Week 7: Functional Data Structures

CS-214 Software Construction

Outline

- Structure Sharing and Path Copying
- Specifications of Data Structures
- Collections and Their Relationships
- Trees with Efficient Append

Structure Sharing and Path Copying

CS-214 Software Construction

A Puzzle with IntSet

```
abstract class IntSet
case class Empty() extends IntSet
case class Node(left: IntSet, elem: Int, right: IntSet) extends IntSet

extension (s: IntSet)
  def size: Int =
    s match
      case Empty() => 0
      case Node(l, _, r) => l.size + 1 + r.size

val t1 = Node(Node(Empty(), 1, Empty()), 1
              Node(Empty(), 1, Empty()))
```

mkTree

What is the running time of this function as a function of n?

```
def mkTree(n: Int): IntSet =
  if n <= 0 then Empty()
  else
    val t1: IntSet = mkTree(n-1)
    Node(t1, n, t1)
```

$O(n)$

Given:

```
val t = mkTree(16)
```

what is `t.size` equal to?

65536 that is, $2^{16} - 1$

mkTreeSize

What is the running time of this function:

```
def mkTreeSize(n: Int): Int = mkTree(n).size
```

$O(2^n)$

How can we make a tree faster than we can compute its size?

Answer: **structure sharing**

“Trees” Are, in Fact, Acyclic Graphs: Drawing

```
def mkTree(n: Int): IntSet =
  if n <= 0 then Empty()
  else
    val t1: IntSet = mkTree(n-1)
    Node(t1, n, t1) // near-constant time

extension (s: IntSet)
  def size: Int =
    s match
      case Empty() => 0
      case Node(l, _, r) => l.size + 1 + r.size
```

List with Concat

```
val l = (1 to p).toList // e.g. p=100
```

We have one list. It uses p units of memory.

```
val xs = (1 to q) ++ l  // e.g. q=5
```

Now we have two lists:

- one p elements long (e.g. 100)
- one p+q elements long (e.g. 105)

How much memory are we using?

1. p units
2. p+q units
3. p + p + q units

p+q, because p of the units are shared

Sharing When Updating Tree Structures: Path Copying

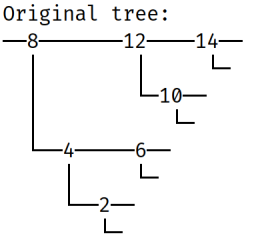
Suppose the tree in IntSet is balanced:

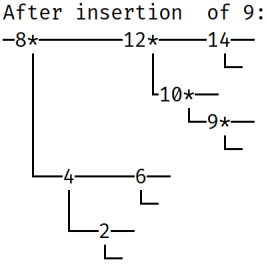
- children of every node have same shape and size

Then this code runs in logarithmic time and makes a logarithmic number of new nodes:

```
extension (s: IntSet)
  def incl(x: Int): IntSet =
    s match
      case Empty() => Node(Empty(), x, Empty())
      case Node(l, e, r) =>
        if x < e then Node(l.incl(x), e, r)
        else if x > e then Node(l, e, r.incl(x))
        else s
```

Illustration of Path Copying





Specifications of Data Structures

CS-214 Software Construction

require and ensuring Specify Contract of a Function

```
def f(x: A): B = {
  require(pre(x))
  body
}.ensuring(res => post(x, res))
```

require (precondition):

- requirement for code to call the function
- it is the responsibility of the caller to make it true

ensuring (postcondition):

- property of the body that it wraps
- it is the responsibility of the function to make it true
- `ensuring` only needs to hold if require was true

Meaning: $\forall x \in A.\ pre(x) \rightarrow post(x, body)$

Data Structure Invariants

Data structure invariant is property of the data structure that is maintained by all (completed) data structure operations

- true for initial values (empty list, or a tree leaf)
- preserved by all operations

Refinement of the type

- example: not simply a list (or tree), but a *sorted* list (or tree)
- $\{x \in \mathit{List}[\mathit{Int}] \mid \mathit{isSorted}(x)\}$

How to express it?

