Specification and Testing Tools
+
Lists, Sets and Relations

Jan van Eijck
CWI & ILLC, Amsterdam

Specification and Testing, Week 4, 2014

Abstract

Specification and (automated) testing tools for Haskell. Hspec: based on Ruby Rspec. Quickcheck: automated testing tool that became very influential.

Working with Lists, Sets and Relations. Checking invariants of operations on datatypes.

Keywords:

Informal and formal specification, test generation, magic square, Sudoku grid, Sudoku puzzle, constraint solving, depth first search, problem generation.


```
module ST where

import Test.Hspec
import Test.QuickCheck
import System.Random
import Data.List
```

Hspec

Use tests as documentation.

See http://hspec.github.io/

Work through the users manual ...

describe

```
describe :: String
         -> [IO (String, Result)] -> IO [Spec]
```

First argument: The name of what is being described, usually a function or type.

Second argument: a list of behaviors and examples, created by a list of `it`.

Create a set of specifications for a specific type being described. Once you know what you want specs for, use this.

```
describe "abs" [
  it "returns a positive number given a negative number"
    (abs (-1) == 1)
   ]
```

it

```
it :: SpecVerifier a =>
      String -> a -> IO (String, Result)
```

First argument: A description of this behavior.

Second argument: An example for this behavior.

Output: description plus result of testing the example.

```
describe "closeEnough" [
  it "is true if two numbers are almost the same"
    (1.001 `closeEnough` 1.002),

  it "is false if two numbers are not almost the same"
    (not $ 1.001 `closeEnough` 1.003)
   ]
```

hspec

```
hspec :: IO [Spec] -> IO ()
```

Create a document of the given specs and write it to stdout. This does track how much time it took to check the examples.

Use this if you want a description of each spec and do need to know how long it tacks to check the examples or want to write to stdout.

pending

```
pending :: String -> Result
```

Input: An explanation for why this behavior is pending.

Declare an example as not successful or failing but pending some other work. If you want to report on a behavior but don't have an example yet, use this.

```
describe "fancyFormatter" [
    it "can format text in a way that everyone likes"
      (pending "waiting for clarification from the designers"
    ]
```

descriptions

```
descriptions :: [IO [Spec]] -> IO [Spec]
```

Combine a list of descriptions.

Exercise: guess how this is implemented with `sequence`.

descriptions

```
descriptions :: [IO [Spec]] -> IO [Spec]
```

Combine a list of descriptions.

Exercise: guess how this is implemented with `sequence`.

```
descriptions :: [IO [Spec]] -> IO [Spec]
descriptions = liftM concat . sequence
```

Further Documentation

https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspe
html

QuickCheck

Very influential random test generation approach to testing. See http://en.wikipedia.org/wiki/QuickCheck.

"In QuickCheck the programmer writes assertions about logical properties that a function should fulfill. Then QuickCheck attempts to generate test cases that falsify these assertions. The project was started in 1999. Besides being used to test regular programs, QuickCheck is also useful for building up a functional specification, for documenting what functions should be doing, and for testing compiler implementations."

https://github.com/nick8325/quickcheck

Tutorial:

http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck2

Original paper: [1]

QuickCheck Example: Pebble Game

```
data Color = W | B deriving (Eq,Show)

drawPebble :: [Color] -> [Color]
drawPebble [] = []
drawPebble [x] = [x]
drawPebble (W:W:xs) = drawPebble (B:xs)
drawPebble (B:B:xs) = drawPebble (B:xs)
drawPebble (W:B:xs) = drawPebble (W:xs)
drawPebble (B:W:xs) = drawPebble (W:xs)
```

What is the colour of the last pebble?

```
*ST> drawPebble [W,W,B,B]
[B]
*ST> drawPebble [W,W,B,B,W,W,W]
[W]
*ST> drawPebble [W,W,B,B,W,W,W,B,W,B,B]
[B]
*ST> drawPebble [W,W,B,B,W,W,W,B,W,B,B,W]
[W]
*ST> drawPebble [W,W,B,B,W,W,W,B,W,B,B,W,W,W,W]
[B]
```

Test Generators

```
newtype Gen a = MkGen{ unGen :: StdGen -> Int -> a }

instance Functor Gen where
  fmap f (MkGen h) =
    MkGen (\r n -> f (h r n))

instance Monad Gen where
  return x =
    MkGen (\_ _ -> x)

  MkGen m >>= k =
    MkGen (\r n ->
      let (r1,r2)  = split r
          MkGen m' = k (m r1 n)
       in m' r2 n
    )
```

RandomGen, next, split

The class RandomGen, defined in System.Random, provides a common interface to random number generators.

Minimal complete definition: `next` and `split`.

```
next :: g -> (Int, g)
```

The next operation returns an Int that is uniformly distributed in the range returned by genRange (including both end points), and a new generator.

```
split :: g -> (g, g)
```

The split operation allows one to obtain two distinct random number generators. This is very useful in functional programs (for example, when passing a random number generator down to recursive calls).

StdGen

Instance of RandomGen.

Defined in System.Random module.

choose

Defined in QuickCheck module.

Generates a random element in the given inclusive range.

```
choose :: Random a => (a,a) -> Gen a
choose rng = MkGen (\r _ ->
                 let (x,_) = randomR rng r in x)
```

Random a

Defined in QuickCheck module.

With a source of random number supply in hand, the Random class allows the programmer to extract random values of a variety of types.

