@AndrzejWasowski

Andrzej Wąsowski

Advanced Programming

Finger Trees Persistent Data Structures, Polymorphic Recursion

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

Goals

Finger trees are more of an excuse than a goal...

- To hint how **pure persistent data structures** are designed
- To **train recursion**
- To see **polymorphic recursion** in action
- To see type classes, higher order types, property-based testing, Gen, and monoids in a slightly larger context of delivering an implementation of a data structure
- To compare Haskell and Scala in a case study setting

What is a deque?

- A linear data structure providing efficient access at both ends
- Efficient add-from-left, and efficient add-from-right
- Efficient remove-from-left, and remove-from-right
- Efficient emptiness check
- A double-ended stack

```scala
trait F[A]
  def addL(q: F[A]) (a: A): F[A]
  def addR(q: F[A]) (a: A): F[A]
  def headL(q: F[A]): A
  def headR(q: F[A]): A
  def tailL(q: F[A]): F[A]
  def tailR(q: F[A]): F[A]
  def empty(q: F[A]): Boolean
  def toTree(l: List[A]): F[A] // nice to have
```

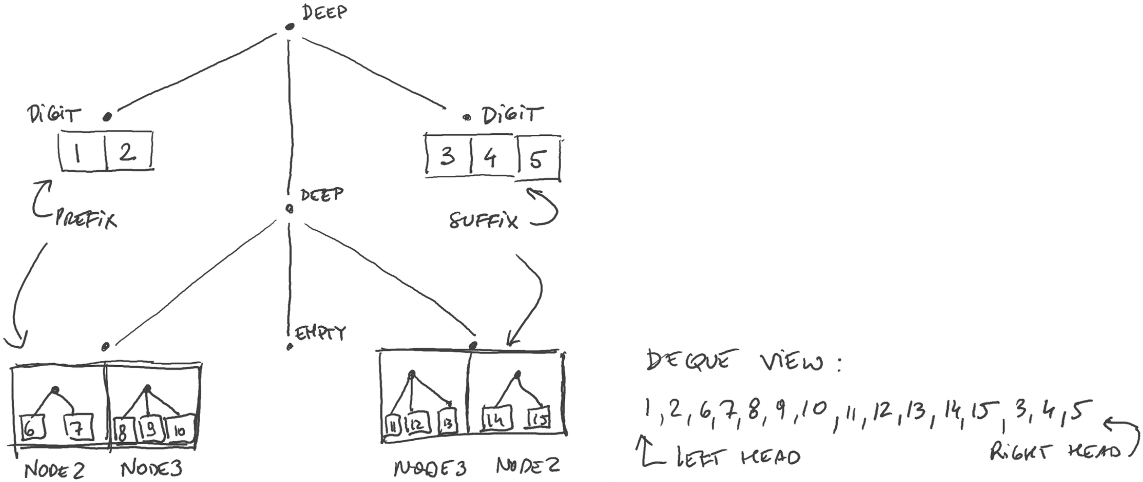
```scala
q match
case ConsL (hd, tl) => ...
case ConsR (tl, hd) => ...
case Nil => ...
```

Finger Trees

- **Balanced trees**, same family as AVL, 2-3-4, red-black trees, etc.
- They maintain the same invariant: logarithmic depth, so depth is in $O(\log n)$ if the tree holds n elements
- Finger trees have four kinds of nodes
  - Leaf nodes (store data elements):
    - **Empty**: stores no elements
    - **Single**: stores exactly one element
    - **Digit**: stores from 1 to 4 data elements
  - Internal nodes (store **finger trees**, with prefix and suffix)
    - **Deep** means somewhere in the middle, not in the prefix, and not in the suffix
    - Deep node contains a **prefix** digit, a **middle** finger tree, and a **suffix** digit
  - Data elements (a twist!):
    - Top level contains just data values directly
    - At depth 1 we use balanced trees of depth 1 as elements
    - Elements at depth n are trees of height n (**Deeper elements are heavier!**)
    - Element trees are always balanced, binary and ternary (Node2, and Node3)

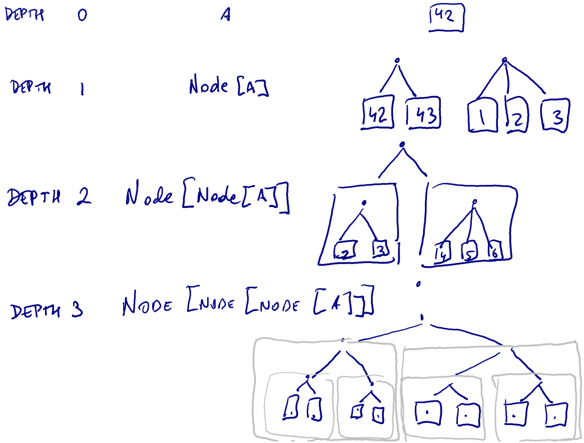
Finger Tree

A Visual Example



Element in the Trees

Representing trees in a type-safe manner



- `Node[ ... Node [A] ... ]`: type of trees of a fixed depth
- The type checker counts the depths of the trees by counting nestings!

