Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 / 05. Dezember 2023

Funktionen höherer Ordnung II

Thomas Barkowsky

Wintersemester 2023/24

Universität Bremen

Organisatorisches

- E-Klausuren
  - Probeklausur: Do, 07.12.2023 10:00 und 10:45
  - **Klausur**: Mo, 11.03.2024 10:00 und 11:45
  - Wiederholungsklausur: Do, 18.04.2024 10:00
- Übungsblatt 7 veröffentlicht: erstes Gruppenübungsblatt
  - Gruppenübungsblätter: doppelte Punktzahl, 14 Tage Bearbeitungszeit
  - Gruppen à 3 Studierende organisieren

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- Testen und Qualitätssicherung
- I/O, Aktionen und Zustände
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung...

- η („eta“) - Kontraktion und punktfreie Notation
- Der Funktionsanwendungsoperator „`$`“
- Funktoren
- Faltung algebraischer Datentypen

η-Kontraktion und punktfreie Notation

Noch einmal: Funktionskomposition

- Beispiel (aus Vorlesung 4):

```
odd n = not (even n)
```

```
twice f x = f (f x)
```

```
odd = not . even
```

```
twice f = f . f
```

- fehlt da nicht etwas?
- warum nicht so:

```
odd n = (not . even) n
```

```
> :t odd
odd :: Int -> Bool
```

- Currying!

η-Kontraktion und punktfreie Notation

- Wir haben häufig folgende Situation
  - ein Ausdruck (Funktion) `E :: a -> b`
  - eine Variable `x :: a`
  - `x` kommt in `E` nicht vor
- Dann gilt: `\x -> E x` = `E`
  - bzw. in Funktionsdefinition: `f x = E x` = `f = E`
  - sog. “punktfreie Notation”: Die Elemente (“Punkte”) des Definitionsbereichs der Funktion tauchen in der Funktionsdefinition nicht auf

Punktfreie Notation: weitere Beispiele

```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

```
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p
```

```
sum :: (Num a) => [a] -> a
sum = foldl (+) 0
```

Der Funktionsanwendungsoperator "`$`"

Der Funktionsanwendungsoperator "$"

- Definition:
```
($) :: (a -> b) -> a -> b
f $ x = f x
```
- Funktionsanwendung.
  Was ist der Unterschied zwischen `f $ x` und `f x` ?
- `f x` : höchste Priorität vs. `f $ x` : niedrigste Priorität

- Vorteil 1: Einsparung von Klammern:

```
sum (map sqrt [1..130])
```
vs.
```
sum $ map sqrt [1..130]
```

```
sqrt (3 + 4 + 9)
```
vs.
```
sqrt $ 3 + 4 + 9
```

Der Funktionsanwendungsoperator "$"

- Vorteil 2: Der Funktionsanwendungsoperator generiert selbst Funktionen
- Beispiel:

```
> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

  (Mapping von Funktionsanwendung auf eine Liste von Funktionen)
- Typ von `($ 3)`:

```
> :t ($ 3)
($ 3) :: Num a => (a -> b) -> b
```

Funktoren

`map` – Für Listen

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

(standard prelude)

- Warum eigentlich nur für Listen?
- Erweiterung der Idee des *mapping* auf andere Datentypen
- z.B. Bäume:
  - Abbildung eines Baums aus Strings in einen Baum, der die Längen der Strings enthält
- In Haskell: Typklasse `Functor`, Funktion `fmap`...

Die Typklasse `Functor`

- Umfasst alle Datentypen, für die es *mapping* gibt
- Definition (*standard prelude*):

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- Erfordert Definition einer einzigen Funktion: `fmap`
  - `fmap` erhält eine Funktion des Typs `a -> b`, sowie eine Struktur des Typs `f a` mit Elementen des Typs `a` und liefert eine Struktur des Typs `f b` mit Elementen des Typs `b`
  - keine Default-Implementierung für `fmap`
  - `f` ist Typkonstruktor mit **einem** Parameter

`fmap` vs. `map`

- Vergleich der Signaturen von `fmap` und `map`
- `fmap`:
  ```
  map  :: (a -> b) -> [a] -> [b]
  ```
- `map`:
  ```
  fmap :: (a -> b) -> f a -> f b
  ```
- `map` ist `fmap`, beschränkt auf Listen
- Datentyp "`[]`" ist Instanz der Typklasse `Functor` mit folgender Definition:
  ```
  instance Functor [] where
      fmap = map
  ```
- "`[]`" ist Typkonstruktor, der in Verbindung mit gegebenen Datentyp Listen verschiedener Typen erzeugt
- `fmap` und `map` haben gleiche Wirkung bei Anwendung auf Listen

Weitere `Functor`-Instanz: `Maybe`

- Datentyp `Maybe` ‘verpackt’ Werte (oder enthält `Nothing`)
- Verwendung als `Functor` liegt nahe…
- Definition:

```
instance Functor Maybe where
     fmap f (Just x)  = Just (f x)
     fmap f Nothing   = Nothing
```

- Beispiele:

```
> fmap (*2) (Just 10)
Just 20
> fmap (*2) Nothing
Nothing
> fmap (++ " -- Thanks!") (Just "Welcome on board!")
Just "Welcome on board! -- Thanks!"
> fmap (++ " -- Thanks!") Nothing
Nothing
```

Eigene Datentypen als Funktoren

- Beispiel Baum:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

- `Funktor` Deklaration:

```
instance Functor Tree where
    fmap g (Leaf x)   = Leaf (g x)
    fmap g (Node l r) = Node (fmap g l)
                             (fmap g r)
```

