Chapter 4: Proof and Synthesis of Programs

Learning targets of this chapter:

1. Significance of referential transparency and equational reasoning
2. Structural induction: natural numbers, lists, trees
3. List laws: decomposition, duality, fusion, homomorphy
4. Homomorphy in the synthesis of algorithms: Mergesort, Fibonacci
5. Program transformations for performance optimization:
   Maximum Segment Sum

Referential Transparency (1)

Def.: An expression E is called referentially transparent, if every subexpression T of E can be replaced by an expression T' with equal value, without effect on the value of E.

Referential transparency implies the Rule of Leibniz: $\boxed{\dfrac{T = T'}{E[x:=T] = E[x:=T']}}$

Example:

```
f x + f x                           = {non-strict let}
let  y = f x  in  f x + f x         = {Leibniz}
let  y = f x  in  y + f x           = {Leibniz} crucial point
let  y = f x  in  y + y             = {arithmetic of integer variables}
let  y = f x  in  2 * y             = {Leibniz}
let  y = f x  in  2 * f x           = {non-strict let}
2 * f x
```

Referential Transparency (2)

- Language with side-effects $\Longrightarrow$ no referential transparency;
  z. B.: `f x + f x` $\neq$ `2 * f x`, cf. Chap. 1
- Referentially transparent functional languages are called purely functional.
- Haskell permits input/output with referential transparency: the meaning of an I/O operation is not the value returned but the operation per se.

Haskell	OCaml

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

output: `haha`	output: `ha`

Proofs of Programs

- Given:
  1. specification, i.e., input/output relation
  2. program
- Goal: proof that the program satisfies the specification

Proofs of Imperative Programs

Established method: Hoare calculus (assertion method)

- Assertions about the program state (the values of the variables)
- Specification: pre- and postcondition for the entire program
- Proof outline: pre- and postcondition for every program part
  - loop invariants
  - backward-substitution rule for assignments

Proof of Functional Programs

Established method: equational reasoning (proofs with equations)

Specification: input/output relation, what remains to be done?

1. Specification need not be executable, e.g.,
   - equations are not always permissible function definitions
   - conditions that cannot be expressed in Haskell
2. Specification ist executable but grossly inefficient
   - program has equal semantics but more efficient execution

Equational proof: stepwise transformation of the specification to the target program, governed by transformation rules (e.g., Leibniz)

Principle of Equational Reasoning

- Sequence of equations with a left and a right expression
- Each equation is justified by application of a proof rule

Procedure to justify equation $P = Q$:

1. Specify a context E with variable v, such that there are expressions L' and R' with the properties $E[v\mathtt{:=}L'] = P$ and $E[v\mathtt{:=}R'] = Q$
2. Choose a suitable equation $L = R$ (or $R = L$) from the proof rule base
3. Specify a substitution(*) φ, such that $L' = \varphi\, L$ and $R' = \varphi\, R$

(*) Substitution $\varphi\, T$ is purely syntactic and simultaneous: for every free variable x in T, replace every free instance of x in T with a fixed expression $(\varphi\, x)$

Example of a Rule Application

Given:

- $P =$ `f (x * (y + g z)) - 1`
- $Q =$ `f (x * y + x * g z) - 1`.
- In the rule base: `(a * (b + c))` $=$ `a * b + a * c` (distributivity of `*` over `+`)

To prove: $P = Q$

1. Specify a context E with variable v, such that there are expressions L' and R', with the properties $E[v\texttt{:=}L'] = P$ and $E[v\texttt{:=}R'] = Q$

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

2. Choose a suitable equation $L = R$ (or $R = L$) from the proof rule base

   choose distributivity: L = `a * (b + c)`, R = `a * b + a * c`

3. Specify a substitution φ, such that $L' = \varphi\, L$ and $R' = \varphi\, R$

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

Proof Outline

Only the name of the rule or an intuitive explanation are stated.

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

Example (proof outline):

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

Correctness of Equational Reasoning

Haskell has the following important properties:

1. Referential transparency
   Each equation maintains partial correctness (correctness modulo termination).
2. Non-strictness of function application (laziness)
   Termination behavior remains unchanged
   - when using a defining equation as proof rule (e.g., `fac.2`)
   - when changing the substitution sequence
3. Static type system
   Each expression has a well-defined semantics.
   - Equalities like `(/2).(*2)` = `id` = ``(`div` 2).(*2)`` do not hold.
   - Rules are type-dependent: `+` is associative for `Rational`, but not for `Float`.

Varying the Substitution Sequence (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
```

Evaluate the argument first

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

Varying the Substitution Sequence (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
```

Evaluate the defining equation first

`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`	

Structural Induction

Here: finite structures, no consideration of laziness

The natural numbers as (structurally) inductively defined set:

```
data Nat = Z                          zero
         | S Nat                      successor
         deriving (Eq,Ord,Show)
```

Elements of `Nat`: `Z` (0), `S Z` (1), `S (S Z)` (2), `S (S (S Z))` (3) ...

