Kapitel 4: Beweis und Herleitung von Programmen

Lernziele dieses Kapitels

1. Bedeutung der referenziellen Transparenz und des Gleichungsbeweisens
2. Strukturelle Induktion: Natürliche Zahlen, Listen, Bäume
3. Listengesetze: Dekomposition, Dualität, Fusion, Homomorphie
4. Homomorphismen zur Herleitung von Algorithmen: Mergesort, Fibonacci
5. Programmtransformationen zur Erhöhung der Effizienz:
   Maximale Segmentsumme

Referenzielle Transparenz (1)

Def.: ein Ausdruck E heisst referenziell transparent, wenn jeder beliebige Teilausdruck T durch einen Ausdruck T' gleichen Wertes ersetzt werden kann, ohne dass sich dabei der Wert von E ändert.

Bei referenzieller Transparenz gilt die Leibniz-Regel: $\frac{T{=}T'}{E[x:{=}T]{=}E[x:{=}T']}$

Beispiel:

```
f x + f x                          = {nicht-striktes let}
let  y = f x  in  f x + f x        = {Leibniz}
let  y = f x  in  y + f x          = {Leibniz} kritische Stelle
let  y = f x  in  y + y            = {Arithmetik auf Integer-Variablen}
let  y = f x  in  2 * y            = {Leibniz}
let  y = f x  in  2 * f x          = {nicht-striktes let}
2 * f x
```

Referenzielle Transparenz (2)

- Sprache mit Seiteneffekten: referenzielle Transparenz nicht gegeben
  dann z.B.: `f x + f x` $\neq$ `2 * f x`, vgl. Kap. 1
- funktionale Sprachen mit referenzieller Transparenz: rein funktional
- Haskell erlaubt Ein-/Ausgabe mit referenzieller Transparenz:
  Wert von I/O-Operationen ist nicht Rückgabewert der Operation, sondern die Operation selbst:

Haskell	OCaml

`let outOp = putStr "ha"`
`in do outOp`
`  outOp`
	`let outOp = printf "ha"`
	`in outOp;`
	`  outOp`

Ausgabe: `haha`	Ausgabe: `ha`

Beweis von Programmen

- gegeben
  1. Spezifikation, d.h. Ein-/Ausgaberelation
  2. Programm
- gesucht: Beweis, dass Programm die Spezifikation erfüllt

Beweis imperativer Programme

etablierte Methode: Hoare-Kalkül (Methode der Zusicherungen)

- Spezifikation: Vor- und Nachbedingung an Programm
- Aussagen über den Zustand der Belegung der Variablen
- Vorbedingung und Nachbedingung an jeden Programmteil
  - Invarianten für Schleifen
  - Substitutionsregel für Statements

Beweis funktionaler Programme

etablierte Methode: Equational Reasoning (Gleichheitsbeweise)

Spezifikation: Ein-/Ausgaberelation, was bleibt zu tun?

1. Spezifikation muss nicht ausführbar sein, z.B.
   - Gleichungen sind keine zulässigen Funktionsdefinitionen
   - nicht-integrierte Zusatzbedingungen
2. Spezifikation ist ausführbar, aber extrem ineffizient
   - Programm hat gleiche Semantik, aber effizienteren Berechnungsweg

Gleichheitsbeweis: schrittweise Überführung der Spezifikation in das Endprogramm durch Anwendung von Regeln (z.B. Leibniz)

Prinzip des Gleichheitsbeweisens

- Gleichungskette zwischen zwei Ausdrücken
- jede Gleichung gerechtfertigt durch Anwendung einer Regel

Regelanwendung, um Gleichheit $P = Q$ zu rechtfertigen:

1. gib einen Kontext E mit einer Variablen v an, sodass es Ausdrücke L' und R' gibt, mit denen gilt: $E[v := L'] = P$ und $E[v := R'] = Q$
2. wähle geeignete Regel $L = R$ (oder $R = L$) aus Regelbasis aus
3. gib eine Substitution(*) φ an, so dass $L' = \varphi\, L$ und $R' = \varphi\, R$

(*) Substitution $\varphi\, T$ rein syntaktisch und simultan: für jede freie Variable x in T ersetze jedes freie Vorkommen von x in T durch einen bestimmten Ausdruck $(\varphi\, x)$

Beispiel einer Regelanwendung

gegeben

- $P =$ `f (x * (y + g z)) - 1`
- $Q =$ `f (x * y + x * g z) - 1`.
- Regelbasis enthält: `(a * (b + c))` = `a * b + a * c` (Distributivgesetz)

zu zeigen: $P = Q$

1. gib einen Kontext E mit einer Variablen v an, sodass es Ausdrücke L' und R' gibt, mit denen gilt: $E[v:=L'] = P$ und $E[v:=R'] = Q$

   E = `f` v `- 1`, $L' =$ `x * (y + g z)`, $R' =$ `x * y + x * g z`

2. wähle geeignete Regel $L = R$ (oder $R = L$) aus Regelbasis aus

   wähle Distributivgesetz: $L =$ `a * (b + c)`, $R =$ `a * b + a * c`

3. gib eine Substitution φ an, so dass $L' = \varphi\, L$ und $R' = \varphi\, R$

   $\varphi =$ [`a:=x, b:=y, c:=g z`]

Beweis-Skizze

nur der Name der Regel wird angegeben oder eine intuitive Bezeichnung

```
fac 0       = 1                          -- fac.1
fac n | n>0 = n * fac (n-1)              -- fac.2
```

Beispiel (Beweis-Skizze):

```
fac 2                 = {fac.2}
2 * fac (2-1)         = {Arithmetik}
2 * fac 1             = {fac.2}
2 * (1 * fac (1-1))   = {Arithmetik}
2 * (1 * fac 0)       = {fac.1}
2 * (1 * 1)           = {Arithmetik}
2 * 1                 = {Arithmetik}
2
```

Korrektheit des Gleichungsbeweisens

Haskell hat wichtige Eigenschaften

1. referenzielle Transparenz
   jede Gleichung erhält die partielle Korrektheit
2. Nicht-Striktheit der Funktionsanwendung (Laziness):
   Terminationsverhalten bleibt unverändert bei
   - Verwendung einer Funktionsgleichung als Regel (z.B. `fac.2`)
   - Änderung von Ersetzungsreihenfolgen
3. statisches Typsystem
   jeder Ausdruck hat eine definierte Semantik
   - soetwas wie `(/2).(*2)` = `id` = `('div' 2).(*2)` gibt es nicht
   - Regeln sind typabhängig: `+` assoziativ für `Rational`, aber nicht für `Float`

Unterschiedliche Ersetzungsreihenfolgen (1)

```
bridge :: (Float,Float) -> Float -> Float -> Float
bridge (x0,y0) x y = if x==x0 then y0 else y        -- bridge

f :: Float -> Float
f x = bridge (0,1) x (sin x / x)                    -- f
```

zuerst Argumentauswertung

`f 0`	$=\{$`f`$\}$
`bridge (0,1) 0 (sin 0 / 0)`	$=\{$Arithmetik$\}$
`bridge (0,1) 0 (0 / 0)`	$=\{$Arithmetik$\}$
`bridge (0,1) 0` $\perp$	$=\{$`bridge`$\}$
`if 0==0 then 1 else` $\perp$	$=\{$`==`$\}$
`if True then 1 else` $\perp$	$=\{$`if-then-else`$\}$
`1`	

Unterschiedliche Ersetzungsreihenfolgen (2)

```
bridge :: (Float,Float) -> Float -> Float -> Float
bridge (x0,y0) x y = if x==x0 then y0 else y        -- bridge

f :: Float -> Float
f x = bridge (0,1) x (sin x / x)                    -- f
```

zuerst Anwendung der Funktionsgleichung

```
f 0                                  = {f}
bridge (0,1) 0 (sin 0 / 0)           = {bridge}
if 0==0 then 1 else (sin 0 / 0)      = {==}
if True then 1 else (sin 0 / 0)      = {if-then-else}
1
```

Strukturelle Induktion

hier: endliche Strukturen, keine Betrachtung von Laziness

Die natürlichen Zahlen als (strukturell) induktiv definierte Menge:

```
data Nat = Zero                    Null
         | Succ Nat                Nachfolger
         deriving (Eq,Ord,Show)
```

Wir verwenden `Z` und `S` als Abkürzung für `Zero` bzw. `Succ`.

Elemente von `Nat`: `Z` (`0`), `S Z` (`1`), `S (S Z)` (`2`), `S (S (S Z))` (`3`) ...

