Praktische Informatik 3: Funktionale Programmierung
Vorlesung 9 vom 11.12.2012: Signaturen und Eigenschaften

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Spezifikation und Beweis
  - Aktionen und Zustände
- 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

$\mathbf{type}\ \mathrm{Map}\ \alpha\ \beta$

- Leere Abbildung:

$\mathtt{empty} :: \mathtt{Tree}\ \alpha$

- Abbildung auslesen:

$\mathtt{lookup} :: \mathrm{Ord}\ \alpha \Rightarrow \alpha \to \mathrm{Map}\ \alpha\ \beta \to \mathrm{Maybe}\ \beta$

- Abbildung ändern:

$\mathtt{insert} :: \mathrm{Ord}\ \alpha \Rightarrow \alpha \to \beta \to \mathrm{Map}\ \alpha\ \beta \to \mathrm{Map}\ \alpha\ \beta$

- Abbildung löschen:

$\mathtt{delete} :: \mathrm{Ord}\ \alpha \Rightarrow \alpha \to \mathrm{Map}\ \alpha\ \beta \to \mathrm{Map}\ \alpha\ \beta$

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: Sprache (Syntax) um Eigenschaften zu beschreiben

Beschreibung von Eigenschaften

Definition (Axiome)
Axiome sind Prädikate über den Operationen der Signatur

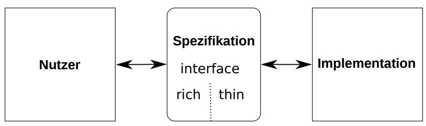
- Elementare Prädikate P :
  - Gleichheit s == t
  - Ordnung s < t
  - Selbstdefinierte Prädikate
- Zusammengesetzte Prädikate
  - Negation not p
  - Konjunktion p && q
  - Disjunktion p || q
  - Implikation p $\Longrightarrow$ q

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
  - Wenige Eigenschaften bekannt, Gleichheit nur wenn definiert
- Beispiel Map:
  - beobachtbar: Adressen und Werte
  - abstrakt: Speicher

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



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

- 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 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)
```

Thin: 5 Axiome
Rich: 13 Axiome

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
  - Coverage-based (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:

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

```
prop_readPut :: Int → Maybe String →
                Map Int String → Bool
prop_readPut a v s =
  lookup a (put a v s) == v
```

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

Axiome mit QuickCheck testen

- Bedingte Eigenschaften:
  - A ⇒ 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_readPutOther :: Int → Int → Maybe String →
                     Map Int String → Property
prop_readPutOther a b v s =
  a ≠ b ⇒ lookup a (put b v s) == lookup a s
```


Axiome mit QuickCheck testen

- Schreiben:

```
prop_putPut :: Int → Maybe String → Maybe String →
               Map Int String → Bool
prop_putPut a v w s =
  put a w (put a v s) == put a w s
```

- Schreiben an anderer Stelle:

```
prop_putPutOther :: Int → Maybe String → Int →
                    Maybe String → Map Int String →
                    Property
prop_putPutOther a v b w s =
  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

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?

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, unterschiedliche Semantik.

Eigenschaften von Stack

- Last in first out (LIFO):

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

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

```
push a s ≠ empty
```


Eigenschaften von Queue

- First in first out (FIFO):

```
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

```
newtype 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	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])
enq 3		3 →5 →7 →4	([4, 7], [3, 5])
deq	4	3 →5 →7	([7], [3, 5])
deq	7	3 →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 [] []
```

- 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:

```
instance Eq α⇒ Eq (Qu α) where
  Qu xs1 ys1 == Qu xs2 ys2 =
    xs1 ++ reverse ys1 == xs2 ++ reverse ys2
```


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