Radboud University Nijmegen
Course: Functional Programming
Professor: Sjaak Smetsers

Functional Programming Summary

Author:
Martan van der Straaten

June 13, 2021

Contents

Week 1: Haskell

What is Functional Programming all about?

1. It's pure, so no side effects (Mostly)
2. Recursion
3. (infinite) Lists
4. Lazy evaluation
5. reduction

This is different from a regular, imperative, programming language. Here we work using assignments (side effects) in an iterative style rather than recursive, and we use eager evaluation rather than lazy evaluation.

Example program

Quicksort - example

```
quickSort [] = []
quickSort (x:xs) = quickSort littles ++ [x] ++ quickSort bigs
  where littles = [a | a <- xs, a < x]
        bigs= [a | a <- xs, x<=a]
```

This program looks very neat, and is way shorter and clearer than its C or Java counter-part. This is the main advantage of a functional-programming language, next to the ease in proving something about a program written in it.

Writing a program in Haskell

Type-declaration

```
quickSort :: Ord(a) => [a] -> [a]
```

We have the function name left of the ::, then we have some domain parameters like the Ord(a) for this example, followed by the arguments the function takes and what result it requires. The domain parameters tell us something about the type of the arguments the function expects, here they need to have an implementation of Ord(), such that we can actually compare them. The function arguments are all things between the arrows. They can be passed to the function, in order to obtain a result. Not all arguments have to be provided, in case only a partial application is possible the function will apply it with the arguments provided, and return a function of the remaining type.

Lambda expressions

Lambda expressions

```
square = \x -> x * x
power = \x -> (\y -> x * (power x (y-1)))
```

Takes an argument x and squares it, speaks for itself right?

Composition of functions

Function composition

```
quad x = double . double x
```

Here f.g x is the same as f(g(x)). This is the function composition as we know it from mathematics. This allows us to write cleaner, more compact code.

Conditional definitions

Conditional definitions

```
smaller :: (Integer, Integer) -> Integer
smaller (x,y)
  | x <= y    = x
  | otherwise = y
```

Each clause here has a guard and an expression, seperated by =. We can have many different guards, with the last one (possibly) being otherwise. This is declaration-style, while using if... then... else... is expression-style.

Pattern-matching

Pattern-matching

```
fibonnaci :: Integer -> Integer
fibonnaci 0 = 1
fibonnaci 1 = 1
fibonnaci n = fibonnaci (n-1) + fibonnaci (n-2)

day :: Integer -> String
day 5 -> "Saturday"
day 6 -> "Sunday"
day _ -> "Weekday"
```

We can pattern-match on the arguments passed to the functions. This way we can easily create base-cases for certain functions, like fibonnaci, or we can create a case distinction like we did for day. We can also use wildcards '_' these will accept any input at that particular place. If no equation matches the incoming pattern, then the result is undefined.

where-clauses

Where-clause

```
demo :: Integer -> Integer -> Integer
demo x y = (a + 1) * (b + 2)
where a = x - y
      b = x + y
```

Allows us to define local functions that can only be used in this function, and before we defined them in the where. Where is in declaration-style.

let-clauses

Let-clause

```
demo :: Integer -> Integer -> Integer
demo x y = let a = x - y
               b = x + y
           in (a + 1) * (b + 2)
```

Same as where, but in expression-style. Let is slightly more flexible.

Week 2: Types

Strong typing

1. Haskell supports strong typing, meaning that every expression is typed.
2. Each type supports certain operations, that are meaningless on other types.
3. Haskell is statically typed, type checking occurs at compile-time.
4. Haskell can infer types, but it is wise to specify them anyway as a start of your program.

Booleans

1. type Bool
2. two constants, True and False

Design Pattern

- task: define a function f:: Bool → S
- step1: solve problem for False
- step2: solve problem for True

Chars

1. type Char
2. constants in single quotes
3. special characters are escapes '\', '\n'
4. ascii ordering

Polymorphism

This is when we use type variables rather than types in a type-declaration. This makes the function very easy to use, any type works, but it is harder to define. An example of such polymorphism would be

Polymorphic program

```
fst :: (a,b) -> a
fst (a,b) = a
```

Type synonyms

Type synonyms are alternative names for already existing types, they are just a 'macro' **no** new type is introduced, and type synonyms are **not** recursive.

Type synonym

```
type Card = (Rank, Suit)
type Name = String
type Date = (Int, Int, Int)
type Person = (Name, Date)
```

Record types

Much like tuples a record type contains multiple expressions that do not have to be of the same type. However here they are identified using a field name. A record type **does** introduce a newtype (rather than type synonym) and this **can** be recursive.

Record type

```
data Date = Date { year :: Int , month :: Int , day :: Int }
data Person = Person { name: Name, birthdate: Date}
data Parent = Parent { name: Name, birthdate: Date, child:Person}
```

Type classes

We cannot always use full polymorphism, for example

```
(+) :: a -> a -> a
```

is too general. Solution is using type-classes.

Type classes

```
(+) :: (Num a) => a -> a -> a
```

Num is the type class for any numeric value, so (+) is now polymorphic for any numeric input.

Week 3: Lists

List notation

List are very essential to functional programming. They are enclosed in square-brackets and comma separated.

Library functions

- length :: [a] → [Int]
- reverse :: [a] → [a]
- concat :: [[a]] → [a]
- map :: (a→b) → ([a]->[b])
- filter :: (a→Bool) → ([a]→[a])
- tails :: [String] → [String]
- take :: Int → [a] → [a]
- group :: (Eq a) => [a] → [[a]]

List Constructors

A list is either empty ([]) or an element x followed by xs (x:xs). Every finite list can be build using [] and :.

List design pattern

1. Solve for []
2. Solve for (x:xs) assume xs as induction hypothesis

List comprehensions

list comprehensions provide a convenient syntax for expressions involving map, filter, concat, etc. They allow us to create new lists from lists

Map

Map

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

Filter

Filter

```
filter :: (a->Bool) -> ([a]->[a])
filter f [] = []
filter f (x:xs)
  | f x == True = x: (filter f xs)
  | otherwise = filter f xs
```

List comprehensions

List comprehensions is a special, convenient syntax for list expressions. They allow us to generate, filter and map easily.

