Funktionales Programmieren
(Praktische Informatik 3)

Berthold Hoffmann
Studiengang Informatik
Universität Bremen

Winter 03/04

Vorlesung vom 10.11.2001: Formalisierung und Beweis – Funktionen höherer Ordnung

Inhalt

- Formalisierung und Beweis
  - Vollständige, strukturelle und Fixpunktinduktion
- Verifikation
  - Tut mein Programm, was es soll?
- Funktionen höherer Ordnung
  - Berechnungsmuster (*patterns of computation*)
  - `map` und `filter`: Verallgemeinerte Listenumschreibung
  - `fold`: Primitive Rekursion durch Falten von rechts nach links

Beweisprinzipien

Rekursive Definition, induktiver Beweis

- Definition von Funktionen ist rekursiv
  - Basisfall (z.B. leere Liste `[]`)
  - Rekursion (z.B. nicht-leere Liste `x:xs`)

```
rev :: [a]-> [a]
rev []     = []
rev (x:xs) = rev xs++ [x]
```

  - Reduktion der Eingabe (vom größeren aufs kleinere)
- Beweis durch *Induktion*
  - Schluß vom kleineren aufs größere

Beweis durch *vollständige Induktion*

Zu zeigen:

Für alle natürlichen Zahlen x gilt $P(x)$.

Beweis:

- Induktionsbasis: $P(0)$
- Induktionsschritt: Annahme $P(x)$, zu zeigen $P(x+1)$.

Beweis durch *strukturelle Induktion*

Zu zeigen:

Für alle Listen xs gilt $P(xs)$

Beweis:

- Induktionssbasis: $P([])$
- Induktionssschritt: Annahme $P(xs)$, zu zeigen $P(x : xs)$

Ein einfaches Beispiel

Lemma: $\texttt{len}\,(xs \texttt{ ++ } ys) = \texttt{len}\,xs + \texttt{len}\,ys$

- Induktionsbasis: $xs = []$

$$\begin{aligned}\texttt{len}\,[] + \texttt{len}\,ys &= 0 + \texttt{len}\,ys \\ &= \texttt{len}\,ys \\ &= \texttt{len}\,([] \texttt{ ++ } ys)\end{aligned}$$

- Induktionsschritt:

  Annahme: $\texttt{len}\,xs + \texttt{len}\,ys = \texttt{len}\,(xs \texttt{ ++ } ys)$, dann

$$\begin{aligned}\texttt{len}\,(x : xs) + \texttt{len}\,ys &= 1 + \texttt{len}\,xs + \texttt{len}\,ys \\ &= 1 + \texttt{len}\,(xs \texttt{ ++ } ys) \\ &= \texttt{len}\,(x : xs \texttt{ ++ } ys)\end{aligned}$$

Noch ein Beispiel

Lemma: $\texttt{rev}\,(xs \texttt{ ++ } ys) = \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,xs$

- Induktionsbasis:

$$\begin{array}{lcl} \texttt{rev}\,([] \texttt{ ++ } ys) & = & \texttt{rev}\,ys \\ & = & \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,[] \end{array}$$

- Induktionsschritt:

  Annahme ist $\texttt{rev}\,(xs \texttt{ ++ } ys) = \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,xs$, dann

$$\begin{array}{lcll} \texttt{rev}\,(x : xs \texttt{ ++ } ys) & = & \texttt{rev}\,(xs \texttt{ ++ } ys) \texttt{ ++ } [x] & \text{Def.} \\ & = & (\texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,xs) \texttt{ ++ } [x] & \text{Ind.ann.} \\ & = & \texttt{rev}\,ys \texttt{ ++ } (\texttt{rev}\,xs \texttt{ ++ } [x]) & \texttt{++ assoz.} \\ & = & \texttt{rev}\,ys \texttt{ ++ } \texttt{rev}\,(x : xs) & \end{array}$$

Fixpunktinduktion

- Gegeben: rekursive Definition

  $f\,x = E \quad E$ enthält rekursive Aufrufe $ft_1 \ldots ft_n$

- Zu zeigen: Für alle Eingaben x gilt $P(fx)$
- Verankerung: nichtrekursive Gleichungen von f.
- Annahme:$P(f\,t_i)$ für alle t_i kleiner x
  Zu zeigen: $P(E)$.
  - d.h. ein Rekursionsschritt erhält P
  - Ein Fall für jede rekursive Gleichung.

