Set Haskell Exercises

Young W. Lim

2018-12-04 Tue

Outline

Based on

"The Haskell Road to Logic, Maths, and Programming",
K. Doets and J. V. Eijck

Using STAL.hs

- Sets, Types, and Lists (STaL)

```
module STAL

where

import List
import DB
```

```
:load STAL
```

Sets

```
Prelude> :load STAL.hs
[1 of 2] Compiling DB               ( DB.hs, interpreted )
[2 of 2] Compiling STAL             ( STAL.hs, interpreted )
Ok, modules loaded: STAL, DB.
*STAL>
*STAL>
*STAL>
*STAL> odds1
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,
45,47,49,51,53,55,57,59,61,63,65,67,69, ...

*STAL>
*STAL>
*STAL> evens2
[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,
44,46,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,
86,88,90,92,94,96,98,100,102,104,106,108,110,112,114,116 ...
```

Halting problems

- no general test for checking
  whether a given procedure terminates
  for a particular input
- the halting problem is undecidable
- the existence of an algorithm for the halting problem
  would lead to a paradox like the Russell paradox

Halting problem Examples (1)

- an example for which no proof of termination exists

```
run :: Integer -> [Integer]
run n | n < 1 = error "argument not positive"
      | n == 1 = [1]
      | even n = n: run (div n 2)
      | odd n  = n: run (3*n+1)
```

- run 5

```
run 5  | odd   5 =  5 : run (3*5+1)
run 16 | even 16 = 16 : run (div 16 2)
run 8  | even  8 =  8 : run (div 8 2)
run 4  | even  4 =  4 : run (div 4 2)
run 2  | even  2 =  2 : run (div 2 2)
run 1  | n ==  1 = [1]
[5, 16, 8, 4, 2, 1]
```

Halting problem Examples (2)

- an example for which no proof of termination exists

```
run :: Integer -> [Integer]
run n | n < 1 = error "argument not positive"
      | n == 1 = [1]
      | even n = n: run (div n 2)
      | odd n  = n: run (3*n+1)
```

- run 6

```
run 6  | even  6 =  6 : run (div 6 2)
run 3  | odd   3 =  3 : run (3*3+1)
run 10 | even 10 =  9 : run (div 10 2)
run 5  | odd   5 =  5 : run (3*5+1)
run 16 | even 16 = 16 : run (div 16 2)
run 8  | even  8 =  8 : run (div 8 2)
run 4  | even  4 =  4 : run (div 4 2)
run 2  | even  2 =  2 : run (div 2 2)
run 1  | n ==  1 = [1]
[5, 3, 10, 5, 16, 8, 4, 2, 1]
```

Halting problem Examples (3)

- an example for which no proof of termination exists

```
run :: Integer -> [Integer]
run n | n < 1 = error "argument not positive"
      | n == 1 = [1]
      | even n = n: run (div n 2)
      | odd n  = n: run (3*n+1)
```

- run 7

```
run  7 | odd   7 =  7 : run (3*7+1)
run 22 | even 22 = 22 : run (div 22 2)
run 11 | odd  11 = 11 : run (3*11+1)
run 34 | even 34 = 34 : run (div 34 2)
run 17 | odd  17 = 17 : run (3*17+1)
run 52 | even 52 = 52 : run (div 52 2)
run 26 | even 26 = 26 : run (div 26 2)
run 13 | odd  13 = 13 : run (3*13+1)
run 40 | even 40 = 40 : run (div 40 2)
run 20 | even 20 = 20 : run (div 20 2)
run 10 | even 10 = 10 : run (div 10 2)
...
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

The Russel Paradox (1)

- it is not true that to every property E
  there corresponds a set $\{x|E(x)\}$ of all objects that have E
- consider the property of not having yourself as a member
- most sets are likely to have this property
  - the set of all even numbers is itself not an even natural number
  - the set of all integers is itself not an integer
- call such sets *ordinary*
  - corresponding abstraction $R = \{x|E(x)\}$

The Russel Paradox (2)

- $R = \{x|E(x)\}$
- impossible to answer the question
  whether the set R itself is ordinary or not
- suppose $R \in R$, ie., R is an ordinary set
  - an ordinary set does not have itself as a member
    R does not have itself as a member $R \notin R$
- suppose $R \notin R$, ie., R is an extraordinary set
  - an extraordinary set has itself as a member
    R has itself as a member, i.e., $R \in R$
- if R were a legitimate set,
  this would unavoidably lead us to a contradiction
  $R \in R \iff R \notin R$

The Russel Paradox (3)

- only properties that are unlikely considered give rise to problems
- a restriction can be applied to the forming set on the basis of a previously given set A $\{x \in A | E(x)\}$ instead of $\{x | E(x)\}$

Non-halting problems

- suppose `halt` can be defined
- suppose also the procedure `funny` is defined in terms of `halts`