Minimal complete definition: `randomR` and `random`.

```
randomR :: RandomGen g => (a, a) -> g -> (a, g)
```

Takes a range (lo,hi) and a random number generator g, and returns a random value uniformly distributed in the closed interval [lo,hi], together with a new generator. It is unspecified what happens if lo¿hi. For continuous types there is no requirement that the values lo and hi are ever produced, but they may be, depending on the implementation and the interval.

```
random :: RandomGen g => g -> (a, g)
```

The same as randomR, but using a default range determined by the type:

For bounded types (instances of Bounded, such as Char), the range is normally the whole type. For fractional types, the range is normally the semi-closed interval [0,1). For Integer, the range is (arbitrarily) the range of Int.

Class Arbitrary

Defined in QuickCheck module.

Random generation and shrinking of values.

```
class Arbitrary a where
  -- | A generator for values of the given type.
  arbitrary :: Gen a
  arbitrary = error "no default generator"

  -- | Produces a (possibly) empty list of
  -- all the possible
  -- immediate shrinks of the given value.
  shrink :: a -> [a]
  shrink _ = []
```

Making Color an Instance of Arbitrary

```
instance Arbitrary Color where
   arbitrary = oneof [return W, return B]
```

This allows QuickCheck to derive `Arbitrary [Color]`...

Here is how:

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized $ \n ->
    do k <- choose (0,n)
       sequence [ arbitrary | _ <- [1..k] ]

  shrink xs = removeChunks xs ...
```

sample for generating example values

```
sample' :: Gen a -> IO [a]
sample' (MkGen m) =
  do rnd0 <- newStdGen
     let rnds rnd = rnd1 : rnds rnd2
                    where (rnd1,rnd2) = split rnd
     return [(m r n) |
             (r,n) <- rnds rnd0 `zip` [0,2..20] ]
```

Generating some example values and print them to 'stdout':

```
sample :: Show a => Gen a -> IO ()
sample g =
  do cases <- sample' g
     sequence_ (map print cases)
```

Example Use

```
*ST> sample $ (arbitrary :: Gen [Color])
[]
[W]
[W,W]
[W,B,W]
[W,B,W,B]
[B,B,B,W,B,W,W]
[B,W,B,W,B,W,B,W,B,B,W]
[]
[B,B,B,B,B,W,B,B,W,B,W,W,W,B]
[W,W,B,B,W,W,B,W,B,B,B,B,W,W,B]
[W,W,W,W,W,W,B,B,B,W,B,W,B,B]
```

Stating an Invariant

Stating an Invariant

```
numberW :: [Color] -> Int
numberW = length . (filter (== W))

parityW :: [Color] -> Int
parityW xs =  mod (numberW xs) 2

prop_invariant xs =
  parityW xs == parityW (drawPebble xs)
```

Testing This . . .

```
ST> quickCheck prop_invariant
+++ OK, passed 100 tests.
```

See what happens with an unreasonable property

Strange invariant:

```
prop_length xs = length xs == length (drawPebble xs)
```

```
ST> quickCheck prop_length
*** Failed! Falsifiable (after 7 tests and 1 shrink):
[B,B]
*ST> quickCheck prop_length
*** Failed! Falsifiable (after 6 tests and 1 shrink):
[B,W]
*ST> quickCheck prop_length
*** Failed! Falsifiable (after 3 tests):
[B,B]
```

Lists Versus Sets

Background:

Lists Versus Sets

Background:

1. Chapter 4 of “The Haskell Road”: Sets, Types and Lists

Lists Versus Sets

Background:

1. Chapter 4 of “The Haskell Road”: Sets, Types and Lists
2. Chapter 5 of “The Haskell Road”: Relations

Extensionality and Subsets

Sets that have the same elements are equal.

For all sets A and B, it holds that:

$$\forall x(x \in A \iff x \in B) \implies A = B.$$

Extensionality and Subsets

Sets that have the same elements are equal.

For all sets A and B, it holds that:

$$\forall x(x \in A \Longleftrightarrow x \in B) \Longrightarrow A = B.$$

The set A is a subset of the set B, and B a superset of A; notations: $A \subseteq B$, and $B \supseteq A$, if every member of A is also a member of B.

$$\forall x \, (x \in A \implies x \in B).$$

Extensionality and Subsets

Sets that have the same elements are equal.

For all sets A and B, it holds that:

$$\forall x(x \in A \iff x \in B) \implies A = B.$$

The set A is a subset of the set B, and B a superset of A; notations: $A \subseteq B$, and $B \supseteq A$, if every member of A is also a member of B.

$$\forall x\ (x \in A \implies x \in B).$$

If $A \subseteq B$ and $A \neq B$, then A is a proper subset of B.

Abstraction, Comprehension

A set is a collection into a whole of definite, distinct objects of our intuition or of our thought. The objects are called the elements (members) of the set.

$$\{x \mid x \in U, E(x)\}$$

Abstraction, Comprehension

A set is a collection into a whole of definite, distinct objects of our intuition or of our thought. The objects are called the elements (members) of the set.

$$\{x \mid x \in U, E(x)\}$$

Cf list comprehension in Haskell.

Assume `list :: [a]` and `foo :: a -> Bool`. Then a list can be defined with: `[ x | x <- list, foo x ]`.

