Begleitunterlagen zur Vorlesung vom 16.12.2014

Eigenschaften von funktionalen Programmen werden als Gleichungen formuliert, und durch Gleichungsumformung bewiesen. Die Gleichungsumformung wird dabei durch eine Kette von Gleichungen notiert. Als einfaches Beispiel, um die Notation einzuführen, sei folgendes Haskell-Programm gegeben

```
addTwice :: Int→ Int→ Int
addTwice x y = 2∗(x+ y)
```

Hierüber beweisen wir jetzt eine triviale Eigenschaft (ein Art Distributivregel über $+$):

Lemma (1) addTwice x (y+z)= addTwice (x+y) z

```
    addTwice x (y+ z)
=   2*(x+(y+z))          — Def. addTwice
=   2*((x+y)+z)          — Assoziativität von +
=   addTwice (x+y) z     — Def. addTwice
```

□

Die Umformung beginnt mit der linken Seite der zu beweisenden Gleichung, und endet mit der rechten Seite. Hinter jeden Schritt schreiben wir rechts eine Rechtfertigung der Umformung von der vorhergehenden auf diese Zeile.

Fallunterscheidung.

Funktionen, die über Fallunterscheidung definiert sind, erfordern Fallunterscheidung in den Beweisen. Auch hier ein Beispiel, welches die Notation einführt:

```
max, min :: Int→ Int→ Int
max x y = if x < y then y else x
min x y = if x < y then x else y
```

Lemma (2) max x y - min x y = $|x-y|$

```
    max x y- min x y
    • Fall: x < y
=   y- min x y      — Def. max
=   y - x           — Def. min
=   |x−y|           — Wenn x < y, dann y − x = |x − y|
    • Fall: x ≥ y
=   x- min x y      — Def. max
=   x - y           — Def. min
=   |x−y|           — Wenn x ≥ y, dann x − y = |x − y|
=   |x−y|
```

□

Bei einer Fallunterscheidung werden die Fälle gegebenüber dem Hauptbeweis eingerückt. Die Disjunktion der Bedingungen der Fälle muss True ergeben, oder mit anderen Worten, ein Fall muss immer zutreffen (hier $x < y$ oder $x \geq y$). Jeder Fall startet eine eigene Umformungskette, die aber alle zu demselben Ergebnis führen müssen (hier $|x-y|$), danach wird der Hauptbeweis mit diesem Ergebnis fortgeführt (hier ist er gleich zu Ende).

Induktion

Eigenschaften rekursiv definierter Funktionen werden durch Induktion bewiesen. Der Induktionsbeweis besteht aus der Induktionsbasis und dem Induktionssschritt; beides sind vollständige Gleichungsumformungen. Es muss jeweils angegeben werden, über welcher Variablen der Behauptung die Induktion stattfindet.

Induktion über Listen. Die Listenkonkatenation $+\!\!+$ ist definiert als

```
(++) :: [α] → [α] → [α]
[] ++ ys     = ys
(x:xs) ++ ys = x: (xs++ ys)
```

Wir zeigen zwei leichte Eigenschaften von $+\!\!+$ durch Induktion über Listen.

Lemma (3) $\mathtt{s}+\!\!+[\,]=\mathtt{s}$

Induktion über s

- Induktionsbasis

```
  []++ []
= []              Def. ++
```

- Induktionsschritt

```
  (x:xs)++ []
= x:(xs++ [])     Def. ++
= x:xs            IA (Induktionsannahme)
```

□

Der folgende Beweis zeigt, warum es wichtig ist, über welche Variable die Induktion erfolgt. Möglich wäre Induktion sowohl über `r` als auch über `s` und `t`, erfolgversprechend ist nur das erstere. Der Grund dafür ist, dass die Konkatenation rekursiv über dem *ersten* Argument definiert ist.

