Praktische Informatik 3: Funktionale Programmierung

Vorlesung 8 / 12. Dezember 2023

Module und Abstrakte Datentypen

Thomas Barkowsky

Wintersemester 2023/24

Universität Bremen

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...

- Noch einmal: der Binärbaum in der Typklasse `foldable`
  - (aus Vorlesung 7)
- Abstrakte Datentypen: Definition
- Module in Haskell
  - importieren und exportieren von Modulen
- Abstrakte Datentypen in Haskell
  - Beispiele
  - Design von ADTs

Binärbaum als Instanz von `Foldable`

- `Foldable`:

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

- Signatur der übergebenen Funktion an Faltung von Listen orientiert
  - Kombination von je zwei Werten

```
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
```

Frage:

```
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
```

- Warum wird in dieser Definition von `foldr` der linke Teilbaum überhaupt gefaltet?
- Beobachtungen:
  - rekursive Bearbeitung von `foldr` erforderlich, wg. Baumstruktur
  - Funktion `f` bleibt immer gleich
  - Basiswert `v` wird im rekursiven Aufruf ersetzt durch (Ergebnis der) Faltung eines Teilbaums
  - vgl. Signatur von `foldr`:

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

Auswertung am Beispiel

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

```
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
```

Frage:

```
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
```

- Spielt es eine Rolle, dass linke und rechte Teilbäume in der Rekursion vertauscht sind?
- Nein! – oder doch?
- Kein Unterschied bei kommutativer Faltungsfunktion
  - selbst ausprobieren
- Jedoch …

```
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 (-) 0 (Node (Leaf 3) (Leaf 4))
==>>  foldr (-) (foldr (-) 0 (Leaf 4)) (Leaf 3)
==>>  foldr (-)         (4 - 0)         (Leaf 3)
==>>                  3 - 4                          ==>>    -1
```

- vs.

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

```
>     foldr (-) 0 (Node (Leaf 3) (Leaf 4))
==>>  foldr (-) (foldr (-) 0 (Leaf 3)) (Leaf 4)
==>>  foldr (-)         (3 - 0)         (Leaf 4)
==>>                  4 - 3                          ==>>     1
```

Abstrakte Datentypen (ADTs)

Abstrakte Datentypen (ADT)

- Nicht verwechseln mit *algebraischen* Datentypen!
- ADT:
  - mathematische Beschreibung von Datentypen
  - Definition durch Verhalten (Semantik)
  - Beschreibung, **was** der Datentyp tut, nicht aber **wie** (Implementierung)
    - Wertebereich
    - mögliche Operationen
    - Verhalten der Operationen
  - ADT ist **gekapselt** (eingeschränkte Sichtbarkeit)
  - Interne Realisierung des ADT kann jederzeit geändert werden
  - in Haskell: **Module**!

ADT: Definition

- Ein ADT besteht aus
  - einem (oder mehreren) Datentypen, sowie
  - Operationen darauf
- Eigenschaften:
  - Werte des Datentypen werden ausschließlich über bereitgestellte Operationen **erzeugt** und **modifiziert**
  - Eigenschaften von Werten werden nur über bereitgestellte Operationen **beobachtet**
  - **Invarianten** des ADT sind garantiert durch bereitgestellte Operationen
    - keine Beeinträchtigung der Invarianten von außen

Module in Haskell

Gründe für Modularisierung

- Übersichtlichkeit & Lesbarkeit
  - Thematische Zuordnung von Code in Modulen
- Separate Kompilierung: technische Handhabbarkeit
  - Änderungen an der Implementierung lokal
- Verkapselung: konzeptuelle Handhabbarkeit
  - Arbeit im Team, Fehlersuche, Sicherheit
- Wiederverwendbarkeit von Komponenten
  - Import von Bibliotheken mit spezifischer Funktionalität

Das Haskell Modulsystem

- Module in Haskell bestehen jeweils aus
  - einer Reihe von Typ- und Funktionsdefinitionen und
  - einem definierten Interface, das beschreibt, welche der Typ- und Funktionsdefinitionen an andere Module exportiert werden
- Module importieren Typ- und Funktionsdefinitionen von anderen Modulen
- Möglichkeit der Hierarchisierung von Modulen

Haskell Module: Syntax und Konventionen

- Ein Modul pro Datei
  - nicht in Sprachdefinition, aber Anforderung von GHC
- Modulheader
  - Schlüsselwort `module` + Name des Moduls + `where`
  - Alle Definition beginnen in Spalte unter `module`
  - Beispiel:

```
module Ant where

data Ants  = ...
anteater x = ...
```

- Modulname = Dateiname!
  - keine formale Anforderung, aber sinnvolle Konvention
  - Beispiel: Modul `Foo` in Datei `Foo.hs`

Importieren von Modulen

- Module **importieren** andere Module:
- Die sichtbaren Definitionen des importierten Moduls stehen in dem importierenden Modul zur Verfügung
- Nicht transitiv:
  (Definitionen von `Ant` nicht sichtbar in `Cow`)

```
module Bee where

import Ant

beekeper = ...
```

```
module Cow where

import Bee

...
```

Was wird aus einem Modul exportiert?