Addition auf Nat

Definition von Funktionen auf induktiv definierten Typen durch Angabe der Gleichung für jeden Datenkonstruktor

```
add :: Nat -> Nat -> Nat
add x Z         = x                          -- add.1
add x (S y)     = S (add x y)                -- add.2

add' :: Integer -> Integer -> Integer
add' x 0        = x                          -- add'.1
add' x y | y>0 = 1 + add' x (y-1)            -- add'.2
```

```
add (S Z) (S Z) ⇝ S (S Z)
add' 1 1 ⇝ 2
```

Multiplikation auf Nat

```
mul :: Nat -> Nat -> Nat
mul x Z         = Z                         -- mul.1
mul x (S y)     = add x (mul x y)           -- mul.2

mul' :: Integer -> Integer -> Integer
mul' x 0        = 0                         -- mul'.1
mul' x y | y>0 = x + mul' x (y-1)           -- mul'.2


mul (S (S Z)) (S (S (S Z)))  ⇝ S (S (S (S (S (S Z)))))
mul' 2 3  ⇝ 6
```

Schrittweise Berechnung

```
          mul (S (S Z)) (S (S (S Z)))
{mul.2}→ add (S (S Z)) (mul (S (S Z)) (S (S Z)))
{mul.2}→ add (S (S Z)) (add (S (S Z)) (mul (S (S Z)) (S Z)))
{mul.2}→ add (S (S Z)) (add (S (S Z))
                            (add (S (S Z)) (mul (S (S Z)) Z)))
{mul.1}→ add (S (S Z)) (add (S (S Z)) (add (S (S Z)) Z))
{add.1}→ add (S (S Z)) (add (S (S Z)) (S (S Z)))
{add.2}→ add (S (S Z)) (S (add (S (S Z)) (S Z)))
{add.2}→ S (add (S (S Z)) (add (S (S Z)) (S Z)))
{add.2}→ S (add (S (S Z)) (S (add (S (S Z)) Z)))
{add.2}→ S (S (add (S (S Z)) (add (S (S Z)) Z)))
{add.1}→ S (S (add (S (S Z)) (S (S Z))))
{add.2}→ S (S (S (add (S (S Z)) (S Z))))
{add.2}→ S (S (S (S (add (S (S Z)) Z))))
{add.1}→ S (S (S (S (S (S Z)))))
```

Der induktive Datentyp Liste

```
data List a = Nil              leere Liste
            | C a (List a)     cons
```

In Haskell (keine legale Quellprogrammsyntax)

```
-- data [a] = []
--          | a : [a]  deriving (Eq, Ord)
```

Elemente:	
`Nil`	`[]`
`C x Nil`	`x:[]`
`C y (C x Nil)`	`y:x:[]`
`C z (C y (C x Nil))`	`z:y:x:[]`
...	

Die Längenfunktion auf Listen

```
len :: List a -> Nat
len Nil      = Z              -- len.1
len (C _ xs) = S (len xs)     -- len.2

len' :: [a] -> Int  --Integer nicht nötig, weil Speicher <= Adressraum
len' []      = 0              -- len'.1
len' (_:xs)  = 1 + len' xs    -- len'.2
```

```
len (C z (C y (C x Nil)))  ⇝ S (S (S Z))
len' (z:y:x:[])  ⇝ 3
```

Die Summenfunktion auf Listen

```
su :: List Nat -> Nat
su Nil       = Z                -- su.1
su (C x xs) = add x (su xs) -- su.2

su' :: Num a => [a] -> a
su' []       = 0                -- su'.1
su' (x:xs)   = x + su' xs       -- su'.2
```

```
su (C (S Z) (C (S (S Z)) (C (S Z) Nil))) ⇝ S (S (S (S Z)))
su' (1:2:1:[])  ⇝ 4
```

Induktive Definition der append-Funktion

append (++) verkettet zwei Listen

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys                    -- (++).1
(x:xs) ++ ys = x : (xs ++ ys)        -- (++).2
```

```
              (1:2:3:[]) ++ (4:5:6:7:8:9:[])
{(++).2}⟶ 1:((2:3:[]) ++ (4:5:6:7:8:9:[]))
{(++).2}⟶ 1:2:((3:[]) ++ (4:5:6:7:8:9:[]))
{(++).2}⟶ 1:2:3:([] ++ (4:5:6:7:8:9:[]))
{(++).1}⟶ 1:2:3:4:5:6:7:8:9:[]
```

Induktive Definition des map-Kombinators

```
map :: (a->b) -> [a] -> [b]
map f []     = []                    -- map.1
map f (x:xs) = f x : map f xs        -- map.2
```

```
          map (^2) (1:2:3:4:[])
{map.2}→ (^2) 1 : map (^2) (2:3:4:[])
{(^2) }→ 1 : map (^2) (2:3:4:[])
{map.2}→ 1 : (^2) 2 : map (^2) (3:4:[])
{(^2) }→ 1 : 4 : map (^2) (3:4:[])
{map.2}→ 1 : 4 : (^2) 3 : map (^2) (4:[])
{(^2) }→ 1 : 4 : 9 : map (^2) (4:[])
{map.2}→ 1 : 4 : 9 : (^2) 4 : map (^2) []
{(^2) }→ 1 : 4 : 9 : 16 : map (^2) []
{map.1}→ 1 : 4 : 9 : 16 : []
```

Struktureller Induktionsbeweis (informell)

- Induktionsaxiom: (zum Beweis einer Aussage P)
  für jeden Datenkonstruktor C eines induktiv definierten Typs S:
  1. sei x ein beliebiges Element von S mit Wurzelsymbol C
  2. Induktionsannahme: $P(x_i)$ gelte für jede echte Teilstruktur x_i von x
  3. man zeige, dass $P(x)$ gilt

  $\Rightarrow$ dann gilt die Aussage P für alle endlichen Elemente von S
- Basisfall, gdw. kein x_i enthält Element vom Typ S
- Induktionsfall, gdw. mindestens ein x_i enthält Element vom Typ S
- Problem: Wahl der Induktionsvariablen x in $P(x)$
- Spezialfälle
  - Vollständige Induktion: Konstruktoren `Zero` (0) und `Succ` ($x \rightarrow x+1$)
  - Listeninduktion: Konstruktoren `[]` (nil) und `(:)` (cons)

Strukturelles Induktionsaxiom (halbformal)

- sei S ein algebraischer Datentyp mit Konstruktoren $\{ C_j \mid j \in \mathbb{N} \wedge 0 \leq j < n \}$
- $\#j$ die Anzahl der Argumente von Konstruktor C_j
- P ein zu beweisendes Prädikat
- Induktionsaxiom

$$\frac{\begin{array}{l}\forall j : 0 \leq j < n :: \\ \forall x_{j,0}, \ldots, x_{j,\#j-1} :: \\ (\forall i : (0 \leq i < \#j \wedge x_{j,i} \in S) : P(x_{j,i})) \implies P(C_j\ x_{j,0} \ldots x_{j,\#j-1})\end{array}}{\forall x\ :\ x \in S \wedge x \text{ endlich}\ :\ P(x)}$$

- Vollständige Induktion als Spezialfall

$$\frac{P(\texttt{Zero}) \ \wedge \ (\forall x : x \in \texttt{Nat} :\ P(x) \implies P(\texttt{Succ}\ x))}{\forall x\ :\ x \in \texttt{Nat} \wedge x \text{ endlich}\ :\ P(x)}$$

Beweis: `map f xs ++ map f ys == map f (xs++ys)` {Fall.1}

Induktionsvariable: `xs`

{Fall.1}: `xs==[]`

```
            map f xs ++ map f ys
{Fall.1}== map f [] ++ map f ys
{ map.1}== [] ++ map f ys
{(++).1}== map f ys
{(++).1}== map f ([]++ys)
{Fall.1}== map f (xs++ys)
```

Beweis: `map f xs ++ map f ys == map f (xs++ys)` {Fall.2}

Induktionsvariable: `xs`

{Fall.2}: `xs==(z:zs)`

```
          map f xs ++ map f ys
{Fall.2}== map f (z:zs) ++ map f ys
{ map.2}== (f z : map f zs) ++ map f ys
{(++).2}== f z : (map f zs ++ map f ys)
{Ind-A.}== f z : (map f (zs++ys))
{ map.2}== map f (z:(zs++ys))
{(++).2}== map f ((z:zs)++ys)
{Fall.2}== map f (xs++ys)
```

Beweis: `(map f . map g) xs == map (f . g) xs` {Fall.1}

```
(q . p) x  = q (p x)     -- compose
```

Induktionsvariable: xs

{Fall.1}: xs==[]