Lemma (4) $(\mathtt{r}+\!\!+\mathtt{s})+\!\!+\mathtt{t}=\mathtt{r}+\!\!+(\mathtt{s}+\!\!+\mathtt{t})$

Induktion über r

- Induktionsbasis

```
  ([]++ s)++ t
= s++ t             Def. ++
= []++ (s++ t)      Def. ++
```

- Induktionsschritt

```
  ((c:r)++ s)++ t
= (c:(r++ s))++ t   Def. ++
= c:((r++ s)++ t)   Def. ++
= c:(r++ (s++ t))   IA
= (c:r)++ (s++ t)   Def. ++
```

□

Natürliche Induktion. Die Funktion `replicate` ist durch Rekursion über dem Argument `n` definiert, wobei `n` eine natürliche Zahl ist:

```
replicate :: Int → α → [α]
replicate 0 c = []
replicate n c = c : replicate (n−1) c
```

Daher muss ein Beweis über `replicate` durch *natürliche Induktion* erfolgen:

Lemma (5) `length ( replicate n c)=n`

Induktion über n

- Induktionsbasis

  length (replicate 0 c)
  = length [] Def. replicate
  = 0 Def. length
- Induktionsschritt

  length (replicate (n+1) c)
  = length (c: replicate n c) Def. replicate
  = 1+ length (replicate n c) Def. length
  = 1+ n IA
  = n+ 1

□

Kombinationen. Der folgende Beweis kombiniert notwendigerweise Induktion und Fallunterscheidung, weil die Funktionen take und drop sowohl rekursiv definiert sind als auch Fallunterscheidungen beinhalten:

```
take :: Int → [α] → [α]
take n _  | n ≤ 0 =  []
take _ []         =  []
take n (x:xs)     =  x : take (n−1) xs

drop :: Int → [α] → [α]
drop n xs | n ≤ 0 =  xs
drop _ []         =  []
drop n (_:xs)     =  drop (n−1) xs
```

Interessanterweise können wir über take und drop sowohl natürliche als auch Listeninduktion anwenden. Wir zeigen das gleiche Lemma auf zwei Arten:

Lemma (6) `take n s ++drop n s =s`

Induktion über s

- Induktionsbasis

  take n []++ drop n []
  = []++ [] Def. take, drop
  = [] Def. ++
- Induktionsschritt

  take n (c:s) ++ drop n (c:s)
  = • n<= 0
    = []++ (c:s) Def. take, drop
    = c:s Def. ++
    • n > 0
    = c: (take (n−1) s ++ drop (n−1) s) Def. take, drop
    = c: s IA
  = c:s

□

Lemma (7) `take n s ++ drop n s = s`

Induktion über n

- Induktionsbasis

```
    take 0 s++ drop 0 s
=   []++ s                                          Def. take, drop
=   s                                               Def. ++
```

- Induktionsschritt

```
    take (n+1) s++ drop n s
=   • s == []
    =   []++ []                                     Def. take, drop
    =   []                                          Def. ++
    =   s
    • s == x:xs
    =   x: (take (n−1) xs ++ drop (n−1) xs)         Def. take, drop
    =   x: xs                                       IA
    =   s
=   s
```

□

Beispiel: rev

Im folgenden zeigen wir einige Eigenschaften der Funktion `reverse`, die wir aus Gründen der Lesbarkeit zu `rev` abkürzen, und die wie folgt definiert sei:

```
rev :: [α]→ [α]
rev []     = []
rev (x:xs) = rev xs++ [x]
```

Auch hier ist wichtig, die richtige Variable für die Induktion zu wählen:

Lemma (8) `rev (xs++ ys)= rev ys++ rev xs`

Induktion über xs

- Induktionsbasis

```
    rev ( []++ ys)
=   rev ys                          Def. ++
=   rev ys++ []                     Lemma (3)
=   rev ys ++ rev xs                Def. rev
```

- Induktionsschritt

```
    rev (x:xs ++ ys)
=   rev (x:(xs++ ys))               Def. ++
=   rev (xs++ ys) ++ [x]            Def. rev
=   (rev ys++ rev xs)++ [x]         IA
=   rev ys++ (rev xs++ [x])         Lemma: (4)
=   rev ys++ (rev (x:xs))           Def. rev
```

