Purely Functional Data Structures and Monoids

Donnacha Oisín Kidney

May 9, 2020

Purely Functional Data Structures

Why Do We Need Them?

Why do pure functional languages need a different way to do data structures? Why can't we just use traditional algorithms from imperative programming?

Why Do We Need Them?

Why do pure functional languages need a different way to do data structures? Why can't we just use traditional algorithms from imperative programming?

To answer that question, we're going to look at a very simple algorithm in an imperative language, and we're going to see how *not* to translate it into Haskell.

Why Do We Need Them?

Why do pure functional languages need a different way to do data structures? Why can't we just use traditional algorithms from imperative programming?

To answer that question, we're going to look at a very simple algorithm in an imperative language, and we're going to see how *not* to translate it into Haskell.

The mistake we make may well be one which you have made in past!

A Simple Imperative Algorithm

A Simple Imperative Algorithm

(in Python)

A Simple Imperative Algorithm

We're going to write a function to create an array filled with some `ints`.

A Simple Imperative Algorithm

It works like this.

```
>>> create_array_up_to(5)
[0,1,2,3,4]
```

A Simple Imperative Algorithm

This is its implementation.

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

A Simple Imperative Algorithm

We first initialise an empty array.

```
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$\Leftarrow$

A Simple Imperative Algorithm

And then we loop through the numbers from `0` to `n-1`.

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

A Simple Imperative Algorithm

We append each number on to the array.

```
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

A Simple Imperative Algorithm

And we return the array.

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

A Simple Imperative Algorithm

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

```
>>> create_array_up_to(5)
[0,1,2,3,4]
```

Trying to Translate it to Haskell

Trying to Translate it to Haskell

We're going to run into a problem with this line.

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

Trying to Translate it to Haskell

We're going to run into a problem with this line.

```
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

The append function *mutates* `array`: after calling append, the value of the variable `array` changes.

Trying to Translate it to Haskell

We're going to run into a problem with this line.

```
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

```
1 array = [1,2,3]
2 print(array)
3 array.append(4)
4 print(array)
```

The append function *mutates* `array`: after calling append, the value of the variable `array` changes.
`array` has different values before and after line 3.

Trying to Translate it to Haskell

We're going to run into a problem with this line.

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

```python
1 array = [1,2,3]
2 print(array)
3 array.append(4)
4 print(array)
```

The append function *mutates* `array`: after calling append, the value of the variable `array` changes.
`array` has different values before and after line 3.

We can't do that in an immutable language! A variable's value cannot change from one line to the next in Haskell.

Append in Haskell

Instead of mutating variables, in Haskell when we want to change a data structure we usually write a function which returns a new variable equal to the old data structure with the change applied.

Append in Haskell

Instead of mutating variables, in Haskell when we want to change a data structure we usually write a function which returns a new variable equal to the old data structure with the change applied.

$$append :: \mathrm{Array}\ a \rightarrow a \rightarrow \mathrm{Array}\ a$$

Append in Haskell

Instead of mutating variables, in Haskell when we want to change a data structure we usually write a function which returns a new variable equal to the old data structure with the change applied.

$$\mathit{append} :: \text{Array } a \to a \to \text{Array } a$$

$$\begin{aligned} &\mathit{myArray} = [1, 2, 3] \\ &\mathit{myArray}_2 = \mathit{myArray} \text{ `}\mathit{append}\text{` } 4 \end{aligned}$$

$$\begin{aligned} &\mathit{main} = \mathbf{do} \\ &\quad \mathit{print}\ \mathit{myArray} \\ &\quad \mathit{print}\ \mathit{myArray}_2 \end{aligned}$$

Translating it to Haskell

Let's look at the imperative algorithm, and try to translate it bit-by-bit.

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

First we'll need to write the type signature and skeleton of the Haskell function.
What should the type be?

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$\mathit{createArrayUpTo} :: \mathrm{Int} \rightarrow \mathrm{Array\ Int}$$
$$\mathit{createArrayUpTo}\ n =$$

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$\mathit{createArrayUpTo} :: \mathrm{Int} \rightarrow \mathrm{Array\ Int}$$
$$\mathit{createArrayUpTo}\ n =$$

We tend not to use loops in functional languages, but this loop in particular follows a very common pattern which has a name and function in Haskell.
What is it?

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$\begin{aligned}
&\mathit{createArrayUpTo} :: \mathrm{Int} \to \mathrm{Array\ Int}\\
&\mathit{createArrayUpTo}\ n =\\
&\quad \mathit{foldl}\\
&\qquad\qquad [0 \mathinner{\ldotp\ldotp} n - 1]
\end{aligned}$$

foldl is the function we need.