```
              (map f . map g) xs
{Fall.1 }== (map f . map g) []
{compose}== map f (map g [])
{ map.1 }== map f []
{ map.1 }== []
{ map.1 }== map (f . g) []
{Fall.1 }== map (f . g) xs
```

Beweis: `(map f . map g) xs == map (f . g) xs` {Fall.2}

Induktionsvariable: `xs`

{Fall.2}: `xs==(z:zs)`

```
            (map f . map g) xs
{Fall.2 }== (map f . map g) (z:zs)
{compose}== map f (map g (z:zs))
{ map.2 }== map f (g z : map g zs)
{ map.2 }== f (g z) : map f (map g zs)
{compose}== (f . g) z : map f (map g zs)
{compose}== (f . g) z : (map f . map g) zs
{Ind-A. }== (f . g) z : map (f . g) zs
{ map.2 }== map (f . g) (z:zs)
{Fall.2 }== map (f . g) xs
```

Verwendung der vordefinierten Arithmetik

- Konstruktor-basierte Berechnung: langsam und inflexibel
- besser: eingebaute Arithmetik verwenden
- Bsp.: Fakultätsfunktion

```
fac :: Integer -> Integer
fac 0       = 1
fac n | n>0 = n * fac (n-1)
```

- Vorsicht: Wertebereich berücksichtigen (Termination)
- Induktionsbeweise analog mit `0` statt `Zero` und `(+1)` statt `Succ`

Bäume als induktive Datentypen

```
data BinTree1 a
   = Empty1
   | Fork1 { elem1::a, left1, right1 :: BinTree1 a }

data BinTree2 a
   = Leaf2 { elem2::a }
   | Fork2 { left2, right2 :: BinTree2 a }

data BinTree3 a
   = Leaf3 { elem3::a }
   | Fork3 { elem3::a, left3, right3 :: BinTree3 a }

data RoseTree a
   = Fork4 { elem4::a, children::[RoseTree a] }
```

Summenfunktion sumT auf BinTree2

```
sumT :: Num a => BinTree2 a -> a
sumT (Leaf2 elem)        = elem                          -- sumT.1
sumT (Fork2 left right) = sumT left + sumT right   -- sumT.2
```

```
            sumT (Fork2 (Fork2 (Leaf2 1) (Leaf2 2)) (Leaf2 1))
{sumT.2}→ sumT (Fork2 (Leaf2 1) (Leaf2 2)) + sumT (Leaf2 1)
{sumT.2}→ sumT (Leaf2 1) + sumT (Leaf2 2) + sumT (Leaf2 1)
{sumT.1}→ 1 + sumT (Leaf2 2) + sumT (Leaf2 1)
{sumT.1}→ 1 + 2 + sumT (Leaf2 1)
{  +   }→ 3 + sumT (Leaf2 1)
{sumT.1}→ 3 + 1
{  +   }→ 4
```

Beweis: `leaves t == forks t + 1`

```
data BinTree2 a = Leaf2 a
                | Fork2 (BinTree2 a) (BinTree2 a)

leaves, forks :: BinTree2 a -> Integer

leaves (Leaf2 _)   = 1                          -- leaves.1
leaves (Fork2 l r) = leaves l + leaves r        -- leaves.2

forks  (Leaf2 _)   = 0                          -- forks.1
forks  (Fork2 l r) = 1 + forks l + forks r      -- forks.2
```

Beweis: `leaves t == forks t + 1` {Fall.1}

Induktionsvariable: `t`

{Fall.1}: `t == Leaf2 _`

```
                leaves t
{  Fall.1  }== leaves (Leaf2 _)
{ leaves.1 }== 1
{Arithmetik}== 0 + 1
{  forks.1 }== forks (Leaf2 _) + 1
{  Fall.1  }== forks t + 1
```

Beweis: `leaves t == forks t + 1` {Fall.2}

Induktionsvariable: `t`

{Fall.2}: `t == Fork2 l r`

```
              leaves t
{ Fall.2 }== leaves (Fork2 l r)
{leaves.2}== leaves l + leaves r
{ Ind.A. }== forks l + 1 + leaves r
{ Kommut.}== 1 + forks l + leaves r
{ Ind.A. }== 1 + forks l + forks r + 1
{ forks.2}== forks (Fork2 l r) + 1
{ Fall.2 }== forks t + 1
```

Ungleichungs-Beweis: `sumT t <= leaves t * maxT t`

Kette von `<=` und `==` statt Kette von `==`

```
leaves (Leaf2 _)   = 1                          -- leaves.1
leaves (Fork2 l r) = leaves l + leaves r        -- leaves.2

sumT   (Leaf2 x)   = x                          -- sumT.1
sumT   (Fork2 l r) = sumT l + sumT r            -- sumT.2

maxT   (Leaf2 x)   = x                          -- maxT.1
maxT   (Fork2 l r) = max (maxT l) (maxT r)      -- maxT.2
```

Ungleichungs-Beweis: `sumT t <= leaves t * maxT t` {Fall.1}

Induktionsvariable: `t`

{Fall.1}: `t == Leaf2 x`

```
                sumT t
{ Fall.1 }== sumT (Leaf2 x)
{ sumT.1 }== x
{Neutr. *}== 1 * x
{leaves.1}== leaves (Leaf2 x) * x
{ maxT.1 }== leaves (Leaf2 x) * maxT (Leaf2 x)
{ Fall.1 }== leaves t * maxT (Leaf2 x)
{ Fall.1 }== leaves t * maxT t
```

Ungleichungs-Beweis: `sumT t <= leaves t * maxT t` {Fall.2}

Induktionsvariable: `t`

{Fall.2}: `t == Fork2 l r`

```
                sumT t
{ Fall.2 }==    sumT (Fork2 l r)
{ sumT.2 }==    sumT l + sumT r
{ Ind.A. }<=    leaves l * maxT l + sumT r
{ Ind.A. }<=    leaves l * maxT l + leaves r * maxT r
{  max   }<=    leaves l * (max (maxT l) (maxT r)) + leaves r * maxT r
{  max   }<=    leaves l * (max (maxT l) (maxT r))
              + leaves r * (max (maxT l) (maxT r))
{ Distr. }==    (leaves l + leaves r) * max (maxT l) (maxT r)
{leaves.2}==    leaves (Fork2 l r) * max (maxT l) (maxT r)
{ maxT.2 }==    leaves (Fork2 l r) * maxT (Fork2 l r)
{ Fall.2 }==    leaves t * maxT (Fork2 l r)
{ Fall.2 }==    leaves t * maxT t
```

Beobachtungen

- mehrmalige Anwendung der Induktionsannahme kann nötig oder nützlich sein
  - bei Strukturen mit mehr als einer rekursiven Teilstruktur (z.B. Bäume)
  - bei mehrmaligem Auftreten der Induktionsvariablen
- oft kann man schnell die Gleichungskette finden, wenn man
  1. von oben abwärts zur Mitte bzw.
  2. von unten aufwärts zur Mitte

  Funktionsdefinitionen einsetzt (Unfolding) und vereinfacht
- bei Strukturen mit einem Basisfall und einem Induktionsfall werden oft die {.1}-Regeln beim Beweis des Basisfalles und die {.2}-Regeln beim Beweis des Induktionsfalles gebraucht.

Listengesetze und Fold

Achtung: Einschränkung auf endliche Listen ohne explizite Erwähnung

- Dekompositionssätze: zerlegen einer Liste
- Dualitätssätze: Beziehung zwischen `foldl` und `foldr`
- Fusionssätze: verschmelzen einer Berechnung nach einem `fold` mit dem `fold`
- Homomorphiesätze: wann ist eine Funktion ein Listenhomomorphismus?

Definitionen

```
foldl f e []     = e                        -- foldl.1
foldl f e (x:xs) = foldl f (f e x) xs       -- foldl.2

foldr f e []     = e                        -- foldr.1
foldr f e (x:xs) = f x (foldr f e xs)       -- foldr.2
```

Fold-Dekompositionssätze

- (`foldl-dec.`):
  `foldl f a (xs++ys)` = `foldl f (foldl f a xs) ys`
- (`foldr-dec.`):
  `foldr f a (xs++ys)` = `foldr f (foldr f a ys) xs`

falls `f` assoziativ mit neutralem Element `e`:

- (`foldl-ass.`):
  `foldl f e (xs++ys)` = `f (foldl f e xs) (foldl f e ys)`
- (`foldr-ass.`):
  `foldr f e (xs++ys)` = `f (foldr f e xs) (foldr f e ys)`

Beweise: zum üben

Erster Dualitätssatz (1)

Sei `f` assoziativ mit neutralem Element `e`.
Dann gilt für jede endliche Liste `xs`: `foldr f e xs` = `foldl f e xs`

Beweis durch Induktion über die Struktur von `xs`

{Fall.1}: `xs == []`

