(Week 7)

Structural Induction on Trees

to prove a property $P(t)$ for all trees of a certain type

- Show that $P(l)$ holds for all leaves l of a tree
- for each type of internal node t with subtrees $s_1 \dots s_n$ show that

$$P(s_1) \wedge \dots \wedge P(s_n) \text{ implies } P(t)$$

Example

IntSet $\xleftarrow{}$ Empty
def incl
def contains $\xleftarrow{}$ NonEmpty

How to show the correctness of the implementation?

Laws:

1) Empty contains x = false
2) (s incl x) contains x = true
3) (s incl x) contains y = s contains y if $x \neq y$

How to prove?

2. by structural induction on s

Base case: Empty

(Empty incl x) contains x
= NonEmpty(x, E, E) contains x
= true

Induction Step: • NonEmpty(x, l, r)

(NE(x, l, r) incl x) contains x
= NonEmpty(x, l, r) contains x
= true

• NonEmpty(y, l, r), y < x

(NE(y, l, r) incl x) contains x
= NE(y, l, r incl x) cont x
= (r incl x) contains x
= true

• NE(y, l, r), y > x is analogous

Etc.

Streams

to find the 2^{nd} prime number between 1000 and 10000

((1000 to 10000) filter isPrime)(1)

But it constructs all prime numbers in [1000, 10000] in a list, but ~~too~~ uses only 2^{nd}.

We can reduce the upper bound, but at risk of missing the 2^{nd} number altogether

However, we can avoid computing the tail of a sequence until it's needed for the evaluation results (which might be never)

Streams are built on this idea and similar to lists

Stream.cons(1, Stream.cons (2, Stream.empty))

or

Stream(1, 2, 3)

collections
~~list~~ have toStream method:

(1 to 1000).toStream > Stream[Int] = Stream(1, ?)

||

def streamRange(lo: Int, hi: Int): Stream[Int] =
  if (lo >= hi) Stream.empty
  else Stream.cons(lo, streamRange(lo+1, hi))

Methods on Stream

Stream supports all methods of lists

Eg. ((1000 to 10000).toStream filter isPrime)(1)

But :: always produces a list, not a stream.

so x #:: xs == Stream.cons(x, xs)

$\underbrace{\#::}_{\text{alternative cons operator}}$

#:: can also be used in expressions and patterns

Implementation

```
trait Stream[+A] extends Seq[A] {

    def isEmpty: Boolean
    def head: A
    def tail: Stream[A]
    ...
}
```

concrete impl in the Stream companion object

```
def cons[T](hd: T, (tl: => Stream)) =
    new Stream[T] {

    def isEmpty = false
    def head = hd
    def tail = tl
}
```

the only difference from List, a by-name parameter

That's why it's not evaluated on the call, but on demand

```
def filter(p: T => Boolean): Stream[T] =
    if (isEmpty) this
    else if (p(head)) cons(head, tail.filter(p))
    else tail.filter(p)
```

exactly the same impl.

Lazy Evaluation

The proposed implementation suffers from a serious performance problem:

if tail is called several times, the corresponding stream will be recomputed each time

This can be avoided by storing the results of the first evaluation of tail and re-using it afterwards

This is called lazy evaluation (as opposed to by-name evaluation and strict evaluation - (for normal parameters and val definitions))

Default evaluation in Scala is strict

```
lazy val x = expr
```

```
def cons[T](hd: T, tl: => Stream[T]) =
    new Stream[T] {

        def head = hd
        lazy val tail = tl
        ...
    }
```

Infinite Sequences

```
def from(n: Int): Stream[Int] = n #:: from(n+1)
```

stream of all integers starting from a given number

`val nat = from(0)` - all natural numbers

`nats map (_ * 4)` - all natural multiplied by 4

The Sieve of Eratosthenes to calculate prime numbers

- start from 2
- eliminate all multiplies of 2
- the first element of the resulting list is 3,
- eliminate all multiplies of 3
- iterate forever
  - at each step, the first number in the list is a prime number
  - and we eliminate all its multiplies

```
def sieve(s: Stream[Int]): Stream[Int] =
    s.head #:: sieve(s.tail filter (_ % s.head != 0))

val primes = sieve(from(2))

(primes take N).toList
```

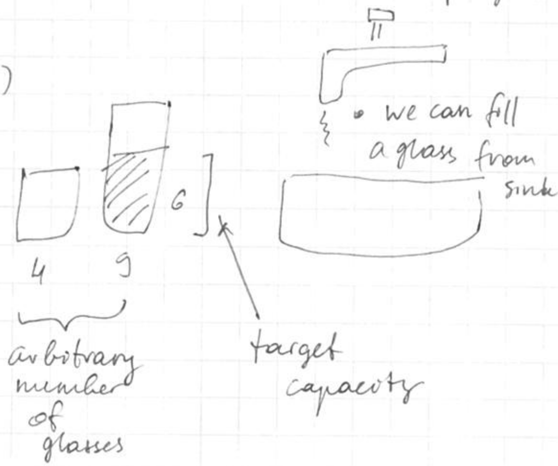
Case Study (Check Udacity!)
Peter Norvig's course