- state property in both `require` and `ensuring`
- `require` on the class constructor

Example: insert Maintains isSorted of a List

```
def isSorted(l: List[Int]): Boolean =
  l match
    case List()|List(_) => true
    case x :: y :: ys => x < y && isSorted(y :: ys)

def insert(x: Int, l: List[Int]): List[Int] = {
  require(isSorted(l))
  l match
    case List() => List(x)
    case y :: ys =>
      if x < y then x :: l
      else if x > y then y :: insert(x, ys)
      else l
}.ensuring(res => isSorted(res) && res.contains(x))
```

IntSet and its Abstraction Function

We often specify a data structure by mapping it to a simpler one

A function that peforms this mapping is called **abstraction function**

Example: toList of the IntSet tree (like one in week 3):

```
abstract class IntSet
case class Empty() extends IntSet
case class Node(left: IntSet, elem: Int, right: IntSet) extends IntSet

extension (s: IntSet)
  // abstraction function:
  def toList: List[Int] = s match
    case Empty() => List()
    case Node(l, e, r) => l.toList ++ List(e) ++ r.toList
```

Specification of `incl` on a IntSet tree

```
extension (s: IntSet)
  // invariant defined via abstraction function
  def valid: Boolean = isSorted(s.toList)

  def incl(x: Int): IntSet = {
    require(s.valid)   // precondition
    s match
      case Empty() => Node(Empty(), x, Empty())
      case Node(l, e, r) =>
        if x < e then Node(l.incl(x), e, r)
        else if x > e then Node(l, e, r.incl(x))
        else s
  }.ensuring(res => res.valid) // postcondition
```

Inserting an element preserves the validity (sortedness) of the tree.

Example: Quadtrees

Quadtree generalizes binary search tree by ordering elements along two dimensions.

```
type Point = (Int, Int)
def inRange(p: Point, minP: Point, maxP: Point): Boolean =
  minP._1 <= p._1 && p._1 < maxP._1 &&
  minP._2 <= p._2 && p._2 < maxP._2

sealed abstract class QuadTree
case class Empty() extends QuadTree
case class Leaf(p: List[Point]) extends QuadTree
case class Quad(center: Point,
                nw: QuadTree, ne: QuadTree,
                sw: QuadTree, se: QuadTree) extends QuadTree
```

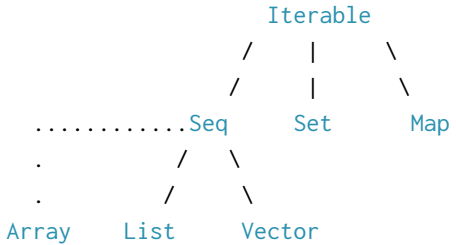
A Possible Quadtree Invariant

```
extension(t: QuadTree)
  def valid(minP: Point, maxP: Point): Boolean =
    t match
      case Empty() => true
      case Leaf(ps) => ps.forall(inRange(_, minP, maxP))
      case Quad(c, nw, ne, sw, se) =>
        // minP  |
        //   nw  |  ne
        // ------c---------
        //   sw  |  se
        //       |     maxP
        inRange(c, minP, maxP) &&
        nw.valid(minP, c) && se.valid(c, maxP) &&
        ne.valid((c._1, minP._2), (maxP._1, c._2)) &&
        sw.valid((minP._1, c._2), (c._1, maxP._2))
```

Collections and Their Relationship

CS-214 Software Construction

Collection Hierarchy from a Previous Week



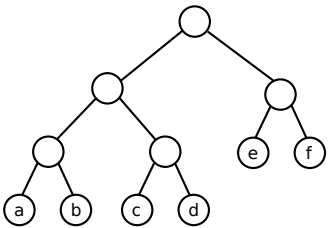
Relationships Between Collections

- strictly sorted sequence $\Longleftrightarrow$ set
  List(10,30,70) $\Longleftrightarrow$ Set(10,30,70)
- non-strictly sorted sequence $\Longleftrightarrow$ bag (multiset)
  List(10, 30, 30, 70) $\Longleftrightarrow$ Multiset(10,30,30,70) // See the exercises. akka Bag
- Map[K,V] $\Longleftrightarrow$ Set[(K,V)] where no two (k,v1),(k,v2) in set
  Map(1 $\rightarrow$ "x", 2 $\rightarrow$ "y") $\Longleftrightarrow$ Set((1,"x"), (2, "y"))
- Bag[T] $\Longleftrightarrow$ Map[T, Int] with positive values
  - counts how many times each element appears

  Bag(10,30,30,70) $\Longleftrightarrow$ Map(10 $\rightarrow$ 1, 30 $\rightarrow$ 2, 70 $\rightarrow$ 1)
- sequence is a map with `[0,n-1]` as a domain
  List(10,30,70) $\Longleftrightarrow$ Map(1 $\rightarrow$ 10, 2 $\rightarrow$ 30, 3 $\rightarrow$ 70)

Relating Collections

Suppose we have a tree that implements a sequence, e.g. `Seq(a,b,c,d,e,f)` as



To use it as a set, insert elements in the appropriate position, keeping it sorted

- see IntSet in a previous lecture

To implement, e.g., a map, use pairs of keys and values, sorting on unique keys.