```
funny x | halts x x = undefined
        | otherwise = True
```

  - suppose `funny` does not halt
    the first case, when x is applied to x, `halts` halts
    `funny` is bound to `x`, then `funny funny` does not halt
    contradiciton
  - suppose `funny funny` does halt
    the second case, `funny x = True` halt
    `funny funny` binds `x` to `funny`
    this becomes the first case, it does not halt
    contradiction

Haskell type discipline

- paradoxical defintions are avoided
  in functional programming by keeping track of
  the types of all objects and operations
- derived types : new types can be constructed from old
  pairs of integers, lists of characters, lists of reals, etc.
- type discipline avoids the halting paradoxes

Haskell type discipline for `funny`

- the definition of `funny` calls `halts`
- the type of `halts`
  the 1st argument : a procedure `proc`
  the 2nd argument : the argument of that procedure `arg`
- the types of 2 arguments : `a -> b` and `a`
  the type of the result of application of (`proc arg`) : `b`
- therefore the application `halts x x` is mal-formed
  the types of 2 arguments must be different
  thus the arguments must themselves must be different

elem examples

- `elem:: a -> [a] -> Bool`
  checks whether an object is element of a list
  - the 1st argument : a certain type a
  - the 2nd argument : a list over a
  - in Haskell, $R \in R$ does not make sense
- ```
  elem 'R' "Russsel"    ......    ['R', 'u', 's', 's', 'e', 'l']
  True

  elem 'R' "Cantor"     ......    ['C', 'a', 'n', 't', 'o', 'r']
  False

  elem "Russel" "Cantor"
  Error: Type error in application
  ...
  ```
  ```

Identifiable objects

- to check if some thing x is
  an elment of a list l of some things
  one has to be able to identify
  things of the type of x
- the objects that can be identified are
  the objects of the kinds for which
  equality and inequality are defined
  - neither texts, potentially *infinite* stream of characters are of this kind
  - nor the Haskell operation denoted as computation procedures r
    - no principled way to check whether two procedures
      are doing the *same* task

Procedure equality test examples (1)

- assume there is an equality test on procedures
- consider a test for whether a procedure *f* halts on input x
- ```
  halts f x = f /= g
    where g y | y == x    = undefined
              | otherwise = f y
  ```
- `where` is used to define an aux function `g`
- `g` diverges when `x` is equal to `y`
- on all other inputs `g` is equal to `f`
  ```

Procedure equality test examples (2)

- ```
  halts f x = f /= g
    where g y | y == x     = undefined
              | otherwise = f y
  ```
- if g is not equal to f
  that difference must come from the input x
  since g *diverges* (undefined) on this input
  on this input f must halt which is not equal to g
- if g and f are equal
  then f and g behaves the same to the same input x
  this means f *diverges* on that input
- this will not work
  ```

Eq class (1)

- the types of object for which
  the question equal or not makes sense
  are grouped into a collection of types called a class Eq
- == for equality of objects of types in the `Eq` class
  /= for inequality of objects of types in the `Eq` class

Eq class (2)

- `:t elem`
  `elem :: Eq a => a -> [a] -> Bool`
- `:t` shows the type of a defined operation
- if a is a type for which equality is defined
  (if a is in the `Eq` class)
  then `a -> [a] -> Bool` is an appropriate type for `elem`
- `elem` can be used to check
  whether an integer is a member of a list of integers
  whether a character is a member of strings
- `elem` cannot be used to check
  whether an operation is a member of a list of operations
  whether a text is a member of a list texts

Ord class

- the class of the types of things which not only can be tested for quality and inequality, but also for order
- in addition to == and /=, the relation < and <= are defined
- has the `min` function for the minimal element and the `max` function for the maximal element
- the class `Ord` is a subclass of the class `Eq`

Class (1)

- classes are useful, because they allow objects and operations on those objects to the instances of several tyhpe at once
- the numeral 1 can be used as an integer, as a rational, as a real, and so on
- ```
  :t 1
  1 :: Num a => a
  ```
  ```

Class (2)

- all of the types integer, rational, real, complex numbers are instances of the same class, called `Num`
- the class `Num` is a subclass of the class `Eq` because it also has equality and inequality
- for all types in the class `Num` certain basic operations such as + and $*$ are defined
- operator overloading
  one could the same name for different operations depending on whether we operate on N, Z, Q . . .
  and depending on the representation we choose

Defining infinite lists

- ```
  ones = 1 : ones

  ones
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1^C Interrupted
  ```
  ```

Data type of lists

- `data [a] = [] | a : [a] deriving (Eq, Ord)`
- in Haskell, every set has a type
- `[a]` specifies that lists over type a
  are either empty
  or consist of an element of type a
  put in front of a list.
- the operation `:` combines an object
  with a list of objects of the same type
  to form a new list of objects of that type

```
:t (:)
(:) :: a -> [a] -> [a]
```

List Equality

- lists are ordered sets
- two lists are the same if
  1. they are both empty
  2. they start with the same element
     and their tails are the same
-
```
instance Eq a => Eq [a] where
  []     == []       = True
  (x:xs) == (y:ys)   = x==y && xs==ys
  _      == _        = False
```
- if a is an instance of class `Eq`, then `[a]` is so

a type of class Ord

- a type on which the binary operation `compare` is defined with a result of type `Ordering`
- the type `Ordering` is the set $\{LT, EQ, GT\}$

List Order (1)

