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 Map α β, Operationen:

```
data Map α β
empty  :: Map α β
lookup :: Ord α⇒ α→ Map α β→ Maybe β
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
delete :: Ord α⇒ α→ Map α β→ Map α β
```


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

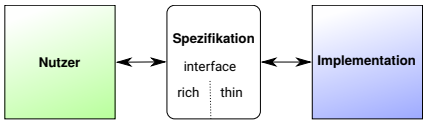
Definition (Axiome)

Axiome sind Prädikate über den Operationen der Signatur

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

```
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 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:
  - 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 (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:

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

- Schreiben an dieselbe Stelle überschreibt alten Wert:

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

- Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ put a v (put b w s) ==
          put b w (put a v s :: Map Int 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 **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" $ λa→
  lookup a (empty :: Map Int String) == Nothing
```

```
prop2 :: TestTree
prop2 = QC.testProperty "lookup_put␣eq" $ λa v s→
  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:
  - $A \Longrightarrow B$ mit A, B Eigenschaften
  - Typ ist `Property`
  - Es werden solange Zufallswerte generiert, bis N die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.

```
prop3 :: TestTree
prop3 = QC.testProperty "lookup_put␣other" $ λa 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

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 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
```

  - 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: 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):

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

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

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

```
push a s ≠ empty
```


Eigenschaften von Queue

- First in first out (FIFO):

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

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

Implementation

- Datentyp:

```
data Qu α = Qu [α] [α]
```

- Leere Schlange: alles leer

```
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
```

- Gleichheit:

```
valid :: Qu α→ 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
  - $\Longrightarrow$ für bedingte Eigenschaften