

Praktische Informatik 3: Funktionale Programmierung  
Vorlesung 9 vom 09.12.2012: Signaturen und Eigenschaften

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

# Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
  - ▶ Abstrakte Datentypen
  - ▶ Signaturen und Eigenschaften
  - ▶ Spezifikation und Beweis
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

# Abstrakte Datentypen und Signaturen

- ▶ Letzte Vorlesung: **Abstrakte Datentypen**
  - ▶ Typ plus Operationen
- ▶ Heute: **Signaturen** und **Eigenschaften**

## Definition (Signatur)

Die **Signatur** eines abstrakten Datentyps besteht aus den Typen, und der Signatur der darüber definierten Funktionen.

- ▶ Keine direkte Repräsentation in Haskell
- ▶ Signatur: **Typ** eines Moduls

# Endliche Abbildung: Signatur

- ▶ Adressen und Werte sind Parameter

```
data Map  $\alpha$   $\beta$ 
```

- ▶ Leere Abbildung:

```
empty :: Map  $\alpha$   $\beta$ 
```

- ▶ Abbildung auslesen:

```
lookup :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Maybe  $\beta$ 
```

- ▶ Abbildung ändern:

```
insert :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$ 
```

- ▶ Abbildung löschen:

```
delete :: Ord  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Map  $\alpha$   $\beta$   $\rightarrow$  Map  $\alpha$   $\beta$ 
```

# Signatur und Eigenschaften

- ▶ Signatur genug, um ADT **typkorrekt** zu benutzen
  - ▶ Insbesondere **Anwendbarkeit** und **Reihenfolge**
- ▶ Signatur beschreibt nicht die **Bedeutung** (Semantik):
  - ▶ Was wird **gelesen**?
  - ▶ Wie **verhält** sich die Abbildung?
- ▶ Signatur ist **Sprache** (Syntax) um **Eigenschaften** zu beschreiben

# Beschreibung von Eigenschaften

## Definition (Axiome)

**Axiome** sind Prädikate über den Operationen der Signatur

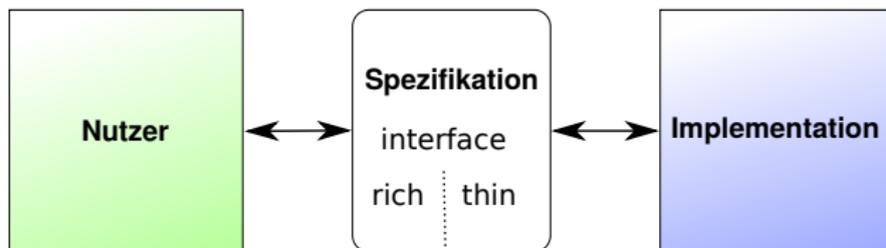
- ▶ Elementare Prädikate  $P$  :
  - ▶ Gleichheit  $s = t$
  - ▶ Ordnung  $s < t$
  - ▶ Selbstdefinierte Prädikate
- ▶ Zusammengesetzte Prädikate
  - ▶ Negation  $\text{not } p$
  - ▶ Konjunktion  $p \ \&\& \ q$
  - ▶ Disjunktion  $p \ || \ q$
  - ▶ **Implikation**  $p \implies q$

# Beobachtbare und Abstrakte Typen

- ▶ **Beobachtbare** Typen: interne Struktur bekannt
  - ▶ Vordefinierte Typen (**Zahlen**, **Zeichen**), algebraische Datentypen (**Listen**)
  - ▶ Viele Eigenschaften und Prädikate bekannt
- ▶ **Abstrakte** Typen: interne Struktur unbekannt
  - ▶ Wenige Eigenschaften bekannt, Gleichheit nur wenn definiert
- ▶ Beispiel Map:
  - ▶ **beobachtbar**: Adressen und Werte
  - ▶ **abstrakt**: Speicher

# Axiome als Interface

- ▶ Axiome müssen **gelten**
  - ▶ für **alle** Werte der freien Variablen zu True auswerten
- ▶ Axiome **spezifizieren**:
  - ▶ nach außen das **Verhalten**
  - ▶ nach innen die **Implementation**
- ▶ **Signatur** + Axiome = **Spezifikation**



# Thin vs. Rich Interfaces

- ▶ Benutzersicht: **reiches** Interface
  - ▶ Viele Operationen und Eigenschaften
- ▶ Implementationssicht: **schlankes** Interface
  - ▶ Wenig Operation und Eigenschaften

- ▶ Beispiel Map:

- ▶ Rich interface:

```
insert :: Ord α => α → β → Map α β → Map α β
delete :: Ord α => α → Map α β → Map α β
```

- ▶ Thin interface:

```
put :: Ord α => α → Maybe β → Map α β → Map α β
```

- ▶ Thin-to-rich:

