

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 9 vom 11.12.2018: Signaturen und Eigenschaften

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- ▶ Anmeldung zur **Probeklausur**:
 - ▶ Ab sofort auf der stud.ip-Seite.
 - ▶ Bis **Do 12:00**
- ▶ Termin: Montag, 17.12.2018 10:00 (**pünktlich**) und 10:30
- ▶ Ort: Testzentrum des ZMML, neben der Uni-Bücherei auf dem Boulevard
- ▶ Dauer: 30 Minuten
- ▶ Inhalt: zwei kleine Funktionen implementieren, vier M/C-Fragen

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ **Teil II: Funktionale Programmierung im Großen**
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
- ▶ 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
- ▶ Typ $\text{Map } \alpha \ \beta$, Operationen:

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

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

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

```
insert :: Ord  $\alpha \Rightarrow \alpha \rightarrow \beta \rightarrow \text{Map } \alpha \ \beta \rightarrow \text{Map } \alpha \ \beta$ 
```

```
delete :: Ord  $\alpha \Rightarrow \alpha \rightarrow \text{Map } \alpha \ \beta \rightarrow \text{Map } \alpha \ \beta$ 
```

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

Eigenschaften Endlicher Abbildungen

- 1 Aus der **leeren** Abbildung kann **nichts** gelesen werden.
- 2 Wenn etwas **geschrieben** wird, und an der **gleichen** Stelle wieder **gelesen**, erhalte ich den geschriebenen Wert.
- 3 Wenn etwas **geschrieben** wird, und an **anderer** Stelle etwas **gelesen** wird, kann das Schreiben vernachlässigt werden.
- 4 An der **gleichen** Stelle **zweimal geschrieben** überschreibt der zweite den ersten Wert.
- 5 An unterschiedlichen Stellen **geschrieben** kommutiert.

Formalisierung von Eigenschaften

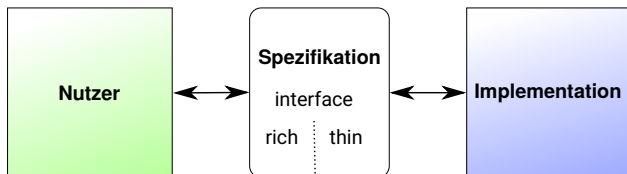
Definition (Axiome)

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** $p \implies q$

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**



Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

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

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

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

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

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

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

- ▶ Lesen an anderer Stelle liefert alten Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (insert } a \text{ v (s :: Map Int String))} = \text{Just v}$$
$$\text{lookup } a \text{ (delete } a \text{ (s :: Map Int String))} = \text{Nothing}$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (delete } b \text{ s)} = \text{lookup } a \text{ (s :: Map Int String)}$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (insert } a \text{ v (s :: Map Int String))} = \text{Just v}$$
$$\text{lookup } a \text{ (delete } a \text{ (s :: Map Int String))} = \text{Nothing}$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (delete } b \text{ s)} = \text{lookup } a \text{ (s :: Map Int String)}$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{insert } a \text{ w (insert } a \text{ v s)} = \text{insert } a \text{ w (s :: Map Int String)}$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (insert } a \text{ v (s :: Map Int String))} = \text{Just v}$$
$$\text{lookup } a \text{ (delete } a \text{ (s :: Map Int String))} = \text{Nothing}$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (delete } b \text{ s)} = \text{lookup } a \text{ (s :: Map Int String)}$$

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{insert } a \text{ w (insert } a \text{ v s)} = \text{insert } a \text{ w (s :: Map Int String)}$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

$$a \neq b \implies \text{insert } a \text{ v (delete } b \text{ s)} = \\ \text{delete } b \text{ (insert } a \text{ v s :: Map Int String)}$$

- ▶ Sehr **viele** Axiome (insgesamt 13)!

Thin vs. Rich Interfaces

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

```
insert :: Ord  $\alpha \Rightarrow \alpha \rightarrow \beta \rightarrow \text{Map } \alpha \beta \rightarrow \text{Map } \alpha \beta$ 
```

```
delete :: Ord  $\alpha \Rightarrow \alpha \rightarrow \text{Map } \alpha \beta \rightarrow \text{Map } \alpha \beta$ 
```

- ▶ Thin interface:

```
put :: Ord  $\alpha \Rightarrow \alpha \rightarrow \text{Maybe } \beta \rightarrow \text{Map } \alpha \beta \rightarrow \text{Map } \alpha \beta$ 
```

- ▶ Thin-to-rich:

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

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

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

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

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

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

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

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

- ▶ Lesen an anderer Stelle liefert alten Wert:

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \text{ } v \text{ (s :: Map Int String))} = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

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

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \text{ } v \text{ (s :: Map Int String))} = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

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

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ s)} = \text{put } a \text{ } w \text{ (s :: Map Int String)}$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \text{ } v \text{ (s :: Map Int String))} = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

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

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ s)} = \text{put } a \text{ } w \text{ (s :: Map Int String)}$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

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

Axiome für Map (thin interface)

- ▶ Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ (empty :: Map Int String)} = \text{Nothing}$$

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \text{ } v \text{ (s :: Map Int String))} = v$$