List comprehension

Map
```
[f x | x<-xs]
```

Filter
```
[x | x<-xs, pred x]
```

Cartesian product
```
[(x,y) | x<-xs, y<-ys]
```

The zip function

traversing two lists xs and ys simultaneously while combining each element x from xs pairwise with element y from ys.
Example is function odds that returns elements at odd indexes in array. This could now be defined as:

Odds

```
odds l = [ x | (i,x) <- zip [1..] l,    odd i]
```

Week 4: Datatypes

New datatypes

We've already seen type synonyms for existing datatypes, and we've also seen enumerations as new datatypes. However data is much more general than this.

Product datatypes

constructors of enumerated types are constants, constructors may be functions too.e.g. people with names and ages.

Product datatypes

```
type Name = String
type Age = Int
data Person = P Name Age
P :: Name -> Age -> Person
```

Such constructor functions do not simplify, they are in (head) normal form; Moreover, they can be used in pattern-matching

```
showPerson :: Person->String
showPerson (P n a) = "Name: " ++ n ++ ", Age: " ++ show a
```

Sum datatypes

Datatypes can have multiple variant

Sum datatypes

```
data Suit= Spades | Hearts | Diamonds | Clubs
data Rank= Faceless Integer| Jack | Queen | King
data Card= Card (Rank Suit) | Joker
```

So a Rank is either of the form Faceless n for some n, or a constant Jack, Queen, or King. The name Card is used both for a type and for a constructor.

Parametric datatypes

Datatypes may be parametric. Then constructors are polymorphic functions.

Parametric datatypes

```
dataMaybe a= Nothing | Just a. a
Just 13 :: Maybe Int
Nothing :: Maybe a
Just :: a  Maybe a
```

Useful for modelling exceptions

Recursive datatypes

Datatypes may be recursive too.
We can for example define natural numbers recursively.

Natural Numbers

Recursive datatypes - Natural numbers

```
data Nat   = Zero | Succ Nat
nat2int  :: Nat->Int
nat2int  Zero = 0
nat2int  (Succ n) = 1 + nat2int n

int2nat ::    Int->Nat
int2nat  0 = Zero
int2nat  n = Succ (int2nat (n-1))
```

Expressions
We can also define expressions for addition and multiplication using a recursive datatype.

Recursive datatypes - Expressions

```
data Expr= Lit Integer| Add Expr Expr| Mul Expr Expr

--using infix operators as constructors
 infixl6  :+:
 infixl7  :*:
dataExpr = Lit Integer-- a literal
         | Expr:+: Expr-- addition
         | Expr:*: Expr-- multiplication
```

Trees

Binary rees are an often used datatype in computing science. We can easily create a datatype that allows us to work with trees in haskell. We use the sum datatype for this, with a choice between a leafnode, or a recursive call to this datatype itself.

Binary Trees

```
data Btree a = Tip a| Bin (Btree a) (Btree a)
--e.g
Bin (Tip1) (Bin (Tip2) (Tip3))

--example function:
size  :: Btreea->Int
size  (Tip _) = 1
size  (Bin t  u) = size  t  + size  u
```

We can also generalize this datatype to not only work for binary trees, but for any tree in general.

General Trees

```
--internally-labelled trees with arbitrary branching (rose trees)
data Gtree a = Branch a [Gtree a]

--e.g.
Branch 1 [Branch 2 [], Branch 3 [Branch 4 []], Branch 5 []]
```

Design pattern Binary Trees

Remember: every datatype comes with a pattern of definition.
If we want to define a function f :: Btree a->s, data type consists of a two clauses, so we have 2 steps.

Step 1: solve problem for the tips.
Step 2: solve the problem for branches; assume that you already have a solution for the branches themselves.

```
f (Tip a) = f a
f (Bin l r) = Bin (f l) (f r)
```

Design pattern General Trees

Remember: every datatype comes with a pattern of definition.
If we want to define a function f :: Gtree P->S, data type consists of a single clause: so we only have 1 step.

Step 1: solve the problem for branches; assume that you already have a list of solutions of trs at hand; extend the intermediate solutions to a solution for Branch e trs.

```
f (Branch e trs) = ...e ... trs ... (map f trs)
```

Overloaded function

A polymorphic function is called overloaded if its type contains one or more class constraints. E.g

```
(+) :: Num a => a->a->a
```

For any numeric type a, (+) takes two values of type a and returns a value of type a.
Constrained type variables can be instantiated to any types that satisfy the constraints. Haskell has a number of type classes, including Num, Eq, Ord. A type class is essentially a set of types. E.g. the prelude adds instances of Double,Float,Int,Integer to Num, but you can also add new instances yourself.

Class declarations

New classes can be declared using the class mechanism. E.g. the class Eq of equality types is declared in the standard prelude as follows:

```
class Eq a where
    (==), (/=) :: a->a->Bool
    x /=y  = not (x == y)
```

This declaration states that for a type a to be an instance of the class Eq, it must support equality and inequality operators of the specified types. Declaring an instance only requires a definition for ==.

Instance declarations

Instance Declarations

```
-- the type
data Blood= A | B | AB | O

-- can be made into an equality type as follows:

instance Eq Blood where
    A == A = True
    B == B = True
    AB == AB = True
    O == O = True
    _ == _ = False
```

Week 5: Higher Order Functions

Functions as first-class citizens

Functions are first-class citizens of a functional programming language. Functions have all the rights of other types: They may be passed as arguments, may be returned as results and may be stored in data structures. Functions that manipulate functions are called higher order functions.

Functions as arguments

We have already seen examples of higher-order operators encapsulating patterns of computation, for example the map function. Which takes a function f and applies it to all elements in a list.

