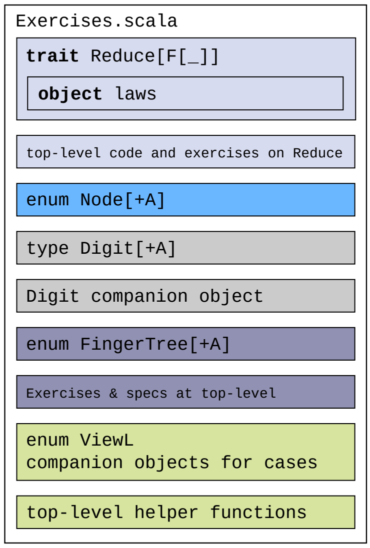
Finger Trees: Persistent Data Structures & Polymorphic Recursion



Read the paper Hinze and Paterson and solve the preparatory exercises 1–5 before starting to program in Scala. Then use the remaining exercises to complete the implementation of finger trees in Scala.

We train the following skills during this exercise:

- Reimplementing a design based on a description in a research paper in the area of data structures
- Implementing persistent pure data structures without pointer/reference manipulation
- Using *polymorphic recursion* and advanced types to enforce complex properties (here the trees being balanced)
- Property-based testing and test-data generation
- Reflecting on differences between two programming languages: Haskell and Scala

The file with the exercises have similar structure to the ones before—it combines several traits, classes, and objects. Extensions are heavily used to allow the exercises to appear in the file linearly—in a real project we would rather group the functions under the right types without extensions. The diagram on top of the page, may help you to orient yourself a bit (also use the overview support in your editor).

Our implementation is designed to be eager, following the regular strictness of Scala. However it would be an interesting exercise to extend it so that it is possibly lazy, like in the paper of Hinze and Paterson. The obvious choice is to make values of elements stored in the queue lazy. Then there is also a discussion of possible suspension of the middle element of the tree on page 7 (we do not explore this).

Hand-in: `Exercises.scala`

Exercise 1. (skip if you know what a deque is) Assume that we have a simple double-ended queue (a deque). We will make one diagram per each point below.

1. Draw how the dequeue looks after adding 1 on the left, and then adding 2 on the left, and then adding 3 on the left.
2. Now take the dequeue from the previous point and draw how it looks after adding 4 on the **right**, and then after adding 5 on the **right**.
3. Now take the dequeue resulting from the previous point and draw how it looks after popping two elements from the **left**, and one element from the **right**.

Exercise 2. Assume that we have an empty finger tree ready to store integer numbers. We perform the following operations on the tree. Draw how the tree looks after the series of operations in each point.

1. We add number 1, and then number 2 (both from the left).
2. Then add numbers 3, 4 and 5 from the left (in this order)
3. Then add number 6 from left
4. Then add numbers 7, 8 and 9 from the **left**.
5. Then add number 10 from the **right**.
6. Then remove a number from the **right** twice.
7. Then remove a number from the **right**.

Exercise 3. This exercise uses Haskell notation to stay close to the paper. So `(+)` denotes the binary plus operator and `(:)` denotes the cons operator for lists (in Scala denoted using double colon `(_::_)` or `Cons(_,_)`). Square brackets in Haskell are used to denote list literals, with empty square brackets representing the empty list (`Nil` or `List()` in Scala). The reduce right and reduce left functions are defined on page 3 in the paper.

What is the result of running:

1. `reducer (+) [1,2,3,4] 0`
2. `reducel (+) 0 [1,2,3,4]`
3. `reducer (+) (Node3 1 2 3) 0`
4. `reducel (+) 0 (Deep [1,2] (Single (Node3 3 4 5)) [0])`
5. `reducer (:) (Deep [1,2] (Single (Node3 3 4 5)) [0]) []`
6. `reducel (:) [] (Deep [1,2] (Single (Node3 3 4 5)) [0])`

Exercise 4. Write the following Haskell data terms using the corresponding Scala type constructors from `Exercises.scala`

1. `[1,2,3]`
2. `[]`
3. `Node2 1 2`
   Observe that we work with `Node2[Int]` here (the type parameter is inferred).
4. `Node2 (Node2 1 2) (Node2 3 4)`
   After solving, make type parameters to `Node2` explicit, to see whether you understand the types.
5. `Deep [1,2] (FingerTree (Node3 3 4 5)) [0]`
   Also make the type parameters explicit.

Exercise 5. Translate to Scala the following Haskell type expressions. Use the algebraic data types for finger trees as defined in `Exercises.scala`. Double colon in Haskell denotes a typing annotation, like a single colon in Scala. Haskell uses single arrows (`->`) as function type constructors, whereas Scala uses double arrows (`=>`). Small letters (free names) in Haskell types denote type variables. These can be both usual type parameters and higher kinded type parameters. Square brackets are also the list type constructor in Haskell, so `[a]` roughly corresponds to `List[A]` in Scala.

NB. Some of these are solved in our Scala file, but please try to solve them yourself – then you will be able to work with the Scala file and with the paper more easily.