```
evens1 = [ n | n <- [0..], even n ]
```

Notation

If f is an operation, then

$$\{ f(x) \mid P(x) \}$$

denotes the set of things of the form $f(x)$ where the object x has the property P.

Notation

If f is an operation, then

$$\{ f(x) \mid P(x) \}$$

denotes the set of things of the form $f(x)$ where the object x has the property P.

$$\{ 2n \mid n \in \mathbb{N} \}$$

is another notation for the set of even natural numbers.

Notation

If f is an operation, then

$$\{\ f(x) \mid P(x)\ \}$$

denotes the set of things of the form $f(x)$ where the object x has the property P.

$$\{\ 2n \mid n \in \mathbb{N}\ \}$$

is another notation for the set of even natural numbers.

Haskell counterpart for lists:

```
evens2 = [ 2*n | n <- [0..] ]
```

Notation

If f is an operation, then

$$\{\, f(x) \mid P(x) \,\}$$

denotes the set of things of the form $f(x)$ where the object x has the property P.

$$\{\, 2n \mid n \in \mathbb{N} \,\}$$

is another notation for the set of even natural numbers.

Haskell counterpart for lists:

```
evens2 = [ 2*n | n <- [0..] ]
```

Compare:

```
naturals = [0..]
small_squares1 = [ n^2 | n <- [0..999] ]
small_squares2 = [ n^2 | n <- naturals , n < 1000 ]
small_squares3 = take 1000 [ n^2 | n <- naturals ]
```

Halting Problem

Suppose there is a total function `halts :: String -> String -> Bool` that checks whether a function (a program in some language, given by a string) is defined on a given input (also given by a string). Consider:

```
funny :: String -> Bool
funny = \x -> if halts x x then undefined else True
```

Halting Problem

Suppose there is a total function `halts :: String -> String -> Bool` that checks whether a function (a program in some language, given by a string) is defined on a given input (also given by a string). Consider:

```
funny :: String -> Bool
funny = \x -> if halts x x then undefined else True
```

Note that if `halts` behaves as supposed, then this is well-defined.

Halting Problem

Suppose there is a total function `halts :: String -> String -> Bool` that checks whether a function (a program in some language, given by a string) is defined on a given input (also given by a string). Consider:

```
funny :: String -> Bool
funny = \x -> if halts x x then undefined else True
```

Note that if `halts` behaves as supposed, then this is well-defined.

Now what about `funny "\x -> if halts x x then undefined else True"`

Suppose There is a Test for Equality of Functions

Such a test would solve the halting problem:

```
halts f x =  f /= g where g y | y == x    = undefined
                              | otherwise = f y
```

Suppose There is a Test for Equality of Functions

Such a test would solve the halting problem:

```
halts f x =  f /= g where g y | y == x     = undefined
                              | otherwise = f y
```

Conclusion: functions cannot be in the class `Eq`.

Suppose There is a Test for Equality of Functions

Such a test would solve the halting problem:

```
halts f x =  f /= g where g y | y == x     = undefined
                              | otherwise = f y
```

Conclusion: functions cannot be in the class `Eq`.

Types and Type Classes are a regulation of the language to rule out paradoxes . . .

Empty Set, Singletons, Empty List, Unit Lists

Empty Set, Singletons, Empty List, Unit Lists

- A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.

Empty Set, Singletons, Empty List, Unit Lists

- A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.
- A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$.

Empty Set, Singletons, Empty List, Unit Lists

- A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.
- A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$.
- Do not confuse d with $\{d\}$.

Empty Set, Singletons, Empty List, Unit Lists

- A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.
- A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$.
- Do not confuse d with $\{d\}$.
- Empty list: `[]`.

Empty Set, Singletons, Empty List, Unit Lists

- A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.
- A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$.
- Do not confuse d with $\{d\}$.
- Empty list: `[]`.
- Unit list: `[d]`.

Empty Set, Singletons, Empty List, Unit Lists

- A set A is empty if it has no elements. By extensionality, there is just one empty set, so we may give it a name: $\emptyset$.
- A set A that has just one member d is called a singleton. The singleton whose only element is d is $\{d\}$.
- Do not confuse d with $\{d\}$.
- Empty list: `[]`.
- Unit list: `[d]`.
- If `d :: a` then `[d] :: [a]`.

Operations on Sets

Operations on Sets

- Intersection: $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$

Operations on Sets

- Intersection: $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$
- Union: $A \cup B = \{\, x \mid x \in A \vee x \in B \,\}$.

Operations on Sets

- Intersection: $A \cap B = \{ x \mid x \in A \wedge x \in B \}$
- Union: $A \cup B = \{ x \mid x \in A \vee x \in B \}$.
- Difference: $A - B = \{ x \mid x \in A \wedge x \notin B \}$.

Properties:

Operations on Sets

- Intersection: $A \cap B = \{ x \mid x \in A \wedge x \in B \}$
- Union: $A \cup B = \{ x \mid x \in A \vee x \in B \}$.
- Difference: $A - B = \{ x \mid x \in A \wedge x \notin B \}$.

Properties:

- $A \cap \emptyset = \emptyset$, $A \cup \emptyset = A$

Operations on Sets

- Intersection: $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$
- Union: $A \cup B = \{\, x \mid x \in A \vee x \in B \,\}$.
- Difference: $A - B = \{\, x \mid x \in A \wedge x \notin B \,\}$.

Properties:

- $A \cap \emptyset = \emptyset$, $A \cup \emptyset = A$
- $A \cap A = A$, $A \cup A = A$ (idempotence)

Operations on Sets

- Intersection: $A \cap B = \{ x \mid x \in A \wedge x \in B \}$
- Union: $A \cup B = \{ x \mid x \in A \vee x \in B \}$.
- Difference: $A - B = \{ x \mid x \in A \wedge x \notin B \}$.

Properties:

- $A \cap \emptyset = \emptyset$, $A \cup \emptyset = A$
- $A \cap A = A$, $A \cup A = A$ (idempotence)
- $A \cap B = B \cap A$, $A \cup B = B \cup A$ (commutativity)

Operations on Sets

- Intersection: $A \cap B = \{\, x \mid x \in A \wedge x \in B \,\}$
- Union: $A \cup B = \{\, x \mid x \in A \vee x \in B \,\}$.
- Difference: $A - B = \{\, x \mid x \in A \wedge x \notin B \,\}$.

Properties:

- $A \cap \emptyset = \emptyset$, $A \cup \emptyset = A$
- $A \cap A = A$, $A \cup A = A$ (idempotence)
- $A \cap B = B \cap A$, $A \cup B = B \cup A$ (commutativity)
- $A \cap (B \cap C) = (A \cap B) \cap C$, $A \cup (B \cup C) = (A \cup B) \cup C$ (associativity)

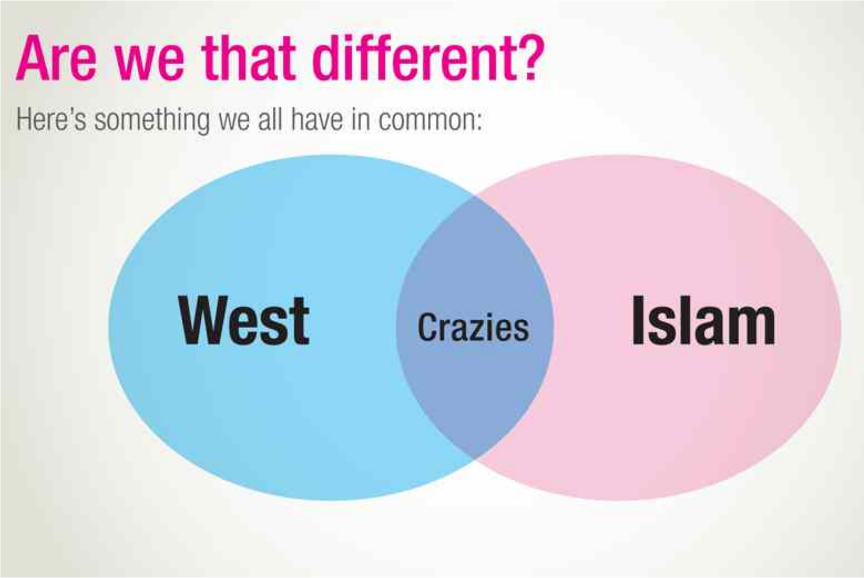
Operations on Sets

- Intersection: $A \cap B = \{ x \mid x \in A \land x \in B \}$
- Union: $A \cup B = \{ x \mid x \in A \lor x \in B \}$.
- Difference: $A - B = \{ x \mid x \in A \land x \notin B \}$.

Properties:

- $A \cap \emptyset = \emptyset$, $A \cup \emptyset = A$
- $A \cap A = A$, $A \cup A = A$ (idempotence)
- $A \cap B = B \cap A$, $A \cup B = B \cup A$ (commutativity)
- $A \cap (B \cap C) = (A \cap B) \cap C$, $A \cup (B \cup C) = (A \cup B) \cup C$ (associativity)
- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

Venn Diagram of Set Intersection



Powerset and Powerlist

The powerset of the set X is the set $\mathcal{P}(X) = \{ A \mid A \subseteq X \}$.

We have: $\emptyset \in \mathcal{P}(X)$ and $X \in \mathcal{P}(X)$.

Powerset and Powerlist

The powerset of the set X is the set $\mathcal{P}(X) = \{ A \mid A \subseteq X \}$.

We have: $\emptyset \in \mathcal{P}(X)$ and $X \in \mathcal{P}(X)$.

Sublists and the Power List Operation:

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

```
Main> powerList [1,2,3]
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

Lists and List Equality

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

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

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x==y && xs==ys
    _      == _      =  False
```

List Ordering

```
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
```

Fundamental List Operations

```
head                :: [a] -> a
head (x:_)           = x
tail                :: [a] -> [a]
tail (_:xs)          = xs
last                :: [a] -> a
last [x]             = x
last (_:xs)          = last xs
init                :: [a] -> [a]
init [x]             = []
init (x:xs)          = x : init xs
null                :: [a] -> Bool
null []              = True
null (_:_)           = False
```

Using Lists to Represent Sets

Representing Sets as Lists Without Duplicates:

Removing duplicates with `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
```

Deleting Elements, Finding Elements

```
delete :: Eq a => a -> [a] -> [a]
delete x [] = []
delete x (y:ys) | x == y     = ys
                | otherwise = y: delete x ys
```

Deleting Elements, Finding Elements

```
delete :: Eq a => a -> [a] -> [a]
delete x [] = []
delete x (y:ys) | x == y     = ys
                | otherwise = y: delete x ys
```

Note: this only deletes the first occurrence of the item!

Deleting Elements, Finding Elements

```
delete :: Eq a => a -> [a] -> [a]
delete x [] = []
delete x (y:ys) | x == y    = ys
                | otherwise = y: delete x ys
```

Note: this only deletes the first occurrence of the item!