- Beispiele:

```
> fmap length (Leaf "abc")
Leaf 3
> fmap even (Node (Leaf 1) (Leaf 2))
Node (Leaf False) (Leaf True)
```

`fmap`: Bedingungen

- `fmap` muss die folgenden zwei Bedingungen erfüllen:

1) `fmap id = id` (Wahrung der Identität)

```
> fmap id (Node (Leaf 1) (Leaf 2))
Node (Leaf 1) (Leaf 2)
```

2) `fmap (g . h) = fmap g . fmap h`
(Wahrung der Funktionskomposition)

```
> (fmap even . fmap length) (Just "twelve")
Just True
> fmap (even . length) (Just "twelve")
Just True
```

`fmap`: Infix Notation

- `fmap` gelegentlich auch in Infix-Notation (Operator-Schreibweise):

```
> (^2) `fmap` [1,2,3]
[1,4,9]
```

- oder kürzer ("`<$>`" als Alias):

```
> (^2) <$> [1,2,3]
[1,4,9]
```

Warum Funktoren?

- `fmap` für alle Typen der Klasse `Functor` verwendbar
  - eine Funktion für alle passenden Strukturen
- Definition generischer Funktionen für alle Funktoren
  - durch Verwendung von `fmap`
  - Beispiel:

```
inc :: Functor f => f Int -> f Int
inc = fmap (+1)
```

```
> inc (Just 1)
Just 2
> inc [2,3,4]
[3,4,5]
> inc (Node (Leaf 4) (Leaf 6))
Node (Leaf 5) (Leaf 7)
```

Faltung algebraischer Datentypen

Faltung jenseits von Listen

- Verallgemeinerung von `map` zu `fmap`: Geht Ähnliches auch für Faltungen?
- Ja! – Für jeden algebraischen Datentyp gibt es (genau) ein `foldr`
- Zur Erinnerung: `foldr` ist die kanonische strukturell rekursive Funktion
  - siehe Vorlesung 4
  - für Listen:

```
f []     = v
f (x:xs) = x # f xs
```

- Listen besitzen zwei Konstruktoren:

```
data [a] = [] | a : [a]
```

- Definition von `foldr`...

`foldr` für Listen

- `foldr` für Listen besitzt für jeden Konstruktor ein Funktionsargument und eine Gleichung:

Funktionsargument für ‘:’ Funktionsargument für ‘[]’

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Gleichung für ‘[]’
Gleichung für ‘:’

- Verallgemeinerung von `foldr`...

`foldr` verallgemeinert

- Für einen gegebenen allgebraischen Datentyp `T` benötigen wir
  - für jeden Konstruktor `C` ein Funktionsargument f_C
  - für jeden Konstruktur `C` eine Gleichung
  - eine freie Typvariable `b` für das jeweilige Resultat
- Beispiel:

```
data IL = Cons Int IL | Err String | Mt

foldIL :: (Int -> b -> b) -> (String -> b) -> b -> IL -> b
foldIL f e a (Cons i il) = f i (foldIL f e a il)
foldIL f e a (Err str)   = e str
foldIL f e a Mt          = a
```

Faltung bekannter Datentypen

- `Bool`: Fallunterscheidung

```
data Bool = False | True

foldBool :: b -> b -> Bool -> b
foldBool a1 a2 False = a1
foldBool a1 a2 True  = a2
```

Faltung bekannter Datentypen

- `Maybe`: Auswertung

```
data Maybe a = Nothing | Just a

foldMaybe :: b -> (a -> b) -> Maybe a -> b
foldMaybe b f Nothing  = b
foldMaybe b f (Just a) = f a
```

- Vordefiniert als Funktion `maybe`:
  - `maybe` erhält einen Default-Wert, eine Funktion und einen `Maybe`-Wert. Ist der `Maybe`-Wert `Nothing`, gibt die Funktion den Default-Wert zurück, sonst wird die Funktion auf den Wert innerhalb des `Just` angewendet

Faltung für Binärbäume

- Definition `Tree` (Label an den Knoten):

```
data Tree a = Mt | Node a (Tree a) (Tree a)
```

- Definition foldT:

```
foldT :: (a -> b -> b -> b)-> b -> Tree a -> b
foldT f e Mt              = e
foldT f e (Node a l r) =  f a (foldT f e l) (foldT f e r)
```

- Beispiel:

```
> foldT (\a b c -> a+b+c) 0 (Node 1 (Node 5 Mt Mt) (Node 2 Mt Mt))
8
```

Die Typklasse `Foldable`

- schränkt die Signatur von `foldr` ein
- `import Data.Foldable`
- Deklaration:

```
class Foldable t where
    ...
    foldr :: (a -> b -> b) -> b -> t a -> b
    ...
```

- Für Datentypen mit zwei Konstruktoren
- Signatur der übergebenen Funktion an Faltung von Listen orientiert
  - Kombination von je zwei Werten

Binärbaum als Instanz von Foldable

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
instance Foldable Tree where
    foldr f v (Leaf x)   = f x v
    foldr f v (Node l r) = foldr f (foldr f v r) l
```

```
> foldr (*) 1 (Node (Node (Leaf 2) (Leaf 4)) (Leaf 6))
48
```

Zusammenfassung

- η-Kontraktion und punktfreie Notation
- Funktionsanwendungsoperator „`$`“
- Funktoren: Verallgemeinerung von `map` durch `fmap`
- Beispiele für Funktor-Instanzen
- Faltung algebraischer Datentypen: Verallgemeinerung von `foldr`
- Beispiele

nächstes Mal...

- Abstrakte Datentypen (ADT): Definition
- Module in Haskell
  - importieren und exportieren von Modulen
- Abstrakte Datentypen in Haskell
  - Beispiele
  - Design von ADTs