□

Lemma (9) `rev (rev s)=s`

Induktion über s

- Induktionsbasis

```
  rev (rev [])
= rev []                    Def. rev
= []                        Def. rev
```

- Induktionsschritt

```
  rev (rev (c:s))
= rev (rev s++ [c])         Def. rev
= rev [c] ++ rev (rev s)    Lemma (8)
```

□

Der folgende Beweis zeigt, wie man die Annahme stärken muss, um eine stärkere Induktionsannahme zu erhalten. Wir wollen zeigen, dass die in der Vorlesung vorgestellte kürzere Definition von `rev` als

```
rev xs = foldl (flip (:)) [] xs
```

äquivalent ist zu der rekursiven Definition. Als Erinnerung hier noch die Definition von `foldl`:

```
foldl :: (β→ α→ β)→ β→ [α]→ β
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Der direkt Beweis von `rev s = foldl (flip (:)) [] s` mit Induktion über `s` scheitert an der zu schwachen Induktionsvoraussetzung. Wir zeigen daher

Lemma (10) `foldl (flip (:)) ys xs = rev xs++ ys`

Induktion über xs

- Induktionsbasis

```
  foldl (flip (:)) ys []
= ys                                    Def. foldl
= []++ ys                               Def. ++
= rev [] ++ ys                          Def. rev
```

- Induktionsschritt

```
  foldl (flip (:)) ys (x:xs)
= foldl (flip (:)) (flip (:) ys x) xs   Def. foldl
= foldl (flip (:)) (x:ys) xs            Def. flip
= rev xs++ (x:ys)                       IA
= rev xs++ ([x]++ ys)                   Def. ++
= (rev xs++ [x])++ ys                   Lemma (s.o.)
= rev (x:xs) ++ ys                      Def. rev
```

□

Jetzt können wir die gewünschte Eigenschaft zeigen:

Lemma (11) `rev s = foldl (flip (:)) [] s`

```
  rev s
= rev s++ []                Lemma
= foldl (flip (:)) [] s     Lemma (10)
```

□

Queues und Stacks

Wir zeigen zuerst einige der Eigenschaften von Stack und Qu.

Für Stacks sind diese Eigenschaften sehr einfach zu zeigen:

Lemma (12) pop (push a s) ==s

	pop (push a (St s))	
=	pop (St (a:s))	Def. push
=	St (tail (a:s))	Def. pop
=	s	Def. tail

□

Für Schlangen ist wichtig, dass die *Invariante* immer gilt: für eine Schlange Qu s t gilt, dass wenn s== [] dann ist t== []. Wir zeigen exemplarisch:

Lemma (13) q ≠empty ⟹deq (enq a q) ==enq a (deq q)

	deq (enq a (Qu (x:xs) ys))	Da q ≠ empty, hat q die Form Qu (x:xs) ys
=	deq (check (x:xs) (a:ys))	Def. enq
=	deq (Qu (x:xs) (a:ys))	Def. check
=	check xs (a:ys)	Def. deq
=	enq a (Qu xs ys)	
Def. enq		
=	enq a (check xs ys)	Def. check
=	enq a (deq (Qu (x:xs) ys))	Def. deq

□

Auf diese Weise lassen sich die in *QuickCheck* spezifizierten Eigenschaften leicht zeigen. Aber sind diese Eigenschaften auch wirklich vollständig? Der Unterschied zwischen Stack und Schlange ist ja, dass ein Stack einen LIFO-Zugriff ermöglicht, während eine Schlangen einen FIFO-Zugriff bietet. Wie spezifizieren wir das?

Wir definieren dazu zwei Hilfsfunktionen, welche einen Stack oder eine Schlange aufbauen (build), und wieder konsumieren (use):