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

List Union and Intersection

```
union :: Eq a => [a] -> [a] -> [a]
union [] ys     = ys
union (x:xs) ys = x : union xs (delete x ys)

intersect :: Eq a => [a] -> [a] -> [a]
intersect []     s               = []
intersect (x:xs) s | elem x s    = x : intersect xs s
                   | otherwise   =     intersect xs s
```

A Datatype for Sets

```
module SetOrd (Set(..),emptySet,isEmpty,inSet,subSet,insertSet
               deleteSet,powerSet,takeSet,(!!!),list2set)

where

import List (sort)
{-- Sets implemented as ordered lists without duplicates --}

newtype Set a = Set [a] deriving (Eq,Ord)

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
```

This gives:

```
SetEq> Set [1..10]
{1,2,3,4,5,6,7,8,9,10}
```

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

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

inSet  :: (Ord a) => a -> Set a -> Bool
inSet x (Set s) = elem x (takeWhile (<= x) s)

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

insertSet :: (Ord a) => a -> Set a -> Set a
```

```haskell
insertSet x (Set s) = Set (insertList x s)

insertList x [] = [x]
insertList x ys@(y:ys') = case compare x y of
                                 GT -> y : insertList x ys'
                                 EQ -> ys
                                 _  -> x : ys

deleteSet :: Ord a => a -> Set a -> Set a
deleteSet x (Set s) = Set (deleteList x s)

deleteList x [] = []
deleteList x ys@(y:ys') = case compare x y of
                                 GT -> y : deleteList x ys'
                                 EQ -> ys'
                                 _  -> ys

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

```
-- list2set xs = Set (foldr insertList [] xs)

powerSet :: Ord a => Set a -> Set (Set a)
powerSet (Set xs) =
   Set (sort (map (\xs -> (list2set xs)) (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
```

This gives:

```
SetEq> Set [1..10]
{1,2,3,4,5,6,7,8,9,10}
emptySet  :: Set a
emptySet = Set []
```

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

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)

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))