Beispiel: Quicksort

```
qsort :: [Int]-> [Int]
qsort [] = []
qsort (x:xs) = (qsort l) ++ [x] ++ (qsort r)
     where l = [ y | y <- xs, y<= x ]
           r = [ y | y <- xs, y > x ]
```

Lemma: $\forall l \in$ `[Int]` $: \mathrm{Perm}(l, \texttt{qsort}\, l) \wedge \mathrm{Sorted}(\texttt{qsort}\, l)$

wobei

$\mathrm{Perm}([x_1, \ldots, x_n], [x_{i_1}, \ldots, x_{i_n}]) := \{i_1, \ldots, i_n\} = \{1, \ldots, n\}$

$\mathrm{Sorted}([x_1, \ldots, x_n]) := \forall 1 \leq i < n \bullet x_i \leq x_{i+1}$

- Induktionsbasis:
  $\mathrm{Perm}([], []) \wedge \mathrm{Sorted}([])$ ist offensichtlich
- Induktionsschritt (Perm):
  Annahme: $\mathrm{Perm}(l, \texttt{qsort}\ l) \wedge \mathrm{Perm}(r, \texttt{qsort}\ r)$
  - $\mathrm{Perm}(xs, l \texttt{ ++ } [x] \texttt{ ++ } r)$ folgt aus der Definition von l und r
  - $\mathrm{Perm}(l \texttt{ ++ } [x] \texttt{ ++ } r, \texttt{qsort}\ l \texttt{ ++ } [x] \texttt{ ++ } \texttt{qsort}\ r)$ nach Induktionsannahme und Verträglichkeit von Perm mit ++
    - $\forall l_1, l_2, l_1', l_2' \in \texttt{[Int]} \bullet$
      $\mathrm{Perm}(l_1, l_1') \wedge \mathrm{Perm}(l_2, l_2') \Rightarrow \mathrm{Perm}(l_1 \texttt{ ++ } l_2, l_1' \texttt{ ++ } l_2')$
  - $\mathrm{Perm}(xs, \texttt{qsort}\ l \texttt{ ++ } [x] \texttt{ ++ } \texttt{qsort}\ r)$ wegen Transitivität von Perm
    - $\forall l_1, l_2, l_3 \in \texttt{[Int]} \bullet \mathrm{Perm}(l_1, l_2) \wedge \mathrm{Perm}(l_2, l_3) \Rightarrow \mathrm{Perm}(l_1, l_3)$

- Induktionsschritt (Sorted):
  Annahme: $\mathrm{Sorted}(\texttt{qsort}\ l) \wedge \mathrm{Sorted}(\texttt{qsort}\ r)$

  - $\forall y \in l : y \leq x$ (nach Definition von l)
  - $\forall y \in \texttt{qsort}\ l : y \leq x$ (wegen $\mathrm{Perm}(l, \texttt{qsort}\ l)$)
  - Analog kann für r gezeigt werden: $\forall z \in \texttt{qsort}\ r : x < z$
  - Also gilt: $\forall y \in \texttt{qsort}\ l, z \in \texttt{qsort}\ r : y \leq x < z$
  - Dann folgt $\mathrm{Sorted}(\texttt{qsort}\ l \texttt{ ++ } [x] \texttt{ ++ } \texttt{qsort}\ r)$
    aus der Induktionsannahme (. . .)

Funktionen Höherer Ordnung I

Berechnungsmuster

- Funktion auf alle Elemente einer Liste anwenden
  - `toLower`, `move`, ...
- Elemente aus einer Liste herausfiltern
  - `books`, `returnLoan`, ...
- Primitive Rekursion
  - `++`, `length`, `concat`, ...
- Listen zerlegen
  - `take`, `drop`
- Sonstige
  - `qsort`

Funktionen Höherer Ordnung

- Grundprinzip des funktionalen Programmierens
- Funktionen sind Werte erster Klasse
- Funktionen als Argumente: allgemeinere Berechungsmuster
- Höhere Wiederverwendbarkeit
- Größere Abstraktion