Functions as arguments

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

quickSortBy:: (a->a->Bool) -> [a] -> [a]
quickSortBy cmp [] = [ ]
quickSortBy cmp (x:xs) = quickSortBy cmp lefts ++ [x] ++ quickSortBy cmp rights
where lefts= [a | a <- xs, not (x `cmp` a)]
      rights= [a | a <- xs, x `cmp` a]
```

The quicksort example is more general than the quicksort on page 1, since we can now pass our own comparison function which will determine what we sort on.

Functions as results

Partial application
Consider the function add' x y = x + y, of type Integer->Integer->Integer. Another view would be that it has type Integer->(Integer->Integer) where -> associates to the right.
We now say that add' takes a single Integer and returns an Integer->Integer function. E.g. add' 3 is the Integer-transformer that adds three.

This is called partial application, it means that we need not apply a function to all its arguments at once. The result will then be a function, awaiting remaining arguments.

Currying
In Haskell, every function takes exactly one argument. A function taking pair of arguments can be transformed into a function taking two successive arguments, and vice versa. E.g

```
add    :: (Integer, Integer) -> Integer
add    (x, y) = x + y

--becomes'
add:: Integer->Integer->Integer'
add x y = x + y
```

Add' is called the curried version of add.

Folds

Many recursive definitions on lists share a pattern of computation. We would like to capture that pattern as a function (abstraction, conciseness, general properties, familiarity, . . .). We already have map and filter, which capture two common patterns, folds capture many more.

foldr

One instance of a fold function is the foldr-function. It takes a function that can be put between two elements of a list, and a neutral last element which will be applied as the second argument for the last function call. Below we will show some examples of translations from regular functions to foldr-calls.

foldr

```
-- Foldr function
foldr:: (a->b->b)->b->[a]->b
foldr st ba [] = ba
foldr st ba (x:xs) = x `st` foldr st ba xs

sum :: Num p=>[p]->p
sum [] = 0
sum (x : xs) = x + sum xs
-- foldr
sum= foldr (+) 0

product:: Num p=>[p]=>p
product [] = 1
product (x : xs) = x * product xs
-- foldr
product= foldr (*) 1

and :: [Bool]->Bool
and []       = True
and (x : xs) = x && and xs
-- foldr
and= foldr (&&) True

or :: [Bool]->Bool
or [] = False
or (x : xs) = x || or xs
--foldr
or= foldr (||) False
```

Some more difficult examples:

foldr

```
length [] = 0
length (x : xs) = 1 + length xs
--foldr
length= foldr (\x y ->y + 1) 0
      = foldr(const (+1)) 0

id [] = []
id (x : xs) = x : id xs
--foldr
id = foldr (:) []

map f [] = []
map f (x : xs) = f x : map f xs
--foldr
map f = foldr (\x ys -> f x : ys) []
      = foldr((:) . f) []

concat [] = []
concat (x : xs) = x ++ concat xs
--foldr
concat= foldr (++) []

reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
--foldr
reverse= foldr snoc []
  where snoc x xs = xs ++ [x]
      = foldr (flip (++).(:[])) []

filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise= filter p xs
--foldr
filter p = foldr (\x r -> if p x then x:r else r) []
```

foldl

Not every list function is a foldr, e.g. decimal[1,2,3] = 123. Efficient algorithm using Horner's rule: decimal [1,2,3]= ((0 * 10 + 1) * 10 + 2) * 10 + 3. We now fold to the left, meaning we cannot use foldr, but have to use foldl.

foldl

```
foldl :: (b->a->b)->b->[a]->b
foldl op ac [] = ac
foldl op ac (x:xs) = foldl op (ac `op` x) xs

decimal :: [Integer]-> Integer
decimal= foldl (\n d -> 10 * n + d) 0

-foldr
reverse:: [a]->[a]
reverse= foldr (\x xs->xs ++ [x]) [ ]
-foldl'
reverse:: [a]->[a]'
reverse= foldl (flip (:)) [ ]
```

Scanl/r

Gives list of intermediate (and end) result of folds. Meaning we get e, e*x1, (e*x1)*x2, etc.

Producers

So far we have focused on consumers. foldr consumes a list to provide a single value. Producers are important too, producers (unfolds) are dual to consumers (folds). Unfoldr creates a list out of some seed.

unfoldr

```
--definition
unfoldr:: (t->Maybe (a, t)) -> t -> [a]
unfoldr rep seed = produce seed
  where
    produceseed = case rep seed of
      Just (a, new_seed) -> a : produce new_seed
      Nothing -> []

repeat:: a->[a]
repeat x = x : repeat x
repeatU x = unfoldr(\x -> Just (x,x)) x


iterate:: (a->a) -> a-> [a]
iterate f x = x : iterate f (f x)
iterateU f x = unfoldr(\x -> Just (x, f x)) x

enumFrom :: Int->[Int]
enumFrom n = n : enumFrom (n+1)
enumFromUn = unfoldr(\x -> Just (x,x+1)) n

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate n x = x : replicate (n-1)
replicateU n x = unfoldr(\n -> if n==0 then Nothing else Just (x, n-1)) n

enumFromTo :: Int -> Int -> [Int]
enumFromTo n m
  | n <= m = n : enumFromTo (n+1) m
  | otherwise = []
enumFromToUn m = unfoldr(\n -> if n<=m then Just (n, n+1) else Nothing) n
```

Week 6: Classes

Quick lookback - Overloading

Sometimes we wish to use the same name for semantically different, but related functions.
We want to overload these identifiers.

Functions

Since == is overloaded, x == y can be ambiguous. When the compiler can't resolve the ambiguity, then the function become overloaded as well!

```
elem :: (Eq a) => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elemx ys
```

Parametric types

To define equality on a parametric type, say, Tree a we require equality on the element type a.

Btree equality

```
data Tree a= Leaf a | Fork (Tree a) (Tree a)
instance (Eq a) => Eq (Tree a) where
  Leaf x1     == Leaf x2    = x1 == x2
  Leaf _      == Fork _ _   = False
  Fork _ _    == Leaf _     = False
  Fork l1 r1  == Fork l2 r2 = l1 == l2 && r1 == r2
```

Functions as arguments

Instead of overloading we can also pass a function as an argument.

Generic programming

Generic programming is the art of defining functions that can operate on a large range of datatypes. For exmple defining instances of a class for different types at once. In our example we will assume that we can express datatypes in terms of:

- unit: ()=()
- product:(a,b)= (a,b)
- sum: Either a b = Left a | Right b

Suppose we have a class C, and instances of for unit, product and sum then by expressing a data type T in terms of unit, product and sum we get a T instance of C for free.

Generic instances for equality

```
instance Eq () where
  () == () = True

instance (Eq a, Eq b) => Eq (a, b) where
  (x1,y1) == (x2,y2) = x1 == x2 && y1 == y2

instance(Eq a, Eq b) => Eq (Either a b) where
  Left l1  == Left l2 = l1 == l2
  Right r1 == Right r2 = r1 == r2
  _        == _       = False
```

Lists

To derive the equality class for list types all we have to do now is translate the list data-type to terms of our 3 base-types. See the example below.

Equality for lists

```
type List elem = Either () (elem,[elem])

toList :: [elem] -> List elem
toList [] = Left ()
toList (x:xs) = Right (x,xs)

instance (Eq a) => Eq [a] where
  l1 == l2 = toList l1 == toList l2
```

Monoids

From Wikipedia: Suppose that S is a set and $\oplus$ is some binary operation $S \times S \to S$, then S with $\oplus$ is a monoid if it satisfies the following two axioms:

- Associativity: For all a, b and c in S, the equation (a $\oplus$ b) $\oplus$ c = a $\oplus$ (b $\oplus$ c) holds
- Identity element: There exists an element e in S such that for every element a in S, the equations e $\oplus$ a = a $\oplus$ e = a hold.

The monoid class

```
class Monoid m where
mempty :: m
(<>)   :: m -> m -> m
```

In Haskell the monoid operation <> is defined in a separate class Semigroup, extended by Monoid. Monoid laws: the operation <> should be associative with mempty as its unit element.