Addition on `Nat`

Definition of functions on inductively defined types by specification of one equation per data constructor

```
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) $\rightsquigarrow$ S (S Z)

add' 1 1 $\rightsquigarrow$ 2

Multiplication on 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

Stepwise Evaluation

```
          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)))))
```

The Inductive Data Type List

```
data List a = Nil                 empty list
            | C a (List a)        cons
```

In Haskell (not legal source program syntax)

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

Elements:	Nil	[]
	C x Nil	x:[]
	C y (C x Nil)	y:x:[]
	C z (C y (C x Nil))	z:y:x:[]
	...	

The Length Function on Lists

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

len' :: [a] -> Int    -- Unlimited integer not needed
                      -- since memory smaller than address space
len' []       = 0              -- len'.1
len' (_:xs)   = 1 + len' xs    -- len'.2
```

`len (C z (C y (C x Nil)))` $\rightsquigarrow$ `S (S (S Z))`

`len' (z:y:x:[])` $\rightsquigarrow$ `3`

The Sum Function on Lists

```
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

Inductive Definition of List Concatenation

append `(++)` joins two lists

```
(++) :: [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:[]
```

Inductive Definition of Combinator `map`

```
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 : []
```

Structurally Inductive Proof (informal)

- Axiom of induction: (to prove an assertion P)
  for every data constructor C of an inductively defined type S:
  1. Let x be an arbitrary element of S with root symbol C
  2. Induction hypothesis: assume $P(x_i)$ for every strict substructure x_i of x
  3. Prove $P(x)$

  $\Longrightarrow$ P holds for all finite elements of S
- Base case $\Longleftrightarrow$ no x_i contains an element of type S
- Induction case $\Longleftrightarrow$ at least one x_i contains an element of type S
- Problem: choice of induction variable x in $P(x)$
- Special cases:
  - mathematical induction: constructors `Z` (0) and `S` ($x \to x{+}1$)
  - list induction: constructors `[]` (nil) and `(:)` (cons)

Axiom of Structural Induction (semiformal)

- Let S be an algebraic data type with constructors $\{\, C_j \mid j \in \mathbb{N} \wedge 0 \leq j < n \,\}$
- Let $\#j$ be the number of arguments of constructor C_j
- Let P be the predicate to be proved
- Induction axiom

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

- Mathematical induction as special case

$$\frac{P(\mathtt{Z}) \quad \wedge \quad (\forall x : x \in \mathtt{Nat} : \ P(x) \implies P(\mathtt{S}\ x))}{\forall x \ : \ x \in \mathtt{Nat} \wedge x \text{ finite} \ : \ P(x)}$$

Proof: `map f xs ++ map f ys == map f (xs++ys)` {Case.1}

Induction variable: `xs`

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

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

Proof: `map f xs ++ map f ys == map f (xs++ys)` {`Case.2`}

Induction variable: `xs`

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

Induction hypothesis: `map f zs ++ map f ys == map f (zs++ys)`

```
             map f xs ++ map f ys
{ Case.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.hyp.}== f z : (map f (zs++ys))
{  map.2 }== map f (z:(zs++ys))
{ (++).2 }== map f ((z:zs)++ys)
{ Case.2 }== map f (xs++ys)
```

Proof: `(map f . map g) xs == map (f . g) xs` {Case.1}

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

Induction variable: `xs`

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

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

Proof: `(map f . map g) xs == map (f . g) xs` {Case.2}

Induction variable: `xs`

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

Induction hypothesis: `(map f . map g) zs == map (f . g) zs`

```
               (map f . map g) xs
{ Case.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.hyp.}== (f . g) z : map (f . g) zs
{  map.2 }== map (f . g) (z:zs)
{ Case.2 }== map (f . g) xs
```

Use of Predefined Arithmetic

- Constructor-based evaluation: slow and inflexible
- Better: use built-in arithmetic
- Example: factorial

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

- Attention: consider value range (termination)
- Inductive proofs analogous with `0` for `Z` and `(+1)` for `S`

Trees as Inductive Data Types

```
data BinTree1 a    -- only forks have a payload
   = Empty1
   | Fork1 { elem1::a, left1, right1 :: BinTree1 a }

data BinTree2 a    -- only leaves have a payload
   = Leaf2 { elem2::a }
   | Fork2 { left2, right2 :: BinTree2 a }

data BinTree3 a    -- forks and leaves have a payload...
   = Leaf3 { elem3::a }
   | Fork3 { elem3::a, left3, right3 :: BinTree3 a }

data RoseTree a    -- ...plus arbitrary finite number of successors
   = Fork4 { elem4::a, children::[RoseTree a] }
```

Sum Function sumT on 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
```