- *Default*: alles was in einem Modul definiert wird
- Einschränkungen? – z.B. Hilfsfunktionen
- Erweiterungen? – z.B. Export von aus einem anderen Modul importierten Funktionen
- Lösung: Explizite Angabe der zu exportierenden Definitionen (Funktionen und Datentypen)...

Was wird aus einem Modul exportiert?

- Lösung: Explizite Angabe der zu exportierenden Definitionen (Funktionen und Datentypen)...

```
module Bee ( beekeper, Ants(..), anteater ) where ...
```

- `( . . )` = Export des Datentyps **inklusive Konstruktoren** (optional)
  - siehe später: ADTs in Haskell

- auch möglich:

```
module Bee ( beekeper, module Ant ) where ...
```

(alle Definitionen aus dem Modul `Ant` werden exportiert)

Das `Main` Modul

- Das Top-Level Modul in jedem Modul-System sollte `Main` heißen
  - inklusive der Definition der Funktion `main`
- `main` wird ausgeführt, wenn das kompilierte Programm gestartet wird
  - unwichtig bei interaktiver Arbeit mit GHCi
- Module ohne *Header* werden per Default behandelt wie

```
module Main (main) where ...
```

Kontrolle über `import`

- Explizite Angabe, was aus einem Modul importiert wird, z.B.

```
import Ant ( Ants(..) )
```

  (ausschließlich `Ants` inkl. Konstruktoren wird importiert)

- Explizites Ausschließen von Definitionen beim Import, z.B.

```
import Ant hiding ( anteater )
```

- *Standard Prelude* wird automatisch in jedem Modul importiert
  - auch hier:

```
import Prelude hiding ( ... )
```

`import qualified`

- Was tun bei Namenskonflikten (gleicher Name in aktuellem und in importiertem Modul)?
  - Beispiel: `bear` sowohl in aktuellem Modul als auch im importierten Modul `Ant`
- Lösung: Nutzung **qualifizierter** Namen (`bear` vs. `Ant.bear`)
- Beim `qualified` Import (`import qualified Ant`) **ausschließlich** qualifizierte Namen verwendbar
- Verwendung lokaler Namen beim Import: `import Insect as Ant` (importiertes Modul `Insect` erhält lokalen Namen `Ant`)

ADTs in Haskell

ADTs in Haskell: Beispiel

- Nehmen wir an, wir wollen Werte (`Integer`) benannten Speicherplätzen (`Var`) zuordnen: Datentyp `Store`
- Verschiedene Realisierungsmöglichkeiten, z.B.
  `[(Integer, Var)]` oder `(Var -> Integer)`
- Wir wollen
  - einen Store erzeugen:
    ```
    initial :: Store
    ```
  - Werte aus Speicherplätzen auslesen:
    ```
    value   :: Store -> Var -> Integer
    ```
  - Speicherplätzen Werte zuweisen:
    ```
    update  :: Store -> Var -> Integer -> Store
    ```

Verwendung von `Store`

- Je nach Realisierung von `Store` können wir mit dem Datentyp unterschiedlich umgehen
  - z.B. Listenmanipulation, Funktionskomposition, ...
- Wie können wir ungewollte Verwendungsweisen verhindern?
- Lösung:
  - Zugriffsbeschränkung auf die Funktionen `initial`, `value`, und `update` **und**
  - Definition von `Store` verstecken
- Verwendung von Modulen mit Exportbeschränkung = **ADT**

`Store` als ADT

- Signaturen von `initial`, `value` **und** `update` dienen als **Interface** für `Store`
- = **Signatur des ADT**
- Signatur ist unveränderlich, Implementierung kann sich ändern
- Somit alle Vorteile von ADTs
- Realisierung in Haskell mit Store durch `[(Integer, Var)]` ...

`Store` als ADT in Haskell

- Modul *Header*: Export des Datentyps `Store` **ohne** Konstruktoren:

```
module Store
    ( Store,
      initial,      -- Store
      value,        -- Store -> Var -> Integer
      update        -- Store -> Var -> Integer -> Store
     ) where
```

- Nutzung von `Store` ausschließlich durch `initial`, `value` und `update`
- Nützliche Konvention: Typen der exportierten Funktionen als Kommentare im Header

Definitionen in `Store`

```
type Var = Char
data Store = Store [(Integer,Var)] deriving (Eq, Show)

initial :: Store
initial = Store []

value  :: Store -> Var -> Integer
value (Store []) v          = 0
value (Store ((n,w):sto)) v
  | v==w                    = n
  | otherwise               = value (Store sto) v

update  :: Store -> Var -> Integer -> Store
update (Store sto) v n = Store ((n,v):sto)
```

Alternativ: `Store` als Funktion `(Var -> Integer)`

- Signatur des ADT wie oben
- `Store` als Funktion `Var -> Integer`:

```
type Var = Char
newtype Store = Store (Var -> Integer)
```

- Leerer `Store` bildet jedes `Var` auf `0` ab:

```
initial :: Store
initial = Store (\v -> 0)
```

