Case Studies of Data Structures in Leon

Maëlle Colussi & Mathieu Demarne

Implementation of "*Catenable List*"
and "*Binomial Heap*"

Overview

We implemented two data structures from the book "*Purely Functional Datastructures*" from Chris Okasaki, 1998.

Chosen data structures:

- Catenable List: recursive structure based on queues (7.2.1, p93)
- Binomial Heap: structure using tree representations (6.2.2, p68)

Goals

- Discover new data structures
- Play with Leon verifier
- Assess its boundaries

Catenable List

Recursive list structure based on queues

Structure Properties

Catenable Lists supports (Okasaki p15):

- `head`
- `tail`
- `cons` (adds an element at the beginning)
- `snoc` (adds an element at the end)
- concatenation (`++`)

… in *O(1)* amortized time.

Structure Details[1]

- Catenable Lists are based on Queues

- We implemented a classical representation of queues in Leon and did formal proofs on it as well
    - supports `head`, `tail`, `snoc`

- The implementation for Queues we used also comes from Okasaki (3.1.1, p15)

Queue Implementation[1]

Queues are implemented as a pair of lists

```
Q  :=    QEmpty[T]
       | QCons[T](l: List[T], r: List[T])
```

Invariant: `l` *is never empty*

The order of elements in the queue is

`l :: r.reverse`

Ex: `QCons(1::2::3,5::4)` $\Rightarrow$ `1::2::3::4::5`

Queue Implementation²

- `head` takes the head of the left hand-side list
- `snoc` adds an element to the right hand-side list if the queue is not empty
  - if the queue is empty, to the left hand-side, so that the invariant holds
- `tail` removes the head of the left hand-side list
  - if the list becomes empty, it reverses the right hand-side list and put it on the left, so that invariant holds

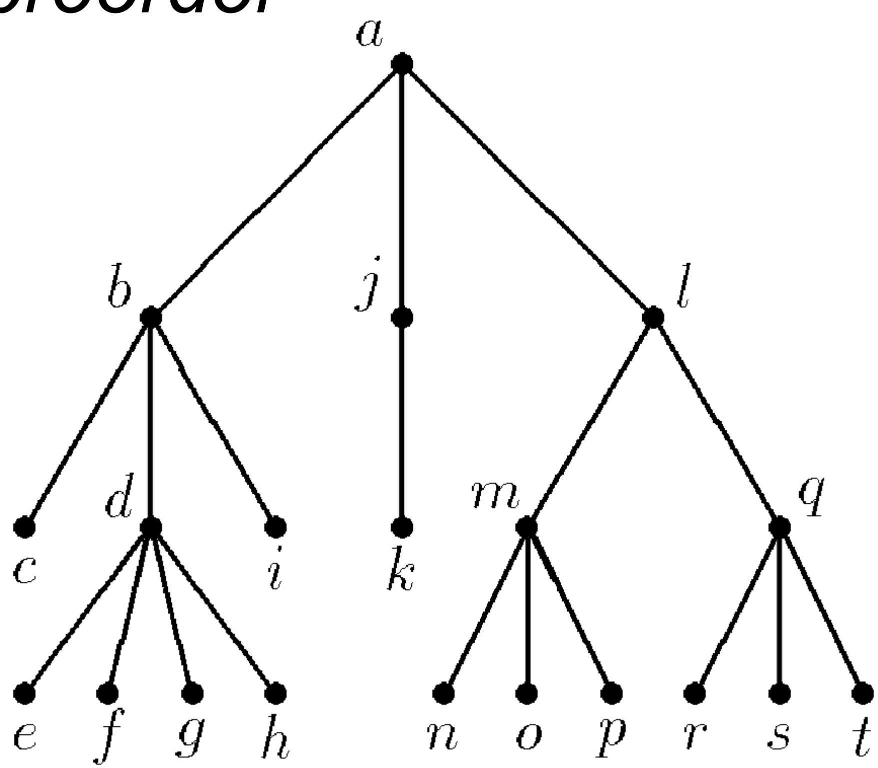
Catenable List Implementation[1]

Catenable Lists are implemented as:

```
CL := CEmpty[T]
    | CCons[T](h: T, q: Queue[CL[T]])
```

Catenable List Implementation^2

Traverse tree in preorder



Represents a list a, b, …, t

Catenable List Implementation[3]