Proof: `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
```

Proof: `leaves t == forks t + 1` {Case.1}

Induction variable: `t`

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

```
                 leaves t
{  Case.1   }== leaves (Leaf2 _)
{ leaves.1  }== 1
{ neutr. +  }== 0 + 1
{  forks.1  }== forks (Leaf2 _) + 1
{  Case.1   }== forks t + 1
```

Proof: `leaves t == forks t + 1` {Case.2}

Induction variable: `t`

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

Induction hypothesis: `leaves l == forks l + 1`, `leaves r == forks r + 1`

```
               leaves t
{ Case.2 }==   leaves (Fork2 l r)
{leaves.2}==   leaves l + leaves r
{ ind.hyp. }== forks l + 1 + leaves r
{ commut.}==   1 + forks l + leaves r
{ ind.hyp. }== 1 + forks l + forks r + 1
{ forks.2}==   forks (Fork2 l r) + 1
{ Case.2 }==   forks t + 1
```

InEquational Proof: `sumT t <= leaves t * maxT t`

sequence of `<=` and `==` for sequence of `==`

```
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
```

InEquational Proof: `sumT t <= leaves t * maxT t` {Case.1}

Induction variable: `t`

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

```
              sumT t
{ Case.1 }== sumT (Leaf2 x)
{ sumT.1 }== x
{neutr. *}== 1 * x
{leaves.1}== leaves (Leaf2 x) * x
{ maxT.1 }== leaves (Leaf2 x) * maxT (Leaf2 x)
{ Case.1 }== leaves t * maxT (Leaf2 x)
{ Case.1 }== leaves t * maxT t
```

InEquational Proof: `sumT t <= leaves t * maxT t` {Case.2}

Induction variable: `t`

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

```
                sumT t
{ Case.2 }==    sumT (Fork2 l r)
{ sumT.2 }==    sumT l + sumT r
{ind.hyp.}<=    leaves l * maxT l + sumT r
{ind.hyp.}<=    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)
{ Case.2 }==    leaves t * maxT (Fork2 l r)
{ Case.2 }==    leaves t * maxT t
```

Observations

- Multiple use of the induction hypothesis may be useful or necessary
  - in case of structures with multiple recursive parts (e.g., trees)
  - in case of multiple instances of the induction variable
- Often one can identify a sequence of equations quickly if one expands function definitions
  1. top-down towards the middle and
  2. bottom-up towards the middle

  and then simplifies.
- In case of structures with one base case and one induction case,
  the `{.1}`-rules are usually used in the proof of the base case
  and the `{.2}`-rules in the proof of the induction case.

List Laws and Fold

Attention: overall restriction to finite lists, not explicitly mentioned

- Decomposition laws: deconstruction of a list
- Duality laws: relation between `foldl` and `foldr`
- Fusion laws: merge of evaluation following `fold` with evaluation of `fold`
- Homomorphism laws: when is a function a list homomorphism?

Definitions

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

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

Fold-Decomposition Laws

- (`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`

If `f` associative with neutral 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)`

Examples: exercise

First Duality Law (1)

Let `f` be associative with neutral element `e`.
Then, for every finite list `xs`: `foldr f e xs` = `foldl f e xs`

Proof by induction on the structure of `xs`

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

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

First Duality Law (2)

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

```
                foldr f e xs
{  Case.2   }== foldr f e (y:ys)
{ foldr.2   }== f y (foldr f e ys)
{ ind.hyp.  }== f y (foldl f e ys)
{side proof}== 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)
{  Case.2   }== foldl f e xs
```

First Duality Law, Side Proof (1) unsuccessful!

To prove: `f y (foldl f e ys)` = `foldl f (f y e) ys`

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

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

First Duality Law, Side Proof (2) unsuccessful!

To prove: `f y (foldl f e ys)` = `foldl f (f y e) ys`

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

```
                 f y (foldl f e ys)
{ Case.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)
{ Case.2  }== foldl f (f y e) ys
```

In general, `z` is not the neutral element.
Induction hypothesis must be generalized!

First Duality Law, Side Proof (1) successful

To prove: `f y (foldl f x ys)` = `foldl f (f y x) ys`

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

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

First Duality Law, Side Proof (2) successful

To prove: `f y (foldl f x ys)` = `foldl f (f y x) ys`

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

```
               f y (foldl f x ys)
{ Case.2   }== f y (foldl f x (z:zs))
{ foldl.2  }== f y (foldl f (f x z) zs)
{ ind.hyp. }== 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)
{ Case.2   }== foldl f (f y x) ys
```

Second Duality Law (1)

$\boxed{\texttt{foldr f x xs} = \texttt{foldl (flip f) x (reverse xs)}}$

Proof by induction on the structure of `xs`

<u>{Case.1}: `xs == []`</u>

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

Second Duality Law (2)

{Case.2}: `xs == y:ys`