- lexicographical order
- the empty list comes first
- non-empty lists L_1, L_2
  1. compare their first elements using `compare` for objects of type `a`
  2. if they are the same, etermine the order of their remaining lists
  3. if the first element of L_1 comes first, L_1 comes first before L_2
  4. if the first element of L_2 comes first, L_2 comes first before L_1

List Order (2)

- lexicographical order
- the empty list comes first
- ```
  instance Ord a => Ord [a] where
      compare []     (_:_)  = LT
      compare []     []     = EQ
      compare (_:_)  []     = GT
      compare (x:xs) (y:ys) = primCompAux x y (compare xs ys)
  ```
- if a is an instance of class `Ord`, then `[a]` is so
  ```

List Order (3)

- non-empty lists L_1, L_2
  1. if they are the same, `compare x y == EQ`
     etermine the order of their remaining lists `compare xs ys`
  2. if the first element of L_1 comes first, `compare x y == LT`
     L_1 comes first before L_2 `LT`
  3. if the first element of L_2 comes first, `compare x y == GT`
     L_2 comes first before L_1 `GT`

- `compare (x:xs) (y:ys)   = primCompAux x y (compare xs ys)`

```
primCompAux  :: Ord a => a -> a -> Ordering -> Ordering
primCompAux x y o  =
    case compare x y of EQ -> o;
                        LT -> LT;
                        GT -> GT;
```

List Order (4)

- ```
  instance Ord a => Ord [a] where
      compare []      (_:_)   = LT
      compare []      []      = EQ
      compare (_:_)   []      = GT
      compare (x:xs) (y:ys)   = primCompAux x y (compare xs ys)

  primCompAux  :: Ord a => a -> a -> Ordering -> Ordering
  primCompAux x y o  =
      case compare x y of EQ -> o;
                          LT -> LT;
                          GT -> GT;
  ```
- `primCompAux` covers the case of two non-empty lists
- type `Odering` is the set $\{LT, EQ, GR\}$
  ```

Head and Tail

- ```
  head :: [a] -> a
  head (x:_) = x
  ```
- ```
  tail :: [a] -> a
  tail (_:xs) = xs
  ```
- ```
  Prelude> head [1, 2, 3, 4]
  1
  Prelude> tail [1, 2, 3, 4]
  [2,3,4]
  ```
  ```

Last and Init

- ```
  last :: [a] -> a
  last [x] = x
  last (_:xs) = last xs
  ```
- ```
  init :: [a] -> [a]
  init [x] = []
  int (x:xs) = x : init xs
  ```
    - ```
      Prelude> last [1, 2, 3, 4]
      4
      Prelude> init [1, 2, 3, 4]
      [1,2,3]
      ```
      ```

Null

- ```
  null :: [a] -> Bool
  null [] = True
  null (_:_) = False
  ```

- ```
  Prelude> null [1, 2, 3, 4]
  False
  Prelude> null []
  True
  ```

Nub

- ```
  nub : (Eq a) => [a] -> [a]
  nub [] = []
  nub (x:xs) = x : nub (remove x xs)
      where
      remove y []                   = []
      remove y (z:zs) | y == z      = remove y zs
                      | otherwise = z : remove y zs
  ```

- in Haskell, strings of characters are represented as lists

  ```
  "abc"
  ['a', 'b', 'c']
  ```

- ```
  nub "Mississippy"
  "Mispy"

  nub ["aa", "bb", "aa", "cc"]
  ["aa", "bb", "cc"]
  ```

Database Module

```
module DB
where
type WordList = [String]
type DB = [WordList]

db :: DB
db = [ ["release", "MV1", "YR1"],          -- MV1 was released in YR1
       ["release", "MV2", "YR2"],          -- MV2 was released in YR2
       ["release", "MV3", "YR3"],          -- MV3 was released in YR3
       {- ... -}

       ["direct", "DRTR1", "MV1"],         -- DRTR1 directed the film MV1
       ["direct", "DRTR2", "MV2"],         -- DRTR2 directed the film MV2
       ["direct", "DRTR3", "MV3"],         -- DRTR3 directed the film MV3
       {- ... -}

       ["play", "ACT1", "MV1", "CHR1"],    -- ACT1 played CHR1 in MV1
       ["play", "ACT2", "MV1", "CHR2"],    -- ACT2 played CHR2 in MV2
       ["play", "ACT3", "MV3", "CHR3"],    -- ACT3 played CHR3 in MV3
       {- ... -} ]
```

Comment

- Everything between {- followed by a space and -} is a block comment.
- ```
  {-
    hello
    world
  -}
  ```
- {- ... -}

https://wiki.haskell.org/Keywords#.7B-.2C_-.7D
  ```

Database and List Comprehension (1)

```
db :: DB

characters = nub [x | ["play",_,_,x]  <- db]  -- played x
movies     =     [x | ["release",x,_] <- db]  -- x was released
actors     = nub [x | ["play",x,_,_]  <- db]  -- x played
directors  = nub [x | ["direct",x,_]  <- db]  -- x directed
dates      = nub [x | ["release",_,x] <- db]  -- was released in x
universe   = nub(characters++actors++directors++movies++dates)


--   ["release", "MV1", "YR1"],          -- MV1 was released in YR1
--   ["direct", "DRTR1", "MV1"],         -- DRTR1 directed the film MV1
--   ["play", "ACT1", "MV1", "CHR1"],    -- ACT1 played CHR1 in MV1
```

