# Praktische Informatik 3: Funktionale Programmierung

Vorlesung 10 / 09. Januar 2024

**Testen und Qualitätssicherung**

Thomas Barkowsky

Wintersemester 2023/24

# Übersicht

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

# heute in dieser Vorlesung...

- Testen in Haskell
- Signaturen und Eigenschaften
- Axiome
- Zufallsbasiertes Testen mit QuickCheck

# Testen in Haskell

# Motivation

- Guter Code sollte **robust** und **korrekt** sein: Qualitätssicherung!
- Viele Features durch Haskell mitgegeben:
  - Typisierung
  - Referentielle Transparenz
  - Polymorphie
  - Modularisierung
- Dadurch:
  - Gute Codequalität
  - gute Refakturierbarkeit und
  - gute Testbarkeit des Codes

# Testen mit Haskell

- Herkömmliches *Unit testing* – bekannt aus den Übungen
- *Property-based testing*
  - Nutzung abstrakter Invarianten
  - typbasiert
  - automatisiert
  - Werkzeug: **Quickcheck**
- Test durch Vergleich mit Referenzimplementierung
  - (ebenfalls eine Art von *property-based testing*)
- Analyse der Codeabdeckung der Tests

- Wie funktioniert eigenschaftsbasiertes Testen?

# Signaturen und Eigenschaften

# noch einmal: Signaturen

- Siehe Vorlesung 8 zu abstrakten Datentypen (ADTs)
- Definition:
  - Die **Signatur eines abstrakten Datentypen** besteht aus dem Typen selbst und den Signaturen der darüber definierten Funktionen
- Generell: Signatur = "Typ" eines Moduls
- Keine direkte Repräsentation in Haskell!
- Aber: Signaturen können genutzt werden, um die **Eigenschaften** der in einem Modul realisierten Funktionen zu beschreiben
- Und: Eigenschaften sind überprüfbar (also **testbar**)
  - (wenn formal repräsentiert)

# Beispiel: Signatur für endliche Abbildung

```
data Map a b
```

```
empty :: Map a b
```

```
lookup :: Eq a => a -> Map a b -> Maybe b
```

```
insert :: Eq a => a -> b -> Map a b -> Map a b
```

```
delete :: Eq a => a -> Map a b -> Map a b
```

# Was leistet die Signatur, was nicht?

- Signatur informiert über:
  - typkorrekte Nutzung des ADTs
  - Anwendbarkeit von Funktionen
  - Reihenfolge von Parametern
- Signatur beschreibt **nicht**:
  - Bedeutung und Verhalten der Funktionen (Semantik), also z.B.
  - Wie werden Daten aus dem ADT ausgelesen?
  - Welches Verhalten hat die Abbildung?
  - etc.

# Signatur und Eigenschaften

- Bedeutung und Verhalten werden durch die **Eigenschaften** beschrieben
- Die Signatur kann genutzt werden um Eigenschaften formal (syntaktisch) zu beschreiben
- Die Formalisierung der Eigenschaften erfolgt mithilfe von Axiomen
- Axiome sind formal untersuchbar / ihre Gültigkeit kann getestet werden
- Axiome **müssen gelten**!

# Eigenschaften endlicher Abbildungen

- Aus einer leeren Abbildung kann nichts ausgelesen werden
- Wenn etwas in die Abbildung geschrieben und an der gleichen Stelle ausgelesen wird, dann erhält man den geschriebenen Wert
- Wenn etwas in die Abbildung geschrieben und an einer anderen Stelle etwas ausgelesen wird, dann hat der Schreibvorgang keinen Einfluss
- Wenn zweimal an der gleichen Stelle in die Abbildung geschrieben wird, dann überschreibt der zweite den ersten Wert
- Schreiben an unterschiedlichen Stellen verhält sich kommutativ

# Formalisierung von Eigenschaften & Prädikate

- ... durch Axiome!