```
               foldr f x xs
{ Case.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.hyp.  }== 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))
{ Case.2    }== foldl (flip f) x (reverse xs)
```

Use of the Second Duality Law

synthesis of an efficient implementation of `reverse`

```
              reverse xs                      specification, see below
{   id    }== id (reverse xs)
{foldr/id}== foldr (:) [] (reverse xs)
{ 2.Dual }== foldl (flip (:)) [] (reverse (reverse xs))
{rev/rev }== foldl (flip (:)) [] xs   implementation in linear time
```

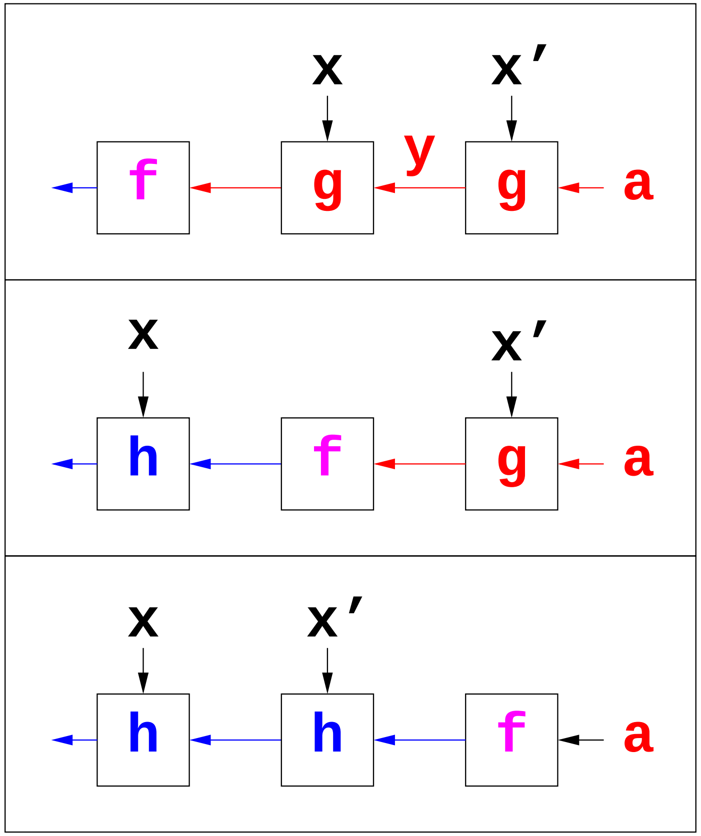
The second duality law uses the specification of `reverse` of quadratic time!

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

Fusion Laws (1: `foldr`-Fusion)

Let `f` be strict and assume `f (g x y) = h x (f y)`.

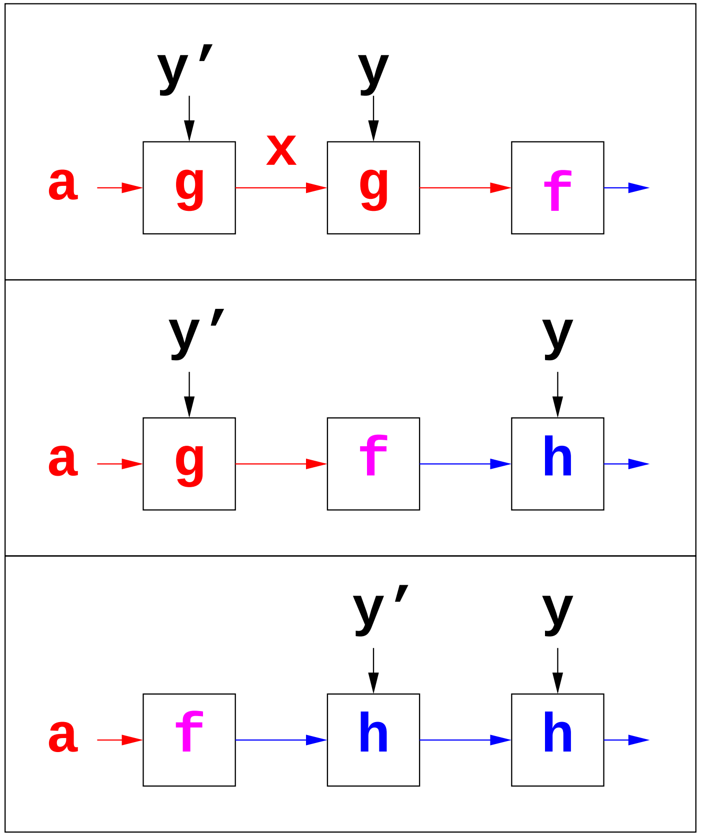
Then: $\boxed{\texttt{f . foldr g a} = \texttt{foldr h (f a)}}$



Fusion Laws (2: `foldl`-Fusion)

Let `f` be strict and assume `f (g x y) = h (f x) y`.

Then: $\boxed{\texttt{f . foldl g a} = \texttt{foldl h (f a)}}$



Further Fusion Laws

- fold/map-Fusion

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

- Book-keeping law: for associative `f` with neutral element `e`

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

  Instance of book-keeping (`f`=`(+)`, `e`=`0`):

  `sum . concat = sum . map sum`

Monoid

Monoid: semigroup with neutral element, notation $(M, \oplus, e)$

- M set
- $(\oplus)$: $M \to M \to M$ total function, $\oplus$ associative
- e neutral element of $\oplus$

Haskell examples:

- (`Integer`,`+`,`0`), natural numbers with addition
- (`[a]`,`++`,`[]`), (finite) lists with append
- (`a->a`, `.` ,`id`), functions with composition

Monoid Homomorphism

- Let $\underline{M_1} = (M_1, \oplus_1, e_1)$ and $\underline{M_2} = (M_2, \oplus_2, e_2)$ be monoids
- Monoid homomorphism: mapping $h : M_1 \to M_2$ with the properties
  1. $h(e_1) = e_2$
  2. $\forall m, m' \in M_1 : h(m \oplus_1 m') = h(m) \oplus_2 h(m')$

List Homomorphisms

Examples of homomorphisms from $\underline{M_1} = (\texttt{[a]}, \texttt{++}, \texttt{[]})$ to $\underline{M_2} = (\texttt{Integer}, \texttt{+}, \texttt{0})$:

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

There are many homomorphisms from $\underline{M_1}$ to $\underline{M_2}$.
Characterization: by the generating set

Generating Set

- Let $\underline{M} = (M, \oplus, e)$ be a monoid
- Generating set: subset $A \subseteq M$, such that every $m \in M$ can be constructed by repeated application of $\oplus$ from elements of A and e.

- Let $\underline{M_1}$ = (`[a]` , `++`, `[]`)
- Theorem: the one-element lists form a generating set for $\underline{M_1}$
(Proof: Peter Thiemann, *Grundlagen der funktionalen Programmierung*, Teubner, 1994, Chap. 6.1.)