Database and List Comprehension (2)

```
direct  = [(x,y)   | ["direct", x, y]  <- db]  -- x directed y
act     = [(x,y)   | ["play", x, y, _] <- db]  -- x acted in y
play    = [(x,y,z) | ["play", x, y, z] <- db]  -- x played z in y
release = [(x,y)   | ["release", x, y] <- db]  -- x was released in y

--   ["release", "MV1", "YR1"],          -- MV1 was released in YR1
--   ["direct", "DRTR1", "MV1"],         -- DRTR1 directed the film MV1
--   ["play", "ACT1", "MV1", "CHR1"],    -- ACT1 played CHR1 in MV1
```

Database and Lambda abstraction

```
charP     = \x       -> elem x characters   -- is x a character
actorP    = \x       -> elem x actors       -- is x an actor
movieP    = \x       -> elem x movies       -- is x a movie
directorP = \x       -> elem x directors    -- is x a director
dateP     = \x       -> elem x dates        -- is x a date
actP      = \(x,y)   -> elem (x,y) act      -- did x act y
releaseP  = \(x,y)   -> elem (x,y) release  -- was x released in y
directP   = \(x,y)   -> elem (x,y) direct   -- did x direct y
playP     = \(x,y,z) -> elem (x,y,z) play   -- did x played z in y

--   ["release", "MV1", "YR1"],          -- MV1 was released in YR1
--   ["direct", "DRTR1", "MV1"],         -- DRTR1 directed the film MV1
--   ["play", "ACT1", "MV1", "CHR1"],    -- ACT1 played CHR1 in MV1
```

Database variables (1)

```
*Main> characters
["CHR1","CHR2","CHR3"]
*Main> movies
["MV1","MV2","MV3"]
*Main> actors
["ACT1","ACT2","ACT3"]
*Main> directors
["DRTR1","DRTR2","DRTR3"]
*Main> dates
["YR1","YR2","YR3"]
*Main> universe
["CHR1","CHR2","CHR3","ACT1","ACT2","ACT3","DRTR1","DRTR2",
"DRTR3","MV1","MV2","MV3","YR1","YR2","YR3"]

*Main>
*Main> direct
[("DRTR1","MV1"),("DRTR2","MV2"),("DRTR3","MV3")]
*Main> act
[("ACT1","MV1"),("ACT2","MV1"),("ACT3","MV3")]
*Main> play
[("ACT1","MV1","CHR1"),("ACT2","MV1","CHR2"),("ACT3","MV3","CHR3")]
*Main> release
[("MV1","YR1"),("MV2","YR2"),("MV3","YR3")]
```

Database variables (2)

```
*Main> fmap charP characters
[True,True,True]
*Main> fmap actorP actors
[True,True,True]
*Main> fmap movieP movies
[True,True,True]
*Main> fmap directorP directors
[True,True,True]
*Main> fmap actP act
[True,True,True]
*Main> fmap releaseP release
[True,True,True]
*Main> fmap directP direct
[True,True,True]
*Main> fmap playP play
[True,True,True]
```

Database Queries (1)

- ```
  q1 = [ x | x <- actors, directorP x]
  q2 = [ (x,y) | (x,y)  <- act, directorP x]

  actors     = nub [x | ["play",x,_,_]  <- db]      -- x played
  directorP = \x -> elem x directors               -- is x a director
  act        = [(x,y) | ["play", x, y, _] <- db]  -- x acted in y
  ```
- q1: give me the actors that also are directors
  (conjuctive queries)
- q2: give me the actors that also are directors
  together with the films in which they were acting
  ```

Database Queries (2)

- ```
  q3 = [ (x,y,z) | (x,y) <- direct, (y,z) <- release ]
  q4 = [ (x,y,z) | (x,y) <- direct, (u,z) <- release, y == u ]


  direct    = [(x,y) | ["direct", x, y]  <- db]  -- x directed y
  release   = [(x,y) | ["release", x, y] <- db]  -- x was released in y
  ```
- q3: not working two `y`'s are unrelated
- q4: give me all diredctors together with their films
  and their release dates
  ```

Database Queries (3)

- ```
  q5 = [ (x,y) | (x,y) <- direct, (u,"YR1") <- release, y == u ]
  q6 = [ (x,y,z) | (x,y) <- direct, (u,z) <- release, y == u, z > "YR1" ]

  direct  = [(x,y) | ["direct", x, y]  <- db]  -- x directed y
  release = [(x,y) | ["release", x, y] <- db]  -- x was released in y
  ```
- q5: give me all directors of films released in YR1, together with these films
- q6: give me all directors of films released after YR1, together with these films and their rlease dates
  ```

Database Queries (4)

- ```
  q7 = [ x | ("ACT1", x) <- act]
  q8 = [ x | (x, y) <- release, y > "YR1", actP("ACT2", x)]

  act     = [(x,y) | ["play", x, y, _] <- db]  -- x acted in y
  release = [(x,y) | ["release", x, y] <- db]  -- x was released in y
  actP    = \(x,y) -> elem (x,y) acts      -- did x act y
  ```
- q7: give me the films in which ACT1 acted
- q8: give me all films released after YR1
  in which ACT2 acted
  ```

