Praktische Informatik 3: Funktionale Programmierung

Vorlesung 9 (13.12.2022): Signaturen und Eigenschaften

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Universität Bremen

Wintersemester 2022/23

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

Lernziele

Wir wollen die Eigenschaften eines Moduls **abstrakt** spezifizieren. Wir können diese Eigenschaften nutzen, um Implementierungen zu testen.

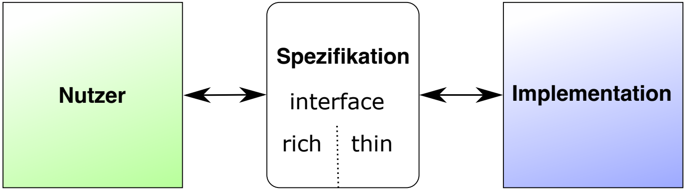
I. Eigenschaften

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**



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 `not p`, Konjunktion `p && q`, Disjunktion `p || q`
  - **Implikation** `p ==> q`

Endliche Abbildung: Signatur für Map

- Adressen und Werte sind Parameter
- Typ Map α β, Operationen:

```haskell
data Map α β
```

```haskell
empty  :: Map α β
```

```haskell
lookup  :: Ord α ⇒ α → Map α β → Maybe β
```

```haskell
insert  :: Ord α ⇒ α → β → Map α β → Map α β
```

```haskell
delete  :: Ord α ⇒ α → Map α β → Map α β
```

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

```
lookup a (delete a s) == Nothing
```

- **Lesen** an anderer Stelle liefert alten Wert:

Axiome für Map

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v s) == Just v
```

```
lookup a (delete a s) == Nothing
```

- **Lesen** an anderer Stelle liefert alten Wert:

$$a \neq b \Longrightarrow \texttt{lookup a (insert b v s)} == \texttt{lookup a s}$$

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

Axiome für `Map`

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v s) == Just v
```

```
lookup a (delete a s) == Nothing
```

- **Lesen** an anderer Stelle liefert alten Wert:

```
a ≠ b ⟹ lookup a (insert b v s) == lookup a s
```

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

```
insert a w (insert a v s) == insert a w s
```

- **Schreiben** und **Löschen** über verschiedene Stellen kommutiert:

Axiome für Map

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v s) == Just v
```

```
lookup a (delete a s) == Nothing
```

- **Lesen** an anderer Stelle liefert alten Wert:

$$a \neq b \Longrightarrow \texttt{lookup a (insert b v s)} == \texttt{lookup a s}$$

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

```
insert a w (insert a v s) == insert a w s
```

- **Schreiben** und **Löschen** über verschiedene Stellen kommutiert:

$$a \neq b \Longrightarrow \texttt{insert a v (delete b s)} == \texttt{delete b (insert a v s)}$$

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

☞ Siehe Übung 9.1

Thin vs. Rich Interfaces

- Benutzersicht: **reiches** Interface
  - Viele Operationen und Eigenschaften
- Implementationssicht: **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
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

Axiome für `Map` (thin interface)

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für `Map` (thin interface)

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **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
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

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

- **Lesen** an anderer Stelle liefert alten Wert:

$$\texttt{a} \neq \texttt{b} \Longrightarrow \texttt{lookup a (put b v s)} == \texttt{lookup a s}$$

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

Axiome für `Map` (thin interface)

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **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 v s) == lookup a s
```

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

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

- **Schreiben** über verschiedene Stellen kommutiert:

Axiome für `Map` (thin interface)

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **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 v s) == lookup a s
```

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

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

- **Schreiben** über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ put a v (put b w s) == put b w (put a v s)
```

Axiome für `Map` (thin interface)

- **Lesen** aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- **Löschen** in der leeren Abbilding bleibt die leere Abbildung:

```
put a Nothing empty == empty
```

- **Lesen** an vorher geschriebener Stelle liefert geschriebenen Wert:

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

- **Lesen** an anderer Stelle liefert alten Wert:

$$\mathtt{a} \neq \mathtt{b} \Longrightarrow \mathtt{lookup\ a\ (put\ b\ v\ s)} == \mathtt{lookup\ a\ s}$$

- **Schreiben** an dieselbe Stelle überschreibt alten Wert:

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

- **Schreiben** über verschiedene Stellen kommutiert:

$$\mathtt{a} \neq \mathtt{b} \Longrightarrow \mathtt{put\ a\ v\ (put\ b\ w\ s)} == \mathtt{put\ b\ w\ (put\ a\ v\ s)}$$

Thin: 6 Axiome
Rich: 13 Axiome

☞ Siehe Übung 9.2

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. Pfadabdeckung, MC/DC)
  - Zufallsbasiertes Testen
- Funktionale Programme eignen sich **sehr gut** zum Testen

Zufallsbasiertes Testen in Haskell

- Idee: Eigenschaften sind Konstante vom Typ `Bool`
- Für **freie** Variablen werden zufällige Werte eingesetzt:

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

- Erweiterungen zu Bool: **Implikation** $\Longrightarrow$, Allquantor (Typ `Property`)
- Polymorphe Variablen nicht testbar
  - Deshalb Typvariablen **instantiieren**
  - Typ muss genug Element haben (hier `Map Int String`)
  - Durch Signatur Typinstanz erzwingen
- Werkzeug: *QuickCheck*

Axiome mit *QuickCheck* testen

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

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

```
prop3 = QC.testProperty "lookup_put␣eq" $ λa v (s :: Map Int String)→
  lookup a (put a v s) == 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` $\Longrightarrow$ `B` mit `A`, `B` Eigenschaften
  - Es werden solange Zufallswerte generiert, bis N die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.
  - Warum?

Axiome mit *QuickCheck* testen

- **Bedingte** Eigenschaften:
  - `A` $\Longrightarrow$ `B` mit `A`, `B` Eigenschaften
  - Es werden solange Zufallswerte generiert, bis N die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.
  - Warum?
  - Implikation *false* $\Longrightarrow \phi$ ist immer wahr (und sagt **nichts** über ϕ).