```
reduce :: (Monoid m) => [m] -> m
reduce []     = mempty
reduce (x:xs) = x <> reduce xs
```

The art now is to find a suitable monoid
Examples

Examples of monoids

```
--Lists
instance Monoid [a] where
  mempty= []
  (<>) = (++)

--Integers
newtype Additive= Sum { fromSum:: Int}
  deriving(Show)

instance Monoid Additive where
  mempty= Sum 0
  x <>y = Sum (fromSum x + fromSum y)

-etc
```

Week 7: Reasoning

Equational reasoning

We can prove certain things about programs by rewriting the inputs using formulas. For example:
Given

```
not :: Bool->Bool
not False = True
not True= False
```

we can show/prove that: not (not False) = False in the following way:

not (not False) = not True	(definition of not (equation 1))
= False	(definition of not (equation2))

Prove that: reverse [x] = [x]
Given

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

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

Proof:

reverse [x] = reverse (x:[])	list notation
= reverse [] ++ [x]	definition of reverse (equation 2)
= [] ++ [x]	definition of reverse (equation 1)
= [x]	definition of ++ (equation 1)

For any non-recursive equations this is a very easy and concise way of proving. However when we want to do the same for functions with recursive definitions then we will most likely have to use induction. This will be treated in the next part.

Proof by induction

proofs about recursive programs typically require induction, we can use explicit induction (ad hoc) or alternatively we can use "canned" induction which uses properties of predefined program schemes.

List induction pattern

Every recursive datatype comes with a pattern of induction, we will look at the induction pattern for lists now.
We want to prove that property P(xs) holds for every list xs.

1. show that the property holds for the empty listP([])(base case)
2. show that the property holds for non-empty lists, assume that you already have established P(xs), based on this assumption prove P(xs) => P(x:xs) (inductive step)

The assumption that P holds for xs is called the induction assumption or induction hypothesis, induction is valid for the same reason that recursive programs are valid: every list is either [] or of the form x:xs.

Reverse append - example

Given:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

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

We want to prove that reverse (xs ++ ys) = reverse ys ++ reverse xs. We do this using a proof by induction on xs.

case xs = [](base case):

reverse ([] ++ ys)	= reverse ys	definition of ++
	= reverse ys ++ []	unitality ++
	= reverse ys ++ reverse []	definition of reverse

case xs = a:as (inductive step), assuming:
reverse(as ++ ys) = reverse ys ++ reverse as (IH)

reverse ((a:as) ++ ys)	= reverse (a:(as ++ ys)	definition of ++
	= reverse (as ++ ys) ++ [a]	definition of reverse
	$\overset{IH}{=}$ (reverse ys ++ reverse as) ++ [a]	
	= reverse ys ++ (reverse as ++ [a])	associativity of ++
	= reverse ys ++ reverse (a:as)	definition of reverse

Program Synthesis using Induction

reverse is slow because ++ traverses its first argument, to eliminate ++ specify reverseCatxs ys = reverse xs ++ ys . We now use induction to derive/synthesize an efficient implementation of reverseCat.

case xs = [](base case):

reverseCat[] ys	$= reverse[] + +ys$	specification of reverseCat
	$= [] + +ys$	definition of reverse
	$= ys$	definition of ++

case xs = a:as (inductive step), assuming: for all lists es:reverseCat as es = reverse as ++ es(IH)

reverseCat(a:as) ys	= reverse (a:as) ++ ys	specification of reverseCat
	= (reverse as ++ [a]) ++ ys	definition of reverse
	= reverse as ([a] ++ ys)	associativity of ++
	= reverse as (a:ys)	list notation, definition of ++
	$\overset{IH}{=}$ reverseCat as (a:ys)	(right-to-left substituting a:ys for es)

We obtained

reverseCat

```
reverseCat [] ys = ys
reverseCat (x:xs) ys = reverseCat xs (x:ys)

--Now we can write:
reverse xs = reverseCat xs []
```

Foldr Fusion

Program schemes

Use of program schemes (higher-order functions) improves modularity of programs, and hence understanding, modification, and reuse. This not only applies to programming but also to proving, use of properties of program schemes improves modularity of proofs, and hence understanding, modification, and reuse. E.g. map preserves identity and composition.

Properties of foldr

```
foldr :: (a → b → b) → b → [a] → b
foldr st ba []     = ba
foldr st ba (x:xs) = x `st` foldr st ba xs

          list   = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list   = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

- frequent rewrite: reordering, pushing a function inside

```
f (foldr ⊕ e list)   =
      f (1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )     =
        (1 ⊞ f ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )   =
        (1 ⊞ ( 2 ⊞ f ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )   =
                          :
        (1 ⊞ ( 2 ⊞ ( 3 ⊞ ( 4 ⊞ ( 5 ⊞ f e  ) ) ) ) )   =
      foldr ⊞ (f e) list
```

Figure 1: Foldr fusion

captured by the all-important (foldr) fusion law:

```
f (foldr g e xs)   = foldr h (f e) xs if for all x y: f (g x y) = h x (f y)
```

Example

We want to prove:
reverse . reverse = id

1. First, we write reverse and id as folds.

```
reverse= foldr(\a r -> r ++ [a]) []
id= foldr(:) []
```

2. Filling in the FF-law:

```
reverse . foldr(\a r -> r ++ [a]) [] foldr(:)   []
```

3. Hence:

```
f = reverse g = \a r -> r ++ [a]h = (:)
```

4. To prove now, it suffices to show that

   (a) `[] reverse []`

   which follows from the definition of reverse

   (b) `reverse (y ++ [x]) (:) x (reverse y) = x : (reverse y)`

   which we can easily find as follows:

reverse (y ++ [x]) = reverse [x] ++ reverse y	property reverse-append
= reverse [x] ++ reverse y	property reverse-singleton
= [x] ++ reverse y	property reverse-singleton
= x : reverse y	list notation, definition of ++

Week 8: Lazy Evaluation

Evaluation orders

evaluation = reduction, you select a reducible expression(redex) and rewrite it according to its definition. When evaluating, different evaluation orders are possible, it matters which evaluation strategy we choose. The ones that are discussed in this course are applicative-order evaluation, normal-order evaluation, lazy evaluation.