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

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

# Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

## Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

`lookup a empty = Nothing`

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

## Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

`lookup a empty == Nothing`

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

`lookup a (put a v s) == v`

- ▶ Lesen an anderer Stelle liefert alten Wert:

# Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ empty} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \ v \ s) = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (put } b \ v \ s) = \text{lookup } a \ s$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

# Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ empty} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \ v \ s) = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (put } b \ v \ s) = \text{lookup } a \ s$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \ w \text{ (put } a \ v \ s) = \text{put } a \ w \ s$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

# Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ empty} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \ v \ s) = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (put } b \ v \ s) = \text{lookup } a \ s$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \ w \text{ (put } a \ v \ s) = \text{put } a \ w \ s$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

$$a \neq b \implies \text{put } a \ v \text{ (put } b \ w \ s) = \\ \text{put } b \ w \text{ (put } a \ v \ s)$$

# Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ empty} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \ v \ s) = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (put } b \ v \ s) = \text{lookup } a \ s$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \ w \text{ (put } a \ v \ s) = \text{put } a \ w \ s$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

$$a \neq b \implies \text{put } a \ v \text{ (put } b \ w \ s) = \\ \text{put } b \ w \text{ (put } a \ v \ s)$$

Thin: 5 Axiome  
Rich: 13 Axiome

# Axiome als Eigenschaften

- ▶ Axiome können **getestet** oder **bewiesen** werden
- ▶ Tests finden **Fehler**, Beweis zeigt **Korrektheit**

E. W. Dijkstra, 1972

Program testing can be used to show the presence of bugs, but never to show their absence.

- ▶ Arten von Tests:
  - ▶ Unit tests (JUnit, HUnit)
  - ▶ Black Box vs. White Box
  - ▶ Coverage-based (z.B. path coverage, MC/DC)
  - ▶ Zufallsbasiertes Testen
- ▶ Funktionale Programme eignen sich **sehr gut** zum Testen

# Zufallsbasiertes Testen in Haskell

- ▶ Werkzeug: QuickCheck
- ▶ Zufällige Werte einsetzen, Auswertung auf True prüfen
- ▶ Polymorphe Variablen nicht `testbar`
  - ▶ Deshalb Typvariablen `instanziieren`
  - ▶ Typ muss genug Element haben (hier `Map Int String`)
  - ▶ Durch Signatur `Typinstanz` erzwingen
- ▶ Freie Variablen der Eigenschaft werden `Parameter` der Testfunktion

# Axiome mit QuickCheck testen

- ▶ Für das Lesen:

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

```
prop_readPut :: Int → Maybe String →
              Map Int String → Bool
prop_readPut a v s =
  lookup a (put a v s) == v
```

- ▶ Eigenschaften als **Haskell-Prädikate**
- ▶ Es werden  $N$  Zufallswerte generiert und getestet ( $N = 100$ )

# Axiome mit QuickCheck testen

- ▶ **Bedingte** Eigenschaften:
  - ▶  $A \implies B$  mit A, B Eigenschaften
  - ▶ Typ ist Property
  - ▶ Es werden solange Zufallswerte generiert, bis  $N$  die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.

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

# Axiome mit QuickCheck testen

- ▶ Schreiben:

```
prop_putPut :: Int → Maybe String → Maybe String →  
             Map Int String → Bool  
prop_putPut a v w s =  
  put a w (put a v s) == put a w s
```

- ▶ Schreiben an anderer Stelle:

```
prop_putPutOther :: Int → Maybe String → Int →  
                 Maybe String → Map Int String →  
                 Property  
prop_putPutOther a v b w s =  
  a ≠ b ⇒ put a v (put b w s) ==  
          put b w (put a v s)
```

- ▶ Test benötigt Gleichheit und Zufallswerte für Map a b

# Zufallswerte selbst erzeugen

- ▶ Problem: **Zufällige** Werte von **selbstdefinierten** Datentypen
  - ▶ Gleichverteiltheit nicht immer erwünscht (e.g.  $[\alpha]$ )
  - ▶ Konstruktion nicht immer offensichtlich (e.g. Map)
- ▶ In QuickCheck:
  - ▶ **Typklasse class** Arbitrary  $\alpha$  für Zufallswerte
  - ▶ Eigene **Instanziierung** kann Verteilung und Konstruktion berücksichtigen
  - ▶ E.g. **Konstruktion** einer Map:
    - ▶ Zufällige Länge, dann aus sovielen zufälligen Werten **Map** konstruieren
    - ▶ Zufallswerte in Haskell?

# Signatur und Semantik

## Stacks

Typ:  $\text{St } \alpha$

Initialwert:

`empty` ::  $\text{St } \alpha$

Wert ein/auslesen:

`push` ::  $\alpha \rightarrow \text{St } \alpha \rightarrow \text{St } \alpha$

`top` ::  $\text{St } \alpha \rightarrow \alpha$

`pop` ::  $\text{St } \alpha \rightarrow \text{St } \alpha$

Last in first out (**LIFO**).