The water pouring problem

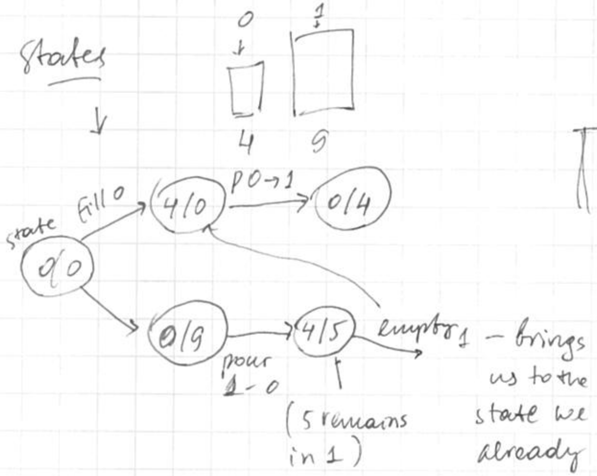
Glass : Int
State : Vector[Int] (one entry per glass) ← number of occupied ~~dvot~~ volume units per glass
Moves :

Empty (glass)
Fill (glass)
Pour (from, to)

∘ we can pour from one glass to another



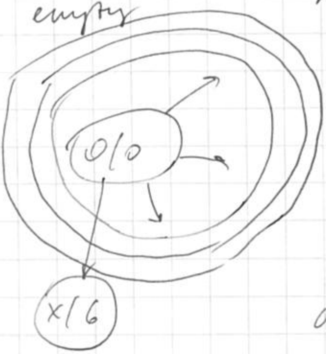
States



how do we generate moves to find the sequence that leads to the target capacity?

Idea: to generate all possible moves called „paths" – and find one of the glassed with the right target

start with all empty



generate all possible moves of length 1

then generate all possible moves of length 2

and so on

until we hit a state with our target amount in one of the glasses.

or we exausted all possible combinations and no solution could be found

Code

for each glass

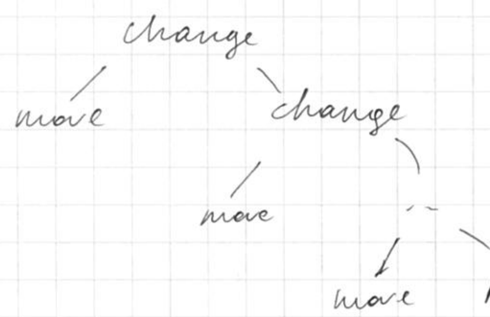
```
class Pouring(capacity: Vector[Int]) {

    // states
    type State = Vector[Int]
                                                  glass
    val initialState = capacity map (x => 0)

    // moves
    trait Move
    case class Empty(glass: Int) extends Move
    case class Fill(glass: Int) extends Move
    case class Pour(from: Int, to: Int) extends Move

    val glasses = 0 until capacity.length
        // all glasses
                                              all moves
    val moves =
        (for (g <- glasses) yield Empty(g)) ++
        (for (g <- glasses) yield Fill(g)) ++
        ~~(for (g <- glasses) yield~~
        (for (from <- glasses; to <- glasses
              if from != to)
               yield Pour(from, to))

    // transition logic

    trait Move {
        def change(state: State): State
    }

    Empty(glass: Int) .. {                    create new
        def change(state: State) =            vector, with
            state updated (glass, 0) }        0 @ glass
```

Fill: ch
```
def ~~c~~change( ) =
    state updated (glass, capacity(glass))
```

Pour:
```
def change (..) = {
    val amount = state(from) min
        (capacity(to) - state(to))

    state updated (from, state(from) - amount)
        updated (to, state(to) + amount)
```

// paths - sequences of moves // last move comes first in the list

```
class Path (history: List [Move]) {
    def endState: State = trackState (history)

    private def trackState (xs: List[Move]): State = {
        xs match {

        case Nil => initialState
        case move :: xs1 => trackState (xs1)
                                        ^
                                   move change
    }
```



← fold Right!

So we can reformulate it

```
def endState = (history foldRight initialState)
                  (_ change _)

def extend (move: Move) = new Path (move :: history)

override def toString =
    (history.reverse mkString " ") + " --> " +
      + endState.

}

val initialPath = Path(Nil)

def from (paths: Set[Path]): Stream [Set[Path]]
     =
  if (paths.isEmpty) Stream.empty
  else {
     val more = for {
       path <- paths
       next <- moves map path.extend
     } yield next
     paths #:: from(more)
```

all possible paths

val pathSets = from (Set(initialPath)) ← "onion" of path layers — up to ∞

(pathSets.take(3).toList — take 3 layers)

```
def solutions(target: Int): Stream[Path] =

    for {
        pathSet <- pathSets
        path <- pathSet
        if path.end contains target
    } yield path
```

sequence of solutions ordered by lengths.

Problems with this implementation:
- we move blindly - and generate states that are already created - and those states don't bring anything to the solution
- so we need to exclude everything we visited before

```
def from(paths: Set[Path], explored: Set[State]):
        Stream[Set[Path]] =

    if ...
    else {
        val more = for {
            path <- paths
            next <- moves map path.extend
            if !(explored contains next.endState)
        } yield next

        paths #:: from(more,
                explored ++ (more map (_.endState)))
```

But there is still room for improvement:

Path.endState is called many times
why recomplute it?

```
class Path (history: List[Move], val endState: State) {

  def extend(move: Move) = new Path (move :: history,
    move change endState)

val initialPath = Path(Nil, initialState)
```

Principles of Good Design

- Name everything you can
- put operations into natural scopes
- keep degrees of freedom for future ~~fo~~ refinements.