Non-terminating evaluations

Even though all evaluation orders will come to the same answer if they terminate, termination is not always guaranteed. It might be that evaluation of an expression is perfectly okay in one, but will never terminate in the other. An example of this are the following two functions.

Non-termination

```
three :: Integer->Integer
three_ = 3

infinity :: Integer
infinity = 1 + infinity
```

If we first fully try to evaluate infinity then we'll easily see it will never terminate since we are trying to calculate infinity.

Applicative-order evaluation

To reduce the application f e we first reduce e to normal form and then expand definition of f and continue reducing. This is a very simple and obvious way of evaluating and rather easy to implement, however it may not terminate when other evaluation orders would. Other names for it are: innermost evaluation, call-by-value evaluation

Normal order evaluation

To reduce the application f e we expand the definition of f, substituting e, then we reduce the result of expansion. This avoids non-termination, if any evaluation order will, however it may involve repeating work. Other names for this are outermost evaluation, call-by-name evaluation.

Lazy Evaluation

Lazy order is like normal-order evaluation, but instead of copying arguments we share them. This

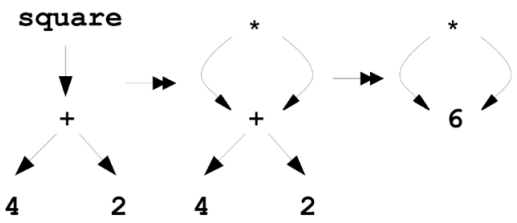

Figure 2: Lazy evaluation - visualized

means that terms are directed graphs, not trees. This has the best of both worlds, it evaluates argument only when needed so it is terminating, but never evaluates argument more than once, so efficient. This is the strategy used by Haskell.

Undefined and strictness

Some expressions denote no normal value (e.g. infinity, 1 / 0). We have a special value denoted by $\perp$, when evaluating such an expression, evaluator may hang or may give error message. We can apply functions to $\perp$; strict functions (square) give $\perp$ as a result, non-strict functions (three) may give some non-$\perp$value.

Normal forms

An expression is in normal form(NF) when it cannot be reduced any further.
An expression is in weak head normal form (WHNF) if it is a lambda expression, or if it is a constructor applied to zero or more arguments
An expression in normal form is in weak head normal form, but converse does not hold

Lazy Evaluation

Demand driven evaluation

Pattern-matching may trigger reduction of arguments to WHNF, for example

- `head [1, ..., 1000000]= head (1 : [(1 +1), ..., 1000000])= 1`
- Patterns matched top to bottom, left to right

```
False && x = False
True  && x = x
```

- guards may also trigger reduction

```
f z
 |    fst   z > 0 = fst z
 | otherwise = snd z
```

- local definitions not reduced until needed

```
g x = (x /=  0 && y < 10) where y = 1/x
```

A pipeline

The outermost function drives the evaluation, for example in

```
foldl (+) 0 (map square [1..1000])
    => foldl (+) 0 (map square (1:[2..1000]))
    => foldl (+) 0 (square 1 : map square [2..1000])
    => foldl (+) 1 (map square [2..1000])
    => foldl (+) 1 (square 2 : map square [3..1000])
    => ...
    => foldl (+) 14 (map square [4..1000])
    => 333833500
```

The entire list does not exist at one (memory efficient)

Demand driven programming

Lazy evaluation has useful implications for program design, many computations can be thought of as pipelines they are expressed with lazy evaluation, and intermediate data structures need not exist all at once.

Arrays

The library Data.Array provides lazy functional arrays:

```
data Array ix val
```

It is based on class Ix that maps a contiguous range of values onto integers. Ix is used as the index type of an array.

The call array (l,u) lazily constructs an array from a list of index/value pairs with indices within bounds (l,u), elements of many types may serve as indices e.g. tuples of indices yield multi-dimensional arrays.

array indexing :

```
(!) :: (Ix ix) => Array ix val -> ix -> val
```

Dynamic Programming

The idea of dynamic programming is to replace a function that computes data by a look-up table that contains data. When doing this we trade space for time: we decrease the running-time at the cost of increased space consumption.

Example - Fibonacci

Fibonacci is the classical recursion example, lets see how it would look if we defined it dynamically.

Fibonacci

```
-- a naive recursive implementation
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib(n -1) + fib (n-2)

-- we simply replace recursive calls by array look-ups
fibFn = fibArray ! n where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) + fibArray! (n-2)

  fibArray = array(0,n) [(i, fib i) | i <- [0..n]]
```

Example - Postage

Fibonacci

```
-- a naive recursive implementation
stamps:: [Stamp] -> Integer -> Integer
stamps ds 0 = 0
stamps ds n = minimum [ stamps ds (n-d) + 1 | d <- ds, d <= n]

-- we simply replace recursive calls by array look-ups
stampsDP:: [Stamp] -> Integer -> Integer
stampsDP ds n = stampsArray ! n where
  stamps 0 = 0
  stamps i = minimum [ stampsArray! (i - d) + 1 | d <- ds , d <= i]

  stampsArray= array (0,n) [(i,stamps i) | i <- [0..n]]
```

Week 9: IO

Side-effects

A pure functional language such as Haskell is referentially transparent, a function has a side effect if, it modifies a mutable data structure or variable, uses IO, wipes out your hard disk, launches a cruise missile, etc. A function with side-effects (sometimes called procedure) can be unpredictable depending on the state of the system, while function with no side-effects can be executed anytime:it will always return the same result, given the same input.