```
prop4 = QC.testProperty "lookup_put other" $ λa b v (s :: Map Int String)→
  a ≠ b ⟹ lookup a (put b v s) == lookup a s
```

Axiome mit *QuickCheck* testen

- **Schreiben**:

```
prop5 = QC.testProperty "put_put␣eq" $ λa v w (s :: Map Int String)→
  put a w (put a v s) == put a w s
```

- **Schreiben** an anderer Stelle:

```
prop6 = QC.testProperty "put_put␣other" $ λa v b w (s :: Map Int String)→
  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`

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 auf die Konstruktoren nicht immer erwünscht (z.B. $[\alpha]$)
  - 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

☞ Siehe Übung 9.3

III. Syntax und Semantik

Signatur und Semantik

Stacks

Typ: `St` α

Initialwert:

```
empty :: St α
```

Wert ein/auslesen:

```
push :: α → St α → St α
top  :: St α → α
pop  :: St α → St α
```

Last in, first out (LIFO).

Queues

Typ: `Qu` α

Initialwert:

```
empty :: Qu α
```

Wert ein/auslesen:

```
enq   :: α → Qu α → Qu α
first :: Qu α → α
deq   :: Qu α → Qu α
```

First in, first out (FIFO)

Gleiche Signatur, unterscheidliche Semantik.

Eigenschaften von `Stack`

- Last in first out (LIFO):

$$\texttt{top}\ (\texttt{push}\ a_1\ (\texttt{push}\ a_2\ \ldots\ (\texttt{push}\ a_n\ \texttt{empty}))) = a_1$$

Eigenschaften von `Stack`

- Last in first out (LIFO):

$$\texttt{top}\ (\texttt{push}\ a_1\ (\texttt{push}\ a_2\ \ldots\ (\texttt{push}\ a_n\ \texttt{empty}))) = a_1$$

```
top (push a s) == a
```

```
pop (push a s) == s
```

```
push a s ≠ empty
```

Eigenschaften von `Queue`

- First in, first out (FIFO):

$$\texttt{first}\ (\texttt{enq}\ a_1\ (\texttt{enq}\ a_2\ \ldots\ (\texttt{enq}\ a_n\ \texttt{empty}))) = a_n$$

Eigenschaften von `Queue`

- First in, first out (FIFO):

$$\texttt{first (enq } a_1 \texttt{ (enq } a_2 \; \ldots \; \texttt{(enq } a_n \texttt{ empty)))} = a_n$$

```
first (enq a empty) == a
```

```
q ≠ empty ⟹ first (enq a q) == first q
```

Eigenschaften von `Queue`

- First in, first out (FIFO):

$$\mathtt{first}\ (\mathtt{enq}\ a_1\ (\mathtt{enq}\ a_2\ \ldots\ (\mathtt{enq}\ a_n\ \mathtt{empty}))) = a_n$$

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

$$\mathtt{q} \neq \mathtt{empty} \Longrightarrow \mathtt{first\ (enq\ a\ q)} == \mathtt{first\ q}$$

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

$$\mathtt{q} \neq \mathtt{empty} \Longrightarrow \mathtt{deq\ (enq\ a\ q)} == \mathtt{enq\ a\ (deq\ q)}$$

Eigenschaften von `Queue`

- First in, first out (FIFO):

$$\texttt{first}\ (\texttt{enq}\ a_1\ (\texttt{enq}\ a_2\ \ldots\ (\texttt{enq}\ a_n\ \texttt{empty}))) = a_n$$

```
first (enq a empty) == a
```

```
q ≠ empty ==> first (enq a q) == first q
```

```
deq (enq a empty) == empty
```

```
q ≠ empty ==> deq (enq a q) == enq a (deq q)
```

```
enq a q ≠ empty
```

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data St α = St [α]  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
- Beispiel für guten **amortisierten** Aufwand.

Repräsentation von `Queue`

Operation	Resultat	Interne Repräsentation	`first`

Repräsentation von `Queue`

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

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`	$\langle 4 \to 9 \rangle$	([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`	$\langle 4 \to 9 \rangle$	([9], [4])	9
`enq 7`	$\langle 7 \to 4 \to 9 \rangle$	([9], [7, 4])	9
`deq`	$\langle 7 \to 4 \rangle$	([4, 7], [])	4

Repräsentation von `Queue`

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

Implementation: Datentyp

- Datentyp:

```
data Qu α = Qu [α] [α]
```

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

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
```

☞ Siehe Übung 9.4

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
  - $\Longrightarrow$ für bedingte Eigenschaften