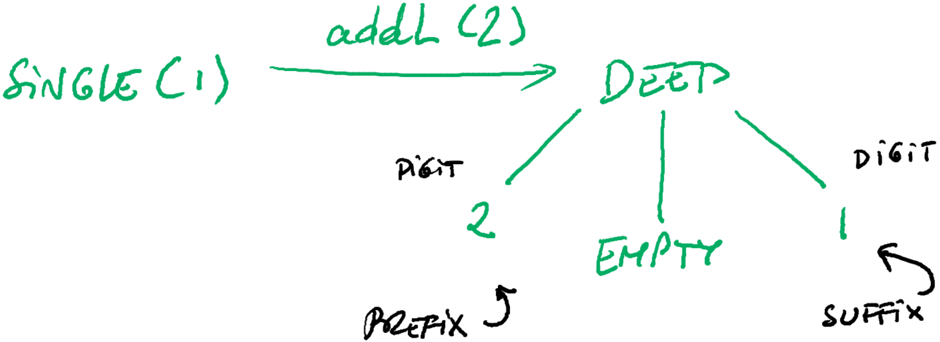
```
enum Node[+A]:
  case Node2(l: A, r: A)
  case Node3(l: A, m: A r: A)


enum FingerTree[+A]
  case Empty
  case Single(data: A)
  case Deep[l: Digit[A], m: FingerTree[Node[A]], r: Digit[A])
```

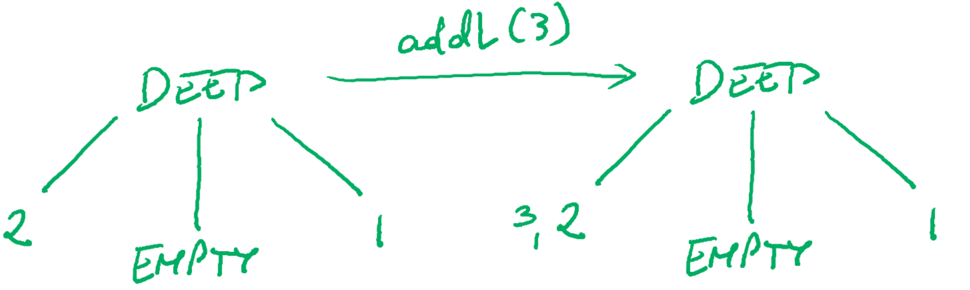

Adding from the left (addL)

$$\text{EMPTY} \xrightarrow{\text{addL}(1)} \text{SINGLE}(1)$$

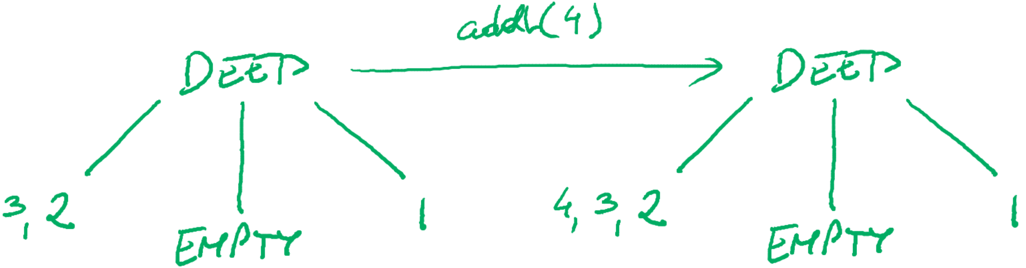
Adding from the left (addL)



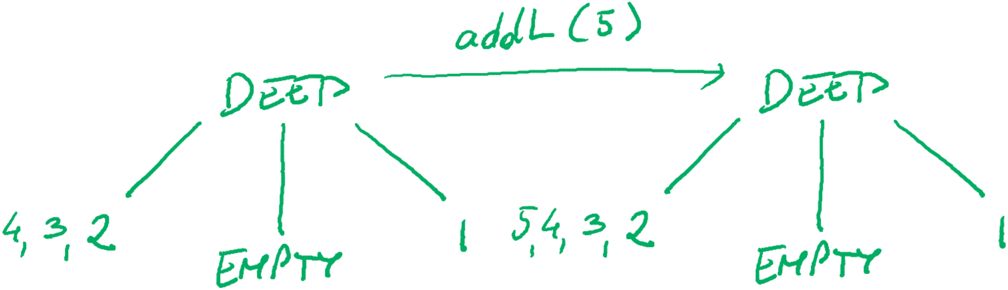
Adding from the left (addL)