- `head` simply returns the element on the left

- `cons` and `snoc` uses a *concatenation* function in a trivial way
  - Create a Catenable List `t` with the element to add
  - Concatenates the original list and `t`

Catenable List Implementation[4]

- *concatenation*:
  - Is trivial with one empty Catenable List
  - Produces one Catenable List by `link`ing both otherwise

- `link` puts the second Catenable List at the end of the first one (*snoc* on its queue)

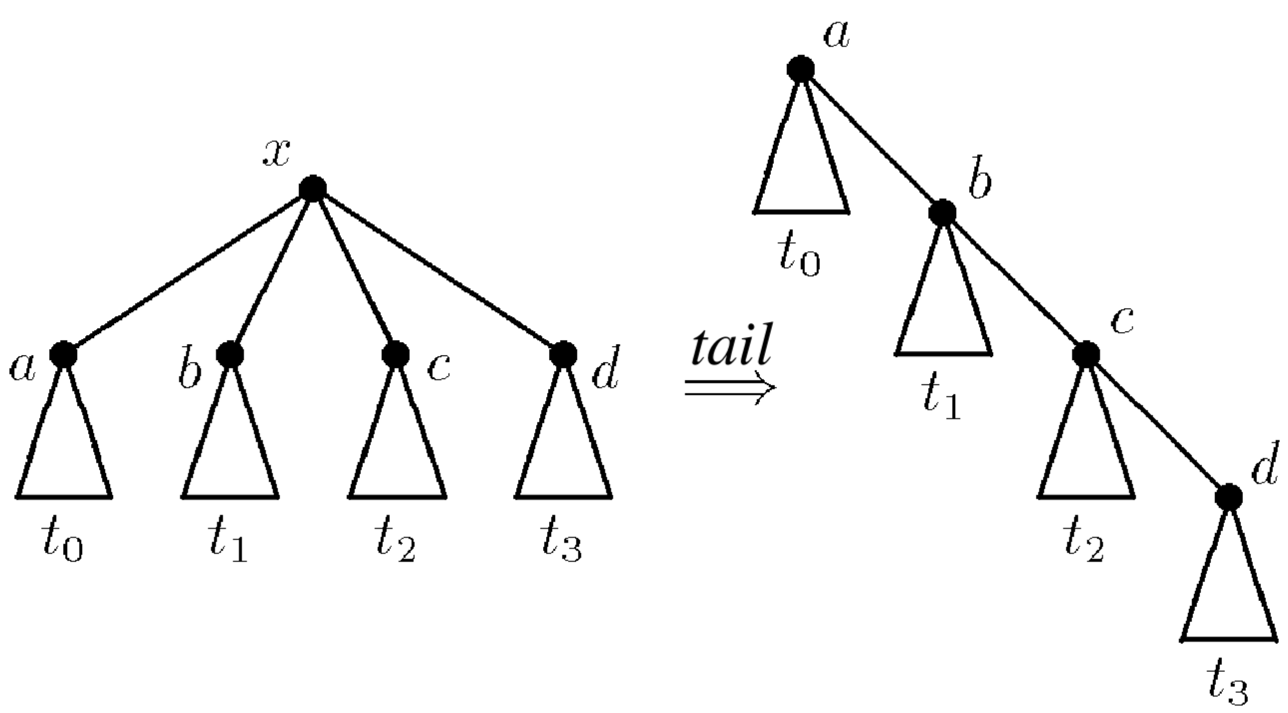
Catenable List Implementation[5]

- `tail` needs to make a Catenable List from the Queue of the Catenable List
  - Trivial if empty
  - If non-empty, `tail` uses a procedure `linkAll`

- `linkAll`:
  - Is trivial if the Queue has only one element
  - Recursively `link`s the head of the queue with the `linkAll` of its tail otherwise

Catenable List Implementation[6]

Concretely tail:



Proofs And Checks[1]

Our Leon post/pre-conditions are based on:

- `size`, `content` and `toList`
- Invariant functions
  - On both data structures, to ensure and check proper form of Queues

We wrote some Scala tests to be evaluated:

- Using `--eval`
- Using `scalac`

Proofs And Checks²

- Queue completely proved
  - Based on list
  - Not a surprise
- `linkAll` is unknown
  - Complex operation
  - Recursive
  - Could not check that all elements were in the resulting list
    - Using high-order function (`forall`)

```scala
def linkAll[T](q: Queue[CatenableList[T]]): CatenableList[T] = {
  require(q.isDefined && queueHasProperShapeIn(q))
  q.tail match {
    case QEmpty() => q.head
    case qTail => q.head.link(linkAll(qTail))
  }
} ensuring(res => q.forall(_.forall(res.contains(_))))
```

Verification Summary

CatenableList$$plus$plus	postcondition	62:20	valid	Z3-f	0.085
CatenableList$$plus$plus	precond. (call $this.link(t...	60:35	valid	Z3-f	0.014
CatenableList$cons	postcondition	48:20	valid	Z3-f	0.425
CatenableList$cons	precond. (call CCons[T](x, ...	47:17	valid	Z3-f	0.033
CatenableList$contains	match exhaustiveness	114:39	valid	Z3-f	0.011
CatenableList$content	match exhaustiveness	85:35	valid	Z3-f	0.006
CatenableList$content	precond. (call queueOfCatTo...	90:44	valid	Z3-f	0.010
CatenableList$forall	match exhaustiveness	109:51	valid	Z3-f	0.010
CatenableList$hasProperShape	match exhaustiveness	131:30	valid	Z3-f	0.007
CatenableList$head	match exhaustiveness	66:17	valid	Z3-f	0.010
CatenableList$head	postcondition	69:20	valid	Z3-f	0.019
CatenableList$link	match exhaustiveness	124:17	valid	Z3-f	0.009
CatenableList$link	postcondition	127:20	unknown	Z3-f	20.109
CatenableList$link	precond. (call $this.t.snoc...	125:54	valid	Z3-f	0.013
CatenableList$size	match exhaustiveness	35:35	valid	Z3-f	0.015
CatenableList$size	postcondition	43:21	valid	Z3-f	0.119
CatenableList$size	precond. (call sumTail[T]($...	40:37	valid	Z3-f	0.079
CatenableList$snoc	postcondition	53:20	valid	Z3-f	0.152
CatenableList$snoc	precond. (call $this ++ CCo...	52:17	valid	Z3-f	0.021
CatenableList$tail	match exhaustiveness	74:45	valid	Z3-f	0.011
CatenableList$tail	postcondition	79:20	unknown	Z3-f	20.108
CatenableList$tail	precond. (call linkAll[T]($...	76:45	valid	Z3-f	0.011
CatenableList$toList	match exhaustiveness	97:36	valid	Z3-f	0.013
CatenableList$toList	postcondition	105:20	unknown	Z3-f	20.128
CatenableList$toList	precond. (call queueOfCatTo...	102:39	valid	Z3-f	0.012
Queue$$plus$plus	match exhaustiveness	57:17	valid	Z3-f	0.003
Queue$$plus$plus	postcondition	66:20	valid	Z3-f	0.015
Queue$content	match exhaustiveness	82:17	valid	Z3-f	0.004
Queue$exists	match exhaustiveness	115:51	valid	Z3-f	0.004
Queue$foldLeft	precond. (call $this.head())	124:58	valid	Z3-f	0.006
Queue$foldLeft	precond. (call $this.tail())	124:35	valid	Z3-f	0.005
Queue$foldLeft	precond. (call $this.tail()...	124:35	valid	Z3-f	0.019
Queue$forall	match exhaustiveness	110:51	valid	Z3-f	0.003
Queue$hasProperShape	match exhaustiveness	130:30	valid	Z3-f	0.003
Queue$head	match exhaustiveness	31:17	valid	Z3-f	0.014
Queue$head	postcondition	34:21	valid	Z3-f	0.011
Queue$head	precond. (call $this.f.head())	32:45	valid	Z3-f	0.018
Queue$map	match exhaustiveness	92:37	valid	Z3-f	0.003
Queue$map	postcondition	97:21	valid	Z3-f	0.028
Queue$size	match exhaustiveness	22:35	valid	Z3-f	0.005
Queue$size	postcondition	27:21	valid	Z3-f	0.007
Queue$snoc	match exhaustiveness	48:17	valid	Z3-f	0.005
Queue$snoc	postcondition	52:21	valid	Z3-f	0.019
Queue$tail	match exhaustiveness	38:37	valid	Z3-f	0.007
Queue$tail	postcondition	44:21	valid	Z3-f	0.031
Queue$toList	match exhaustiveness	72:36	valid	Z3-f	0.005
Queue$toList	postcondition	77:22	valid	Z3-f	0.010
apply	precond. (call empty[T]().c...	146:30	valid	Z3-f	0.013
apply	precond. (call empty[T]().s...	141:30	valid	Z3-f	0.009
empty	postcondition	?:?	valid	Z3-f	0.010
empty	postcondition	?:?	valid	Z3-f	0.003
linkAll	postcondition	156:20	unknown	Z3-f	20.200
linkAll	precond. (call linkAll[T](q...	154:51	unknown	Z3-f	20.050
linkAll	precond. (call q.head())	154:39	valid	Z3-f	0.013
linkAll	precond. (call q.head())	153:42	valid	Z3-f	0.010
linkAll	precond. (call q.head().lin...	154:39	unknown	Z3-f	20.176
linkAll	precond. (call q.tail())	152:17	valid	Z3-f	0.007
linkAll	precond. (call q.tail())	152:17	valid	Z3-f	0.012
listOfCatToContent	match exhaustiveness	185:17	valid	Z3-f	0.010
listOfCatToContent	precond. (call l.h.content())	187:44	valid	Z3-f	0.008
listOfCatToContent	precond. (call listOfCatToC...	187:57	valid	Z3-f	0.007
	match exhaustiveness	201:17	valid	Z3-f	0.006
	precond. (call l.h.toList())	203:44	valid	Z3-f	0.006
	precond. (call listOfCatToL...	203:56	valid	Z3-f	0.006
	match exhaustiveness	177:17	valid	Z3-f	0.003
	precond. (call listOfCatToC...	179:45	unknown	Z3-f	20.149
	precond. (call listOfCatToC...	179:70	unknown	Z3-f	20.136
	match exhaustiveness	193:17	valid	Z3-f	0.004
	precond. (call listOfCatToL...	195:45	unknown	Z3-f	20.118
	precond. (call listOfCatToL...	195:67	unknown	Z3-f	20.073
	match exhaustiveness	169:17	valid	Z3-f	0.004
	postcondition	173:20	valid	Z3-f	0.009
	precond. (call lst.h.size())	171:63	valid	Z3-f	0.010
	precond. (call sumInList[T]...	171:44	valid	Z3-f	0.011
	match exhaustiveness	160:35	valid	Z3-f	0.003
	postcondition	165:20	unknown	Z3-f	20.150
	precond. (call sumInList[T]...	162:45	unknown	Z3-f	20.141
sumTail	precond. (call sumInList[T]...	162:63	unknown	Z3-f	20.132