I/O in Haskell

In haskell they introduce a new special type of I/O computations (sometimes called actions)

```
type IO a
```

IO a is a type of computation that may do I/O, then returns an element of type a. It can be seen as the type of a to-do list. The type of putStr

```
putStr::    String -> IO ()
```

putStr does I/O, then returns () (similar to void).

Sequencing

Sequences of actions can be combined as a single composite action using the do-notation. For example:

do-notation

```
welcome:: IO ()
welcome= do putStr "Please enter your name.\n"
            s <- getLine
            putStr ("Welcome " ++ s ++ "!\n")
```

v <- a is called a generator (pronounce "v is drawn from a"). Note that a has type IO t, whereas v has type t.

Returning

With return we turn pure values into impure actions.

```
return::    a -> IO
```

Combining I/O operations

m >> n first executes m and then n.

```
(>>) :: IO a->IO b->IO b
```

m >>= n additionally feeds the result of the first computation into the second

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

I/O computations as first-class citizens

I/O computations are first-class citizens, juist like functions. This means we can freely mix I/O computations with, say, lists. Much like function values, we cannot diplay I/O computations.

```
printNumbers:: [IO ()]
printNumbers= [ print i | i <- [0..9]]
```

We can lift the I/O computations

```
sequence :: [IO ()]-> IO ()
sequence []     = return ()
sequence (a:as) = a >> sequence as

main::    IO ()
main= sequence printNumbers
```

Composition of effectful functions

As we know pure functions can be chained with function composition (.). Effectful functions can be chained with <=<.

```
(<=<) :: (b -> IO c) -> (a -> IO b) -> (a -> IO c)
(f <=< g) x = do y <- g x
                 f y
```

We can turn a pure into an effectful function using

```
lift :: (a -> b) -> (a -> IO b)
lift f x = return (f x)
```

References

Referential transparency: updatable variables live in the IO world. For IORefs (Updatable variables) we have the following interface:

```
type IORef a

newIORef:: a->IO (IORef a)
readIORef :: IORef a->IO a
writeIORef :: IORef a->a->IO ()
```

- newIORef creates a new IORef and initializes
- readIORef reads the value of an IORef
- writeIORef writes a new value into an IORef

Example: swapping "variables"

IORefs

```
swap:: IORef a -> IORef a -> IO ()
swap x y = do a <- readIORef x
              b <- readIORef y
              writeIORef x b
              writeIORef y a
```

Week 10: Applicatives

container types

Different containers we already know

The type of lists is the prime example of a container type. recall: map applies a given function to each element of a list, in doing that it changes the elements but keeps the structure intact.

Maybe is also an example of a container, it namely is either an empty or a singleton container. Maybe also supports a mapping function

```
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)
mapMaybe f Nothing  = Nothing
mapMaybe f (Just a) = Just (f a)
```

The same can be done for Btrees and Gtrees, they can also have a mapping function.

The functor class

The types of the mapping functions are very similar. The functor class abstracts away from the specific container type.

```
class Functor f where
  fmap ::(a -> b) -> (f a -> f b)
```

Here, f is a type constructor, not a type! f ranges over type constructors ([ ], Maybe,Btree, etc). Functor is a so-called constructor class ▪ an infix synonym for fmap is (<$>).

For the functor class to work, every container type should be made an instance of the functor class. E.g

```
instance Functor []
  where fmap= map

instance Functor Maybe
  where fmap= mapMaybe
```

More peculiar Functor instances

We can define

```
instance Functor((,) a)
  where fmap f (x,y) = (x, f y)
```

(,) is the binary tuple type constructor.

What about the type constructor -> (note that -> is a binary type constructor)?

```
instance Functor((->) a) where
  --fmap :: (b -> c) -> ((->) a b -> (->) a c)
  fmap= (.)
```

Applicatives

```
infixl4 <*>

class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Pure turns a value into a structure of type f a, and <*> is a generalized function application where function, argument, and result are contained in f structures.

Maybe instace - example

```
instance Applicative Maybe where
-- pure :: a -> Maybe a
pure= Just
-- (<*>):: Maybe(a -> b) -> Maybe a -> Maybe b
Nothing <*>_ = Nothing
(Just g) <*> my = fmap g my
```

Applicatives thus supports exceptional programming: We can start applying pure functions to arguments that may fail without managing the propagation of failure explicitly.

Lists

This is contained in the standard prelude. For lists, pure transforms a value into a singleton list and <*>takes a list of functions and a list of arguments and applies each function to each argument. We can View [a] as a generalisation of Maybe a: the empty list denotes no success and a non-empty list represents all possible ways a result may succeed. Hence applicative style for lists supports non-deterministic programming.

```
instance Applicative [] where
-- pure :: a -> [a]
pure x = [x]

-- (<*>):: [a -> b] -> [a] -> [b]
gs <*> xs = [g x | g <- gs, x <- xs]
```

IO instance

IOtype can be made into an applicative functor using the following declaration:

```
instance Applicative IO where
-- pure :: a -> IO a
pure= return
-- (<*>):: IO (a -> b) -> IO a -> IO b
mg <*> mx = do {g <- mg; x <- mx; return (g x)}

-- example
getChars :: Int -> IO String
getChars 0 = pure []
getChars n = pure (:) <*> getChar <*> getChars(n -1)
```

This enables programming with effects

Derived operators

There also one-sided versions of <*>, which can be useful if a computation is only executed for its effect.

```
(*>) :: Applicative f => f a -> f b -> f b
a *> b = pure (\x y -> y) <*> a <*> b

(<*) :: Applicative f => f a -> f b -> f a
a <* b = pure (\x y -> x) <*> a <*> b
```

Example of applicative

Example of Applicatives

```
-- equations
 infixl6  :+:
 infixl7  :*:

dataExpr
  = Lit Integer-- a literal
  | Expr:+: Expr-- addition
  | Expr:*: Expr-- multiplication
  | Div ExprExpr-- integer division

-- original evaluator
eval :: Expr ->Integer
eval(Lit i) = i
eval(e1 :+: e2) = eval e1 + eval e2
eval(e1 :*: e2) = eval e1 * eval e2
eval(Div e1 e2) = div (eval e1) (eval e2)