```
buildQ :: [α]→ Qu α
buildQ xs = foldl (flip Q.enq) Q.empty xs

useQ :: Eq α⇒ Qu α→ [α]
useQ q | q == Q.empty = []
       | otherwise    = Q.first q : useQ (Q.deq q)

buildS :: [α]→ St α
buildS xs = foldl (flip S.push) S.empty xs

useS :: Eq α⇒ St α→ [α]
useS s | s == S.empty = []
       | otherwise    = S.top s : useS (S.pop s)
```

Wir zeigen dann, dass wenn wir eine Schlange erst aufbauen, und dann wieder konsumieren, die Reihenfolge der Elemente gleich bleibt (useQ (buildQ s)=s), während sie sich bei einem Stack umdreht (useS (buildS s)= rev s).

Wir behandeln zuerst Schlangen. Wieder benötigen wir zuerst ein allgemeineres Lemma, bevor wir die

eigentliche Eigenschaft zeigen, weil wir eine stärkere Induktionsvoraussetzung benötigen.

Lemma (14) useQ (foldl (flip S.enq) q s) = rev (useQ q) ++ s

Induktion über s

- Induktionsbasis

```
= useQ (foldl (flip S.enq) q [])
= useQ q                                                          Def. foldl
= []++ useQ q                                                     Def. ++
```

- Induktionsschritt

```
= useQ (foldl (flip Q.enq) t (c:s))
= useQ (foldl (flip Q.enq) (Q.enq c q) s)                         Def. foldl
= rev (useQ (Q.enq c q))++ s                                      IA
= rev (Q.first (Q.enq c q) : useQ (S.deq (S.enq c q))) ++ s       Def. useQ, Q.enq c q ≠ Q.empty
= rev (c: useQ q)++ s                                             Lemma (??)
= (rev (useQ q) ++ [c])++ s                                       Def. rev
= rev (useQ q) ++ (c: s)                                          Lemma (4), Def ++
```

□

Lemma (15) useQ (buildQ s) = s

Induktion über s

- Induktionsbasis

```
useQ (buildQ [])
= useQ (foldl (flip Q.enq) Q.empty [])                            Def. buildQ
= useQ Q.empy                                                     Def. foldl
= []                                                              Def. useQ
```

- Induktionsschritt

```
useQ (buildQ (c:s))
= useQ (foldl (flip Q.enq) Q.empty (c:s))
= useQ (foldl (flip Q.enq) (Q.enq c empty) s)                     Def. foldl
= rev (useQ (Q.enq c empty)) ++ s                                 Lemma (14)
= rev (Q.first (Q.enc c empty) : (Q.deq (Q.enc c empty))) ++ s     Def. useQ
= rev (c:[])++ s                                                  Def. Q.first, Q.deq
= c:s                                                             Def. rev, ++
```

□

Es folgt der Beweis der Behauptung für Stacks.

Lemma (16) `useS (foldl (flip S.push) t s) = rev s ++ useS t`

Induktion über s

- Induktionsbasis

```
=  useS (foldl (flip S.push) t [])
=  useS t                                                    Def. foldl
=  rev []++ useS t                                           Def. rev, Def. ++
```

- Induktionsschritt

```
=  useS (foldl (flip S.push) t (c:s))                        Def. buildS
=  useS (foldl (flip S.push) (S.push c t) s)                 Def. foldl
=  rev s ++ useS (S.push c t)                                IA
=  rev s ++ S.top (S.push c t) : useS (S.pop (S.push c t))   Def. useS, S.push ≠ S.empty
=  rev s ++ c:useS t                                         Lemma (12)
=  (rev s ++ [c])++ useS t                                   Def. ++, Lemma (4)
=  rev (c:s) ++ useS t                                       Def. rev
```

□

Damit folgt jetzt leicht

Lemma (17) `useS (buildS s) = rev s`

```
   useS (buildS s)
=  useS (foldl (flip S.push) S.empty s)   Def. buildS
=  rev s ++ useS S.empty                  Lemma (16)
=  rev s ++ []                            Def. useS
=  rev s                                  Lemma (3)
```

□

Interessant ist, dass diese Beweise lediglich die *Eigenschaften* von Stacks resp. Schlangen nutzen (z.B. Lemma (16)), und nicht die Implementation. Mit anderen Worten, jede Implementation von Schlangen (egal ob mit zwei Listen oder einer) erfüllt die FIFO-Eigenschaft, und jede Implementation von Stacks die LIFO-Eigenschaft.