```
              foldr f e xs
{ Fall.1 }== foldr f e []
{foldr.1 }== e
{foldl.1 }== foldl f e []
{ Fall.1 }== foldl f e xs
```

Erster Dualitätssatz (2)

{Fall.2}: xs == (y:ys)

```
                foldr f e xs
{ Fall.2  }==   foldr f e (y:ys)
{ foldr.2 }==   f y (foldr f e ys)
{ Ind.Ann.}==   f y (foldl f e ys)
{Hilfsbew.}==   foldl f (f y e) ys
{Neut. e  } ==  foldl f y ys
{Neut. e  } ==  foldl f (f e y) ys
{ foldl.2 }==   foldl f e (y:ys)
{ Fall.2  }==   foldl f e xs
```

Erster Dualitätssatz, Hilfsbeweis (1) Fehlversuch!

zu zeigen: `f y (foldl f e ys)` = `foldl f (f y e) ys`

{Fall.1}: `ys == []`

```
                f y (foldl f e ys)
{ Fall.1   }== f y (foldl f e [])
{ foldl.1  }== f y e
{ foldl.1  }== foldl f (f y e) []
{ Fall.1   }== foldl f (f y e) ys
```

Erster Dualitätssatz, Hilfsbeweis (2) Fehlversuch!

zu zeigen: `f y (foldl f e ys)` = `foldl f (f y e) ys`

{Fall.2}: `ys == (z:zs)`

```
                f y (foldl f e ys)
{ Fall.2   }==  f y (foldl f e (z:zs))
{ foldl.2  }==  f y (foldl f (f e z) zs)
{ Neut.e.  }==  f y (foldl f z zs)
{    ?     }==  foldl f (f y z) zs
{ foldl.2  }==  foldl f y (z:zs)
{ Neut. e  }==  foldl f (f y e) (z:zs)
{ Fall.2   }==  foldl f (f y e) ys
```

z ist i.a. nicht gleich dem neutralen Element.
Induktionsannahme muss verallgemeinert werden!

Erster Dualitätssatz, Hilfsbeweis (1) richtig

zu zeigen: `f y (foldl f x ys)` = `foldl f (f y x) ys`

{Fall.1}: `ys == []`

```
                f y (foldl f x ys)
{ Fall.1   }== f y (foldl f x [])
{ foldl.1  }== f y x
{ foldl.1  }== foldl f (f y x) []
{ Fall.1   }== foldl f (f y x) ys
```

Erster Dualitätssatz, Hilfsbeweis (2) richtig

zu zeigen: `f y (foldl f x ys)` = `foldl f (f y x) ys`

{Fall.2}: `ys == (z:zs)`

```
                f y (foldl f x ys)
{ Fall.2   }== f y (foldl f x (z:zs))
{ foldl.2  }== f y (foldl f (f x z) zs)
{ Ind.Ann. }== foldl f (f y (f x z)) zs
{ Ass. f   }== foldl f (f (f y x) z) zs
{ foldl.2  }== foldl f (f y x) (z:zs)
{ Fall.2   }== foldl f (f y x) ys
```

Zweiter Dualitätssatz (1)

```
foldr f x xs = foldl (flip f) x (reverse xs)
```

Beweis durch Induktion über die Struktur von `xs`

{Fall.1}: `xs == []`

```
            foldr f x xs
{Fall.1 }== foldr f x []
{foldr.1}== x
{foldl.1}== foldl (flip f) x []
{ rev.1 }== foldl (flip f) x (reverse [])
{Fall.1 }== foldl (flip f) x (reverse xs)
```

Zweiter Dualitätssatz (2)

{Fall.2}: xs == y:ys

```
                foldr f x xs
{ Fall.2    }== foldr f x (y:ys)
{ foldr.2   }== f y (foldr f x ys)
{  flip     }== flip f (foldr f x ys) y
{ foldl.1   }== foldl (flip f) (flip f (foldr f x ys) y) []
{ foldl.2   }== foldl (flip f) (foldr f x ys) [y]
{ Ind.Ann.  }== foldl (flip f) (foldl (flip f) x (reverse ys)) [y]
{foldl-dec.}== foldl (flip f) x (reverse ys ++ [y])
{reverse.2  }== foldl (flip f) x (reverse (y:ys))
{ Fall.2    }== foldl (flip f) x (reverse xs)
```

Anwendung des zweiten Dualitätssatzes

Herleitung einer effizienten Implementierung von `reverse`

```
              reverse xs                    Spezifikation, s.u.
{   id   }== id (reverse xs)
{foldr/id}== foldr (:) [] (reverse xs)
{ 2.Dual }== foldl (flip (:)) [] (reverse (reverse xs))
{rev/rev }== foldl (flip (:)) [] xs    Implementierung linearer Zeit
```

---

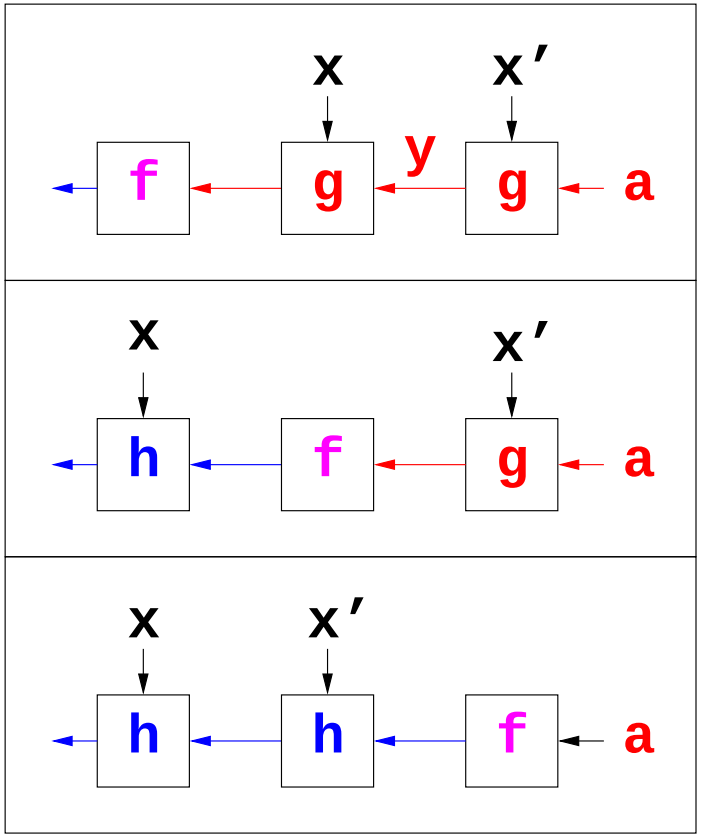
2. Dualitätssatz verwendet Spezifikation von `reverse` mit quadratischer Zeit!

```
reverse []     = []                    -- reverse.1
reverse (x:xs) = reverse xs ++ [x]     -- reverse.2
```

Fusionssätze (1: `foldr`-Fusion)

Sei `f` strikt und gelte `f (g x y) = h x (f y)`.

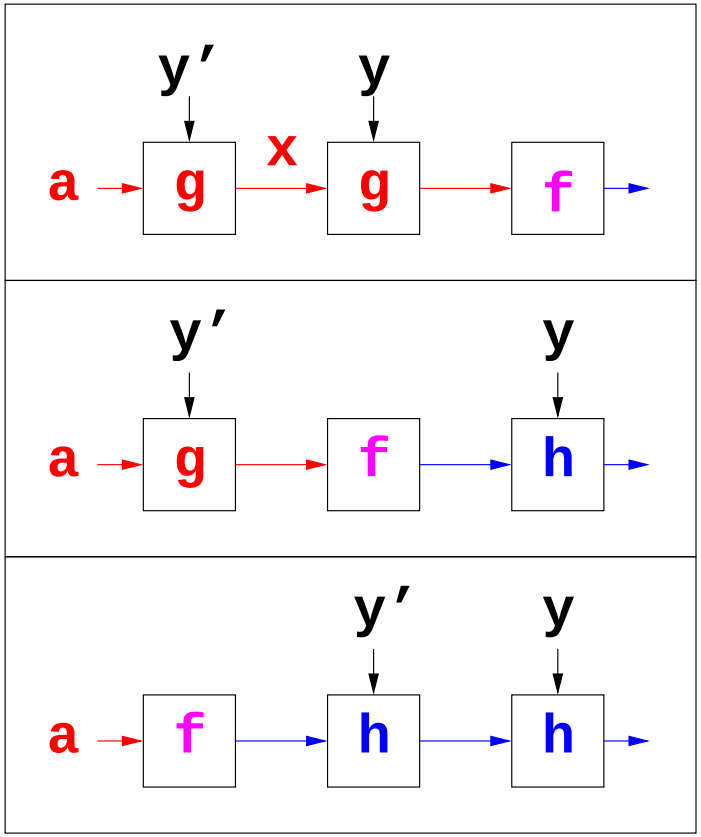
Dann gilt: `f . foldr g a = foldr h (f a)`



Fusionssätze (2: `foldl`-Fusion)

Sei `f` strikt und gelte `f (g x y) = h (f x) y`.

Dann gilt: `f . foldl g a = foldl h (f a)`



Spezialfälle der Fusionssätze

- fold/map-Fusion