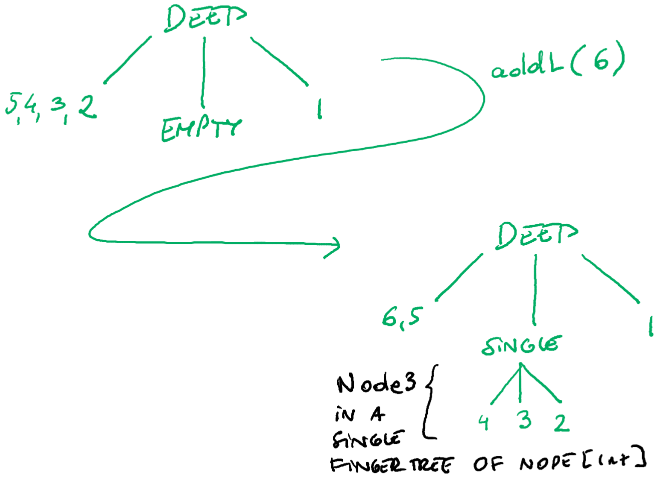
Adding from the left (addL)



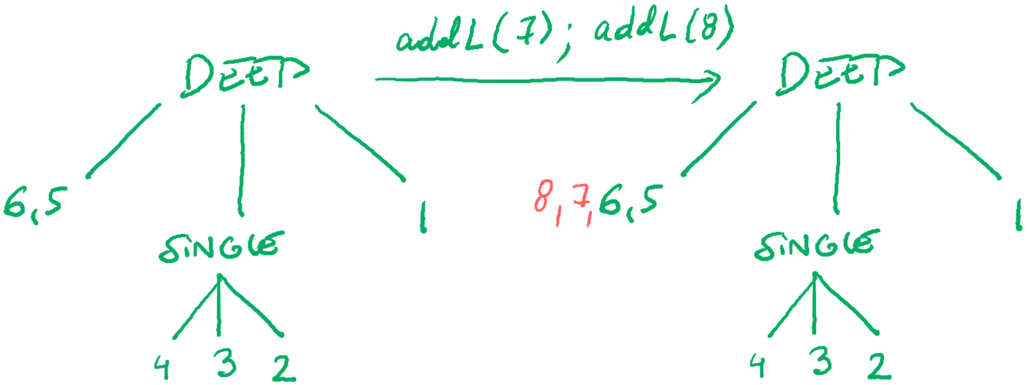
Adding from the left (addL)



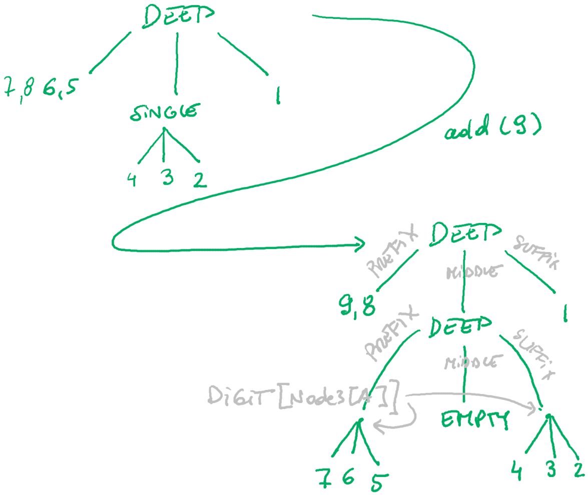
Adding from the left (addL)



Adding from the left (addL)



Adding from the left (addL)



Finger Trees

Sketch of complexity of addition

- Insertion from the right is symmetric (so we don't even implement it)
- Insertion may recurse down the spine and take $O(\log n)$ time worst case
- This gives $O(\log n)$ worst case cost per insertion
- Still, the cost is $O(1)$ amortized time
- Insertion can only propagate to the next level if a Digit is full,
- This makes the digit size 2, so next operation on it will not propagate
- At most half operations descend one level, half of that 2 levels etc.
- For n operations we get: $O(1 + 1/2 + 1/4 + \ldots + 1/2^n) = O(2) = O(1)$
- The amortized cost is constant