Examples of List Homomorphisms

Examples of homomorphisms from $\underline{M_1} = ($`[a]`$,$ `++`$,$ `[]`$)$ to $\underline{M_2} = (M_2, \oplus, e)$:

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` is associative with neutral element `e`

First Homomorphism Law

Original: Richard Bird, 1988

Function `h::[a]->b` is a homomorphism

$\Longleftrightarrow$

there are

- one associative function `g` with neutral element `e` and
- one function `f` with the property `f x = h [x]`,

such that `h` can be written: `h = foldr g e . map f`

Definitions

A list function h is called

- ⊕-left-momomorphic for a binary operator $\oplus$ if and only if for all elements a and lists y: $h\,([a] \mathbin{+\!\!+} y) = a \oplus h\,y$
- ⊗-right-homomorphic for a binary operator $\otimes$ if and only if for all lists x and elements a: $h\,(x \mathbin{+\!\!+} [a]) = h\,x \otimes a$

Note: $\oplus$ and $\otimes$ need not be associative.

Second Homomorphism Law (Specialization Law)

Original: Richard Bird, 1987

Every list homorphism can be expressed as left-homomorphic and also as right-homomorphic list function, i.e.,

- Let mapping h from $([\alpha], +\!\!+, [\,])$ to $(\beta, \circledast, e)$ be a homomorphism.
- Then there exist (with $f\,a := h\,[a]$):
  1. an operator $\oplus$ with $a \oplus y \;=\; f\,a \circledast y$, such that $h \;=\; \mathit{foldr}\,(\oplus)\;e$,
  2. an operator $\otimes$ with $x \otimes b \;=\; x \circledast f\,b$, such that $h \;=\; \mathit{foldl}\,(\otimes)\;e$.

Third Homomorphism Law

due to Jeremy Gibbons, 1995

$\boxed{\text{If } h \text{ is left- and right homomorphic, then it is a homomorphism.}}$

To prove: If h is left- and right-homomorphic, then there is an operator $\odot$ such that $h\,(x \mathbin{+\!\!+} y) = h\,x \odot h\,y$. $\odot$ is associative since $\mathbin{+\!\!+}$ is associative.
We need an explicit definition of $\odot$, i.e., a function g with

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

Idea: g yields representatives in the domain of h that are equivalent to x and y.

$$h\,x \odot h\,y = h\,\big(g\,(h\,x) \mathbin{+\!\!+} g\,(h\,y)\big) = h\,(x \mathbin{+\!\!+} y)$$

Side Lemma 1 for the Proof of the Third Homomorphism Law

Side lemma 1.
For every computable function h with enumerable domain, there is a computable (possibly non-total) function g such that: $h \circ g \circ h = h$.

Proof. To compute $g\,t$, one can enumerate the domain of h and return the first x with: $h\,x = t$. This procedure terminates if t falls inside the range of h.

Side Lemma 2 for the Proof of the Third Homomorphism Law

Side Lemma 2:

List function h is a homomorphism if and only if for all lists v, w, x und y:

$h\,v = h\,x \;\wedge\; h\,w = h\,y \implies h\,(v +\!\!+ w) = h\,(x +\!\!+ y)$.

Proof:

- h is homomorphism $\Rightarrow \exists \oplus :: h(v +\!\!+ w) = h\,v \oplus h\,w = h\,x \oplus h\,y = h(x +\!\!+ y)$
- Assume $h\,v = h\,x \;\wedge\; h\,w = h\,y \implies h\,(v +\!\!+\; w) = h\,(x +\!\!+\; y)$ (*)
  - choose g such that $h \circ g \circ h = h$ (Side Lemma 1)
  - define $\odot$ as $t \odot u = h\,(g\,t +\!\!+ g\,u)$

$$
\begin{array}{ll}
h\,x \odot h\,y & = \{\text{definition of } \odot\} \\
h\,(g\,(h\,x)\;+\!\!+\;g\,(h\,y)) & = \{\text{ let } v = g\,(h\,x) \text{ and } w = g\,(h\,y)\ \} \\
h\,(v +\!\!+ w) & = \{\ h\,v = h\,x \wedge h\,w = h\,y \wedge (*)\ \} \\
h\,(x +\!\!+ y) &
\end{array}
$$

Thus: h is homomorphism with operator $\odot$ (neutral element: $h\,[\,]$).

Proof of the Third Homomorphism Law

To prove: h is left- and right-homomorphic $\implies$ $h$is homomorphism

- Assume: h is left- and right-homomorphic: $h = \mathit{foldr}\,(\oplus)\ e\ =\ \mathit{foldl}\,(\otimes)\ e$
- To prove: h is homomorphism
- Show the according to Side Lemma 2 equivalent property:
  $h\,v = h\,x \wedge h\,w = h\,y \implies h(v +\!\!+\ w) = h(x +\!\!+ y)$
  - Assume $h\,v = h\,x\ \wedge\ h\,w = h\,y$
  - To prove: $h\,(v +\!\!+\ w) = h\,(x +\!\!+\ y)$ [next slide]

<u>Proof:</u>

$$
\begin{array}{ll}
h\,(v +\!\!+\, w) & \\
= & \{\ h \text{ is left-homomorphic }\} \\
\mathit{foldr}\,(\oplus)\ e\ (v +\!\!+\, w) & \\
= & \{\ \mathit{foldr}\text{-decomposition }\} \\
\mathit{foldr}\,(\oplus)\ (\mathit{foldr}\,(\oplus)\ e\ w)\ v & \\
= & \{\ h\,w = h\,y\ \} \\
\mathit{foldr}\,(\oplus)\ (\mathit{foldr}\,(\oplus)\ e\ y)\ v & \\
= & \{\ \mathit{foldr}\text{-decomposition }\} \\
\mathit{foldr}\,(\oplus)\ e\ (v +\!\!+\, y) & \\
= & \{\ h \text{ is left-homomorphic }\} \\
h\ (v +\!\!+\, y) & \\
= & \{\ h \text{ is right-homomorphic }\} \\
\mathit{foldl}\,(\otimes)\ e\ (v +\!\!+\, y) & \\
= & \{\ \mathit{foldl}\text{-decomposition }\} \\
\mathit{foldl}\,(\otimes)\ (\mathit{foldl}\,(\otimes)\ e\ v)\ y & \\
= & \{\ h\,v = h\,x\ \} \\
\mathit{foldl}\,(\otimes)\ (\mathit{foldl}\,(\otimes)\ e\ x)\ y & \\
= & \{\ \mathit{foldl}\text{-decomposition }\} \\
\mathit{foldl}\,(\otimes)\ e\ (x +\!\!+\, y) & \\
= & \{\ h \text{ is right-homomorphic }\} \\
h\ (x +\!\!+\, y) &
\end{array}
$$

Benefits of Homomorphisms

Mathematical:

Homomorphisms are structure-preserving functions. One can operate in either the domain monoid or the range monoid (commuting diagram).

For programming:

Homomorphisms support component-based software composition and generic software.

For performance:

Homomorphisms help to avoid intermediate data structures and to gain parallelism. The corresponding optimizations can be identified and applied by the compiler.

Use: Synthesis of a Sorting Algorithm

Insertion into a sorted list: `ins`

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

ins = flip ins'
```