- Definition:
  - **Axiome** sind Prädikate über den Operationen der Signatur

- Prädikate: ...

- Elementare Prädikate
  - Gleichheit: `s == t`
  - Ordnung: `s < t`, etc.
  - selbstdefinierte Prädikate
- Zusammengesetzte Prädikate
  - Negation: `not p`
  - Konjunktion: `p && q`
  - Disjunktion: `p || q`
  - Implikation: `p => q`

# Axiome

# Axiome für `Map`

- Aus einer leeren Abbildung kann nichts ausgelesen werden

```
lookup a (empty :: Map Int String) == Nothing
```

- Wenn etwas in die Abbildung geschrieben und an der gleichen Stelle ausgelesen wird, dann erhält man den geschriebenen Wert

```
lookup a (insert a v (s :: Map Int String)) == Just v
```

```
lookup a (delete a (s :: Map Int String)) == Nothing
```

- Wenn etwas in die Abbildung geschrieben und an einer anderen Stelle etwas ausgelesen wird, dann hat der Schreibvorgang keinen Einfluss

```
a /= b ==> lookup a (insert b v s) == lookup a (s :: Map Int String)
```

```
a /= b ==> lookup a (delete b s) == lookup a (s :: Map Int String)
```

# Axiome für `Map` (2)

- Wenn zweimal an der gleichen Stelle in die Abbildung geschrieben wird, dann überschreibt der zweite den ersten Wert

```
insert a w (insert a v s) == insert a w (s :: Map Int String)
```

  - insgesamt 4 Axiome zu dieser Eigenschaft:

```
insert a w (delete a s) == insert a w (s :: Map Int String)
```

```
delete a (insert a v s) == delete a (s :: Map Int String)
```

```
delete a (delete a s) == delete a (s :: Map Int String)
```

# Axiome für `Map` (3)

- Schreiben an unterschiedlichen Stellen verhält sich kommutativ

```
a /= b ==> insert a v (insert b w s) ==
           insert b w (insert a v (s :: Map Int String))
```

  - auch hier insgesamt 4 Axiome zu dieser Eigenschaft:

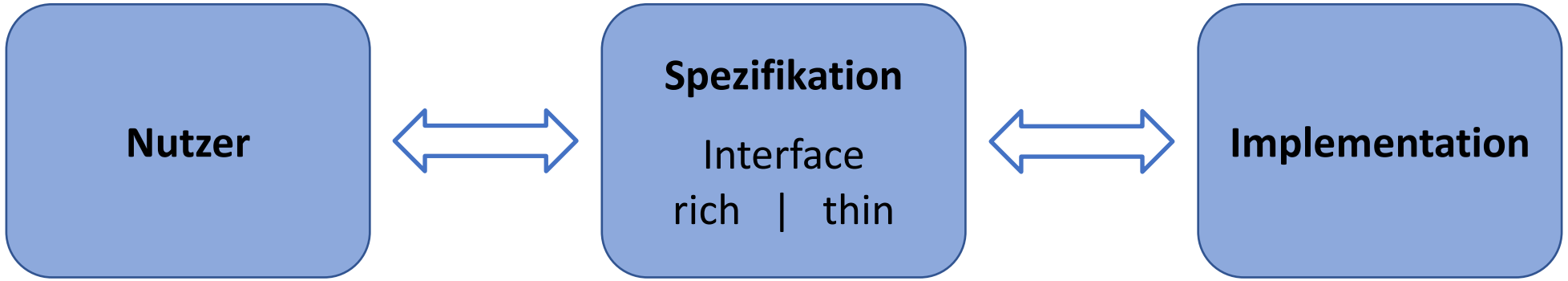
```
a /= b ==> delete a (delete b s) ==
           delete b (delete a (s :: Map Int String))
```

  - davon 2 identisch wg. Kommutativität von "==":

```
a /= b ==> insert a v (delete b s) ==
           delete b (insert a v (s :: Map Int String))
```

