

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 9 vom 11.01.2021: Signaturen und Eigenschaften

Christoph Lüth



Wintersemester 2020/21

Organisatorisches

- ▶ Anmeldung zur **Klausur**:
 - ▶ Ab **Dienstag** bis **Ende der Woche** auf stud.ip (unverbindlich)
 - ▶ Ersetzt **nicht** die **Modulanmeldung**
- ▶ Klausurtermine:
 - ▶ Klausur: 03.02.2020, 10:00/11:30/15:00
 - ▶ Wiederholungstermin: 21.04.2020, 10:00/11:30/15:00
- ▶ Probeklausur (alte Klausuren vom letzten Jahr) werden veröffentlicht.
- ▶ Fragenkatalog für mündliche Prüfung
- ▶ Es gibt noch eine Extra-Sendung zur mündlichen Prüfung.

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

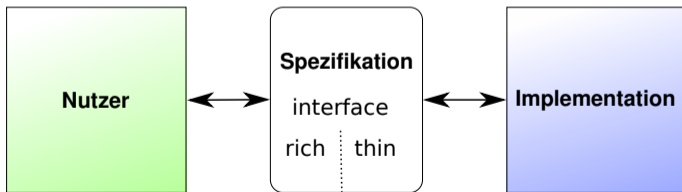
I. Eigenschaften

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

Axiome als Interface

- ▶ Axiome müssen **gelten**
 - ▶ für **alle** Werte der freien Variablen zu True auswerten
- ▶ Axiome **spezifizieren**:
 - ▶ nach außen das **Verhalten** (viele Operationen und Eigenschaften — *rich interface*)
 - ▶ nach innen die **Implementation** (wenig Operationen und Eigenschaften — *thin interface*)
- ▶ Signatur + Axiome = **Spezifikation**



Eigenschaften endlicher Abbildungen

Übung 9.1: Was denkt ihr?

Überlegt mindestens **drei** weitere Eigenschaften endlicher Abbildungen!

- 1 Aus der **leeren** Abbildung kann **nichts** gelesen werden.

Eigenschaften endlicher Abbildungen

Übung 9.1: Was denkt ihr?

Überlegt mindestens **drei** weitere Eigenschaften endlicher Abbildungen!

- 1 Aus der **leeren** Abbildung kann **nichts** gelesen werden.
- 2 Wenn etwas **gelesen** wird an der **gleichen** Stelle, an der etwas **geschrieben** worden ist, erhalte ich den geschriebenen Wert.
- 3 Wenn etwas **gelesen** wird an einer **anderen** Stelle, an der etwas **geschrieben** worden ist, 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

- ▶ Ziel: Eigenschaften **formal** beschreiben, um sie testen oder beweisen zu können.

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** $p \implies q$

Endliche Abbildung: Signatur für Map

- ▶ 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  $\alpha \Rightarrow \alpha \rightarrow \text{Map } \alpha \beta \rightarrow \text{Map } \alpha \beta$ 
```

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:

```
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:

```
a ≠ b ⇒ lookup a (delete b s) = lookup a (s :: Map Int String)
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten 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:

```
a ≠ b ⇒ lookup a (delete b s) = lookup a (s :: Map Int String)
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

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

- ▶ Schreiben über verschiedene Stellen kommutiert:

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:

```
a ≠ b ⇒ lookup a (delete b s) = lookup a (s :: Map Int String)
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

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

- ▶ Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⇒ insert a v (delete b s) = delete b (insert a vs)
```

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

Thin vs. Rich Interfaces

- ▶ Benutzersicht: **reiches** Interface
 - ▶ Viele Operationen und Eigenschaften
- ▶ Implementationsicht: **schlankes** Interface
 - ▶ Wenig Operation und Eigenschaften
- ▶ Konversion dazwischen („Adapter“)

Thin vs. Rich Maps

- ▶ 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
```

- ▶ Konversion von thin auf 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 = Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map (thin interface)

- ▶ 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 (thin interface)

- ▶ 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:

```
a ≠ b ⇒ lookup a (put b c s) = lookup a s
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map (thin interface)

- ▶ 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:

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

- ▶ 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 (thin interface)

- ▶ 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:

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

- ▶ 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 (thin interface)

- ▶ 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:

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

- ▶ 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

Quick Question

Übung 9.2: Gleichheiten

Betrachtet die letzten beiden Fälle:

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

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

Wieso müssen wir die Fälle $a = b$ und $a \neq b$, aber nicht $w = v$ und $w \neq v$ unterscheiden?