total: 78 valid: 65 invalid: 0 unknown 13 263.144

Lessons Learned[1]

- High-order functions are problematic in Leon
  - Could not use `flatMap` and `foldLeft`
    - Even if so useful on recursive data structures

Lessons Learned^2

- `ensuring:`
  - `scala` requires explicit return type
  - Leon infers them

```scala
def tail: CatenableList[T] = {
  require(this.isDefined && this.hasProperShape)
  this match {
    case CCons(h, t) if t.isEmpty => CEmpty()
    case CCons(h, t) => CatenableList.linkAll(t)
  }
} ensuring(res => ...)
```

⇒

```scala
def tail: CatenableList[T] = {
  require(this.isDefined && this.hasProperShape)
  val res: CatenableList[T] = this match {
    case CCons(h, t) if t.isEmpty => CEmpty()
    case CCons(h, t) => CatenableList.linkAll(t)
  }
  res
} ensuring(res => ...)
```

Binomial Heaps

Heap structure using trees

Structure Properties

"Classical implementation of mergeable priority queues" (Okasaki p68)

Supports standard functionalities:

- `insert`
- `merge`
- `findMin`
- `deleteMin`

Structure Details[1]

- Binomial heaps are:
  - Collection of binomial trees
  - With particular structure and properties

- Nodes have:
  - a key (which type has a total order)
  - a rank

- Trees must satisfy Minimum Heap Property:
  - the key of a parent node must be smaller or equal to the key of any of its children nodes