Database Queries (5)

- ```
  q9   = q1 /= []
  q10  = [ x | ("DRTR1",x) <- direct ] /= []
  q10' = directorP "DRTR1"


  q1 <- [ x | x <- actors, directorP x]
  direct  = [(x,y) | ["direct", x, y]  <- db]  -- x directed y
  directorP = \x -> elem x directors             -- is x a director
  ```
- q9: are there any films in which the director was also an actor?
- q10: does the database contain films directed by DRT1?
  ```

Database Queries Results

```
*Main> q1
[]
*Main> q2
[]
*Main> q3
[("DRTR1","MV1","YR1"),("DRTR1","MV2","YR2"),("DRTR1","MV3","YR3"),
 ("DRTR2","MV1","YR1"),("DRTR2","MV2","YR2"),("DRTR2","MV3","YR3"),
 ("DRTR3","MV1","YR1"),("DRTR3","MV2","YR2"),("DRTR3","MV3","YR3")]
*Main> q4
[("DRTR1","MV1","YR1"),("DRTR2","MV2","YR2"),("DRTR3","MV3","YR3")]
*Main> q5
[("DRTR1","MV1")]
*Main> q6
[("DRTR2","MV2","YR2"),("DRTR3","MV3","YR3")]
*Main> q7
["MV1"]
*Main> q8
[]
*Main> q9
False
*Main> q10
True
*Main> q10'
True
```

Defining infinite sets

- List comprehension
- Lazy evaluation
- `naturals = [0..]`

```
evens1 = [ n | n <- naturals , even n ]
odds1 =  [ n | n <- naturals , odd n ]

evens2 = [ 2*n | n <- naturals ]
odds2  = [ 2*n+1 | n <- naturals ]

small_squares1 = [ n^2 | n <- [0..999] ]
small_squares2 = [ n^2 | n <- naturals , n < 1000 ]
```

Delete

- ```
  delete :: Eq a => a -> [a] -> [a]
  delete x [] = []
  delete x (y:ys) | x == y       = ys
                  | otherwise    =  y : delete x ys
  ```
- ```
  *Main> delete 3 [1, 2, 3, 4]
  [1,2,4]
  *Main>
  ```

Element

- ```
  elem' :: Eq a => a -> [a] -> Bool
  elem' x []                         = False
  elem' x (y:ys) | x == y            = True
                 | otherwise         = elem' x ys
  ```
- ```
  *Main> elem' 3 [1, 2, 3, 4]
  True
  *Main>
  ```

Union

- ```
  union :: Eq a => [a] -> [a] -> [a]
  union [] ys        = ys
  union (x:xs) ys    = x : union xs (delete x ys)
  ```
- ```
  *Main> union [1, 2, 3] [2, 3, 4, 5]
  [1,2,3,4,5]
  *Main>
  ```

Intersect

- ```
  intersect :: Eq a => [a] -> [a] -> [a]
  intersect [] s        = []
  intersect (x:xs) s | elem x s     = x : intersect xs s
                     | otherwise    = intersect xs s
  ```
- ```
  *Main> intersect [1, 2, 3] [2, 3, 4, 5]
  [2,3]
  *Main>
  ```

elem and notElem

- ```
  elem, notElem :: Eq a => a -> [a] -> Bool
  elem = any . (==)
  notElem = all . (/=)
  ```
- ```
  *Main> elem2 3 [1, 2, 3, 4]
  True
  *Main> notElem 5 [1, 2, 3, 4]
  True
  ```

addElem

- ```
  addElem :: a -> [[a]] -> [[a]]
  addElem x = map(x:)
  ```
- ```
  *Main> addElem 3 [[1], [2,3], [4,5,6]]
  [[3,1],[3,2,3],[3,4,5,6]]
  *Main>
  ```

- ```
  powerList :: [a] -> [[a]]
  powerList [] = [[]]
  powerList (x:xs) = (powerList xs) ++ (map (x:) (powerList xs))
  ```
- ```
  *Main> powerList [1,2]
  [[],[2],[1],[1,2]]
  *Main>
  (1: [2]) => [[], [2]] ++ [[1], [1,2]]
  (2: [ ]) => [] ++ [2]
  ```
- ```
  *Main> powerList [1,2,3]
  [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
  *Main>
  (1, [2,3]) => [[],[3],[2],[2,3]] ++ [[1],[1,3],[1,2],[1,2,3]]
  (2, [3])   => [[],[3]] ++ [[2], [2,3]] = [[],[3],[2],[2,3]]
  (3, [ ])   => [] ++ [3]
  ```
  ```

[]

- ```
  Prelude> :t [[], [[]]]
  [[], [[]]] :: [[[t]]]

  [] :: [[t]]
  [[]] :: [[t]]  --> [] :: [t]

  1st [] :: [[t]]
  2nd [] :: [t]

  Prelude> []
  []
  Prelude> [[], [[]]]
  [[],[[]]]

  *Main> :t []
  [] :: [t]
  *Main> :t [[]]
  [[]] :: [[t]]
  *Main> :t [[[]]]
  [[[]]] :: [[[t]]]
  *Main> :t empty
  empty :: [S]
  ```
  ```

Void