Quick Question

Übung 9.2: Gleichheiten

Betrachtet die letzten beiden Fälle:

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

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

Wieso müssen wir die Fälle $a = b$ und $a \neq b$, aber nicht $w = v$ und $w \neq v$ unterscheiden?

Lösung: Im Gegensatz zu a und b gelten beide Axiome sowohl für $w = v$ als auch für $w \neq v$:

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

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

II. Testen von Eigenschaften

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

- ▶ Eigenschaften als **monomorphe Haskell-Prädikate**
- ▶ Für das Lesen:

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

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

- ▶ *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 ⇒ 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" $ λa 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" $ λa 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

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!

Zufallswerte selbst erzeugen

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

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

- ▶ Zufallswerte in Haskell?

Zufällige Maps erzeugen

- ▶ Erster Ansatz: zufällige Länge, dann aus sovielen zufälligen Werten `Map` konstruieren
 - ▶ Berücksichtigt `delete` nicht
- ▶ Besser: über einen **smart constructor** zufällige Maps erzeugen
 - ▶ Muss entweder in `Map` implementiert werden
 - ▶ oder benötigt Zugriff auf interne Struktur

Was stimmt hier nicht?

Übung 9.3: Map als balancierte Bäume.

Warum ist diese Implementierung von `Map` als binärer Baum falsch?

```
data Map  $\alpha$   $\beta$  = Empty
  | Node  $\alpha$   $\beta$  Int (Map  $\alpha$   $\beta$ ) (Map  $\alpha$   $\beta$ )
  deriving Eq
```

Was stimmt hier nicht?

Übung 9.3: Map als balancierte Bäume.

Warum ist diese Implementierung von `Map` als binärer Baum falsch?

```
data Map  $\alpha$   $\beta$  = Empty
                | Node  $\alpha$   $\beta$  Int (Map  $\alpha$   $\beta$ ) (Map  $\alpha$   $\beta$ )
                deriving Eq
```

Lösung: Weil die abgeleitete Gleichheit nicht die beobachtbare Gleichheit ist. Die Gleichheit darf nur prüfen, ob die gleichen Schlüssel/Wert-Paare enthalten sind:

```
toList :: Map  $\alpha$   $\beta$   $\rightarrow$  [( $\alpha$ ,  $\beta$ )]
toList = fold ( $\lambda$ k x l r  $\rightarrow$  l++[(k,x)]++r) []
```

```
instance (Eq  $\alpha$ , Eq  $\beta$ )  $\Rightarrow$  Eq (Map  $\alpha$   $\beta$ ) where
  t1 == t2 = toList t1 == toList t2
```


III. Syntax und Semantik

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):

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

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

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

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

Eigenschaften von Queue

- ▶ First in first out (FIFO):

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

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 (`last/init`) ist teuer ($O(n)$).
- ▶ 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

`first`

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	<code>first</code>
<code>empty</code>	<code><></code>	<code>([], [])</code>	<code>error</code>

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Implementation: Datentyp

► Datentyp:

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

► Invariante:

- 1 Anfang der Schlange ist der **Kopf** der ersten Liste
- 2 Wenn erste Liste leer, dann ist auch die zweite Liste leer

► Invariante prüfen und ggf. herstellen (**smart constructor**):

```
queue :: [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  Qu  $\alpha$   
queue [] ys = Qu (reverse ys) []  
queue xs ys = Qu xs ys
```

Implementation: Gleichheit

Übung 9.4:

Warum reicht für Gleichheit auf Schlangen nicht `derive Eq` und wie implementieren wir es dann?

Implementation: Gleichheit

Übung 9.4:

Warum reicht für Gleichheit auf Schlangen nicht `derive Eq` und wie implementieren wir es dann?

Lösung:

► Gegenbeispiel:

$$q_1 = \text{deq } (\text{enq } 7 \ (\text{enq } 4 \ (\text{enq } 9 \ \text{empty}))), q_2 = \text{enq } (7 \ (\text{enq } 4 \ \text{empty}))$$

► Zwei Schlangen sind gleich, wenn der **Inhalt gleich** ist:

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


Implementation: Operationen

- ▶ Leere Schlange: alles leer

```
empty :: Qu α
empty = Qu [] []
```

- ▶ Erstes Element steht vorne in erster Liste

```
first :: Qu α → α
first (Qu [] _) = error "Queue: first of empty Q"
first (Qu (x:xs) _) = x
```

- ▶ Bei enq und deq Invariante prüfen (Funktion queue)

```
enq :: α → Qu α → Qu α
enq x (Qu xs ys) = queue xs (x:ys)
```

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