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
```

Invariants

Key Notion: Invariants For Operations

Invariants

Key Notion: Invariants For Operations

- Suppose f is an operation of type `a -> a`, or of type `a -> a -> a`,

Invariants

Key Notion: Invariants For Operations

- Suppose f is an operation of type `a -> a`, or of type `a -> a -> a`,
- Suppose φ has type `a -> Bool`.

Invariants

Key Notion: Invariants For Operations

- Suppose f is an operation of type `a -> a`, or of type `a -> a -> a`,
- Suppose φ has type `a -> Bool`.
- Then φ is an invariant for f if the following holds:

  If $\varphi(x)$ then $\varphi(f(x))$.

  If $\varphi(x)$ and $\varphi(y)$ then $\varphi(f(x, y))$.

Invariants

Key Notion: Invariants For Operations

- Suppose f is an operation of type `a -> a`, or of type `a -> a -> a`,
- Suppose φ has type `a -> Bool`.
- Then φ is an invariant for f if the following holds:

  If $\varphi(x)$ then $\varphi(f(x))$.

  If $\varphi(x)$ and $\varphi(y)$ then $\varphi(f(x, y))$.

Examples:

Invariants

Key Notion: Invariants For Operations

- Suppose f is an operation of type `a -> a`, or of type `a -> a -> a`,
- Suppose φ has type `a -> Bool`.
- Then φ is an invariant for f if the following holds:

  If $\varphi(x)$ then $\varphi(f(x))$.

  If $\varphi(x)$ and $\varphi(y)$ then $\varphi(f(x, y))$.

Examples:

- If `Set ys` has the property that `ys` does not contain duplicates, then `insert x (Set ys)` does not contain duplicates.

Invariants

Key Notion: Invariants For Operations

- Suppose f is an operation of type `a -> a`, or of type `a -> a -> a`,
- Suppose φ has type `a -> Bool`.
- Then φ is an invariant for f if the following holds:

  If $\varphi(x)$ then $\varphi(f(x))$.
  If $\varphi(x)$ and $\varphi(y)$ then $\varphi(f(x, y))$.

Examples:

- If `Set ys` has the property that `ys` does not contain duplicates, then `insert x (Set ys)` does not contain duplicates.
- If `Set ys` has the property that `ys` does not contain duplicates, then `delete x (Set ys)` does not contain duplicates.

Relations

Relations as Boolean Functions and as Sets of Pairs

Relations

Relations as Boolean Functions and as Sets of Pairs

- A relation R on a set A can be viewed as a function of type $A \to A \to \text{Bool}$.

Relations

Relations as Boolean Functions and as Sets of Pairs

- A relation R on a set A can be viewed as a function of type $A \to A \to$ Bool.
- Example: $<$ on $\mathbb{N}$. For every two numbers $n, m \in \mathbb{N}$, the statement $n < m$ is either true or false. E.g., $3 < 5$ is true, whereas $5 < 2$ is false.

Relations

Relations as Boolean Functions and as Sets of Pairs

- A relation R on a set A can be viewed as a function of type $A \to A \to$ Bool.
- Example: $<$ on $\mathbb{N}$. For every two numbers $n, m \in \mathbb{N}$, the statement $n < m$ is either true or false. E.g., $3 < 5$ is true, whereas $5 < 2$ is false.
- In general: to a relation you can "input" a pair of objects, after which it "outputs" either true or false. depending on whether these objects are in the relationship given.

Relations

Relations as Boolean Functions and as Sets of Pairs

- A relation R on a set A can be viewed as a function of type $A \to A \to$ Bool.
- Example: $<$ on $\mathbb{N}$. For every two numbers $n, m \in \mathbb{N}$, the statement $n < m$ is either true or false. E.g., $3 < 5$ is true, whereas $5 < 2$ is false.
- In general: to a relation you can "input" a pair of objects, after which it "outputs" either true or false. depending on whether these objects are in the relationship given.
- Alternative view: relation R on A as a set of pairs:

$$\{(a, b) \in A^2 \mid Rab \text{ is true } \}.$$

Relations

Relations as Boolean Functions and as Sets of Pairs

- A relation R on a set A can be viewed as a function of type $A \to A \to$ Bool.
- Example: $<$ on $\mathbb{N}$. For every two numbers $n, m \in \mathbb{N}$, the statement $n < m$ is either true or false. E.g., $3 < 5$ is true, whereas $5 < 2$ is false.
- In general: to a relation you can "input" a pair of objects, after which it "outputs" either true or false. depending on whether these objects are in the relationship given.
- Alternative view: relation R on A as a set of pairs:

$$\{(a, b) \in A^2 \mid Rab \text{ is true }\}.$$

- $<$ on $\mathbb{N}$ = $\{(0,1),(0,2),(1,2),(0,3),(1,3),(2,3),\ldots\}$

Domain and Range

The set $\mathrm{dom}\,(R) = \{x \mid \exists y\,(\,Rxy\,)\}$, the set of all first coordinates of pairs in R, is called the domain of R.

The set ran$(R) = \{y \mid \exists x\,(\,Rxy\,)\}$, the set of second coordinates of pairs in R, is called the range of R.

The relation R is a relation from A to B or between A and B, if $\mathrm{dom}\,(R) \subseteq A$ and ran$(R) \subseteq B$.

A relation from A to A is called on A.

$R = \{(1,4),(1,5),(2,5)\}$ is a relation from $\{1,2,3\}$ to $\{4,5,6\}$, and it also is a relation on $\{1,2,4,5,6\}$. Furthermore, $\mathrm{dom}\,(R) = \{1,2\}$, ran$(R) = \{4,5\}$.

Identity and Inverse

Identity and Inverse

- $\Delta_A = \{ (a,b) \in A^2 \mid a = b \} = \{ (a,a) \mid a \in A \}$ is a relation on A, the identity on A.

Identity and Inverse

- $\Delta_A = \{ (a, b) \in A^2 \mid a = b \} = \{ (a, a) \mid a \in A \}$ is a relation on A, the identity on A.
- If R is a relation between A and B, then $R^{-1} = \{ (a, b) \mid Rba \}$, the inverse of R, is a relation between B and A.

Identity and Inverse

- $\Delta_A = \{ (a,b) \in A^2 \mid a = b \} = \{ (a,a) \mid a \in A \}$ is a relation on A, the identity on A.
- If R is a relation between A and B, then $R^{-1} = \{ (a,b) \mid Rba \}$, the inverse of R, is a relation between B and A.
- The inverse of the relation 'parent of' is the relation 'child of'.

Identity and Inverse

- $\Delta_A = \{ (a,b) \in A^2 \mid a = b \} = \{ (a,a) \mid a \in A \}$ is a relation on A, the identity on A.
- If R is a relation between A and B, then $R^{-1} = \{ (a,b) \mid Rba \}$, the inverse of R, is a relation between B and A.
- The inverse of the relation 'parent of' is the relation 'child of'.
- $A \times B$ is the biggest relation from A to B.

Identity and Inverse

- $\Delta_A = \{ (a,b) \in A^2 \mid a = b \} = \{ (a,a) \mid a \in A \}$ is a relation on A, the identity on A.
- If R is a relation between A and B, then $R^{-1} = \{ (a,b) \mid Rba \}$, the inverse of R, is a relation between B and A.
- The inverse of the relation 'parent of' is the relation 'child of'.
- $A \times B$ is the biggest relation from A to B.
- $\emptyset$ is the smallest relation from A to B.

Identity and Inverse

- $\Delta_A = \{ (a, b) \in A^2 \mid a = b \} = \{ (a, a) \mid a \in A \}$ is a relation on A, the identity on A.
- If R is a relation between A and B, then $R^{-1} = \{ (a, b) \mid Rba \}$, the inverse of R, is a relation between B and A.
- The inverse of the relation 'parent of' is the relation 'child of'.
- $A \times B$ is the biggest relation from A to B.
- $\emptyset$ is the smallest relation from A to B.
- For the usual ordering $<$ of $\mathbb{R}$, $<^{-1} = >$.

Identity and Inverse

- $\Delta_A = \{ (a,b) \in A^2 \mid a = b \} = \{ (a,a) \mid a \in A \}$ is a relation on A, the identity on A.
- If R is a relation between A and B, then $R^{-1} = \{ (a,b) \mid Rba \}$, the inverse of R, is a relation between B and A.
- The inverse of the relation 'parent of' is the relation 'child of'.
- $A \times B$ is the biggest relation from A to B.
- $\emptyset$ is the smallest relation from A to B.
- For the usual ordering $<$ of $\mathbb{R}$, $<^{-1} = >$.
- $(R^{-1})^{-1} = R$; $\Delta_A^{-1} = \Delta_A$; $\emptyset^{-1} = \emptyset$ and $(A \times B)^{-1} = B \times A$.