# Axiome als Interface

- Axiome spezifizieren
  - **nach außen** das **Verhalten** (des Moduls, ADTs)
  - **nach innen** die **Implementation**
- Signatur + Axiome = **Spezifikation**
- Axiome **müssen gelten**
  - d.h.: für alle möglichen Werte der freien Variablen zu `True` auswerten



# *Rich* vs. *thin interfaces*?

- Benutzersicht
  - viele Operationen und Eigenschaften

  vs.
- Implementationssicht
  - schlankes Interface, wenige Operationen und Eigenschaften

- ... am Beispiel `Map`:

# *Rich* vs. *thin* Interface am Beispiel `Map`

- *rich*:

```
insert :: Eq a => a -> b -> Map a b -> Map a b
```

```
delete :: Eq a => a -> Map a b -> Map a b
```

- *thin*:

```
put :: Eq a => a -> Maybe b -> Map a b -> Map a b
```

- *thin-to-rich*:

```
insert a v = put a (Just v)
```

```
delete a = put a Nothing
```

# Warum *thin interfaces*? – Axiome

- Lesen aus leerer Abbildung (unverändert)

```
lookup a (empty :: Map Int String) == Nothing
```

- Schreiben, dann auslesen an gleicher Stelle: geschriebener Wert

```
lookup a (put a v (s :: Map Int String)) == v
```

- Schreiben, dann auslesen an anderer Stelle: Wert unverändert

```
a /= b ==> lookup a (put b v s) == lookup a (s :: Map Int String)
```

- Zweimal an gleicher Stelle geschrieben: Wert überschrieben

```
put a w (put a v s) == put a w (s :: Map Int String)
```

# Warum *thin interfaces*? – Axiome (2)

- Schreiben an unterschiedlichen Stellen verhält sich kommutativ

```
a /= b ==> put a v (put b w s) ==
           put b w (put a v (s :: Map Int String))
```

- Beobachtung: 5 Axiome (*thin*) vs. 13 Axiome (*rich*)
  - *rich*:
    - je 2 Axiome für *gleiche* und *verschiedene* Stelle Lesen (für `insert` und `delete`) **plus**
    - je 4 Axiome für *Überschreiben* und *Kommutativität* (für alle Kombinationen von `insert` und `delete`)
  - *thin*:
    - je 1 Axiom für *gleiche* und *verschiedene* Stelle Lesen (für `put`) **plus**
    - je 1 Axiom für *Überschreiben* und *Kommutativität* (nur für `put`)

# Zufallsbasiertes Testen in Haskell

# Zufallsbasiertes Testen in Haskell

- **Werkzeug:** `QuickCheck`

```
import Test.Tasty
import qualified Test.Tasty.QuickCheck as QC
import Test.Tasty.QuickCheck((==>))
```

- `QuickCheck` als Bestandteil des Standard-Frameworks `Tasty`
- `==>`: logische Implikation für testbare Eigenschaften
  - `True`, wenn erstes Argument `False` (Test wird nicht ausgeführt) oder wenn zu testende Eigenschaft erfüllt ist.

# Zu testende Implementierung von `Map`

```
module MapList(Map, empty, lookup, put) where

import Prelude hiding(lookup)
import Data.List((\\))

data Map alpha beta = Map [(alpha, beta)]

empty :: Map alpha beta
empty = Map []

lookup :: Eq alpha => alpha -> Map alpha beta-> Maybe beta
lookup a (Map s) =
  case (filter ((a ==). fst) s) of
    []        -> Nothing
    ((_,b):_) -> Just b

-- ...
```

# Testen der Axiome mit `QuickCheck`…

- z.B.: Lesen aus leerer Abbildung

```
prop1 :: TestTree
prop1 = QC.testProperty "read_empty" $ \a ->
  lookup a (empty :: Map Int String) == Nothing
```

- `TestTree`: `Tasty` - Datenstruktur zur Definition von *Test-Suites*
- `testProperty`: `QuickCheck` Funktion zur Testgenerierung
  - Bezeichnung des Tests (`String`)
  - zu testende Eigenschaft (Haskell Prädikat)
- Erzwungene Instantiierung von Typvariablen von `Map`
  - sonst nicht testbar, da polymorph
  - Typ sollte hinreichende Anzahl Elemente haben (hier: `Map Int String`)

# Zufallsbasiertes Testen in Haskell

- Aber was soll nun mit den Tests passieren?