Structure Details^2

- A node with rank:
  - $k \Rightarrow k$ children of ranks $k\text{-}1, k\text{-}2, \ldots, 0$
  - $0 \Rightarrow$ has no children

- Root of rank $k \Rightarrow 2^k$ elements
  - $1 + 2 + 4 + .. + 2^{k-1} + 1(root) = 2^k$

Structure Details[3]

- The binary heap's trees collection must not contain more than one tree of a particular rank
- A binomial heap can be mapped to a binary number: binary representation of its size
  - Trees are mapped to 1-valued bits of the number (ex: tree of rank *k* is the k^{th} bit, counting from 0 and from the right)

Tree Implementation[1]

Trees are implemented as nodes with a rank, a key (element) and a list of Trees:

```
Tree[T] :=
   Node[T](rank: BigInt, elem: T, ch: List[Tree])
```

Type `T` needs to be totally ordered:

```
T <: Ordered[T]
```

Tree Implementation^2

- If we take a node of rank *k*:
  - Its list of Trees needs to have exactly one Tree of each rank between *0* and *k-1*

  ⇒ ***To enforce***

- Implementation keeps the ranks of Trees in list in decreasing order
  - Invariant to check and to use for proofs

  ⇒ ***To enforce***

Tree Implementation[3]

- `link` function creates a Tree of rank k from two Trees of rank k-1
    - makes one Tree one child of the other
    - maintains minimum heap property: the Tree with larger root becomes the child

⇒ like addition of two bits

Tree Checks And Proofs

Our Leon post/pre-conditions are based on:

- `size`, `content` and `toList` functions

- Invariant functions for:
  - Rank uniqueness and ≥ 0
  - Decreasing order in the list of Trees

Binomial Heap Implementation[1]

A Binomial Heap is implemented as:

```
BH[T] := List[Tree[T]]
```

- Must not have more than one Tree with a particular rank

  ⇒ ***To enforce***

- The implementation keeps the ranks in the Trees list in increasing order

  ⇒ ***To enforce***

Binomial Heap Implementation[2]

- ***`merge`*** is like the addition of two binary numbers, recursively done:
  - It goes through the tree lists of both heaps
  - In increasing order of rank
- It compares the two smallest ranked Trees
  - Keep the smallest ranked Tree if exists
    - And recursively **`merge`** the others
  - If both Trees are of same ranks, `link`s them
    - Insert result with ***`insTree`*** in the rest of the recursively merged tree
      - equivalent to addition carry

Binomial Heap Implementation[4]

- ***`insert`***:
    - Can be seen like the **`merge`** with a Heap which has a unique Tree of rank 0
        - incrementation of a binary number
    - But the implementation does not use `merge`

- ***`insert`*** directly inserts the 0-ranked tree on the heap with *insTree* function

Binomial Heap Implementation[5]

- ***insTree*** inserts a tree ***t*** in a heap ***h***
  - **If** the rank of ***t*** *is less than min. rank of the Heap*: insert it before
    - as simple as a list ***cons***
  - **If the ranks are equal** it *link*s them and recursively "***insertTree***s" the resulting tree
  - Rank of *t* cannot be bigger than min. Heap rank:
    - In *insert*, the inserted Tree is 0-ranked
    - In *merge*, Trees of rank *k* are *link*ed and result is inserted in a Heap with only Trees with *rank ≤ k+ 1* ***⇒ To verify***

Binomial Heap Implementation[7]

- ***`findMin`*** simply recursively finds the minimum **root** of the Tree list
  - Can do that because Trees are **heap-ordered**
    - Finds the minimum root of the tail
    - Compares with root of head
    - Keeps the smallest element

Binomial Heap Implementation[8]

- ***deleteMin*** uses the helper function ***getMin***
  - ***`getMin`*** returns tree with the minimal root and the 'rest of the trees list'
    - i.e. all trees, in the same order, but without the one with minimal root
  - ***`deleteMin`*** reverses the tree list of the minimum tree (to get the right order of ranks for heaps) and merges it with the 'rest of trees list'

- ***getMin*** recursively apply itself to the tail of the trees list, and compares with the head to find the tree with minimal root.