Polymorphic Recursion

- Go back to finger tree types slide: what happens in line 9?
- A type constructor `T` over `A` nests a value of type `T[Node[A]]`
- How different from `List`! `Cons[A](hd:A, tl:List[A]) extends List[A]`
- **Mentimeter:** What will be the type nested by `FingerTree[Node[A]]` ?
- Imagine a recursive function `f[A]` traversing `FingerTree[A]`
- When it hits `Deep`, it will call itself recursively on the middle value `m`
- That value `m` will be of type `Node[A]`
- We call `f[Node[A]]` from within `f[A]` **changing the type of the current function at the recursive call**
- **Polymorphic recursion** (AKA Milner–Mycroft type-ability or the Milner–Mycroft calculus) *refers to a recursive parametrically polymorphic function in which the type parameter changes with each recursive invocation made instead of staying constant*

Polymorphic Recursion (2)

- Languages supporting polymorphic recursion known to AW: Haskell, Scala, Ocaml (and maybe F# as per Internet rumours)
- Hinz/Paterson use a common pattern to use polymorphic recursion to ensure that the trees are balanced.
- This limits the difference between the right and left side of the tree
- You get a typing error, if your code can produce an unbalanced tree
- Ingenious!

Removing from the Left

(headL, tailL, ConsL)

$$\text{EMPTY} \xrightarrow{\text{removeL}} \text{Fail!}$$

Removing from the Left

(headL, tailL, ConsL)

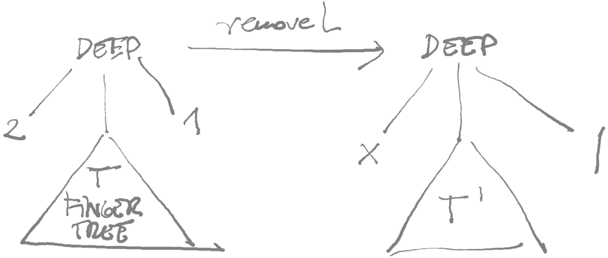
$$\text{Single}(a) \xrightarrow{\text{removeL}} \text{Empty},\ a$$

Removing from the Left

(headL, tailL, ConsL)

Removing from the Left

(headL, tailL, ConsL)



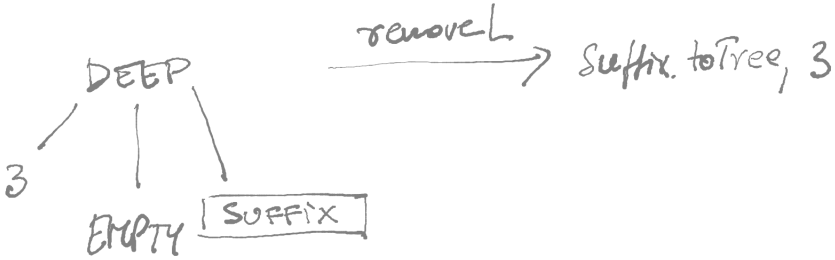
where removeL(T) = T', x

where x has at most 3 elements

(why?)

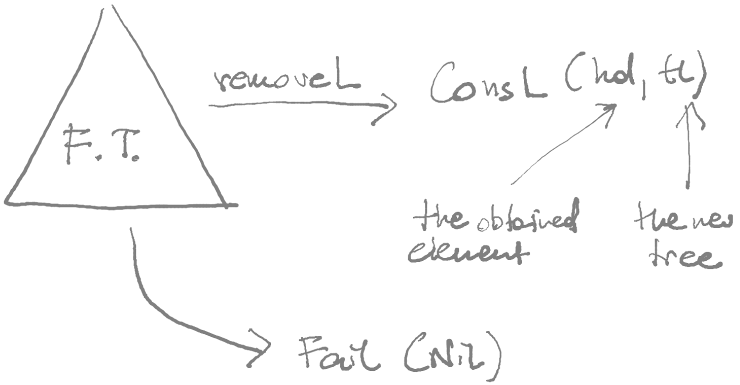
Removing from the Left

(headL, tailL, ConsL)



Removing from the Left

(headL, tailL, ConsL)



Removing Elements

Extractors

Hinze and Paterson implement the previous as a **view function**:

```
data ViewL a = Nil | ConsL a (FingerTree a)
viewL :: FingerTree a -> ViewL a
```

Not entirely convenient. In Scala using this would look like this:

```
viewL(t) match { case Nil =>...; case ConsL(a, t) =>... }
```

Note: For lists we do not need to explicitly match on the view.
In Scala we can make views automatic using **extractors**, so they behave like for lists:

```
t match { case Nil =>...; case ConsL(a,t) =>... }
```

To do this we need two **objects** `Nil` and `ConsL` implementing the `unapply` method.
`unapply` **takes** value to be matched as parameter.
`unapply` **returns** `Option[T]` where `T` the type of parameters of “matching constructor”

For `ConsL(a:A,t:T)` the return type should be `Option[(A,T)]`
In the method, perform matching and return `Some` if successful, `None` if failed.