  ```
  foldr f a . map g = foldr (f . g) a
  ```

- bookkeeping-law: sei `f` assoziativ mit neutralem Element `e`; dann gilt:

  ```
  foldr f e . concat = foldr f e . map (foldr f e)
  ```

  Anwendung Buchführung (`f`=`(+)`,`e`=`0`):

  ```
  sum . concat = sum . map sum
  ```

Monoid

Monoid: Halbgruppe mit neutralem Element, Notation $(M, \oplus, e)$

- M Menge
- $(\oplus)$: $M \to M \to M$ eine totale Funktion, $\oplus$ assoziativ
- e ist neutrales Element bzgl. $\oplus$

Haskell-Beispiele:

- (`Integer`,`+`,`0`), natürliche Zahlen mit Addition
- (`[a]`,`++`,`[]`), (endliche) Listen mit append
- (`a->a`, `.` ,`id`), Funktionen mit Komposition

Monoidhomomorphismus

- seien $\underline{M_1} = (M_1, \oplus_1, e_1)$ und $\underline{M_2} = (M_2, \oplus_2, e_2)$ Monoide
- Monoidhomomorphismus: Abbildung $h : M_1 \rightarrow M_2$ mit den Eigenschaften
  1. $h(e_1) = e_2$
  2. $\forall m, m' \in M_1 : h(m \oplus_1 m') = h(m) \oplus_2 h(m')$

Listenhomomorphismen

Beispiele für Homomorphismen von $\underline{M_1} = (\texttt{[a]}, \texttt{++}, \texttt{[]})$ nach $\underline{M_2} = (\texttt{Integer}, \texttt{+}, \texttt{0})$:

- `length`
- `sum`
- `sum . map f`

es gibt viele Homomorphismen von $\underline{M_1}$ nach $\underline{M_2}$ Unterscheidung: durch Erzeugendensystem

Erzeugendensystem

- sei $\underline{M} = (M, \oplus, e)$ ein Monoid
- Erzeugendensystem: Teilmenge $A \subseteq M$, sodass jedes $m \in M$ durch wiederholte Anwendung von $\oplus$ aus Elementen von A und e konstruiert werden kann.

- sei $\underline{M_1} = (\texttt{[a]}, \texttt{++}, \texttt{[]})$
- Satz: die einelementigen Listen bilden ein Erzeugendensystem für $\underline{M_1}$ (Beweis: siehe Thiemann-Buch (6.1))

Beispiele für Listenhomomorphismen

Beispiele für Homomorphismen von $\underline{M_1} = (\texttt{[a]}, \texttt{++}, \texttt{[]})$ nach $\underline{M_2} = (M_2, \oplus, e)$:

Name	`h [x]`	e	$(\oplus)$
`id`	`[x]`	`[]`	`(++)`
`map f`	`[f x]`	`[]`	`(++)`
`reverse`	`[x]`	`[]`	`(flip (++))`
`length`	`1`	`0`	`(+)`
`sum`	`x`	`0`	`(+)`
`product`	`x`	`1`	`(*)`
`sum . map f`	`f x`	`0`	`(+)`
`foldr g e . map f` (*)	`f x`	`e`	`g`

(*) `g` ist assoziativ mit neutralem Element `e`

Erster Homomorphiesatz

Original: Bird, 1988

eine Funktion `h::[a]->b` ist ein Homomorphismus

$\Longleftrightarrow$

es gibt

- eine assoziative Funktion `g` mit neutralem Element `e` und
- eine Funktion `f` mit der Eigenschaft `f x = h [x]`,

sodass `h` sich schreiben lässt als: `h = foldr g e . map f`

Definitionen

Eine Listenfunktion h heisst

- $\oplus$-linksgerichtet für einen binären Operator $\oplus$ gdw., für alle Elemente a und Listen y gilt: $h\,([a] +\!\!+ y) = a \oplus h\,y$
- $\otimes$-rechtsgerichtet für einen binären Operator $\otimes$ gdw., für alle Listen x und Elemente a gilt: $h\,(x +\!\!+ [a]) = h\,x \otimes a$

Anm.: es wird nicht gefordert, dass $\oplus$ bzw. $\otimes$ assoziativ sind.

Zweiter Homomorphiesatz (Spezialisierungssatz)

Original: Bird, 1987

Jeder Listenhomomorphismus lässt sich sowohl als eine linksgerichtete als auch als eine rechtsgerichtete Listenfunktion ausdrücken, d.h.

- sei h von $([\alpha],+\!\!+,[\,])$ nach $(\beta,\circledast,e)$ ein Homomorphismus
- dann gibt es
  1. einen Operator $\oplus$ mit $a \oplus y \;=\; f\, a \circledast y$, sodass $h \;=\; \mathit{foldr}\,(\oplus)\; e$
  2. einen Operator $\otimes$ mit $x \otimes b \;=\; x \circledast f\, b$, sodass $h \;=\; \mathit{foldl}\,(\otimes)\; e$

Dritter Homomorphiesatz

Beweis und Beispiel nach Gibbons, 1995

> Wenn h linksgerichtet und rechtsgerichtet ist,
> dann ist h ein Homomorphismus.

Zu zeigen: wenn h links- und rechtsgerichtet ist, dann gibt es einen Operator $\odot$, so dass $h\,(x \mathbin{+\!\!+} y) = h\,x \odot h\,y$. $\odot$ assoziativ weil $+\!\!+$ assoziativ.
Wir suchen eine explizite Definition für $\odot$, d.h., eine Funktion g mit

1. $t \odot u = h\,(g\,t \mathbin{+\!\!+} g\,u)$
2. $h \circ g \circ h = h$

Idee: g liefert zu x und y äquivalente Repräsentanten im Definitionsbereich von h.

Hilfslemma 1 zum Beweis des dritten Homomorphiesatzes

Hilfslemma 1.
Für jede berechenbare Funktion h mit aufzählbarem Definitionsbereich gibt es eine berechenbare (mglw. nicht totale) Funktion g, so dass gilt: $h \circ g \circ h = h$.

Beweis. Um $g\,t$ zu berechnen, kann man den Definitionsbereich von h aufzählen, und das erste x zurückgeben, für das gilt: $h\,x = t$. Falls t im Wertebereich von h liegt, terminiert dieses Verfahren.

Hilfslemma 2 zum Beweis des dritten Homomorphiesatzes

Hilfslemma 2:
Die Listenfunktion h ist ein Homomorphismus gdw. für alle Listen v, w, x und y gilt: $h\,v = h\,x \;\wedge\; h\,w = h\,y \implies h\,(v +\!\!+ w) = h\,(x +\!\!+ y)$.

Beweis:

- h-Homomorphismus $\Rightarrow \exists \oplus :: h(v +\!\!+ w) = h\,v \oplus h\,w = h\,x \oplus h\,y = h(x +\!\!+ y)$
- gelte $h\,v = h\,x \;\wedge\; h\,w = h\,y \implies h\,(v +\!\!+\, w) = h\,(x +\!\!+\, y)$ $(*)$
  - wähle g, sodass $h \circ g \circ h = h$ (Hilfslemma 1)
  - definiere $\odot$ durch $t \odot u = h\,(g\,t +\!\!+ g\,u)$

$$
\begin{array}{ll}
h\,x \odot h\,y & = \{\text{Definition } \odot\} \\
h\,(g\,(h\,x) \;+\!\!+\; g\,(h\,y)) & = \{\text{ setze } v = g\,(h\,x) \text{ und } w = g\,(h\,y)\ \} \\
h\,(v +\!\!+ w) & = \{\ h\,v = h\,x \wedge h\,w = h\,y \wedge (*)\ \} \\
h\,(x +\!\!+ y) &
\end{array}
$$

also: h ist Homomorphismus mit Operator $\odot$ (neutrales Element: $h\,[\,]$).

Beweis des dritten Homomorphiesatzes

zu zeigen: h ist links- und rechtsgerichtet $\Longrightarrow$ h Homomorphismus

- gelte: h ist links- und rechtsgerichtet: $h = \mathit{foldr}\,(\oplus)\,e \;=\; \mathit{foldl}\,(\otimes)\,e$
- zu zeigen: h Homomorphismus
- zeige äquivalente Aussage nach Hilfslemma 2:
  $h\,v = h\,x \wedge h\,w = h\,y \;\Longrightarrow\; h(v +\!\!+\; w) = h(x +\!\!+ y)$
  - gelte $h\,v = h\,x \;\wedge\; h\,w = h\,y$
  - zu zeigen: $h\,(v +\!\!+\; w) = h\,(x +\!\!+\; y)$ [nächste Folie]

Beweis:

$$
\begin{array}{ll}
& h\,(v +\!\!+\, w) \\
= & \qquad \{\ h \text{ ist linksgerichtet } \} \\
& \mathit{foldr}\,(\oplus)\ e\ (v +\!\!+\, w) \\
= & \qquad \{\ \mathit{foldr}\text{-Dekomposition } \} \\
& \mathit{foldr}\,(\oplus)\ (\mathit{foldr}\,(\oplus)\ e\ w)\ v \\
= & \qquad \{\ h\,w = h\,y\ \} \\
& \mathit{foldr}\,(\oplus)\ (\mathit{foldr}\,(\oplus)\ e\ y)\ v \\
= & \qquad \{\ \mathit{foldr}\text{-Dekomposition } \} \\
& \mathit{foldr}\,(\oplus)\ e\ (v +\!\!+\, y) \\
= & \qquad \{\ h \text{ ist linksgerichtet } \} \\
& h\ (v +\!\!+\, y) \\
= & \qquad \{\ h \text{ ist rechtsgerichtet } \} \\
& \mathit{foldl}\,(\otimes)\ e\ (v +\!\!+\, y) \\
= & \qquad \{\ \mathit{foldl}\text{-Dekomposition } \} \\
& \mathit{foldl}\,(\otimes)\ (\mathit{foldl}\,(\otimes)\ e\ v)\ y \\
= & \qquad \{\ h\,v = h\,x\ \} \\
& \mathit{foldl}\,(\otimes)\ (\mathit{foldl}\,(\otimes)\ e\ x)\ y \\
= & \qquad \{\ \mathit{foldl}\text{-Dekomposition } \} \\
& \mathit{foldl}\,(\otimes)\ e\ (x +\!\!+\, y) \\
= & \qquad \{\ h \text{ ist rechtsgerichtet } \} \\
& h\ (x +\!\!+\, y)
\end{array}
$$

Anwendung: Herleitung eines Sortierverfahrens

Einfügen in eine sortierte Liste: `ins`

