von n durch Succ n:

```
(m + Succ n) - Succ n = m
≡ {2. Gleichung für (+)}
Succ(m + n) - Succ n = m
≡ {Hypothese (m + n) - n = m}
Succ(m + n) - Succ n = (m + n) - n
```

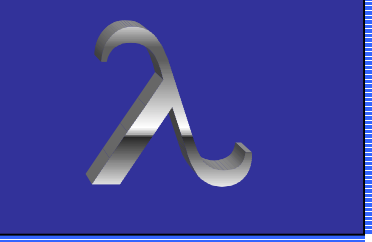
- Ergebnis:

```
m - Zero = m

Succ m - Succ n = m - n
```



Grundlagen Fold

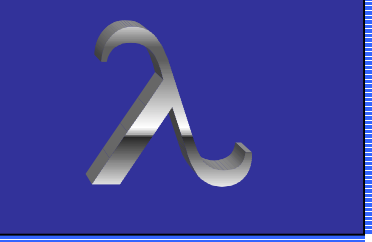
- Mächtige Funktion
- „Zusammenfalten“ von Funktionen
- Vorteil: Definitionen werden kürzer, da nur eine Gleichung geschrieben werden muss statt zwei
- Oft effizienter als direkte rekursive Definition
- Es ist möglich, generelle Eigenschaften von *foldn* zu prüfen für den Beweis von spezifischen Fällen zu nutzen
- Interessant v.a. für Listen und Bäume



Grundlagen Fold


```
foldn :: (α → α) → α → Nat → α
foldn h c Zero = c
foldn h c (Succ n) = h(foldn h c n)
```

```
foldn (+ 6) 5 4 =
  4.  6 +
  3.  6 +
  2.  6 +
  1.  6 +
  0.  5          = 29
```



Fusion

- Gehört in den Bereich der Fold-Funktion
- Fusion = Verschmelzung
- Fusionssatz:

```
f . foldn g a = foldn h b
```

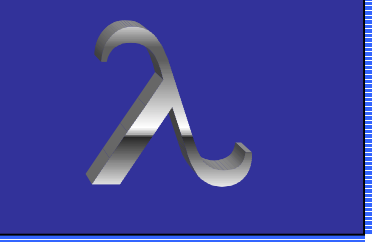
- Gilt für die enthaltenen Variablen, wenn *f* bestimmte Bedingungen erfüllt
- Berechnung von *f* verschmilzt mit der Berechnung von *foldn g a*
- Erhebliche Effizienzsteigerung möglich

Grundlagen Typklassen

- 3 Arten von Integer (positive, Null und negative)

```
data Integer = Neg Positive |
     Zero | Pos Positive

data Positive = One | Succ
     Positive
```

- Approximationen von reellen Zahlen als Sequenz von Dezimalzahlen
- Definition der Arithmetik durch symbolische Rechenwege



Grundlagen Typklassen

- Haskell kennt u.a. folgende Datentypen:
  - *Int*
  - *Integer*
  - *Float*
  - *Double*
  - *Rational*
  - *Complex*
  - *...*

Grundlagen Typklassen

- *Nat* existiert in Haskell nicht (selbst deklariert), aber...
- weiterhin Nutzung von Pattern-Matching durch Rekursion:

```
fak :: Integer → Integer

fak 0 = 1

fak(n + 1) = (n + 1) * fak n
```

- keine partiellen Zahlen in der Arithmetik

Numerische Typklassen

- Alle Zahlentypen in Haskell sind Instanzen der Typklasse *Num*
- Nutzung der gleichen Symbole +, *, ..., jedoch mit unterschiedlicher Bedeutung

```
class (Eq α, Show α) => Num α where
(+),(-),(*) :: α → α → α
negate :: α → α
fromInteger :: Integer → α
x - y = x + negate y
```

Rationale Zahlen

- Integer-Paar (x, y) repräsentiert den Bruch x/y
  - (1, 7), (3, 21), (168, 1176) $\rightarrow$ 1/7
  - Wohldefinierte Werte, wenn $y \neq 0$
- Unendlich viele Möglichkeiten der Darstellung $\rightarrow$ Kanonische Form

  Repräsentation durch ein Paar, für das gilt:
  - $y > 0$
  - $gcd(x, y) = 1$
- *Rational* als Datentyp in Haskell:

```
newtype Rational = Rat (Integer, Integer)
```

Rationale Zahlen

- Erzeugung durch die Funktion *mkRat*:

```
mkRat          :: (Integer, Integer) → Rational
mkRat(x,y)     = Rat((div u d), (div v d))
                       where u = x*(signum y)
                             v = abs y
                             d = ggt u v
```

- Wird benutzt um kanonische Form zu sichern

Rationale Zahlen

- Vergleichsoperationen selbst definieren
  - Paare: (2, 1) < (3, 2), aber rationale Zahlen: 2 > 3/2

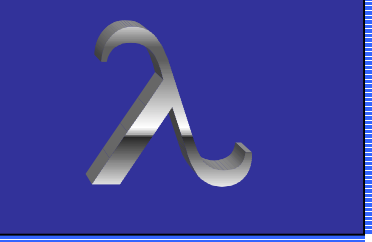
```
instance Eq Rational where
        Rat(x, y) == Rat(u, v) = (x * v) == (v * y)

instance Ord Rational where
        Rat(x, y) < Rat(u, v) = (x * v) < (v * y)
        Rat(x, y) > Rat(u, v) = (x * v) > (v * y)
```

- Ausgabe von rationalen Zahlen

```
showRat(Rat (x,y)) = if y == 1 then show x
else show x ++ "/" ++ show y
```

Binäre Suche

- Lineare Suche ineffizient
- Prinzip der binären Suche ist bekannt:
  - x wird in einem begrenzten Wertebereich gesucht
  - Mittelwert der Bereichsgrenzen bestimmen
  - ist x kleiner, dann vordere Hälfte durchsuchen, sonst hintere
- Anzahl der Schritte verhält sich proportional zu $log_2\ (abs\ x)$



Binäre Suche

- Beispiel Wurzeln:

```
eps :: Float
eps = 0.0001

sqrt :: Float → Float
sqrt x = until done improve x
      where done y = (abs(y * y - x) < eps)
            improve y = (y + x/y)/2
```

Church-Zahlen

- Alonzo Church, 1940-er Jahre
- praktischer Nutzen ?
- Datentypen sind überflüssig, ausschließlich Nutzung von Funktionen
- Beispiel: Bool

```
true, false :: α → α → α
true x y = x
false x y = y

type Cbool α = α → α → α
```

Haben Sie...

- Fragen?
- Anmerkungen?
- Sonstiges?

Vielen Dank

für die

Aufmerksamkeit!