Properties of Relations

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.
- A relation R on A is **irreflexive** if for no $x \in A$: Rxx.

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.
- A relation R on A is **irreflexive** if for no $x \in A$: Rxx.
- A relation R on A is **symmetric** if for all $x, y \in A$: if Rxy then Ryx.

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.
- A relation R on A is **irreflexive** if for no $x \in A$: Rxx.
- A relation R on A is **symmetric** if for all $x, y \in A$: if Rxy then Ryx.
- Fact: A relation R is symmetric iff $R \subseteq R^{-1}$, iff $R = R^{-1}$.

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.
- A relation R on A is **irreflexive** if for no $x \in A$: Rxx.
- A relation R on A is **symmetric** if for all $x, y \in A$: if Rxy then Ryx.
- Fact: A relation R is symmetric iff $R \subseteq R^{-1}$, iff $R = R^{-1}$.
- A relation R on A is **asymmetric** if for all $x, y \in A$: if Rxy then not Ryx.

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.
- A relation R on A is **irreflexive** if for no $x \in A$: Rxx.
- A relation R on A is **symmetric** if for all $x, y \in A$: if Rxy then Ryx.
- Fact: A relation R is symmetric iff $R \subseteq R^{-1}$, iff $R = R^{-1}$.
- A relation R on A is **asymmetric** if for all $x, y \in A$: if Rxy then not Ryx.
- A relation R on A is **transitive** if for all $x, y, z \in A$: if Rxy and Ryz then Rxz.

Properties of Relations

- A relation R is **reflexive** on A if for every $x \in A$: Rxx.
- A relation R on A is **irreflexive** if for no $x \in A$: Rxx.
- A relation R on A is **symmetric** if for all $x, y \in A$: if Rxy then Ryx.
- Fact: A relation R is symmetric iff $R \subseteq R^{-1}$, iff $R = R^{-1}$.
- A relation R on A is **asymmetric** if for all $x, y \in A$: if Rxy then not Ryx.
- A relation R on A is **transitive** if for all $x, y, z \in A$: if Rxy and Ryz then Rxz.
- A relation R on A is **intransitive** if for all $x, y, z \in A$: if Rxy and Ryz then not Rxz.

Classifying Relations with Relational Properties

Classifying Relations with Relational Properties

- A relation R on A is a **pre-order** if R is transitive and reflexive.

Classifying Relations with Relational Properties

- A relation R on A is a **pre-order** if R is transitive and reflexive.
- A relation R on A is a **strict partial order** if R is transitive and irreflexive.

Classifying Relations with Relational Properties

- A relation R on A is a **pre-order** if R is transitive and reflexive.
- A relation R on A is a **strict partial order** if R is transitive and irreflexive.
- Fact: every strict partial order is asymmetric.

Classifying Relations with Relational Properties

- A relation R on A is a **pre-order** if R is transitive and reflexive.
- A relation R on A is a **strict partial order** if R is transitive and irreflexive.
- Fact: every strict partial order is asymmetric.
- Fact: every transitive and asymmetric relation is a strict partial order.

Classifying Relations with Relational Properties

- A relation R on A is a **pre-order** if R is transitive and reflexive.
- A relation R on A is a **strict partial order** if R is transitive and irreflexive.
- Fact: every strict partial order is asymmetric.
- Fact: every transitive and asymmetric relation is a strict partial order.
- A relation R on A is a **partial order** if R can be written as $S \cup \Delta_A$, where S is a strict partial order.

Classifying Relations with Relational Properties

- A relation R on A is a **pre-order** if R is transitive and reflexive.
- A relation R on A is a **strict partial order** if R is transitive and irreflexive.
- Fact: every strict partial order is asymmetric.
- Fact: every transitive and asymmetric relation is a strict partial order.
- A relation R on A is a **partial order** if R can be written as $S \cup \Delta_A$, where S is a strict partial order.
- A relation R on A is an **equivalence relation** if R is reflexive, symmetric and transitive.

Closures of Relations

If $\mathcal{O}$ is a set of properties of relations on a set A, then the $\mathcal{O}$-closure of a relation R is the smallest relation S that includes R and that has all the properties in $\mathcal{O}$.

To show that R is the smallest relation S that has all the properties in $\mathcal{O}$, show the following:

1. R has all the properties in $\mathcal{O}$,
2. If S has all the properties in $\mathcal{O}$, then $R \subseteq S$.

Fact: $R \cup \Delta_A$ is the reflexive closure of R.

Fact: $R \cup R^{-1}$ is the symmetric closure of R.

Composing Relations

Composing Relations

- Suppose that R and S are relations on A.

Composing Relations

- Suppose that R and S are relations on A.
- The composition $R \circ S$ of R and S is the relation on A that is defined by

$$(R \circ S)xz \;:\equiv\; \exists y \in A(Rxy \wedge Syz).$$

Composing Relations

- Suppose that R and S are relations on A.
- The composition $R \circ S$ of R and S is the relation on A that is defined by

$$(R \circ S)xz \;:\equiv\; \exists y \in A(Rxy \wedge Syz).$$

- For $n \in \mathbb{N}, n \geq 1$ we define R^n by means of $R^1 := R$, $R^{n+1} := R^n \circ R$.

Composing Relations

- Suppose that R and S are relations on A.
- The composition $R \circ S$ of R and S is the relation on A that is defined by

$$(R \circ S)xz \;:\equiv\; \exists y \in A(Rxy \wedge Syz).$$

- For $n \in \mathbb{N}, n \geq 1$ we define R^n by means of $R^1 := R$, $R^{n+1} := R^n \circ R$.
- Fact: a relation R is transitive iff $R^2 \subseteq R$.

Composing Relations

- Suppose that R and S are relations on A.
- The composition $R \circ S$ of R and S is the relation on A that is defined by

$$(R \circ S)xz \;:\equiv\; \exists y \in A(Rxy \wedge Syz).$$

- For $n \in \mathbb{N}, n \geq 1$ we define R^n by means of $R^1 := R$, $R^{n+1} := R^n \circ R$.
- Fact: a relation R is transitive iff $R^2 \subseteq R$.
- Fact: for any relation R on A, the relation $R^+ = \bigcup_{n \geq 1} R^n$ is the transitive closure of R.

Composing Relations

- Suppose that R and S are relations on A.
- The composition $R \circ S$ of R and S is the relation on A that is defined by

$$(R \circ S)xz \;:\equiv\; \exists y \in A(Rxy \wedge Syz).$$

- For $n \in \mathbb{N}, n \geq 1$ we define R^n by means of $R^1 := R$, $R^{n+1} := R^n \circ R$.
- Fact: a relation R is transitive iff $R^2 \subseteq R$.
- Fact: for any relation R on A, the relation $R^+ = \bigcup_{n \geq 1} R^n$ is the transitive closure of R.
- Fact: for any relation R on A, the relation $R^+ \cup \Delta_A$ is the reflexive transitive closure of R.

Composing Relations

- Suppose that R and S are relations on A.
- The composition $R \circ S$ of R and S is the relation on A that is defined by

$$(R \circ S)xz \;:\equiv\; \exists y \in A(Rxy \wedge Syz).$$

- For $n \in \mathbb{N}, n \geq 1$ we define R^n by means of $R^1 := R$, $R^{n+1} := R^n \circ R$.
- Fact: a relation R is transitive iff $R^2 \subseteq R$.
- Fact: for any relation R on A, the relation $R^+ = \bigcup_{n \geq 1} R^n$ is the transitive closure of R.
- Fact: for any relation R on A, the relation $R^+ \cup \Delta_A$ is the reflexive transitive closure of R.
- Abbreviation for the reflexive transitive closure of R: R^*.

Equivalence Relations

A relation R on A is an **equivalence relation** or **equivalence** if R is transitive, reflexive on A and symmetric.

Example: For all $n \in \mathbb{N}^+$, the relation

$$(\text{mod } n) = \{(x, y) \mid x - y \text{ is divisible by n } \}$$

is an equivalence on $\mathbb{Z}$. Implementation:

```
modulo :: Integer -> Integer -> Integer -> Bool
modulo n x y = rem (x-y) n == 0
```

Example: the relation 'having the same size' on finite lists is an equivalence relation on the class of all finite lists. Implementation:

```
equalSize :: [a] -> [b] -> Bool
equalSize list1 list2 = (length list1) == (length list2)
```

Equivalence Classes and Partitions

Explain with picture on blackboard ...

Equivalence relations on a set A enable us to partition the set A into equivalence classes.

Suppose R is an equivalence relation on A and that $a \in A$. The set

$|\, a \,| = |\, a \,|_R = \{\, b \in A \mid Rba \,\}$

is called the R-equivalence class of a, or the equivalence class of a modulo R.

Elements of an equivalence class are called representatives of that class.

Suppose that R is an equivalence on A. If $a, b \in A$, then:

$$|\, a \,|_R = |\, b \,|_R \;\Leftrightarrow\; aRb.$$

Equivalences and Partitions

Let R be an equivalence on A. Then:

1. Every equivalence class is non-empty,
2. every element of A belongs to some equivalence class,
3. different equivalence classes are disjoint.

A family $\mathcal{A}$ of subsets of a set A is called a **partition** of A if

- $\emptyset \notin \mathcal{A}$,
- $\bigcup \mathcal{A} = A$,
- for all $X, Y \in \mathcal{A}$: if $X \neq Y$ then $X \cap Y = \emptyset$.

Assume that R is an equivalence on the set A. The collection of equivalence classes of R, $A/R = \{|\, a \,|\ |\ a \in A\}$, is called the quotient of A modulo R.

Fact: Every quotient (of a set, modulo an equivalence) is a partition (of that set), and vice versa.

Here is Another Way . . .

```
module Rel2

where

type Rel a = a -> a -> Bool

emptyR :: Rel a
emptyR _ _ = False

list2rel :: Eq a => [(a,a)] -> Rel a
list2rel [] _ _                   = False
list2rel ((x,y):xys) u v
              | x == u && y == v = True
              | otherwise        = list2rel xys u v

idR :: Eq a => [a] -> Rel a
idR [] _ _ = False
```

```haskell
idR (x:xs) u v | x == u && x == v     = True
               | otherwise            = idR xs u v

invR :: Rel a -> Rel a
invR = flip

inR :: Rel a -> (a,a) -> Bool
inR r  = uncurry r

reflR :: [a] -> Rel a -> Bool
reflR list r = and [ r x x | x <- list ]

irreflR :: [a] -> Rel a -> Bool
irreflR list r = and [ not (r x x) | x <- list ]

symR :: [a] -> Rel a -> Bool
symR list r = and [ not (r x y && not (r y x))
                          | x <- list, y <- list ]

transR :: [a] -> Rel a -> Bool
```

```haskell
transRR list r = and
              [ not (r x y && r y z && not (r x z))
                   | x <- list, y <- list, z <- list ]

unionR :: Rel a -> Rel a -> Rel a
unionR r s x y = r x y  || s x y

intersR :: Rel a -> Rel a -> Rel a
intersR r s x y = r x y && s x y
```

Lab Session This Week

- Read up on relations (if you still need it)
- Operations on Relations in Haskell
- Automated Testing of your implementations

References

[1] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In Proc. Of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000.