## Queues

Typ:  $\text{Qu } \alpha$

Initialwert:

`empty` ::  $\text{Qu } \alpha$

Wert ein/auslesen:

`enq` ::  $\alpha \rightarrow \text{Qu } \alpha \rightarrow \text{Qu } \alpha$

`first` ::  $\text{Qu } \alpha \rightarrow \alpha$

`deq` ::  $\text{Qu } \alpha \rightarrow \text{Qu } \alpha$

First in first out (**FIFO**)

Gleiche **Signatur**, unterschiedliche **Semantik**.

# Eigenschaften von Stack

- ▶ Last in first out (LIFO):

$\text{top}(\text{push } a \text{ } s) = a$

$\text{pop}(\text{push } a \text{ } s) = s$

$\text{push } a \text{ } s \neq \text{empty}$

# Eigenschaften von Queue

- ▶ First in first out (FIFO):

$$\text{first} (\text{enq } a \text{ empty}) = a$$

$$q \neq \text{empty} \implies \text{first} (\text{enq } a \ q) = \text{first } q$$

$$\text{deq} (\text{enq } a \ \text{empty}) = \text{empty}$$

$$q \neq \text{empty} \implies \text{deq} (\text{enq } a \ q) = \text{enq } a \ (\text{deq } q)$$

$$\text{enq } a \ q \neq \text{empty}$$

# Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
newtype St  $\alpha$  = St [ $\alpha$ ] deriving (Show, Eq)
```

```
empty = St []
```

```
push a (St s) = St (a:s)
```

```
top (St []) = error "St:_top_on_empty_stack"
```

```
top (St s) = head s
```

```
pop (St []) = error "St:_pop_on_empty_stack"
```

```
pop (St s) = St (tail s)
```

# Implementation von Queue

- ▶ Mit einer Liste?
  - ▶ Problem: am Ende anfügen oder abnehmen ist teuer.
- ▶ Deshalb zwei Listen:
  - ▶ Erste Liste: zu entnehmende Elemente
  - ▶ Zweite Liste: hinzugefügte Elemente rückwärts
  - ▶ Invariante: erste Liste leer gdw. Queue leer

# Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
-----------	----------	-------	----------------

---

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])

## Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])

# Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])

# Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])

# Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])
deq	5	3	([3], [])

# Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])
deq	5	3	([3], [])
deq	3		([], [])

# Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 → 9	([9], [4])
enq 7		7 → 4 → 9	([9], [7, 4])
deq	9	7 → 4	([4, 7], [])
enq 5		5 → 7 → 4	([4, 7], [5])
enq 3		3 → 5 → 7 → 4	([4, 7], [3, 5])
deq	4	3 → 5 → 7	([7], [3, 5])
deq	7	3 → 5	([5, 3], [])
deq	5	3	([3], [])
deq	3		([], [])
deq	error		([], [])

# Implementation

- ▶ Datentyp:

```
data Qu  $\alpha$  = Qu [  $\alpha$  ] [  $\alpha$  ]
```

- ▶ Leere Schlange: alles leer

```
empty = Qu [] []
```

- ▶ Erstes Element steht vorne in erster Liste

```
first :: Qu  $\alpha$   $\rightarrow$   $\alpha$   
first (Qu [] _) = error "Queue: first of empty Q"  
first (Qu (x:xs) _) = x
```

- ▶ Gleichheit:

```
instance Eq  $\alpha$   $\Rightarrow$  Eq (Qu  $\alpha$ ) where  
  Qu xs1 ys1 == Qu xs2 ys2 =  
    xs1 ++ reverse ys1 == xs2 ++ reverse ys2
```

# Implementation

- ▶ Bei `enq` und `deq` Invariante prüfen

```
enq x (Qu xs ys) = check xs (x:ys)
```

```
deq (Qu [] _)      = error "Queue: deq of empty Q"  
deq (Qu (_:xs) ys) = check xs ys
```

- ▶ Prüfung der Invariante **nach** dem Einfügen und Entnehmen
- ▶ `check` **garantiert** Invariante

```
check :: [α] → [α] → Qu α  
check [] ys = Qu (reverse ys) []  
check xs ys = Qu xs ys
```

# Zusammenfassung

- ▶ **Signatur**: Typ und Operationen eines ADT
- ▶ **Axiome**: über Typen formulierte **Eigenschaften**
- ▶ **Spezifikation** = Signatur + Axiome
  - ▶ **Interface** zwischen Implementierung und Nutzung
  - ▶ **Testen** zur Erhöhung der Konfidenz und zum Fehlerfinden
  - ▶ **Beweisen** der Korrektheit
- ▶ **QuickCheck**:
  - ▶ Freie Variablen der Eigenschaften werden **Parameter** der Testfunktion
  - ▶  $\implies$  für **bedingte** Eigenschaften