-Extending this quickly becomes confusing and will obstruct the clear structure

evalA:: (Applicative f) => Expr -> f Integer
evalA(Lit i) = pure i
evalA(e1 :+: e2) = pure (+) <*> evalA e1 <*> evalA e2
evalA(e1 :*: e2) = pure (*) <*> evalA e1 <*> evalA e2
evalA(Div e1 e2) = pure div <*> evalA e1 <*> evalA e2
```

There are two major changes compared to the vanilla evaluator, namely we now use prefix: (+) a b instead of a + b and the application is made explicit: pure f <*> a <*> b instead of f a b. The function is still pure now, but much easier to extend.

To get the function working like we had before we do the following:

Identity Functor

```
newtype Id a= I { froml:: a }

instance Functor Id where
-- fmap:: (a -> b) -> Id a -> Id b
fmapf (I x) = I (f x)

instance Applicative Id where
-- pure:: a -> Id a
pure a = I a
-- (<*>):: Id (a -> b) -> Id a -> Id b
I f <*> I x = I (f x)
```

We now use froml to unpack values after evaluation, and then we have the same exact evaluation as we had in our original eval(). To add effects we can create a different functor, for instane one that counts the amount of evaluations that have happened.

Counter Functor

```
newtype Counter a= C { fromC:: (a,Int)}
  deriving(Show)

instance Functor Counter where
  -- fmap:: (a -> b) -> Counter a -> Counter b
  fmap f (C (a,n)) = C (f a,n)

instance Applicative Counter where
  -- pure:: a -> Counter a
  pure a = C (a,0)
  -- (<*>):: Counter(a -> b) -> Counter a -> Counter b
  C (f,m) <*> C (x,n) = C (f x, m + n)

-- New evaluator for counting
evalC :: Expr -> Counter Integer
evalC(Lit i) = tick *> pure i
evalC(e1 :+: e2) = tick *> pure (+) <*> evalC e1 <*> evalC e2
evalC(e1 :*: e2) = tick *> pure (*) <*> evalC e1 <*> evalC e2
evalC(Div e1 e2) = tick *> pure div <*> evalC e1 <*> evalC e2
```

Week 11: Monads

Collapsables

If we look at our evaluator and suppose that we have a new div called 'safediv' which raises an exception if its second argument is zero. We have an issue, we cannot 'inject' the result into the background computation (which takes care of the effects). After all,

```
pure safediv <*> evalA e1 <*> evalA e2
```

has type Maybe (Maybe Integer) and not Maybe Integer.

To solve this we extend the interface and introduce an additional combinator that allows us to collapse a value of type f (f a) into a value of type f a.

Monad class

Such a collapable class does not exist in Haskell. Instead Haskell introduces monads.

```
class(Applicative m) => Monad m where
  return:: a-> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

For the maybe class this would look like this

Maybe monad

```
instance Monad Maybe where
-- return:: a -> Maybe a
return a = Just a

-- (>>=):: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= _ = Nothing
Just a  >>= f = f a

-- new evaluator
evalM:: (Monad m) => Expr -> m Integer
evalM(Lit i) = return i
evalM(Div e1 e2) = evalM e1 >>= \n1 ->
                   evalM e2 >>= \n2 ->
                   return (n1 `div` n2)
```

Do notation

Typical expressions built with the >>= operator have the following structure:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
mn >>= \xn ->
f x1 x2 ... xn
```

Haskell has the do notation for this. Which works as follows:

```
do  m1 >>= \x1 ->
    m2 >>= \x2 ->
    ...
    mn >>= \xn ->
    f x1 x2 ... xn
```

Laws

Laws

```
-- functor laws (functors are required to satisfy these)
fmap id = id
fmap (g . f) = fmap g . fmapf

-- Applicative functor laws (instances of Applicative are required to satisfy these)
pure id  <*> v       = v
pure f   <*> pure x  = pure(f x)
u        <*> pure y  = pure(\g -> g y) <*>u
u        <*> (v <*>w) = (pure (.) <*>u <*>v) <*>w

-- Monad laws (instances of Monadare required to satisfy these)
m >>= \a -> return a = m
return a >>= \b -> f b= f a
(m >>=\a -> f a) >>= \b -> g b= m >>= \a -> (f a >>=\b -> g b)
```

Week 12: Parsers

Parsing is an important part of programming. We need to understand input the user gives for a program to function. To come back to our example used so often before, the expressions. It would be nice to be able to input the expressions in a slightly more natural way. Parsers can help with this.

Parsers as functions

We can represent parsers as functions. The parser (for expressions) can be represented as a function of type

```
type Parser :: String ->Expr
```

However, parser might not always consume its entire argument string, so we should account for this by returning any unconsumed part of the argument string

```
type Parser :: String -> (Expr, String)
```

Similarly, a parser may fail. We will thus return a list of solutions, an empty list means fails, a singleton list means success.

```
type Parser :: String -> [(Expr, String)]
```

Finally, we can generalize this from expressions to any type.

```
type Parser a :: String -> [(a, String)]
```

In summary, a parser of type a is a function that takes an input string and produces a list of results, each of which is a pair comprising a result value of type a and an output string.

The problems of adhoc parsing

We could specify a completely new parser for every datatype we want to have. This would be called adhoc parsing, since we just have an adhoc solution for this current problem. This is an ok idea, if you need one or two small parsers, but when using many or more complex parsers this becomes tedious, confusing and rather complex. A better idea might be to create parsers for subparts, and to combine these small parsers for bigger cases.

Combining parsers

Like stated before, it is a good idea to define small parsers and combine these to form more complex parsers. The question now is how do we do this? We will use the operations of class Functor, Applicative and Monad(and more) as combinators, so we want the parser type to be made into instances of these type classes.

Example

Parser instances

```
newtype Parser a = P { parse:: String -> [( a,String)] }

-a simple parsing primitive
item :: Parser Char item= P (\inp -> case inp of
                                       []     -> []
                                       (x:xs) ->[( x,xs)])

-- we can now use fmap
instance Functor Parser where
  -- fmap:: (a -> b) -> Parser a -> Parser b
  -- = (a -> b) -> (String -> [( a,String)]) -> String -> [( b,String)]
  fmap f p = P (\inp -> case parse p inp of
                          []          -> []
                          [( v,out)] -> [(f v, out)])

-- we can now use applicative style (<*>)
instance Applicative Parser where
  -- pure:: a -> Parser a
  pure v = P (\inp -> [(v,inp)])

  -- <*>:: Parser(a -> b) -> Parser a -> Parser b
  abp<*>ap = P (\inp -> case parse abp inp of
                          []          -> []
                          [(g,out)] -> parse (fmap g ap) out)

-- we can now use do-notation
instance Monad Parser where
-- (>>=):: Parser a -> (a -> Parser b) -> Parser b
p >>=f = P (\inp -> case parse p inp of
                      []          -> []
                      [(v,out)] -> parse (f v) out)
```

Choice

Do-notation combines parsers in sequence. The output string from each parser in the sequence becomes the input string for the next. However another way of combining parsers is to apply one parser to the input string, and if this fails to then apply another to the same input instead. This is the choice operator. We can borrow it from the predefined class Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f  a -> f a -> f  a
```

The operation $< | >$ is associative, and has empty as unit element (notice the similarity with class Monoid).