1. `addL :: a -> FingerTree a -> FingerTree a`
2. `addLprime :: f a -> FingerTree a -> FingerTree a`
3. `toTree :: f a -> FingerTree a`
4. `viewL :: FingerTree a -> ViewL FingerTree a`
5. `deepL :: [a] -> FingerTree a -> ViewL FingerTree a`

Exercise 6. Read Section 2.1 in Hinze/Paterson, and recall our own definition of the `Monoid` type class, and its instances. Read through the provided file `Monoid.scala`—this file differs only slightly from the last week's one, as it is no longer constrained by being part of the exercises. I used the file from the book git repository, after introducing only minimal adjustments.

Exercise 7. Study Section 2.2 in Hinze/Paterson. The trait `Reduce` in our exercise file implements the type class *Reduce* in Scala. We implement foldable laws using the monoid instance (to test whether the two reducers are the same in a monoid). This law is based on the argument in the opening of Section 2.2. Implement the instance for *Reduce* for `List`—as `Reduce` is essentially the same as `Foldable`, you can compare with your notes of the prior week.

Exercise 8. The final part of Section 2.2 shows how to implement `toList` using an instance of `Reduce`. Implement it as an extension method for values of type `F[A]` available if an instance `Reduce[F]` exists. This makes the function easy to access for users.

Exercise 9. Read the opening of Section 3 (the first two pages, including the first page of Section 3.1). Find the types `Node` and `FingerTree` in our implementation. (The type `Tree` is not implemented, as it is merely brought up in the paper as a motivating example, and not used later in the paper.)

Understand how these types are realized in Scala and how they relate to each other; compare with the Haskell versions in the paper. Finally, identify the location in Scala's type definitions where *polymorphic recursion* is used (important, do not skip this step!).

No programming in this exercise.

Exercise 10. We are now moving to the final part of Section 3.1 (page 5) with the goal of implementing an instance of the reduce type class for nodes. We place it in the companion object of `Node` towards the end of the file (`Node.scala`). Implement this type class instance providing definitions for `reduceR` and `reduceL` for nodes.

Exercise 11. Implement reducers for finger trees as specified in Haskell in the final part of Section 3.1 (page 5). You will have to use the instances of `Reduce` type class for nodes and digits when solving this exercise. The exercise has week tests. We develop more serious tests for this part much later, in exercises 16–17.

Exercise 12. Read the first half page of Section 3.2, which specifies how to add an element to the left of the deque, implemented in a finger tree. Note that in the paper the name of the function is $\triangleright$, and that it is written infix, so the first argument is to the left of the triangle. Name the function `addL`.

After the tests for `addL` pass, implement also `addR`. We develop tests for `addR` in Exercise 13.

The implementations of `addL` and `addR` should involve *polymorphic recursion*. Find (and mark with a comment) the calls which are polymorphically recursive.

Exercise 13. Write three tests for `addR`:

a) A scenario test that adds 42 to an empty tree, then converts the tree to a list and checks whether this list is a singleton containing 42.

b) A scenario test that adds 42 to an empty tree, then adds 42, and converts the tree to a list and checks whether this list contains 42 and 43.

c) Generalize the above tests to a property-based test: for any list `l` adding elements from the left (from the head) with `addR` will produce a tree that contains the same list (for instance after converting it back to a list)

Exercise 14. We named the functions `addL` and `addR`, but we can use a unicode triangle as an alias to be consistent with the paper. Implement the extensions for infix (operator) versions of `addL` and addR. This is best done by delegating to the other implementations.

Exercise 15. We skip the lifted (primed) versions of addR and addL and proceed to implement toTree from page 6. You can use reduce directly in toTree to save time on implementing the primed versions.

Exercise 16. Implement an instance of the type class Gen for FingerTree, so that we can use trees in property-based tests. Do it in two steps. First, implement fingerTreeOfN[A](n: Int) that creates a tree containing n elements. You can use toTree in this implementation. Second, create fingerTree[A] by invoking fingerTreeOfN with a generated size. Be careful to limit the sizes of the generated trees, not to run out memory.

Finally, expose the latter generator as a given instance of Arbitrary for FingerTree.

We will use these generators in tests for Reduce[FingerTree] in the next exercise.

Exercise 17. Implement the property tests for Reduce[FingerTree] using Reduce.law. You can get inspiration from tests for Reduce[List] and Reduce[Node].

Exercise 18. Find the definition of type $View_L$ in the paper, and then the corresponding Scala implementation ViewL in Exercises.scala. Convince yourself that you understand the proposal for Scala. No coding in this exercise.

The paper uses explicit views to implement deconstruction of finger tree values (removing elements with pattern matching). We want to follow another route and extend Scala to allow pattern matching on finger trees.

Read the Scala documentation about extractors:

- https://docs.scala-lang.org/tour/extractor-objects.html
- https://docs.scala-lang.org/scala3/reference/changed-features/pattern-matching.html

With extractors, we do not have to call the view function explicitly. Thus instead of creating an ADT for views, we will define three extractor objects (in the next exercise).

To check your understanding, find the implementation of the extractor Digit in the object Digit.

Exercise 19 [difficult]. Implement extractors Nil, ConsL, and ConsR. The behavior of Nil is specified in the first case of definition of $view_L$ in the paper. The behavior of ConsL is specified by the two remaining cases. Behaviour of ConsR will be symmetric.

Note that this exercise requires also implementing deepL and deepR, presented on the next page in the paper. The tests use the head and tail functions defined in page 7, and use your pattern extractors in the trait FingerTree. You can check how patterns are used in their code in the trait.