Funktionen als Argumente

- Funktion auf alle Elemente anwenden: `map`
- Signatur:

```
map :: (a-> b)-> [a]-> [b]
```

- Definition

```
map f xs = [ f x | x <- xs ]
```

- oder -

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

- Elemente filtern: `filter`
- Signatur:

```
filter :: (a-> Bool)-> [a]-> [a]
```

- Definition

```
filter p xs = [ x | x <- xs, p x ]
```

- oder -

```
filter p []    = []
filter p (x:xs)
  | p x        = x:(filter p xs)
  | otherwise = filter p xs
```

Skandal: Listenumschreibung entlarvt!

$$[Tx|x \leftarrow l, Px] \equiv \texttt{map t } (\texttt{filter p } l)$$
$$\texttt{where t } x = Tx$$
$$\texttt{p } x = Px$$

Beispiel: Kehrwerte aller ungeraden Zahlen von 1 bis 100

```
[ 1 / x | x <- [1..100], (mod x 2) == 1 ]
```

```
map t (filter p [1..100])
      where t x = 1 / x
            p x = (mod x 2) == 1
```

Primitive Rekursion

- Primitive Rekursion:
  - Basisfall (leere Liste): ein neutraler Wert
  - Rekursionsfall: Verknüpfung von Listenkopf und Rekursionswert
- Signatur

```
foldr :: (a-> b-> b)-> b-> [a]-> b
```

- Definition

```
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

- Beispiel: Summieren von Listenelementen.

```
sum :: [Int]-> Int
sum xs = foldr (+) 0 xs

sum [3,12] = 3 + sum [12]
           = 3+ 12+ sum []
           = 3+ 12+ 0= 15
```

- Beispiel: Summieren von Listenelementen.

```
sum :: [Int]-> Int
sum xs = foldr (+) 0 xs

sum [3,12] = 3 + sum [12]
           = 3+ 12+ sum []
           = 3+ 12+ 0= 15
```

- Beispiel: Flachklopfen von Listen.

```
concat :: [[a]]-> [a]
concat xs = foldr (++) [] xs

concat [l1,l2,l3,l4] = l1++ l2++ l3++ l4++ []
```

Listen zerlegen

- `take, drop`: n Elemente vom Anfang nehmen/weglassen
- Längster Präfix für den Prädikat gilt

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []                =  []
takeWhile p (x:xs)
            | p x             =  x : takeWhile p xs
            | otherwise       =  []
```

- Restliste des längsten Präfix

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

- Es gilt: `takeWhile p xs ++ dropWhile p xs == xs`

- Kombination von `takeWhile` und `dropWhile`

```
span       :: (a -> Bool) -> [a] -> ([a],[a])
span p xs = (takeWhile p xs, dropWhile p xs)
```

- Ordnungserhaltendes Einfügen:

```
ins :: Int-> [Int]-> [Int]
ins x xs = lessx ++ [x] ++ grteqx
    where (lessx, grteqx) = span less xs
          less z = z < x
```

- Damit sortieren durch Einfügen:

```
isort :: [Int]-> [Int]
isort xs = foldr ins [] xs
```

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?

Beliebiges Sortieren

- Wieso eigentlich immer aufsteigend?
- Ordnung als Argument `ord`
  - Totale Ordnung: transitiv, antisymmetrisch, reflexiv, total
  - Insbesondere: x `ord` $y \wedge y$ `ord` $x \Rightarrow x = y$

```
qsortBy :: (a-> a-> Bool)-> [a]-> [a]
qsortBy ord [] = []
qsortBy ord (x:xs) =
      qsortBy ord [y| y<-xs, ord y x] ++ [x] ++
      qsortBy ord [y| y<-xs, not (ord y x)]
```

Zusammenfassung

- Verifikation und Beweis
  - Beweis durch strukturelle und Fixpunktinduktion
  - Verifikation eines nichttrivialen Algorithmus
- Funktionen höherer Ordnung
  - Funktionen als gleichberechtigte Werte
  - Erlaubt Verallgemeinerungen
  - Erhöht Flexibilität und Wiederverwendbarkeit
  - Beispiele: `map`, `filter`, `foldr`
  - Sortieren nach beliebiger Ordnung