Binomial Heap Proofs And Checks[1]

Our Leon post/pre-conditions are based on:

- *`size`*, *`content`* and *`toList`* functions
- Invariant functions: on both data structures, to ensure and check proper forms

We wrote some Scala tests to be evaluated:

- Using `--eval`
- Using `scalac`

Proofs And Checks²

- some *unknown*
- Due to recursive definitions of the data structure
- Did not find a proper way to prove the invariants

⇒ Used tests as well

⇒ Tests proved some “unknown” to be wrong.

Verification Summary

BinHeap$content	precond. (call content($thi...	53:5	valid	Z3-f	0.004
BinHeap$deleteMin	postcondition	43:15	unknown	Z3-f	20.119
BinHeap$deleteMin	precond. (call getMin($this...	40:19	valid	Z3-f	0.012
BinHeap$deleteMin	precond. (call merge(n.chil...	41:32	unknown	Z3-f	20.063
BinHeap$findMin	precond. (call getMin($this...	33:18	valid	Z3-f	0.029
BinHeap$insert	postcondition	23:15	unknown	Z3-f	20.292
BinHeap$insert	precond. (call insTree($thi...	21:32	valid	Z3-f	0.045
BinHeap$merge	postcondition	29:15	unknown	Z3-f	21.807
BinHeap$merge	precond. (call merge($this....	27:32	valid	Z3-f	0.025
BinHeap$size	postcondition	49:15	valid	Z3-f	0.007
BinHeap$size	precond. (call size($this.t...	47:23	valid	Z3-f	0.008
Tree$content	precond. (call treeListToCo...	40:58	valid	Z3-f	0.007
Tree$link	postcondition	24:21	unknown	Z3-f	20.120
Tree$size	postcondition	30:21	valid	Z3-f	0.010
Tree$size	precond. (call treeListToCo...	28:39	valid	Z3-f	0.009
Tree$toList	postcondition	36:21	unknown	Z3-f	20.203
Tree$toList	precond. (call treeListToLi...	34:57	valid	Z3-f	0.017
apply	precond. (call empty().inse...	67:35	valid	Z3-f	0.022
content	match exhaustiveness	145:5	valid	Z3-f	0.009
content	precond. (call content(lhs.t))	147:40	valid	Z3-f	0.021
content	precond. (call lhs.h.conten...	147:27	valid	Z3-f	0.024
getMin	match exhaustiveness	124:5	valid	Z3-f	0.019
getMin	postcondition	132:15	valid	Z3-f	0.110
getMin	precond. (call getMin(lhs.t))	127:9	valid	Z3-f	0.054
getMin	precond. (call getMin(lhs.t))	127:9	valid	Z3-f	0.069
getMin	precond. (call getMin(lhs.t))	127:9	valid	Z3-f	0.063
hasIncrRanks	match exhaustiveness	80:46	valid	Z3-f	0.011
insTree	match exhaustiveness	114:27	valid	Z3-f	0.023
insTree	postcondition	120:15	valid	Z3-f	0.192
insTree	precond. (call insTree(lhs....	117:28	valid	Z3-f	0.093
insTree	precond. (call t1.link(lhs.h))	117:40	valid	Z3-f	0.061
merge	match exhaustiveness	102:27	valid	Z3-f	0.014
merge	postcondition	110:15	unknown	Z3-f	20.653
merge	precond. (call insTree(merg...	107:68	unknown	Z3-f	20.323
merge	precond. (call lhs.h.link(r...	107:93	valid	Z3-f	0.063
merge	precond. (call merge(lhs.t,...	107:76	valid	Z3-f	0.036
merge	precond. (call merge(lhs.t,...	105:73	valid	Z3-f	0.021
merge	precond. (call merge({val x...	106:73	valid	Z3-f	0.024
size	match exhaustiveness	136:23	valid	Z3-f	0.018
size	postcondition	141:15	valid	Z3-f	0.080
size	precond. (call lhs.h.size())	138:27	valid	Z3-f	0.056
size	precond. (call size(lhs.t))	138:36	valid	Z3-f	0.039
treeListHasDecrRanks	match exhaustiveness	66:60	valid	Z3-f	0.005
treeListToContent	match exhaustiveness	95:40	valid	Z3-f	0.005
treeListToContent	precond. (call l.h.content())	97:45	valid	Z3-f	0.007
treeListToContent	precond. (call treeListToCo...	97:58	valid	Z3-f	0.007
treeListToCount	match exhaustiveness	77:35	valid	Z3-f	0.004
treeListToCount	postcondition	82:21	valid	Z3-f	0.010
treeListToCount	precond. (call l.h.size())	79:45	valid	Z3-f	0.006
treeListToCount	precond. (call treeListToCo...	79:54	valid	Z3-f	0.007
treeListToList	match exhaustiveness	86:41	valid	Z3-f	0.010
treeListToList	precond. (call l.h.toList())	88:45	valid	Z3-f	0.007
treeListToList	precond. (call treeListToLi...	88:57	valid	Z3-f	0.007