```
ins a []             = [a]
ins a (b:x) | a<=b = a : b : x
            | a>b  = b : ins a x
```

```
ins' = flip ins
```

Sortieren basieren auf einfügen:

- linksgerichtet: `sort' = foldr ins []`
- rechtsgerichtet: `sort = foldl ins' []`

Folge aus dem dritten Homomorphiesatz: `sort` ist ein Homomorphismus.

Herleitung eines Sortierverfahrens

- sowohl linksgerichtetes als auch rechtsgerichtetes Einfügesortieren braucht Zeit $\theta(n^2)$ für Liste der Länge n
- gesucht: Lösung, bei der die Liste in zwei etwa gleich große Teile geteilt wird, um Zeit $\theta(n \cdot \log\, n)$ zu erreichen.
- Plan: Herleitung eines Sortierverfahrens, das
  - Teilung der unsortierten Liste in zwei beliebige Teile ungleich [] zulässt
  - links- und rechtsgerichtetes Einfügesortieren als Spezialfälle enthält
- bestimme Implementierung des Operators $\odot$, so dass gilt:
  `sort u ⊙ sort v == sort (u ++ v)`.

Herleitung eines Sortierverfahrens

```
                                                u ⊙ v
{ u, v sortiert }                           =
                                                sort u ⊙ sort v
{ sort ist Homomorphismus }                 =
                                                sort (u ++ v)
{ sort }                                    =
                                                foldl ins' [] (u ++ v)
{ foldl-Dekomposition }                     =
                                                foldl ins' (foldl ins' [] u) v
{ sort }                                    =
                                                foldl ins' (sort u) v
{ u sortiert }                              =
                                                foldl ins' u v
{ setze mergeQ = foldl ins' (*) }           =
                                                mergeQ u v
```

(*) Name `mergeQ` wegen Eigenschaft von `foldl ins'`:

```
mergeQ u []    == foldl ins' u [] == u
mergeQ u (b:v) == foldl ins' u (b:v)
               == foldl ins' (ins' u b) v == mergeQ (ins' u b) v
```

Herleitung eines Sortierverfahrens

```
u ⊙ v = mergeQ u v

mergeQ u []     = u
mergeQ u (b:v) = mergeQ (ins' u b) v
```

- `mergeQ` verwendet Eigenschaft, dass `u` sortiert ist
- `mergeQ` verwendet nicht, dass `v` sortiert ist
  $\Rightarrow$ immer noch quadratische Laufzeit!

Verbesserung von `mergeQ` zu `merge`:
Umschreiben unter Ausnutzung der Sortiertheit von `v`

Verbesserung von `mergeQ` zu `merge`

Abkürzung: sei a ein Element und v eine Liste:
$a \leq v$ heisst: für alle b in v gilt $a \leq b$

Hilfslemma:
$\texttt{a} \leq \texttt{x} \wedge \texttt{a} \leq \texttt{y} \Rightarrow (\texttt{foldl ins' (a:x) y}) = (\texttt{a : foldl ins' x y})$

Beweis: durch Induktion

Verbesserung von `mergeQ` zu `merge`

- zweite Liste ist leer

```
                                merge u []
{ merge == mergeQ }             =
                                mergeQ u []
        { mergeQ.1 }            =
                                u
```

- erste Liste ist leer

```
                                merge [] v
{ merge == mergeQ }             =
                                mergeQ [] v
     { Abk. mergeQ }            =
                                foldl ins' [] v
            { sort }            =
                                sort v
{ v ist sortiert }              =
                                v
```

Verbesserung von `mergeQ` zu `merge`

- beide Listen sind nicht leer

```
                          merge (a:u) (b:v)
{ merge == mergeQ }       =
                          mergeQ (a:u) (b:v)
    { Abk. mergeQ }       =
                          foldl ins' (a:u) (b:v)
        { foldl.2 }       =
                          foldl ins' (ins' (a:u) b) v
                          =
                          ...
```

weiter mit Fallunterscheidung `a` $<$ `b` und `a` $\geq$ `b`

Verbesserung von `mergeQ` zu `merge` (`a` < `b`)

Annahme: (`a:u`) und (`b:v`) sortiert und `a` < `b`

also gilt: `a` ≤ `u`, `a` ≤ `v` und `a` ≤ (`ins' u b`)

	`merge (a:u) (b:v)`
{ Schritte von letzter Folie }	=
	`foldl ins' (ins' (a:u) b) v`
{ `ins'` ∧ `a<b` }	=
	`foldl ins' (a : ins' u b) v`
{ Hilfslemma }	=
	`a : foldl ins' (ins' u b) v`
{ `foldl.2` }	=
	`a : foldl ins' u (b:v)`
{ Abk. `mergeQ` }	=
	`a : mergeQ u (b:v)`
{ `merge == mergeQ` }	=
	`a : merge u (b:v)`

Verbesserung von `mergeQ` zu `merge` (`a` $\geq$ `b`)

Annahme: (`a:u`) und (`b:v`) sortiert und `a` $\geq$ `b`

also gilt: `b` $\leq$ (`a:u`) und `b` $\leq$ `v`

	`merge (a:u) (b:v)`
{ Schritte von vorletzter Folie }	=
	`foldl ins' (ins' (a:u) b) v`
{ `ins'` $\wedge$ `a`$\geq$`b` }	=
	`foldl ins' (b:a:u) v`
{ Hilfslemma }	=
	`b : foldl ins' (a:u) v`
{ Abk. `mergeQ` }	=
	`b : mergeQ (a:u) v`
{ `merge == mergeQ` }	=
	`b : merge (a:u) v`

Verbesserung von `mergeQ` zu `merge`, Fazit

damit ergibt sich für `merge` ...