- ```
  data S = Void deriving (Eq,Show)
  empty :: [S]
  empty = []
  ```

- ```
  *Main> [] == []
  True
  *Main> [] == [[]]
  False
  *Main> [[]] == [[]]
  True
  *Main> []
  []
  *Main> [[]]
  [[]]
  *Main>
  ```

- a data type `S` containing a single object `Void`
- `Void` is used only to provide `empty` with a type

powerList empty

```
*Main> powerList empty
[[]]
*Main> powerList (powerList empty)
[[],[[]]]
*Main> powerList (powerList (powerList empty))
[[],[[[]]],[[]],[[],[[]]]]
*Main> powerList (powerList (powerList (powerList empty)))
[[],[[[],[[]]]],[[[]]],[[[]],[[],[[]]]],[[[[]]]],[[[[]]],[[],[[]]]],
[[[[]]],[[]]],[[[[]]],[[]],[[],[[]]]],[[]],[[],[[],[[]]]],[[],[[]]]],
[[],[[]],[[],[[]]]],[[],[[[]]]],[[],[[[]]],[[],[[]]]],[[],[[[]]],[[]]],
[[],[[[]]],[[]],[[],[[]]]]]
```

data

- ```
  data Bool = False | True
  data Color = Red | Green | Blue
  data Point a = Pt a a
  Pt 2.0 3.0 :: Point Float
  Pt 'a' 'b' :: Point Char
  Pt True False :: Point Bool
  data Point a = Point a a
  data Tree a = Leaf a | Branch (Tree a) (Tree a)
  Branch :: Tree a -> Tree a -> Tree a
  Leaf :: a -> Tree a
  ```

https://www.haskell.org/tutorial/classes.html
  ```

derving

- deriving automatically implements functions for a few of Haskell's typeclasses such as Show and Eq.
- This cannot be done with arbitrary typeclasses, but the ones for which deriving does work for are simple enough for automatic implementation.
- The Show typeclass defines functions for how to represent data types as a String.

https://stackoverflow.com/questions/44744884/what-does-deriving-do-mean-in-haskell

Difference Lists as functions

- A difference list representation of a list `xs :: [T]` is a function `f :: [T] -> [T]`
- when given another list `ys :: [T]` returns the list that `f` represents, prepended to `ys` i.e. `f ys = xs ++ ys`
- depending on usage patterns, difference lists can improve performance by effectively flattening the list building computations.

https://wiki.haskell.org/Difference_list

Difference Lists examples - usage patterns

- usage patterns

```
(show L)
++ (show T ++ (show R))

((show LL) ++ (show LT ++ (show LR)))
++ (show T ++ (show R))

(((show LLL) ++ (show LLT ++ (show LLR))) ++ (show LT ++ (show LR)))
++ (show T ++ (show R))
```

- ```
  (show L)
  ((show LL) ++ (show LT ++ (show LR)))
  (((show LLL) ++ (show LLT ++ (show LLR))) ++ (show LT ++ (show LR)))
  ```

https://wiki.haskell.org/Difference_list
  ```

- usage patterns and flattened results

```
(show L) ++ (show T ++ (show R))
 shows L . (shows T . shows R)

((show LL) ++ (show LT ++ (show LR)))  ++ (show T ++ (show R))
(shows LL . (shows LT . shows LR)) . (shows T . shows R)

(((show LLL) ++ (show LLT ++ (show LLR))) ++ (show LT ++ (show LR)))
++ (show T ++ (show R))
((shows LLL . (shows LLT . shows LLR)) . (shows LT . shows LR)) .
(shows T . shows R)
```

https://wiki.haskell.org/Difference_list

Difference Lists examples - efficiency

- flattening results

```
shows L . (shows T . shows R)
(shows LL . (shows LT . shows LR)) . (shows T . shows R)
((shows LLL . (shows LLT . shows LLR)) . (shows LT . shows LR)) .
(shows T . shows R)
```

-
```
((shows LLL.(shows LLT.shows LLR)).(shows LT.shows LR)).(shows T.shows R)
||          |-------------------|| |                 || |              |
||------------------------------| |-----------------|| |              |
|----------------------------------------------------| |--------------|
```

- still need to resolve three (.)
  until the first character of the result string,
  but for the subsequent characters
  you do not need to resolve those dots.
  In the end, resolution of all (.) may need some time
  but then concatenation is performed entirely right-associative.

https://wiki.haskell.org/Difference_list

ShowS type synonym

- `type ShowS = String -> String`
- `shows :: Show a => a -> ShowS`
- `show :: Show a => a -> String`

- The `shows` functions return a function
  that prepends the output String to an existing String
  `shows :: a -> String -> String`
  output string in `a -> String`
  existing string : the second string in `a -> String -> String`
- This allows constant-time concatenation of results
  using function composition.

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html

shows function examples

- `type ShowS = String -> String`
- `shows :: Show a => a -> ShowS`
- `show :: Show a => a -> String`
- ```
  Input : shows 12 "-14-16"     -- "12" : "-14-16"
  Output: "12-14-16"
  Input : shows "A" "SSS"       -- "\"A\"" : "SSS"
  Output: "\"A\"SSS"
  Input: shows 'A' "SSS"        -- "'A'" : "SSS"
  Output: "'A'SSS"
  ```

http://zvon.org/other/haskell/Outputprelude/shows_f.html
  ```