total: 53 valid: 45 invalid: 0 unknown 8 164.950

Lessons Learned

- Problems with the total order of T
  - Tried <: Ordered[T], but Leon does not support this
  - Could use Ordering to pass to each function
  - We decided to implement a BigInt version
    - No loss of generality when it comes to logic
    - But easier to read

- A few boilerplate due to Leon’s limitations
  - No method in a subclass
  - No implicit class (This one is probably tricky)
    - A BinHeap[T] is a List[Tree[T]] - it would have been nice to use case class bodies or implicit classes

Conclusion

- Two structures implemented and checked
  - Some *unknown* results unfortunately
    - Difficult to avoid with recursive definitions

- Leon is powerful but has some limitations:
  - Need to find ways to circumvent some syntactic limitations and some library limitations
  - Leon code cannot be used in Scala as such due to missing syntactic sugar (e.g. repeated parameters for apply methods, etc.).

- *Interesting and powerful tool nevertheless!*

Thank You !

Backup Slides

Queue Complexity[1]

Simple complexity proofs:

- `head` has the same complexity as its counterparts in `List[T]`: O*(1)*
  - This is guaranteed by the invariant that a non-empty queue has a non-empty left hand-side list

- `snoc` has the complexity of `cons` of `List [T]`:
  - O*(1)*

Queue Complexity²

Simple complexity proofs (continued):

- `tail` has the same cost as `tail` on `List[T]` unless the right hand side list has to be reversed:
  - O*(n)* worst-case, *O(1)* amortized
  - **Proof:**
    - Banker's method: `snoc` pays *2*, cost of *1*, `tail` has cost *O(1)* when no reversal
    - When list of size *m* is reversed, cost is *m+1*, we have *m* credits → amortized cost of *1*

Code Samples: Queue, Cat. Lists

```
def cons(x: T): CatenableList[T] = {
  require(this.hasProperShape)
  CCons(x, QEmpty[CatenableList[T]]()) ++ this
} ensuring(res => res.content == this.content ++ Set(x) && res.head == x && res.size == this.size + 1)

def snoc(x: T): CatenableList[T] = {
  require(this.hasProperShape)
  this ++ CCons(x, QEmpty[CatenableList[T]]())
} ensuring(res => res.content == this.content ++ Set(x) && res.size == this.size + 1)

def ++(that: CatenableList[T]): CatenableList[T] = {
  require(this.hasProperShape && that.hasProperShape)
  (this, that) match {
    case (CEmpty(), _) => that
    case (_, CEmpty()) => this
    case _ => this.link(that)
  }
} ensuring(res => res.content == this.content ++ that.content && res.size == this.size + that.size)
private def link(that: CatenableList[T]): CatenableList[T] = {
  require(this.isDefined && this.hasProperShape && that.isDefined && that.hasProperShape)
  this match {
    case CCons(h, t) => CCons(h, t.snoc(that))
  }
} ensuring(res => res.content == this.content ++ that.content && res.size == this.size + that.size)

/* Invariants */

def hasProperShape = this match {
  case CEmpty() => true
  /* The queue must have proper shape according to queue specs, and we cannot have a queue of empty lists */
  case CCons(h, t) => CatenableList.queueHasProperShapeIn(t)
}
```