Hence, we will focus on data structure for sequences.

EPFL

Trees with Concatenation

CS-214 Software Construction

Simplified variant of ConcRope (A. Prokopec and M. Odersky). Formal verification by R.K. Madhavan et al.

Goal: Vector-Like Sequences

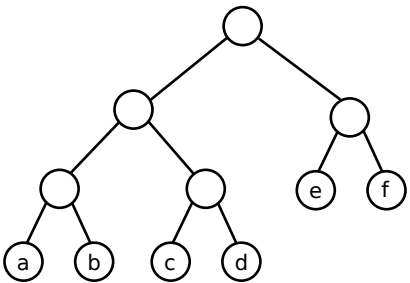
```
trait Seq[T]:
    def toList: List[T]                          // for specification
    def size: Int
    def apply(i: Int): T                         // log(n)
    def ++(that: Seq[T]): Seq[T]                 // log(n)
    def slice(from: Int, until: Int): Seq[T]     // log(n)
```

Using ++ we can implement adding to the front or the end

Using `slice` we can implement tail, init, splitAt

Binary Trees

```
sealed abstract class Conc[T] // Conc-atenation supporting tree
case class Empty[T]() extends Conc[T]
case class Leaf[T](x: T) extends Conc[T]
case class Node[T](left: Conc[T], right: Conc[T],
```



Storing Size and Height

```
sealed abstract class Conc[T]
case class Empty[T]() extends Conc[T]
case class Leaf[T](x: T) extends Conc[T]
case class Node[T](left: Conc[T], right: Conc[T],
                   csize: Int, cheight: Int) extends Conc[T]
```

Invariant Ensures They are Computed Right

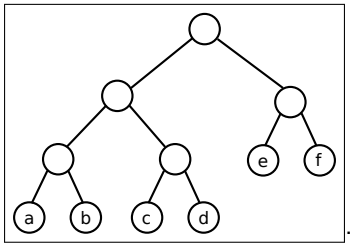
```
sealed abstract class Conc[T]
case class Empty[T]() extends Conc[T]
case class Leaf[T](x: T) extends Conc[T]
case class Node[T](left: Conc[T], right: Conc[T],
                   csize: Int, cheight: Int) extends Conc[T]:
  require(csize == left.size + right.size && left != Empty[T]() && right != Empty[T]()
          cheight == max(left.height, right.height) + 1)
```

Invariant also ensures that children are non-empty

- no need for empty children, the leaves store data

Abstraction Function Maps a Tree to a List

```
extension[T](t: Conc[T])
  def toList: List[T] = t match
    case Empty() => Nil[T]()
    case Leaf(x) => List(x)
    case Node(l, r, _, _) => l.toList ++ r.toList
```



.toList = List(a, b, c, d, e, f)

Abstraction Function Can Specify Correctness of Operations

```
extension[T](t: Conc[T])
  def size: Int = {
    t match
      case Empty() => Int(0)
      case Leaf(_) => Int(1)
      case Node(_, _, csize, _) => csize
  }.ensuring(_ == t.toList.size)

  def height: Int =
    t match
      case Empty() => 0
      case Leaf(x) => 1
      case Node(_, _, _, cheight) => cheight
```

Look up Element at Index i

```
extension[T](t: Conc[T])
  def apply(i: Int): T = {
    require(0 <= i && i < t.size)
    t match
      case Leaf(x) => assert(i == 0); x
      case Node(l, r, _, _) =>
```

Look up Element at Index i

```
extension[T](t: Conc[T])
  def apply(i: Int): T = {
    require(0 <= i && i < t.size)
    t match
      case Leaf(x) => assert(i == 0); x
      case Node(l, r, _, _) =>
        if i < l.size then l(i)   // look up in left subtree
        else r(i - l.size)        // look up in right subtree
  }
```

Look up Element at Index i

```
extension[T](t: Conc[T])
  def apply(i: Int): T = {
    require(0 <= i && i < t.size)
    t match
      case Leaf(x) => assert(i == 0); x
      case Node(l, r, _, _) =>
        if i < l.size then l(i)
        else r(i - l.size)
  }.ensuring(_ ==            )
```

What is the Specification?