How would the output have differed if we used *foldr* instead?

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$\mathit{createArrayUpTo} :: \mathsf{Int} \rightarrow \mathsf{Array\ Int}$$
$$\mathit{createArrayUpTo}\ n =$$
$$\quad \mathit{foldl}$$
$$\qquad [0\mathinner{\ldotp\ldotp} n-1]$$

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$createArrayUpTo :: \mathrm{Int} \rightarrow \mathrm{Array\ Int}$
$createArrayUpTo\ n =$
$\quad foldl$

$\qquad emptyArray$
$\qquad [0\,..\,n-1]$

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$
\begin{aligned}
&\mathit{createArrayUpTo} :: \mathsf{Int} \rightarrow \mathsf{Array\ Int} \\
&\mathit{createArrayUpTo}\ n = \\
&\quad \mathit{foldl} \\
&\qquad\quad \mathit{emptyArray} \\
&\qquad\quad [0 \mathinner{..} n - 1]
\end{aligned}
$$

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$
\begin{aligned}
&\mathit{createArrayUpTo} :: \mathrm{Int} \to \mathrm{Array\ Int}\\
&\mathit{createArrayUpTo}\ n =\\
&\quad \mathit{foldl}\\
&\qquad (\lambda \mathit{array}\ i \to \mathit{append}\ \mathit{array}\ i)\\
&\qquad \mathit{emptyArray}\\
&\qquad [0\mathinner{\ldotp\ldotp} n - 1]
\end{aligned}
$$

Is there a shorter way to write this, that doesn't include a lambda?

Translating it to Haskell

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$
\begin{aligned}
&\mathit{createArrayUpTo} :: \mathsf{Int} \to \mathsf{Array\ Int}\\
&\mathit{createArrayUpTo}\ n =\\
&\quad \mathit{foldl}\\
&\qquad (\lambda \mathit{array}\ i \to \mathit{append}\ \mathit{array}\ i)\\
&\qquad \mathit{emptyArray}\\
&\qquad [0 \mathinner{..} n - 1]
\end{aligned}
$$

$$\mathcal{O}(n)$$

$$\mathcal{O}(n^2)$$

Why the performance difference?

Why the performance difference?

Why the performance difference?

It comes down to the different complexities of *append*.

Why the performance difference?

It comes down to the different complexities of *append*.

Python	Haskell
$\mathcal{O}(1)$	$\mathcal{O}(n)$

Why the performance difference?

It comes down to the different complexities of *append*.

Python	Haskell
$\mathcal{O}(1)$	$\mathcal{O}(n)$

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$
\begin{aligned}
&\mathit{createArrayUpTo} :: \mathsf{Int} \rightarrow \mathsf{Array\ Int} \\
&\mathit{createArrayUpTo}\ n = \\
&\quad \mathit{foldl} \\
&\qquad (\lambda \mathit{array}\ i \rightarrow \mathit{append}\ \mathit{array}\ i) \\
&\qquad \mathit{emptyArray} \\
&\qquad [0\,..\,n-1]
\end{aligned}
$$

Why the performance difference?

It comes down to the different complexities of *append*.

Python	Haskell
$\mathcal{O}(1)$	$\mathcal{O}(n)$

```python
def create_array_up_to(n):
    array = []
    for i in range(n):
        array.append(i)
    return array
```

$$
\begin{array}{l}
\mathit{createArrayUpTo} :: \mathsf{Int} \rightarrow \mathsf{Array\ Int} \\
\mathit{createArrayUpTo}\ n = \\
\quad \mathit{foldl} \\
\qquad (\lambda \mathit{array}\ i \rightarrow \mathit{append}\ \mathit{array}\ i) \\
\qquad \mathit{emptyArray} \\
\qquad [0 \mathinner{\ldotp\ldotp} n - 1]
\end{array}
$$

Both implementations call *append* n times, which causes the difference in asymptotics.

Why is the imperative version so much more efficient? Why is `append` $\mathcal{O}(1)$?

Forgetful Imperative Languages

Why is the imperative version so much more efficient? Why is `append` $\mathcal{O}(1)$?

```
array = [1,2,3]
print(array)
array.append(4)
print(array)
```

Why is the imperative version so much more efficient? Why is `append` $\mathcal{O}(1)$?

To run this code efficiently, most imperative interpreters will look for the space next to 3 in memory, and put 4 there: an $\mathcal{O}(1)$ operation.

```
array = [1,2,3]
print(array)
array.append(4)
print(array)
```

Forgetful Imperative Languages

Why is the imperative version so much more efficient? Why is `append` $\mathcal{O}(1)$?

