Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 05.01.2011: Signaturen und Eigenschaften

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Aktionen und Zustände
- Teil III: Funktionale Programmierung im richtigen Leben

Abstrakte Datentypen

- Letzte Vorlesung: Abstrakte Datentypen
- Typ plus Operationen
  - In Haskell: Module
- Heute: Signaturen und Eigenschaften

Signaturen

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

Zur Erinnerung: Endliche Abbildungen

- Endliche Abbildung (FiniteMap)
- Typen: die Abbildung S, Adressen a, Werte b
- Operationen (Auszug)
  - leere Abbildung: S

Zur Erinnerung: Endliche Abbildungen

- Endliche Abbildung (FiniteMap)
- Typen: die Abbildung S, Adressen a, Werte b
- Operationen (Auszug)
  - leere Abbildung: S
  - Abbildung an einer Stelle schreiben: $S \longrightarrow a \longrightarrow b \longrightarrow S$

Zur Erinnerung: Endliche Abbildungen

- Endliche Abbildung (FiniteMap)
- Typen: die Abbildung S, Adressen a, Werte b
- Operationen (Auszug)
  - leere Abbildung: S
  - Abbildung an einer Stelle schreiben: $S \to a \to b \to S$
  - Abbildung an einer Stelle lesen: $S \to a \rightharpoonup b$ (partiell)

Zur Erinnerung: Endliche Abbildungen

- Endliche Abbildung (FiniteMap)
- Typen: die Abbildung S, Adressen a, Werte b
- Operationen (Auszug)
  - leere Abbildung: S
  - Abbildung an einer Stelle schreiben: $S \to a \to b \to S$
  - Abbildung an einer Stelle lesen: $S \to a \rightharpoonup b$ (partiell)

Endliche Abbildung: Signatur

- Adressen und Werte sind Parameter

```
type Map α β
```

- Leere Abbildung:

```
empty :: Map α β
```

- An eine Stelle einen Wert schreiben:

```
insert :: Map α β→ α → β→ Map α β
```

- An einer Stelle einen Wert lesen:

```
lookup :: Map α β→ α → Maybe β
```

Signatur und Eigenschaften

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

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 not p
  - Konjunktion p && q
  - Disjunktion p || q
  - Implikation p $\Longrightarrow$ 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
  - Wenig Eigenschaft bekannt, Gleichheit nur wenn definiert
- Beispiel Map:
  - beobachtbar: Adressen und Werte
  - abstrakt: Speicher

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- Lesen an anderer Stelle liefert alten Wert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- Lesen an anderer Stelle liefert alten Wert:

```
a1 /= a2 ==> lookup (insert m a1 b) a2 ==
             lookup m a2
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- Lesen an anderer Stelle liefert alten Wert:

```
a1 /= a2 ==> lookup (insert m a1 b) a2 ==
             lookup m a2
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
insert (m a b1) a b2 == insert m a b2
```

- Schreiben über verschiedene Stellen kommutiert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup (insert m a b) a == Just b
```

- Lesen an anderer Stelle liefert alten Wert:

```
a1 /= a2 ==> lookup (insert m a1 b) a2 ==
             lookup m a2
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

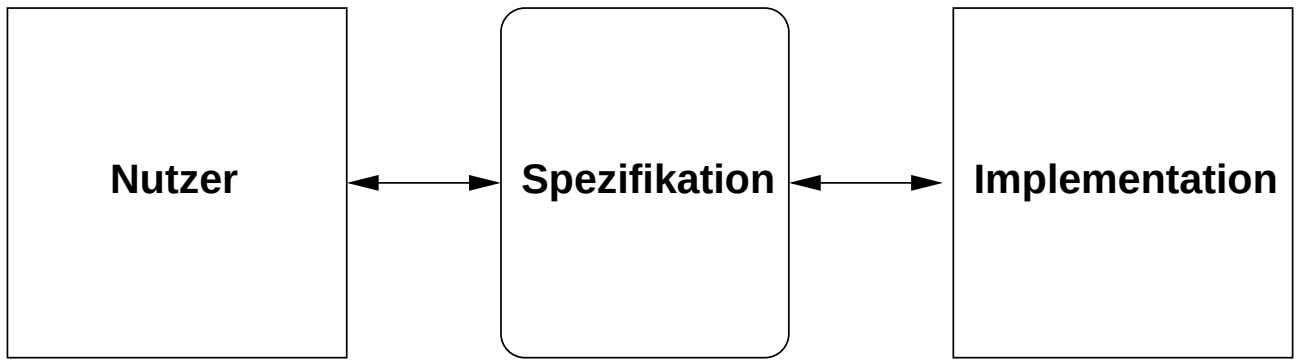
```
insert (m a b1) a b2 == insert m a b2
```

- Schreiben über verschiedene Stellen kommutiert:

```
a1 /= a2 ==> insert (insert m a1 b1) a2 b2 ==
             insert (insert m a2 b2) a1 b1
```

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



- Implementation kann getestet werden
- Axiome können (sollten?) bewiesen werden

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
```

Test auf Leer:

```
isEmpty :: St α → Bool
```

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
```

Test auf Leer:

```
isEmpty :: Qu α → Bool
```

First in first out (FIFO)

Gleiche Signatur, unterscheidliche Semantik.

Eigenschaften von Stack

Last in first out (LIFO):

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

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

```
isEmpty empty
```

```
not (isEmpty (push a s))
```

```
push a s /= empty
```

Eigenschaften von Queue

First in first out (FIFO):

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

```
not (isEmpty q) ==> first (enq a q) == first q
```

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

```
not (isEmpty q) ==> deq (enq a q) = enq a (deq q)
```

```
isEmpty (empty)
```

```
not (isEmpty (enq a q))
```

```
enq a q /= empty
```

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data Stack a= Stack [a]  deriving (Show, Eq)

empty =  Stack []

push a (Stack s) = Stack (a:s)

top  (Stack [])  = error "Stack: top on empty stack"

pop  :: Stack a→ Stack a
```

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 $\longrightarrow$9	([9], [4])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 $\longrightarrow$9	([9], [4])
enq 7		7 $\longrightarrow$4 $\longrightarrow$9	([9], [7, 4])

Repräsentation von Queue

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 $\longrightarrow$9	([9], [4])
enq 7		7 $\longrightarrow$4 $\longrightarrow$9	([9], [7, 4])
deq	9	7 $\longrightarrow$4	([4, 7], [])

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Repräsentation von Queue

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

Implementation

- Datentyp:

```
data Qu α = Qu [α] [α]
```

- Leere Schlange: alles leer

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

- Invariante: erste Liste leer gdw. Queue leer

```
isEmpty (Qu xs _) = null xs
```

- Erstes Element steht vorne in erster Liste

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

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
```

Axiome als Eigenschaften

- Axiome können getestet oder bewiesen werden
- Tests finden Fehler, Beweis zeigt Korrektheit
- Arten von Tests:
  - Unit tests (JUnit, HUnit)
  - Black Box vs.White Box
  - 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 Int)
  - Durch Signatur Typinstanz erzwingen
- Freie Variablen der Eigenschaft werden Parameter der Testfunktion

Axiome mit QuickCheck testen

- Für das Lesen:

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

```
prop_read_write :: Map Int Int → Int → Int → Bool
prop_read_write s a v=
  lookup (insert s a v) a == Just v
```

- Hier: Eigenschaften direkt als Haskell-Prädikate
- Es werden N Zufallswerte generiert und getestet ($N = 100$)

Axiome mit QuickCheck testen

- Bedingte Eigenschaft in `quickCheck`:
  - $A \Longrightarrow B$ 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_read_write_other ::
  Map Int Int → Int → Int → Int → Property
prop_read_write_other s a v b=
  a /= b ⟹ lookup (insert s a v) b == lookup s b
```

Axiome mit QuickCheck testen

- Schreiben:

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

- Schreiben an anderer Stelle:

```
prop_write_other ::
  Map Int Int → Int → Int → Int → Int → Property
prop_write_other s a v b w =
  a /= b ⟹ insert (insert s a v) b w ==
            insert (insert s b w) a v
```

- Test benötigt Gleichheit auf `Map a b`

Zufallswerte selbst erzeugen

- Problem: Zufällige Werte von selbstdefinierten Datentypen
  - Gleichverteiltheit nicht immer erwünscht (e.g. [a])
  - Konstruktion nicht immer offensichtlich (e.g. Map)
- In QuickCheck:
  - Typklasse **class** Arbitrary a 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?

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