Look up Element at Index i

```
extension[T](t: Conc[T])
  def apply(i: Int): T = {
    require(0 <= i && i < t.size)
    t match
      case Leaf(x) => assert(i == 0); x
      case Node(l, r, _, _) =>
        if i < l.size then l(i)
        else r(i - l.size)
  }.ensuring(_ == t.toList(i))
```

Building Tree from Possibly Empty Parts

```
extension[T](t1: Conc[T])
  def <>(t2: Conc[T]) =
    if t1 == Empty[T]() then t2
    else if t2 == Empty[T]() then t1
    else Node(t1, t2, t1.size + t2.size,
                      max(t1.height, t2.height) + 1)
```

Naive Concatenation: Same as <>

```
extension[T](t1: Conc[T])
  def ++(t2: Conc[T]): Conc[T] = {
    t1 <> t2
  }
```

Naive Concatenation: Same as <>

```
extension[T](t1: Conc[T])
  def ++(t2: Conc[T]): Conc[T] = {
    t1 <> t2
  }.ensuring(                                    )
```

What is the specification?

Naive Concatenation: Same as <>

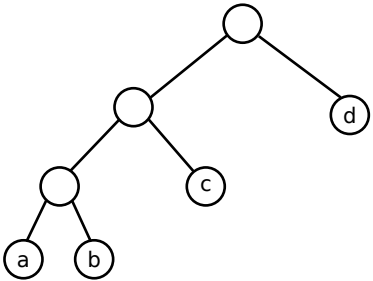
```
extension[T](t1: Conc[T])
  def ++(t2: Conc[T]): Conc[T] = {
    t1 <> t2
  }.ensuring(_.toList == t1.toList ++ t2.toList)
```

What is the specification? - List concatenation!

Does this ensure log(n) time of apply for all trees we build?

Demo of Trees with Naive Concat

Imbalanced Tree



New Invariant

The height difference between the left and the right subtree is always 1 or less.

The following method checks this property:

```
extension[T](t1: Conc[T])
  def isBalanced: Boolean =
    t match
      case Node(l, r, _, _) =>
        -1 <= l.height - r.height && l.height - r.height <= 1 &&
        l.isBalanced && r.isBalanced
      case _ => true
```

Consequence:

- is `t.isBalanced` then its height is logarithmic in its size

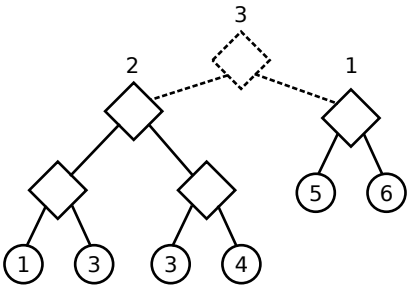
Goal: Design xs ++ ys to preserve this invariant

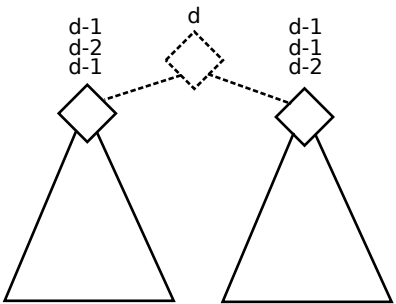
```
extension[T](xs: Conc[T])
  def ++(ys: Conc[T]): Conc[T] = {
    require(xs.isBalanced && ys.isBalanced)
    if xs == Empty[T]() then ys
    else if ys == Empty[T]() then xs
    else
      ...
```

Not difficult, but a number of cases.

Specifications will help us check them.

Are trees close enough in size?





If yes, we are done (it's fast when trees are of close height):

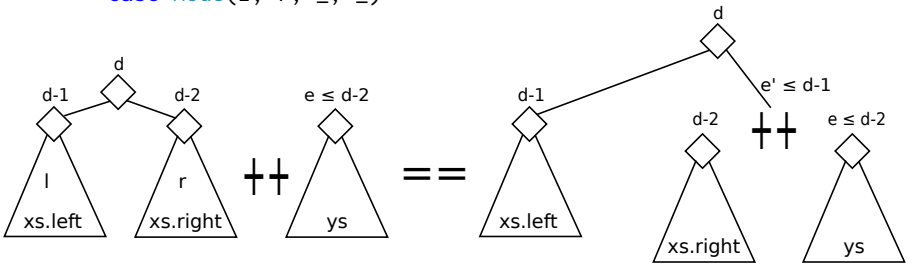
```
val diff = ys.height - xs.height
if -1 <= diff && diff <= 1 then xs <> ys
else
```

If not, one of the tree is at least two higher than the other.