ShowS allows efficient concatenation

- `type ShowS = String -> String`
- a *difference list*
- a string xs is represented as a `ShowS`
  by the function `(xs ++)`
  that prepends it to any other list
- This allows efficient concatenation,
  avoiding the problems of nested left-associative concatenation
  (i.e. `((as ++ bs) ++cs) ++ ds`).
  - concatenate by a function composition
  - make a `String` by passing an empty list:

https://stackoverflow.com/questions/9197913/what-is-the-shows-trick-in-haskell

ShowS efficient concatenation examples

- concatenate by a function composition
- make a String by passing t an empty list:
- ```
  hello = ("hello" ++)
  world = ("world" ++)

  helloworld = hello . world       -- ("helloworld" ++)
  helloworld' = helloworld ""      -- "helloworld"
  ```

https://stackoverflow.com/questions/9197913/what-is-the-shows-trick-in-haskell
  ```

show implementation

- It's called ShowS because ShowS is used
  in the implementation of the standard Show typeclass
  to show efficiently large, deeply-nested structures

- show can be also be implemented by showsPrec,
  which has the type:
  `showsPrec :: (Show a) => Int -> a -> ShowS`
    - handles operator precedence
    - returns a ShowS value
    - The standard instances implement this
      instead of show for efficiency; `showsPrec 0 a ""`

https://stackoverflow.com/questions/9197913/what-is-the-shows-trick-in-haskell

showsPrec

- `showsPrec :: Int -> a -> ShowS`
  - `Int` : the operator precedence of the enclosing context (a number from 0 to 11). Function application has precedence 10.
  - `a` : the value to be converted to a String
  - `ShowS`
- Convert a value to a readable String.
- `showsPrec` should satisfy the law
  `showsPrec d x r ++ s  ==  showsPrec d x (r ++ s)`

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html

Precedence and fixities

```
+--------+-----------------+-----------------+-----------------+
|  Prec- | Left associative | Non-associative | Right associative|
| edence | operators       | operators       | operators       |
+--------+-----------------+-----------------+-----------------+
|      9 | !!              |                 | .               |
|      8 |                 |                 | ^,^^,**         |
|      7 | *,/,div,mod,rem, |                 |                 |
|        | quot            |                 |                 |
|      6 | +,-             |                 |                 |
|      5 |                 |                 | :,++            |
|      4 |                 | ==,/=,<,<=,>,>=, |                 |
|        |                 | elem,notElem    |                 |
|      3 |                 |                 | &&              |
|      2 |                 |                 | ||              |
|      1 | >>,>>=          |                 |                 |
|      0 |                 |                 | $,$!,seq        |
+--------+-----------------+-----------------+-----------------+
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html

Show class

- ```
  class  Show a  where
    {-# MINIMAL showsPrec | show #-}
    showsPrec :: Int -> a -> ShowS
    show      :: a   -> String

    showList  :: [a] -> ShowS
    showsPrec _ x s = show x ++ s
    show x          = shows x ""
    showList ls   s = showList__ shows ls s
  ```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html
  ```

Show instances

- ```
  instance  Show Char  where
    showsPrec _ '\'' = showString "'\\''"
    showsPrec _ c    = showChar '\'' . showLitChar c . showChar '\''

    showList cs = showChar '"' . showLitString cs . showChar '"'
  ```
- ```
  instance Show Int where
    showsPrec = showSignedInt
  ```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html

showsPrec exmples (1)

- ```
  data T = P :# P | T P   -- 1) P :# P (data P, operator :#, data P)
    deriving Show         -- 2) T P

  infix 6 :#              -- :# infix operator  with priority 6

  data P = P              -- type P, data P

  instance Show P where        -- type P
    showsPrec p P = shows p    -- p : priority integer, data P
  ```

https://stackoverflow.com/questions/27471937/
showsprec-and-operator-precedences/27473420
  ```

showsPrec exmples (2)

- ```
  data T = P :# P | T P         -- type T
    deriving Show               -- values 1) P :# P 2) T P

  data P = P                    -- type P, value P
  ```
- use the data keyword to define a new data type
- value constructors specify the different values that this type can have
- both the type name and the value constructors have to be capital cased

https://stackoverflow.com/questions/27471937/showsprec-and-operator-precedences/27473420
  ```

showsPrec exmples (3)

- the type `T` can have a value of `P :# P` or `T P`
- the type `P` can have a value of `P`

- ```
  type ShowS = String -> String
  showsPrec : Int -> a -> ShowS
  showsPrec p P = shows p     -- p : priority integer

  (P :# P) :: T type
  (T P) :: T type

  data P = P                    -- type P, value P
  ```

https://stackoverflow.com/questions/27471937/showsprec-and-operator-precedences/27473420
  ```

showsPrec exmples (4)