```
merge []     v              = v
merge u      []             = u
merge (a:u) (b:v) | a<b     = a : merge u (b:v)
                  | a>=b    = b : merge (a:u) v
```

... und die Einfügefunktionen sind Spezialfälle davon:

```
ins  a  xs == merge [a] xs
ins' xs a  == merge xs [a]
```

Resultierendes Sortierverfahren: mergesort

```
mergesort :: Ord a => [a] -> [a]
mergesort []  = []                          -- neutrales Element
mergesort [x] = [x]                         -- Erzeugendensystem
mergesort xs  = let (ys,zs) = splitAt (length xs `div` 2) xs
                    u = mergesort ys
                    v = mergesort zs
                in merge u v                -- Operator im Wertebereich
                                            -- des Homomorphismus

merge :: Ord a => [a] -> [a] -> [a]
merge []     v            = v
merge u      []           = u
merge (a:u) (b:v) | a<b  = a : merge u (b:v)
                  | a>=b = b : merge (a:u) v
```

Programmtransformationen

gehen zurück auf Burstall/Darlington (1977): "A transformation system for developing recursive programs"

Anwendung: transformiere ineffizientes in effizientes Programm

Arten von Transformationsschritten

- unfold: ersetzen eines Funktionsnamens durch den Funktionsrumpf
- fold: Umkehrung von unfold
- def: einführen lokaler Definitionen (`let`/`where`)
- spec: Spezialisierung

Beispiel: Fibonacci-Funktion

Ausgangspunkt: sehr ineffizientes Programm

```
fib 0        = 0                          -- fib.0
fib 1        = 1                          -- fib.1
fib n | n>1 = fib (n-2) + fib (n-1)
fib (n+2)    = fib n + fib (n+1)          -- fib.2
```

Performance-steigernde Transformationen

(1) einfügen einer lokalen Definition:

```
fib (n+2)    = z1 + z2  where (z1,z2) = (fib n, fib (n+1))
```

(2) Definiere eine Hilfsfunktion für die rechte Seite von `fib (n+2)`

```
fib2 n = (fib n, fib (n+1))                    -- fib2
```

(3) Anwendung von (spec)

(a) `fib2 0 = (fib 0, fib 1) = (0,1)`

(b)

```
                fib2 (n+1)
{ Def. fib2 }   =
                (fib (n+1), fib (n+2))
   { unfold }   =
                (fib (n+1), fib (n+1) + fib n)
   { where }    =
                (z2,z2+z1) where (z1,z2) = fib2 n
```

Damit:

```
                fib n
{ fib.2 }       =
                z1 where (z1,z2) = fib2 n
  { fst }       =
                = fst (fib2 n)
```

```
fib2 :: Integer -> (Integer,Integer)
fib2 0       = (0,1)
fib2 n | n>0 = (z2,z2+z1) where (z1,z2) = fib2 (n-1)

fib :: Integer -> Integer
fib n = fst (fib2 n)
```

oder, Rekursion ersetzt durch `foldl`

```
fib :: Integer -> Integer
fib n = let f (z1,z2) = const (z2,z2+z1)
        in fst (foldl f (0,1) [1..n])
```

Homomorphieeigenschaften bei `fib`

- Ziel: Optimierung der `foldl`-Berechnung
- repräsentieren `n` durch Liste von () der Länge n
- Homomorphismus $h : ([()], +\!\!+, []) \rightarrow (\mathbb{N}^2 \rightarrow \mathbb{N}^2, \circ, id)$

```
fib :: Integer -> Integer
fib n = fst (fibHom [ () | _ <- [1..n] ] (0,1))

fibHom :: [()] -> ((Integer,Integer)->(Integer,Integer))
fibHom xs = let f (z1,z2) = (z2,z2+z1)
            in foldr (.) id (map (const f) xs)
```

Weitere Verbesserungen

- explizite Funktionskomposition ist speicherplatzaufwändig
- suchen nach algebraischen Möglichkeiten zur Kompaktifizierung der Komposition

wir wissen:

- `f` ist eine lineare Abbildung $\texttt{f}(z_1, z_2) = (z_2, z_2 + z_1) = \left(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right)^T$
- Komposition linearer Abbildungen $(g \circ f)$ entspricht Matrixmultiplikation

nächster Schritt: Umstellung des `foldr`-Operators auf Matrixmultiplikation

Einführung eines Typs von 2x2-Matrizen

```
data M2 a = M2 {upperLeft,upperRight,lowerLeft,lowerRight::a}
           deriving (Eq,Show)

unitMat, fibMat :: M2 Integer
unitMat = M2 1 0
             0 1    -- (z1,z2) -> (z1,z2)
fibMat  = M2 0 1
             1 1    -- (z1,z2) -> (z2,z1+z2)

instance Num a => Num (M2 a) where
 (M2 a b
     c d) *
             (M2 e f
                 g h) = M2 (a*e+b*g) (a*f+b*h)
                           (c*e+d*g) (c*f+d*h)
```

Fibonacci-Berechnung mittels Matrixmultiplikation

```
fib :: Integer -> Integer
fib n = lowerLeft (fibHom' [ () | _ <- [1..n] ])

fibHom' :: [()] -> M2 Integer
fibHom' xs = foldr (*) unitMat (map (const fibMat) xs)
-- war:      foldr (.) id      (map (const f     ) xs)
```

weitere Optimierung: effiziente Potenzierung

```
fibOpt :: Integer -> Integer
fibOpt n = lowerLeft $ fibMat^n
```

Laufzeitvergleiche (Zeiten mit ghci in Sekunden, ohne Ausgabe)

n	binär rekursiv	linear rekursiv	`foldr (.)`	`foldr (*)`	`fibOpt`
30	5.38	–	–	–	–
31	8.70	–	–	–	–
1000	–	0.01	0.00	0.01	0.00
10000	–	0.12	0.10	0.42	0.00
100000	–	12.19	11.62	87.28	0.01
1000000	–	–	–	–	0.26
10000000	–	–	–	–	8.39

Problem der maximalen Segmentsumme

gesucht: Maximum der Summen aller Segmente einer Liste ganzer Zahlen

Bsp.: `[-3,4,-7,2,4,-5,2,3,7,-2,-1,9,3,-15,6,-2,9,-7]` $\rightarrow$ 22

imperative Problemlösungen (beschrieben in: Jon Bentley: Programming Pearls)

- Test aller Segmente: $\Theta(n^3)$
- inkrementelles Update von Teilsummen: $\Theta(n^2)$
- Divide-and-Conquer Lösung: $\Theta(n \cdot \log\ n)$
- Scan-Algorithmus: $\Theta(n)$

Im folgenden: formale Herleitungen

- funktionaler Scan-Algorithmus $\Theta(n)$
- paralleler D&C-Algorithms $\Theta(\log\ n)$

Problem der maximalen Segmentsumme

Spezifikation:

```
mss :: [Integer] -> Integer
mss = let segs = concat . map inits . tails
      in maximum . map sum . segs
```

- `tails`: finale Segmente, Bsp.: `tails [1,2,3]` $\rightsquigarrow$ `[[1,2,3],[2,3],[3],[]]`
- `inits`: initiale Segmente, Bsp.: `map inits (tails [1,2,3])` $\rightsquigarrow$
  `[[[],[1],[1,2],[1,2,3]], [[],[2],[2,3]], [[],[3]], [[]]]`
- `segs`: alle Segmente
  Bsp.: `segs [1,2,3]` $\rightsquigarrow$
  `[[],[1],[1,2],[1,2,3],[],[2],[2,3],[],[3],[]]`

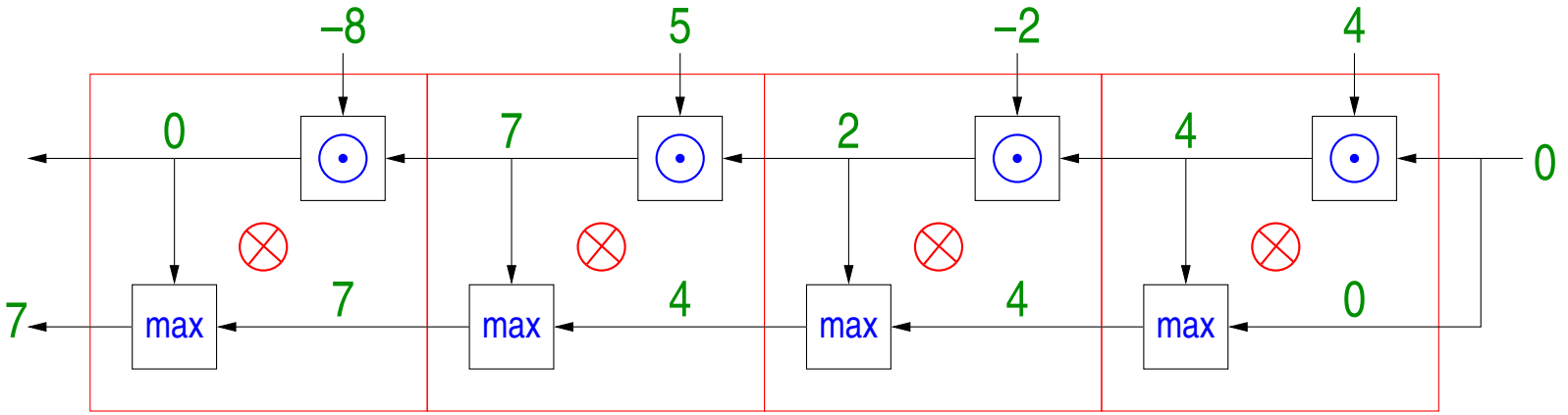
Reduktion der Komplexität von $\Theta(n^3)$ auf $\Theta(n)$