Suppose Left Tree is At Least 2 Higher Than Right

Then let's look at the parts of that big xs tree, call them l,r

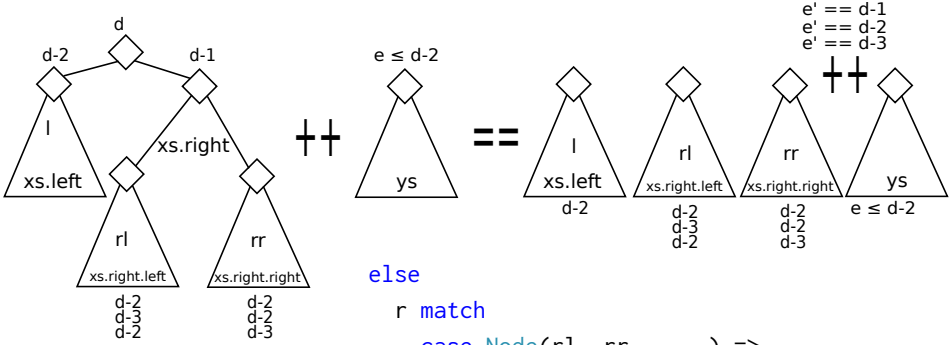
```
else if diff < -1 then
  xs match
    case Node(l, r, _, _) =>
```



If the left tree is left-leaning, we recurse: `(r ++ ys)`; the result is good to use:

```
if l.height >= r.height then l <> (r ++ ys)
```

Left Tree is Right Leaning - Add ys into rr




```
else
  r match
    case Node(rl, rr, _, _) =>
      val nrr = rr ++ ys
      if nrr.height == xs.height - 3 then
        l <> (rl <> nrr)  // keep rl where it was
      else
        (l <> rl) <> nrr  // rotate left: nrr goes up
```

That's It!

The case where right tree is larger is symmetrical

- with respect to `.reverse` symmetry and flipping of a tree
- in total only around 30 lines of code

Some remarkable correctness and balancing properties provably by induction:

```
}.ensuring(res =>
    res.isBalanced &&
    res.height <= max(xs.height, ys.height) + 1 &&
    res.height >= max(xs.height, ys.height) &&
    res.toList == xs.toList ++ ys.toList)
```

Many cases to consider (remember proofs about IntSet). Automation to rescue:

`https://github.com/epfl-lara/stainless/`

Complexity

Question: What is the complexity of ++?

- $O(\log n)$
- $O(h_1 - h_2)$
- $O(n)$
- $O(1)$

Concatenation takes $O(h_1 - h_2)$ time, where h_1 and h_2 are the heights of the two trees.

- indeed, `abs(xs.height - ys.height)` decreases in each recursive call
- we deconstruct larger tree and merge its parts with the smaller tree

Now, let's do slice! We can use ++

Elements: from, from+1, ..., until-1

```
def slice(from: Int, until: Int): Conc[T] = {
  require(0 <= from && from <= until && until <= t.size && t.isBalanced)
  if from == until then Empty[T]()
  else t match
          case Leaf(x) => Leaf(x)
          case Node(l, r, _, _) =>
            if l.size <= from then r.slice(from - l.size, until - l.size)
            else if until <= l.size then l.slice(from, until)
            else
              val l1 = l.slice(from, l.size)
              val r1 = r.slice(0, until - l.size)
              l1 ++ r1
}.ensuring(res => res.isBalanced && res.toList == t.toList.slice(from, until))
```

Constant Time Appends

It is possible to implement more efficient addition of individual elements

- the result is called Conc-Rope
- sequence of increasingly larger binary trees
- alternative: finger trees (basis of `Vector` implementation)

More information

- Chris Okasaki: Purely Functional Data Structures, Cambrige University Press.
  - originally PhD thesis: `https://www.cs.cmu.edu/~rwh/students/okasaki.pdf`
- Tobias Nipkow: Automatic Functional Correctness Proofs for Functional Search Trees. `https://www21.in.tum.de/~nipkow/pubs/itp16.pdf`