Alternativ: `Store` als Funktion `(Var -> Integer)`

- Beim Auslesen eines Wertes wird übergebenes `Var` auf die Funktion angewendet:

```
value :: Store -> Var -> Integer
value (Store sto) v = sto v
```

- Für `update` wird eine identische Funktion zurückgeliefert, außer bzgl. des erneuerten Wertes:

```
update  :: Store -> Var -> Integer -> Store
update (Store sto) v n
  = Store (\w -> if v==w then n else sto w)
```

Testen von ADTs

- Kein direkter Zugriff auf Implementierung des ADT, sondern ausschließlich über Interfacefunktionen
- Prüfen der Eigenschaften des ADT, z.B.
  - Auslesen des leeren Stores ergibt `0`
  - Nach `update` liefert `Var` beim Auslesen den erneuerten Wert
  - Bei `update` eines Werts bleiben andere Werte unverändert
  - …
- Zugesicherte Eigenschaften müssen unabhängig von der zugrundeliegenden Implementierung gelten
- Weiteres Beispiel...

Queues als ADT

- FIFO – *first in, first out* (Warteschlange)

```
module Queue
   (Queue,
    emptyQ,          --  Queue a
    isEmptyQ,        --  Queue a -> Bool
    addQ,            --  a -> Queue a -> Queue a
    remQ             --  Queue a -> (a, Queue a)
    ) where
```

Queue ADT mit Listen

```
newtype Queue a = Queue [a]

emptyQ :: Queue a
emptyQ = Queue []

isEmptyQ :: Queue a -> Bool
isEmptyQ (Queue []) = True
isEmptyQ _          = False
```

Queue ADT: Hinzufügen und Entfernen

- Hinzufügen am Ende der Liste, Entfernen am Anfang

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

remQ :: Queue a -> (a, Queue a)
remQ q@(Queue xs)
  | not (isEmptyQ q)    = (head xs , Queue (tail xs))
  | otherwise           = error "remQ"
```

- @: "*as-Pattern*": ermöglicht Zugriff sowohl auf `Queue` im Ganzen (`q`), als auch auf deren Komponenten (`Queue xs`)

`Queue`: Effizienz?

- `addQ` teuer (Anfügen am Listenende), `remQ` billig
- Alternativ umgekehrt:

```
addQ :: a -> Queue a -> Queue a
addQ x (Queue xs) = Queue (x:xs)

remQ :: Queue a -> (a, Queue a)
remQ q@(Queue xs)
  | not (isEmptyQ q)    = (last xs , Queue (init xs))
  | otherwise           = error "remQ"
```

(auch nicht besser!)

Queue: Effizientere Lösung

- Verwendung von zwei Listen
- Anfügen am Anfang von Liste 1 (billig)
- Entfernen: vom Anfang von Liste 2 (billig)
- Wenn Liste 2 leer:
  - Liste 1 umdrehen (teuer) und als Liste 2 verwenden
  - Liste 1 neu mit leerer Liste starten
- Aufwändige Operation nur noch gelegentlich
- Kein Einfluss auf Signatur des ADT: ...

Queue mit zwei Listen

```
data Queue a = Queue [a] [a]

emptyQ :: Queue a
emptyQ = Queue [] []

isEmptyQ :: Queue a -> Bool
isEmptyQ (Queue [] []) = True
isEmptyQ _             = False

addQ :: a -> Queue a -> Queue a
addQ x (Queue xs ys) = Queue xs (x:ys)

remQ :: Queue a -> (a, Queue a)
remQ (Queue (x:xs) ys)     = (x ,Queue xs ys)
remQ (Queue [] ys@(z:zs)) = remQ (Queue (reverse ys) [])
remQ (Queue [] [])         = error "remQ"
```

Design von ADTs – HowTo

- Identifikation und Benennung der Datentypen
- Informelle Beschreibung der Eigenschaften und Funktionen
- Definition der Signatur des ADT. Funktionen:
  - Generierung von Objekten des Typs (leere Queue, Wurzel des Baums, ...)
  - Diskriminatoren für Objekte – worum handelt es sich? (Knoten oder Blatt?, ...)
  - Selektoren – Extraktion von Komponenten (Informationen auslesen, suchen, ...)
  - Objekte transformieren (hinzufügen, umkehren, ...)
  - Objekte kombinieren (vereinigen von Bäumen, Mengen, ...)
  - Objekte auswerten (aufsummieren, Größe, ...)

Zusammenfassung

- ADTs verkapseln Datentypen
- Zugriff nur über Interfaces: Signatur des ADT
- Zugesicherte Invarianten
- Modulares Softwaredesign in Haskell: `import` und `export`
- Fallbeispiele `Store` und `Queue`
- Modifikation der Implementierung ohne Veränderung der Signatur

nächstes Mal...

- I/O, Aktionen und Zustände
  - I/O in funktionalen Sprachen: Wo ist das Problem?
  - Vordefinierte Aktionen und ihre Verwendung in der richtigen Reihenfolge
  - Umgang mit Dateien
  - Zufallszahlen in Haskell