Sorting based on insertion:

- left-homomorphic: `sort' = foldr ins' []`
- right-homomorphic: `sort = foldl ins []`

Consequence of the third homomorphism law: `sort` is a homomorphism.

Synthesis of a Sorting Algorithm

- Both the left- and right-homomorphic insertion sort require execution time in $\Theta(n^2)$ for a list of length n.
- Goal: solution in which the list is divided into two approx. equally sized parts to reach an execution time in $\Theta(n \cdot \log n)$.
- Plan: synthesis of a sorting algorithm that
  - permits the division of an unsorted lists into two arbitrary non-empty parts
  - contains the left- and right-homomorphic insertion sort as special cases
- Specify the implementation of operator $\odot$, such that:

  `sort xs` $\odot$ `sort ys == sort (xs ++ ys)`.
- `sort` is a homomorphism from (`[a]`,`++`,`[]`) to (`[a]`$_{\text{sorted}}$,$\odot$,`[]`).

Synthesis of a Sorting Algorithm

	`u ⊙ v`
{ `u`, `v` sorted }	=
	`sort u ⊙ sort v`
{ `sort` is homomorphism }	=
	`sort (u ++ v)`
{ `sort` }	=
	`foldl ins [] (u ++ v)`
{ `foldl`-decomposition }	=
	`foldl ins (foldl ins [] u) v`
{ `sort` }	=
	`foldl ins (sort u) v`
{ `u` sorted }	=
	`foldl ins u v`
{ set `mergeQ` = `foldl ins` (*) }	=
	`mergeQ u v`

(*) Named `mergeQ` because of the following property of `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
```

Synthesis of a Sorting Algorithm

```
u ⊙ v = mergeQ u v
```

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

- `mergeQ` exploits the property that `u` is sorted
- `mergeQ` does not exploit that `v` is sorted
  $\Rightarrow$ still quadratic execution time!!

Improvement of `mergeQ` to `merge`:
rewrite and exploit the sortedness of `v`

Improvement of `mergeQ` to `merge`

Shorthand: let a be an element and v a list:

$a \leq v$ means: $a \leq b$ for all b in v

Side lemma:

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

Proof: by induction

Improvement of `mergeQ` to `merge`

- Second list empty

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

- First list empty

	`merge [] v`
{ `merge == mergeQ` }	=
	`mergeQ [] v`
{ def. `mergeQ` }	=
	`foldl ins [] v`
{ `sort` }	=
	`sort v`
{ `v` ist sorted }	=
	`v`

Improvement of `mergeQ` to `merge`

- Both lists not empty

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

continue with case distinction `a` $<$ `b` and `a` $\geq$ `b`

Improvement of `mergeQ` to `merge` ($a < b$)

Assume: (`a:u`) and (`b:v`) sorted and `a` $<$ `b`

thus: `a` $\leq$ `u`, `a` $\leq$ `v` and `a` $\leq$ (`ins u b`)

	`merge (a:u) (b:v)`
{ steps of previous slide }	=
	`foldl ins (ins (a:u) b) v`
{ `ins` ∧ `a<b` }	=
	`foldl ins (a : ins u b) v`
{ side lemma }	=
	`a : foldl ins (ins u b) v`
{ `foldl.2` }	=
	`a : foldl ins u (b:v)`
{ def. `mergeQ` }	=
	`a : mergeQ u (b:v)`
{ `merge == mergeQ` }	=
	`a : merge u (b:v)`

Improvement of `mergeQ` to `merge` ($a \geq b$)

Assume: (`a:u`) and (`b:v`) sorted and `a` $\geq$ `b`
thus: `b` $\leq$ (`a:u`) and `b` $\leq$ `v`

	`merge (a:u) (b:v)`
{ steps of next to previous slide }	=
	`foldl ins (ins (a:u) b) v`
{ `ins` $\wedge$ `a`$\geq$`b` }	=
	`foldl ins (b:a:u) v`
{ side lemma }	=
	`b : foldl ins (a:u) v`
{ def. `mergeQ` }	=
	`b : mergeQ (a:u) v`
{ `merge == mergeQ` }	=
	`b : merge (a:u) v`

Improvement of `mergeQ` to `merge`, Summary

This yields for `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
```

. . . and the two insertions sorts are special cases:

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

Resulting Sorting Algorithm: mergesort

```
mergesort :: Ord a => [a] -> [a]
mergesort []  = []                            -- neutral element
mergesort [x] = [x]                           -- generating set
mergesort xs  = let (ys,zs) = splitAt (length xs `div` 2) xs
                    u = mergesort ys
                    v = mergesort zs
                in merge u v                  -- operator in the domain
                                              -- of the homomorphism

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
```

Program Transformations

Seminal work by Rod Burstall / John Darlington (1977):
"A transformation system for developing recursive programs"

Use: transform an inefficient to an efficient program.