Parser instances - Alternative

```
instance Alternative Parser where
  -- empty:: Parser a
  empty= P (\inp -> [])

  -- (<|>):: Parser a -> Parser a -> Parser a
  p <|> q = P (\inp -> case parse p inp of
                         [] -> parse q inp
                         [( v,out)] -> [(v,out)])
```

Derived primitives

Now that we have defined all these operations for our parser, we can easily combine some parsers into more complex ones. First lets define some primitives:

Primitives

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then return x else empty

digit :: Parser Char
digit = sat isDigit

letter :: Parser Char
letter = sat isAlpha

alphanum:: Parser Char
alphanum= sat isAlphaNum

char :: Char -> Parser Char
char x = sat (== x)
```

And now we combine those to get:

Combined

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)

int :: Parser Int
int = do char '-'
         n <- nat
         return (-n)
       <|>
         nat

many :: f a -> f [a]
some :: f a -> f [a]
many x = some x <|> pure []
some x = pure (:) <*> x <*> many x

space :: Parser ()
space = do many (sat isSpace)
           return ()

token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v
```

As can be seen this process is extremely easy and natural now. It takes barely any effort and you can create rather powerful parsers using this method!

Monadic programming

In practice you don't have to define monads yourself. Instead you can use the Monad Transformer Library. It provides a selection of (basic) monads and their transformer variants (to combine/stack monads) along with type classes that allow uniform handling of a base monad and its transformer.

Week 13: Foldables and traversables

Folds

Collapsing a list of values into a single value. Our minimal assumptions are that the operation $\oplus$ is associative (and has a unit element), thus map-reduce builds on so-called monoids.

```
fold :: Monoid m => [m] -> m
fold []     = mempty
fold (x:xs) = x <> fold xs
```

Which is the same as

```
fold= foldr(<>) mempty
```

Combining fold with map we get

```
foldMap :: Monoid m => (a -> m) -> [m] -> m
foldMap f []     = mempty
foldMap f (x:xs) = f x <> foldMap f xs
```

Foldable

The foldable class abstracts away from the container type in defining folds.

```
class Foldable t where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
```

Foldable includes default implementations, defining either foldr or foldMap is sufficient. Other functions such as length and max are also contained in Foldable.

Functions in terms of functions

The reason why defining foldr or FoldMap are sufficient is because we can define all others in terms of either.

```
-- foldMap as foldr
foldMap f = foldr(\a m -> f a <> m) mempty

--foldr as foldMap
foldr f e x = appEndo (foldMap (Endo . f) x) e
```

Traversables

Mapping functions over a data structure (container) is normally done using the functor class (fmap). But what if we want to map a function that might fail? For that we use traverse.

```
traverse :: (a -> Maybe b) -> [a] -> Maybe [b]
traverse f []     = pure []
traverse f (x:xs) = pure (:) <*> f x <*> traverse f xs
```

Abstracting from the applicative and from the container class we get

```
(Functort, Foldable t) -> Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Traverable - example for lists

```
instance Traversable [] where
--traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
traverse g []     = pure []
traverse g (x:xs) = pure (:) <*> gx <*> traverse g xs
```

Week 14: Type families

Types of types

Int is a type, Maybeis a unary type constructor, the function space "->" is a binary type constructor. These types and type constructors also possess types, called kinds. Examples of kinds are:

```
Int   :: *
Maybe :: *->*
[]    :: *->*
(,,)  :: *->*->*->*
(->)  :: *->*->*(->)
Int   :: *->*
```

Type classes

In Haskell, type classes provide a structured way to control overloading. A method of a type class can be seen as a family of functions, e.g. the family of equality functions(==) is captured by

```
class Eq a where
  (==) :: a -> a -> Bool
instance Eq Char where...
instance Eq Int  where...
```

Propagation of overloading

Overloading is contagious, if overloading cannot be resolved statically it will propagate. The context in which the overloaded method is used contains insufficient information to determine the instance type. E.g.

```
double :: Num a => a -> a
double x = x + x
```

Sometimes overloading can never be resolved.

```
class Foo a where
  foo :: a-> Int

class Bar a where
  bar :: Int -> a

quux= foo . bar
```

The function quux will generate an error (ambiguous overloading). The function

```
xuuq :: (Bar c, Foo a) => a -> c
xuuq= bar . foo
```

xuuq is fine.

Multi-parameter type classes

Type families are quite clear now, but how to capture families where two types vary.

```
insert :: Bool -> BitVector  -> BitVector
insert :: Int  -> SearchTree -> SearchTree
```

We can use multi-paramater type classes.

```
class Set el set where
  insert :: el -> set -> set

instance Set Bool BitVector  where...
instance Set Int  SearchTree where...
```

However they often result in ambiguity and therefore they are hardly used.

Type families

Like we stated, multi-paramter type classes are not a great solution to our problems. Luckily type families exist!

```
type familyContainer el :: *

type instance Container Bool = BitVector
type instance Container Int  = SearchTree

class Set elem where
  insert :: elem -> Container elem -> Container elem

instance Set Bool where
  insert ...
instance Set Int where
  insert ...
```

Container can be seen as a function on types (family = signature, instances = equations), these functions are evaluated compile-time and can be recursive.

Data families

Type synonym families allow us to associate a type name to a set of existing types. They define partial functions from types to types by pattern matching on input types. Data family type families create a **new** data type, each instance introduces a collection of new data constructors.

```
data family X a

data instance X Int  = XI1 Char | XI2
data instance X Char = XC1 Int  | XC2 String
```

Associated type families

A data or type synonym family can be declared as part of a type class.

```
class Set elem where
  type Container elem
  insert :: elem -> Container elem -> Container elem
```

instances can be declared like

```
instance Set Bool where
  type Container Bool = BitVector
  insert el c = ...
```