Vorlesung vom 12.11.2001: Formalisierung und Beweis Funktionen höherer Ordnung

Inhalt

- Formalisierung und Beweis
  - Vollständige, strukturelle und Fixpunktinduktion
- Verifikation
  - Tut mein Programm, was es soll?
- Fallbeispiel: Verifikation von Mergesort
- Funktionen höherer Ordnung
  - Berechnungsmuster (*patterns of computation*)
  - `map` und `filter`: Verallgemeinerte Listenkomprehension
  - `fold`: Primitive Rekursion

Rekursive Definition, induktiver Beweis

- Definition ist rekursiv
  - Basisfall (leere Liste)
  - Rekursion (`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: $P(0)$
- Induktionsssschritt: 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: $\texttt{len}\,(xs \mathbin{\texttt{++}} ys) = \texttt{len}\,xs + \texttt{len}\,ys$

- Induktionsbasis: $xs = []$

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

- Induktionsschritt:

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

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

Noch ein Beispiel

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

- Induktionsbasis:

$$\begin{aligned} \texttt{rev}\ ([] \mathbin{+\!+} ys) &= \texttt{rev}\ ys \\ &= \texttt{rev}\ ys \mathbin{+\!+} \texttt{rev}\ [] \end{aligned}$$

- Induktionsschritt:

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

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

Fixpunktinduktion

- Gegeben: rekursive Definition

$$f x = E \quad E \text{ enthält rekursiven Aufruf } f t$$

- Zu zeigen: Für alle Listen x gilt $P(f x)$
- Beweis: Annahme:$P(f t)$, zu zeigen: $P(E)$.
  - d.h. ein Rekursionsschritt erhält P
  - Ein Fall für jede rekursive Gleichung.
  - Induktionsverankerung: nichtrekursive Gleichungen.

Berechnungsmuster

- Listenkomprehension I: Funktion auf alle Elemente anwenden
  - `toLower`, `move`, . . .
- Listenkomprehension II: Elemente herausfiltern
  - `books`, `returnLoan`, . . .
- Primitive Rekursion
  - `++`, `length`, `concat`, . . .
- Listen zerlegen
  - `take`, `drop`
- Sonstige
  - `qsort`, `msort`

Funktionen Höherer Ordnung

- Grundprinzip der funktionalen Programmierung
- Funktionen sind gleichberechtigt
  - d.h. Werte wie alle anderen
- 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)
```

Funktionen als Argumente

- 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
```

Primitive Rekursion

- Primitive Rekursion:
  - Basisfall (leere Liste)
  - Rekursionsfall: Kombination aus 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: 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
- 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 der beiden

```
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?
- 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