- In Eigenschaften zufällige Werte einsetzen und
- Auswertung auf `True` prüfen
- Freie Variablen der Eigenschaften werden zu Parametern der Testfunktion
- Es wird eine bestimmte Anzahl von Zufallswerten generiert und getestet
  - QuickCheck Standardwert N = 100

# … die anderen Axiome:

```
prop2 :: TestTree
prop2 = QC.testProperty "lookup_put eq" $ \a v s ->
  lookup a (put a v (s :: Map Int String)) == v

prop3 :: TestTree
prop3 = QC.testProperty "lookup_put other" $ \a b v s ->
  a /= b ==> lookup a (put b v s) == lookup a (s :: Map Int String)

prop4 :: TestTree
prop4 = QC.testProperty "put_put eq" $ \a v w s ->
  put a w (put a v s) == put a w (s :: Map Int String)

prop5 :: TestTree
prop5 = QC.testProperty "put_put other" $ \a v b w s ->
  a /= b ==> put a v (put b w s) ==
             put b w (put a v s :: Map Int String)
```

# Steuerung der Zufallswerte-Generierung

- Für selbstdefinierte Datentypen (z.B. `Map`) oder
- wenn Gleichverteilung nicht erwünscht
- In `QuickCheck`: Typklasse `class Arbitrary a` für Zufallswerte
- Einflussnahme auf Zufallswerte durch eigene Instantiierung:

```
instance (Ord a, QC.Arbitrary a, QC.Arbitrary b) =>
         QC.Arbitrary (Map a b) where
  ...
```

- Demo: Test von `MapList` …

# Weiteres Beispiel: `Queue` (fehlerhaft)

- `Queue` mit zwei Listen
  - Schreiben in zweite Liste, löschen aus erster Liste
  - Funktion `First` liefert erstes Element der ersten Liste
- Demo: Test von `Queue` ...
- Getestete Eigenschaft:

```
QC.testProperty "first_enq" $ \a q ->
          q /= empty ==> first (enq a q) == first (q :: Qu Int)
```

  - "Wenn die Queue nicht leer ist, dann ändert sich das vorderste Element beim Hinzufügen eines neuen Elements nicht"

# Queue Implementierung (fehlerhaft)

```
enq   :: alpha-> Qu alpha-> Qu alpha
enq x (Qu [] ys) = Qu (x: reverse ys) []
enq x (Qu xs ys) = Qu xs (x:ys)

first :: Qu alpha-> alpha
first (Qu [] [])    = error "Queue: first of empty Q"
first (Qu [] ys)    = last ys
first (Qu (x:xs) _) = x

deq   :: Qu alpha-> Qu alpha
deq (Qu [] [])     = error "Queue: deq of empty Q"
deq (Qu [] ys)     = Qu (tail (reverse ys)) []
deq (Qu (_:xs) ys) = Qu xs ys
```

# Zusammenfassung

- Signatur: Typ und Operationen eines ADTs
- Eigenschaften, formalisiert in Axiomen
- Axiome als Interface zwischen Nutzer und Implementation
- Signatur + Axiome = Spezifikation
- Axiome können zufallsbasiert getestet werden
- `QuickCheck`
  - Variablen der Axiome als Parameter der Testfunktionen
  - bedingte Eigenschaften durch Implikation in Testfunktionen

# nächstes Mal…

- Monaden