- ▶ Lesen an anderer Stelle liefert alten Wert:

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

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ s)} = \text{put } a \text{ } w \text{ (s :: Map Int String)}$$

- ▶ Schreiben über verschiedene Stellen kommutiert:

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

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 **instantiieren**
 - ▶ 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:

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

```
prop2 :: TestTree
prop2 = QC.testProperty "lookup_put_eq" $  $\lambda a \ v \ s \rightarrow$ 
  lookup a (put a v (s :: Map Int String)) == v
```

- Hier: Eigenschaften als **Haskell-Prädikate**
- *QuickCheck*-Axiome mit `QC.testProperty` in *Tasty* eingebettet
- Es werden N Zufallswerte generiert und getestet (Default $N = 100$)

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.

```
prop3 :: TestTree
prop3 = QC.testProperty "lookup_put_other" $ \a b v s →
  a ≠ b  $\implies$  lookup a (put b v s) ==
    lookup a (s :: Map Int String)
```

Axiome mit *QuickCheck* testen

► Schreiben:

```
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)
```

► Schreiben an anderer Stelle:

```
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)
```

► Test benötigt **Gleichheit** und **Zufallswerte** für Map a b

Zufallswerte selbst erzeugen

- ▶ Problem: **Zufällige** Werte von **selbstdefinierten** Datentypen
 - ▶ Gleichverteiltheit nicht immer erwünscht (z.B. $[\alpha]$)
 - ▶ Konstruktion nicht immer offensichtlich (z.B. `Map`)
- ▶ In *QuickCheck*:
 - ▶ **Typklasse** `class Arbitrary  $\alpha$` für Zufallswerte
 - ▶ Eigene **Instanziierung** kann Verteilung und Konstruktion berücksichtigen

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

- ▶ Bspw. Konstruktion einer `Map`:
 - ▶ Zufällige Länge, dann aus sovielen zufälligen Werten `Map` konstruieren
 - ▶ Zufallswerte in Haskell?

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

Beobachtbare Gleichheit

- ▶ Auf abstrakten Typen: nur **beobachtbare** Gleichheit
 - ▶ Zwei Elemente sind **gleich**, wenn alle Operationen die gleichen Werte liefern
- ▶ Bei **Implementation**: Instanz für **Eq** (**Ord** etc.) entsprechend definieren
 - ▶ Die Gleichheit $=$ muss die **beobachtbare** Gleichheit sein.
- ▶ **Abgeleitete** Gleichheit (**deriving** **Eq**) wird **immer** exportiert!

Signatur und Semantik

Stacks

Typ: $\text{St } \alpha$

Initialwert:

`empty :: St  $\alpha$`

Wert ein/auslesen:

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

`top :: St  $\alpha \rightarrow \alpha$`

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

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

Queues

Typ: $\text{Qu } \alpha$

Initialwert:

`empty :: Qu  $\alpha$`

Wert ein/auslesen:

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

`first :: Qu  $\alpha \rightarrow \alpha$`

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

First in first out (**FIFO**)

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

Eigenschaften von Stack

- Last in first out (LIFO):

$$\text{top} (\text{push } a_1 (\text{push } a_2 \dots (\text{push } a_n \text{ empty}))) = a_1$$

Eigenschaften von Stack

- Last in first out (LIFO):

$$\text{top} (\text{push } a_1 (\text{push } a_2 \dots (\text{push } a_n \text{ empty}))) = a_1$$

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

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

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

Eigenschaften von Queue

- First in first out (FIFO):

$$\text{first} (\text{enq } a_1 (\text{enq } a_2 \dots (\text{enq } a_n \text{ empty}))) = a_n$$

Eigenschaften von Queue

- First in first out (FIFO):

$$\text{first} (\text{enq } a_1 (\text{enq } a_2 \dots (\text{enq } a_n \text{ empty}))) = a_n$$

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

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

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

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

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

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data 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

Interne Repräsentation

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	$([5, 3], [])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	$([5, 3], [])$
first	5	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	$([5, 3], [])$
first	5	
deq	$\langle 3 \rangle$	$([3], [])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	$([5, 3], [])$
first	5	
deq	$\langle 3 \rangle$	$([3], [])$
first	3	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	$([5, 3], [])$
first	5	
deq	$\langle 3 \rangle$	$([3], [])$
first	3	
deq	$\langle \rangle$	$([], [])$

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle \rangle$	$([], [])$
enq 9	$\langle 9 \rangle$	$([9], [])$
enq 4	$\langle 4 \rightarrow 9 \rangle$	$([9], [4])$
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	$([9], [7, 4])$
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	$([4, 7], [])$
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [5])$
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	$([4, 7], [3, 5])$
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	$([7], [3, 5])$
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	$([5, 3], [])$
first	5	
deq	$\langle 3 \rangle$	$([3], [])$
first	3	
deq	$\langle \rangle$	$([], [])$
first	error	
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:

```
valid :: Qu  $\alpha$   $\rightarrow$  Bool  
valid (Qu [] ys) = null ys  
valid (Qu (_:_) _) = True
```

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
 - ▶ $\Rightarrow$ für **bedingte** Eigenschaften