Code Samples: Tree[1]

```scala
def merge(lhs: List[Tree], rhs: List[Tree]): List[Tree] = {
  require(hasProperShape(lhs) && hasProperShape(rhs))
  val res: List[Tree] = (lhs, rhs) match {
    case (t, Nil()) => t
    case (Nil(), t) => t
    case (Cons(t1, ts1), Cons(t2, ts2)) if t1.rank < t2.rank => t1 :: merge(ts1, t2 :: ts2)
    case (Cons(t1, ts1), Cons(t2, ts2)) if t1.rank > t2.rank => t2 :: merge(t1 :: ts1, ts2)
    case (Cons(t1, ts1), Cons(t2, ts2)) if t1.rank == t2.rank => insTree(merge(ts1, ts2), t1 link t2)
  }
  res
} ensuring (res => hasProperShape(res))
```

```scala
def insTree(lhs: List[Tree], t1: Tree): List[Tree] = {
  require(hasInsTreeProperShape(lhs, t1))
  val res: List[Tree] = lhs match {
    case Nil() => t1 :: Nil()
    case Cons(t2, ts) if t1.rank < t2.rank => t1 :: t2 :: ts
    case Cons(t2, ts) => insTree(ts, t1 link t2)
  }
  res
} ensuring (res => hasProperShape(res))
```

```scala
def hasIncrRanks(c: List[Tree]): Boolean = c match {
  case Nil() => true
  case Cons(t, Nil()) => t.rank >= 0
  case Cons(t1, ts @ Cons(t2, _)) => t1.rank >= 0 && t1.rank < t2.rank && hasIncrRanks(ts)
}
```

Code Samples: Bin. Heap^2

```
def getMin(lhs: List[Tree]): (Tree, List[Tree]) = {
  require(!lhs.isEmpty && hasProperShape(lhs))
  lhs match {
    case Cons(t, Nil()) => (t, Nil())
    case Cons(t, ts) =>
      getMin(ts) match {
        case (tp, tsp) if t.root <= tp.root => (t, ts)
        case (tp, tsp) => (tp, t :: tsp)
      }
  }
} ensuring (res => res._1.hasProperShape && hasMinHeapProp(res._2))
```

```
def link(that: Tree) : Tree = {
  require (this.hasProperShape && that.hasProperShape && this.rank == that.rank)
  val res: Tree = (this, that) match {
    case (t1, t2) if t1.root <= t2.root => TreeNode(t1.rank + 1, t1.root, t2 :: t1.children)
    case (t1, t2) => TreeNode(t2.rank + 1, t2.root, t1 :: t2.children)
  }
  res
} ensuring (res => res.size == this.size + that.size &&
  res.hasProperShape && res.content == this.content ++ that.content)
```

Complexities of various data structures

Name	Running Times of Supported Functions	Page
banker's queues	*snoc*/*head*/*tail*: $O(1)$	26
physicist's queues	*snoc*/*head*/*tail*: $O(1)$	31
real-time queues	*snoc*/*head*/*tail*: $O(1)^\dagger$	43
bootstrapped queues	*head*: $O(1)^\dagger$, *snoc*/*tail*: $O(\log^* n)$	89
implicit queues	*snoc*/*head*/*tail*: $O(1)$	113
banker's deques	*cons*/*head*/*tail*/*snoc*/*last*/*init*: $O(1)$	56
real-time deques	*cons*/*head*/*tail*/*snoc*/*last*/*init*: $O(1)^\dagger$	59
implicit deques	*cons*/*head*/*tail*/*snoc*/*last*/*init*: $O(1)$	116
catenable lists	*cons*/*snoc*/*head*/*tail*/$+\!\!+$: $O(1)$	97
simple catenable deques	*cons*/*head*/*tail*/*snoc*/*last*/*init*: $O(1)$, $+\!\!+$: $O(\log n)$	119
catenable deques	*cons*/*head*/*tail*/*snoc*/*last*/*init*/$+\!\!+$: $O(1)$	122
skew-binary random-access lists	*cons*/*head*/*tail*: $O(1)^\dagger$, *lookup*/*update* : $O(\log n)^\dagger$	79
skew binomial heaps	*insert*: $O(1)^\dagger$, *merge*/*findMin*/*deleteMin* : $O(\log n)^\dagger$	83
bootstrapped heaps	*insert*/*merge*/*findMin*: $O(1)^\dagger$, *deleteMin*: $O(\log n)^\dagger$	102
sortable collections	*add*: $O(\log n)$, *sort*: $O(n)$	35
scheduled sortable collections	*add*: $O(\log n)^\dagger$, *sort*: $O(n)^\dagger$	47

Worst-case running times marked with †. All other running times are amortized.