- with `infix 6 :#`,
  the `Show T` instance calls `showsPrec 7`
  on the arguments to :#, and also it
  shows parentheses only at precedences > 6:

- ```
  *Main> showsPrec 6 (P :# P) ""
  "7 :# 7"

  :# (priority 6), showsPrec (recision 7), no need parenthesis

  *Main> showsPrec 7 (P :# P) ""
  "(7 :# 7)"

  :# (priority 7, showsPrec (recision 7),  parenthesis
  ```

https://stackoverflow.com/questions/27471937/showsprec-and-operator-precedences/27473420
  ```

showsPrec exmples (5)

- And for the ordinary constructor `T`,
  he generated instance calls `showsPrec 11` on the argument and shows parens at precedences > 10:

- ```
  *Main> showsPrec 10 (T P) ""
  "T 11"
  *Main> showsPrec 11 (T P) ""
  "(T 11)"
  ```

https://stackoverflow.com/questions/27471937/showsprec-and-operator-precedences/27473420
  ```

ShowString

- `showString :: String -> ShowS`
- utility function converting a String to a show function that simply prepends the string unchanged.
- ```
  Prelude> showString "AAA" ""
  "AAA"
  Prelude> showString "AAA" "BBB"
  "AAABBB"
  ```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html
  ```

ShowChar

- `showChar :: Char -> ShowS`
- utility function converting a Char to a show function that simply prepends the character unchanged.
- ```
  Prelude> showString ’A’ ""
  "A"
  Prelude> showString ’A’ "BBB"
  "ABBB"
  ```

http://hackage.haskell.org/package/base-4.12.0.0/docs/Text-Show.html
  ```

SetEq (1)

- ```
  newtype Set a = Set [a]

  instance (Eq a) => Eq (Set a) where
    set1 == set2 = subSet set1 set2 && subSet set2 set1

  instance (Show a) => Show (Set a) where
      showsPrec _ (Set s) str = showSet s str

  showSet []     str = showString "{}" str
  showSet (x:xs) str = showChar '{' (shows x (showl xs str))
       where showl []     str = showChar '}' str
             showl (x:xs) str = showChar ',' (shows x (showl xs str))
  ```
  ```

SetEq (2)

```
emptySet  :: Set a
emptySet = Set []

isEmpty  :: Set a -> Bool
isEmpty (Set []) = True
isEmpty  _       = False

inSet  :: (Eq a) => a -> Set a -> Bool
inSet x (Set s) = elem x s

subSet :: (Eq a) => Set a -> Set a -> Bool
subSet (Set [])      _    = True
subSet (Set (x:xs)) set = (inSet x set) && subSet (Set xs) set

insertSet :: (Eq a) => a -> Set a -> Set a
insertSet x (Set ys) | inSet x (Set ys) = Set ys
                     | otherwise        = Set (x:ys)

deleteSet :: Eq a => a -> Set a -> Set a
deleteSet x (Set xs) = Set (delete x xs)
```

SetEq (3)

```
list2set :: Eq a => [a] -> Set a
list2set [] = Set []
list2set (x:xs) = insertSet x (list2set xs)

powerSet :: Eq a => Set a -> Set (Set a)
powerSet (Set xs) = Set (map (\xs -> (Set xs)) (powerList xs))

powerList  :: [a] -> [[a]]
powerList  [] = [[]]
powerList  (x:xs) = (powerList xs) ++ (map (x:) (powerList xs))

takeSet :: Eq a => Int -> Set a -> Set a
takeSet n (Set xs) = Set (take n xs)

infixl 9 !!!
(!!!) :: Eq a => Set a -> Int -> a
(Set xs) !!! n = xs !! n
```

SetEq (4)

- ```
  instance Eq a => Eq (Set a) where
    set1 == set2  = subSet set1 set && subSet set2 set1
  ```
  ```

SetEq (5)

- ```
  instance (Show a) => Show (Set a) where
    showsPrec _ (Set s) = showSet s

  showSet []     str = showString "{}" str
  showSet (x:xs) str = showChar '{' (shows x ( showl xs str))
    where showl []     str = showChar '}' str
          showl (x:xs) str = showChar ',' (shows x (showl xs str))
  ```
  ```

showSet

- ```
  showSet []     str = showString "{}" str
  showSet (x:xs) str = showChar '{' (shows x (showl xs str))
       where showl []     str = showChar '}' str
             showl (x:xs) str = showChar ',' (shows x (showl xs str))
  ```
- ```
  *Main> showSet [1,2,3] "AAA"
  "{1,2,3}AAA"
  *Main> showSet [1,2,3] ""
  "{1,2,3}"
  *Main> showSet [1,[2,2],3] ""
  *Main> showSet [1,2,2,3] ""
  "{1,2,2,3}"
  ```

Hierarchy (1)

```
module Hierarchy where

import SetEq

data S = Void deriving (Eq,Show)
empty,v0,v1,v2,v3,v4,v5 :: Set S

empty = Set []
v0 = empty
v1 = powerSet v0
v2 = powerSet v1
v3 = powerSet v2
v4 = powerSet v3
v5 = powerSet v4
```

Hierarchy (2)

```
display :: Int -> String -> IO ()
display n str = putStrLn (display' n 0 str)
  where
  display' _ _ [] = []
  display' n m (x:xs) | n == m = '\n' : display' n 0 (x:xs)
                      | otherwise = x : display' n (m+1) xs
```