```
                                mss
{Definition mss}                =
                                maximum . map sum . concat . map inits . tails
{concat/map}                    =
                                maximum . concat . map (map sum) . map inits . tails
{map/.}                         =
                                maximum . concat . map (map sum . inits) . tails
{ bookkeeping }                 =
                                maximum . map maximum . map (map sum . inits) . tails
{map/.}                         =
                                maximum . map (maximum . map sum . inits) . tails
{map/sum}                       =
                                maximum . map (maximum . scanl (+) 0) . tails
{ Def. maximum }                =
                                maximum . map (foldr1 max . scanl (+) 0) . tails
{ foldr Fusion }                =
                                maximum . map (foldr (⊙) 0) . tails
                                              where x⊙y = max 0 (x+y)
{ map/foldr }                   =
                                maximum . scanr (⊙) 0   where x⊙y = max 0 (x+y)
```

Akkumulierung der Zwischenergebnisse

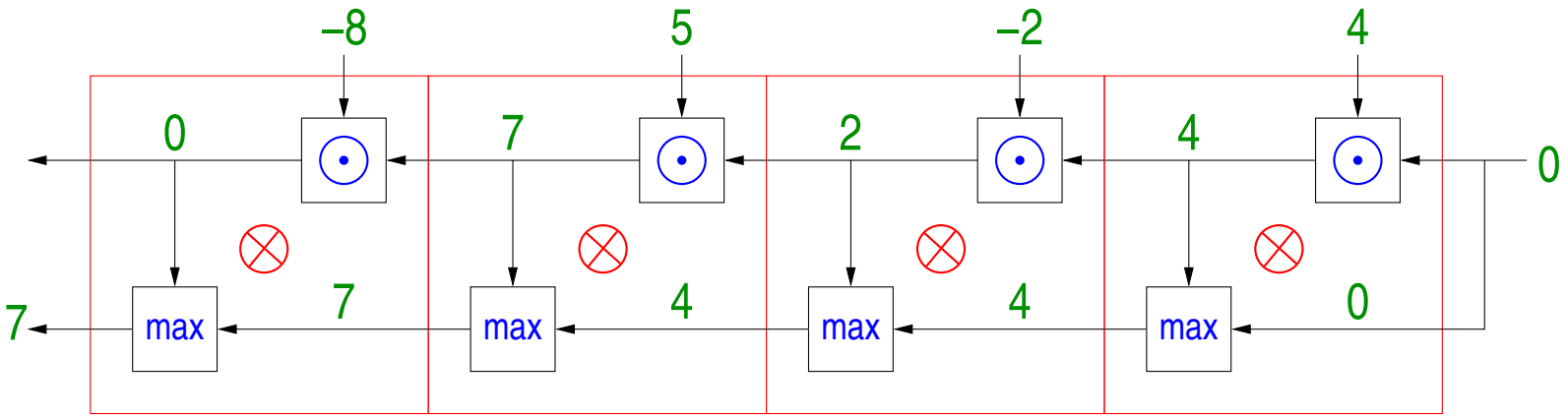
Def.: `x ⊙ y = max 0 (x+y)`

```
                    mss
  { vorige Folie }  =
                    maximum . scanr (⊙) 0
{ Def. maximum }    =
                    foldr1 max . scanr (⊙) 0
      { Tupling }   =
                    snd . foldr (⊗) (0,0)
```



Akkumulierung der Zwischenergebnisse

```
mss = snd . foldr (⊗) (0,0)
```




```
x ⊗ (acc,res) = let acc' = max 0 (x+acc)     -- x ⊙ acc
                in (acc', max res acc')
```

Interpretation des Akkumulatorinhalts: Wert des maximalen initialen Segments

Anwendung von Homomorphieeigenschaften

1. `segs = concat . map inits . tails`:

```
mss = let x ⊗ (acc,res) = let acc' = max 0 (x+acc)
                          in (acc', max acc' res)
      in snd . foldr (⊗) (0,0)
```

   Akkumulator: Wert des maximalen initialen Segments (`mis`)

2. `segs = concat . map tails . inits`:

```
mss = let (acc,res) ⊕ x = let acc' = max 0 (acc+x)
                          in (acc', max acc' res)
      in snd . foldl (⊕) (0,0)
```

   Akkumulator: Wert des maximalen finalen Segments (`mfs`):

`mss` ist eine Projektion einer linksgerichteten und rechtsgerichteten Listenfunktion.

Dritter Homomorphiesatz: `mss` ist Projektion eines Homomorphismus `h`.

Suche nach dem Kombinationsoperator $\circledast$: `h (x++y) == h x ⊛ h y`

Suche nach dem Kombinationsoperator $\circledast$

Der Homomorphismus muss enthalten:

1. maximale Segmentsumme `mss`
2. Summe des maximalen initialen Segments `mis`
3. Summe des maximalen finalen Segments `mfs`

Problem: wie erhält man aus x und y die Werte für $x \circledast y$?

1. $\texttt{mss}_{x \circledast y} = \texttt{max (max mss}_x\ \texttt{mss}_y\texttt{) (mfs}_x\ \texttt{+ mis}_y\texttt{)}$
2. $\texttt{mis}_{x \circledast y} = \texttt{max mis}_x\ \texttt{(sum}_x\ \texttt{+ mis}_y\texttt{)}$
3. $\texttt{mfs}_{x \circledast y} = \texttt{max mfs}_y\ \texttt{(sum}_y\ \texttt{+ mfs}_x\texttt{)}$

zusätzliches Tupling mit der Summe aller Werte nötig:

4. $\texttt{sum}_{x \circledast y} = \texttt{sum}_x\ \texttt{+ sum}_y$

mss als Projektion eines Homomorphismus

```
mss :: [Integer] -> Integer
mss = prj1aus4 . foldr g e . map f
       where e = (0,0,0,0)
             g (mssX,misX,mfsX,sumX)
               (mssY,misY,mfsY,sumY) = (max (max mssX mssY)
                                            (mfsX + misY),
                                        max misX (sumX + misY),
                                        max mfsY (sumY + mfsX),
                                        sumX + sumY)
             f x = let xnn = max 0 x
                   in (xnn,xnn,xnn,x)
             prj1aus4 (a,b,c,d) = a
```

Übungsaufgabe: Erweiterung, so dass auch Segmentgrenzen geliefert werden

mss als Divide-and-Conquer

```
mss :: [Integer] -> Integer
mss [] = 0
mss xs = prj1aus4 (dc pred basic divide combine xs) where
 pred xs    = length xs == 1
 basic [x] = let xnn = max 0 x
             in (xnn,xnn,xnn,x)
 divide xs = let (ys,zs) = splitAt (length xs `div` 2) xs
             in [ys,zs]
 combine _ [(mssX,misX,mfsX,sumX),
            (mssY,misY,mfsY,sumY)] = (max (max mssX mssY)
                                          (mfsX + misY),
                                      max misX (sumX + misY),
                                      max mfsY (sumY + mfsX),
                                      sumX + sumY)
 prj1aus4 (a,b,c,d) = a
```

Beobachtungen für `mss`

- Laufzeitreduktion von $\Theta(n^3)$ (Spezifikation)
  - sequenziell: auf $\Theta(n)$
  - parallel: durch D&C auf $\Theta(\log\ n)$
- wichtige Gesetze/Techniken
  - bookkeeping: vermeidet temporäre Listen
  - fold-Fusion und map-Komposition: vermeidet Mehrfachdurchgänge
  - Tupling: erhöht die Flexibilität der verknüpfenden Operatoren
    - Zwischenergebnisse für Fold-Fusion
    - Assoziativität für Homomorphismen