Types of transformations:

- unfold: replace the instance of a function name by its body
- fold: reverse of unfold
- def: introduce a local definition (`let`/`where`)
- spec: specialize

Example: Fibonacci Numbers

Starting point: an extremely inefficient program

```
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-improving transformations

(1) Introduce a local definition:

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

(2) Define an auxiliary function for the right side of `fib (n+2)`

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

(3) Specialize

(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
```

Thus:

```
               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)
```

or, with `foldl` in place of the recursion

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

Homomorphism Properties of `fib`

- Goal: optimization of the evaluation of `foldl`
- Represent `n` as a list of () of length n
- Homomorphism $h : ([()], +\!\!+, []) \to (\mathbb{N}^2 \to \mathbb{N}^2, \circ, id)$

```
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)
```

Further Improvements

- Explicit function composition is memory-intensive.
- Find algebraic ways of compacting the composition.

We know:

- `f` is a linear function $\texttt{f}(z_1, z_2) = (z_2, z_2{+}z_1) = \left(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} \right)^T$.
- The composition $(g \circ f)$ of linear functions corresponds to matrix multiplication.

Next step: conversion of operator `foldr` to matrix multiplication.

Introduction of a Type of 2x2-Matrices

```
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 Computation as Matrix Multiplication

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

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

Further Optimization: Efficient Computation of Power

```
fibOpt :: Integer -> Integer
fibOpt n = upperRight (fibMat^n)
```

Execution times (obtained with ghci in seconds, ignoring output)

n	binary rec.	linearly rec.	`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

Maximum Segment Sum

Goal: maximum of the sums of all segments in a list of integer numbers

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

Imperative problem solutions (see Jon Bentley: Programming Pearls)

- compute of all segments: $\Theta(n^3)$
- update partial sums incrementally: $\Theta(n^2)$
- divide and conquer: $\Theta(n \cdot \log n)$
- scan algorithm: $\Theta(n)$

Here: formal synthesis

- functional scan algorithm $\Theta(n)$

Maximum Segment Sum

Specification:

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

- `tails`: final segments, e.g., `tails [1,2,3]` $\leadsto$ `[[1,2,3],[2,3],[3],[]]`
- `inits`: initial segment, e.g.,: `map inits (tails [1,2,3])` $\leadsto$ `[[[],[1],[1,2],[1,2,3]], [[],[2],[2,3]], [[],[3]], [[]]]`
- `segs`: all segments,
  e.g., `segs [1,2,3]` $\leadsto$ `[[],[1],[1,2],[1,2,3],[],[2],[2,3],[],[3],[]]`

Reduction of the Complexity from $\Theta(n^3)$ to $\Theta(n)$

```
                     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
{book-keeping}       =
                     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/scanl}        =
                     maximum . map (foldr (⊙) 0) . tails

                                 where x⊙y = max x (x+y)
{map/foldr}          =
                     maximum . scanr (⊙) 0   where x⊙y = max x (x+y)
```

Fusion of `foldr1` and `scanl`

Let $\oplus$ be associative, $\otimes$ associative, $\mathtt{e}$ neutral wrt. $\otimes$ and $\oplus$ left-distributive over $\otimes$ (i.e., $\mathtt{x} \oplus (\mathtt{y} \otimes \mathtt{z}) = (\mathtt{x} \oplus \mathtt{y}) \otimes (\mathtt{x} \oplus \mathtt{z})$). Then:

$$\mathtt{foldr1} \oplus \mathtt{.}\ \mathtt{scanl} \otimes \mathtt{e} = \mathtt{foldr} \odot \mathtt{e}$$

with $\mathtt{x} \odot \mathtt{y} = \mathtt{x} \otimes (\mathtt{e} \oplus \mathtt{y})$.

Example of maximum segment sum:
`(+)` distributes over `max`, thus:

$$\mathtt{foldr1\ max\ .\ scanl\ (+)\ 0 = foldr}\ (\odot)\ \mathtt{0}$$

with $\mathtt{x} \odot \mathtt{y} = \mathtt{x + max\ 0\ y = max\ x\ (x+y)}$.

$$
\begin{aligned}
& (\mathtt{foldr1}\ (\oplus)\ \mathtt{.}\ \mathtt{scanl}\ (\otimes)\ \mathtt{e})\ [a_1, \ldots, a_n] \\
= \quad & \mathtt{e} \oplus \mathtt{e}\otimes a_1 \oplus \mathtt{e}\otimes a_1 \otimes a_2 \oplus \ldots \oplus \mathtt{e}\otimes a_1 \otimes \ldots \otimes a_n \\
= \quad & \mathtt{e} \otimes \left(\mathtt{e}\oplus a_1 \oplus a_1 \otimes a_2 \oplus \ldots \oplus a_1 \otimes \ldots \otimes a_n\right) \\
= \quad & \mathtt{e} \otimes \left(\mathtt{e} \oplus a_1 \otimes \Big(\mathtt{e} \oplus a_2 \otimes \big(\ldots \otimes (\mathtt{e} \oplus a_n)\big)\Big)\right)
\end{aligned}
$$