To run this code efficiently, most imperative interpreters will look for the space next to 3 in memory, and put 4 there: an $\mathcal{O}(1)$ operation.

```
array = [1,2,3]
print(array)
array.append(4)
print(array)
```

(Of course, sometimes the "space next to 3" will already be occupied! There are clever algorithms you can use to handle this case.)

Forgetful Imperative Languages

Why is the imperative version so much more efficient? Why is `append` $\mathcal{O}(1)$?

To run this code efficiently, most imperative interpreters will look for the space next to 3 in memory, and put 4 there: an $\mathcal{O}(1)$ operation.

```python
1 array = [1,2,3]
2 print(array)
3 array.append(4)
4 print(array)
```

Semantically, in an imperative language we are allowed to "forget" the contents of `array` on line 1: `[1,2,3]`. That array has been irreversibly replaced by `[1,2,3,4]`.

Haskell doesn't Forget

The Haskell version of append looks similar at first glance:

$$
\begin{aligned}
&\mathit{myArray} = [1, 2, 3] \\
&\mathit{myArray}_2 = \mathit{myArray} \text{ `}\mathit{append}\text{` } 4
\end{aligned}
$$

Haskell doesn't Forget

The Haskell version of append looks similar at first glance:

$$myArray = [1, 2, 3]$$
$$myArray_2 = myArray \text{ `}append\text{` } 4$$

But we can't edit the array $[1, 2, 3]$ in memory, because *myArray* still exists!

Haskell doesn't Forget

The Haskell version of append looks similar at first glance:

$$myArray = [1, 2, 3]$$
$$myArray_2 = myArray \text{ `}append\text{` } 4$$

But we can't edit the array $[1, 2, 3]$ in memory, because *myArray* still exists!

$$\begin{aligned} &main = \textbf{do} \\ &\quad print\ myArray \\ &\quad print\ myArray_2 \end{aligned}$$

Haskell doesn't Forget

The Haskell version of append looks similar at first glance:

$$\begin{aligned} &\mathit{myArray} = [1, 2, 3] \\ &\mathit{myArray}_2 = \mathit{myArray} \text{ `}\mathit{append}\text{` } 4 \end{aligned}$$

But we can't edit the array $[1, 2, 3]$ in memory, because *myArray* still exists!

$$\begin{aligned} &\mathit{main} = \textbf{do} \\ &\quad \mathit{print}\ \mathit{myArray} \\ &\quad \mathit{print}\ \mathit{myArray}_2 \end{aligned}$$

```
>>> main
[1,2,3]
[1,2,3,4]
```

Haskell doesn't Forget

The Haskell version of append looks similar at first glance:

$$myArray = [1, 2, 3]$$
$$myArray_2 = myArray \text{ `}append\text{` } 4$$

But we can't edit the array $[1, 2, 3]$ in memory, because *myArray* still exists!

$$\begin{aligned} &main = \textbf{do} \\ &\quad print\ myArray \\ &\quad print\ myArray_2 \end{aligned}$$

```
>>> main
[1,2,3]
[1,2,3,4]
```

As a result, our only option is to copy, which is $\mathcal{O}(n)$.

The Problem

In immutable languages, old versions of data structures have to be kept around in case they're looked at.

The Problem

In immutable languages, old versions of data structures have to be kept around in case they're looked at.

For arrays, this means we have to copy on every mutation. (i.e.: `append` is $\mathcal{O}(n)$)

The Problem

In immutable languages, old versions of data structures have to be kept around in case they're looked at.

For arrays, this means we have to copy on every mutation. (i.e.: `append` is $\mathcal{O}(n)$)

Solutions?

The Problem

In immutable languages, old versions of data structures have to be kept around in case they're looked at.

For arrays, this means we have to copy on every mutation. (i.e.: `append` is $\mathcal{O}(n)$)

Solutions?

1. Find a way to disallow access of old versions of data structures.

This approach is beyond the scope of this lecture!
However, for interested students: linear type systems can enforce this property. You may have heard of Rust, a programming language with linear types.

The Problem

In immutable languages, old versions of data structures have to be kept around in case they're looked at.

For arrays, this means we have to copy on every mutation. (i.e.: `append` is $\mathcal{O}(n)$)

Solutions?

1. Find a way to disallow access of old versions of data structures.
2. Find a way to implement data structures that keep their old versions efficiently.

This is the approach we're going to look at today.

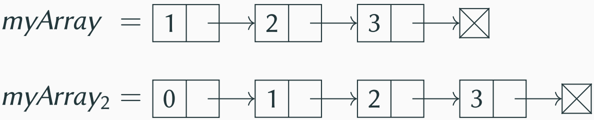
Keeping History Efficiently

Consider the linked list.

$myArray =$ 

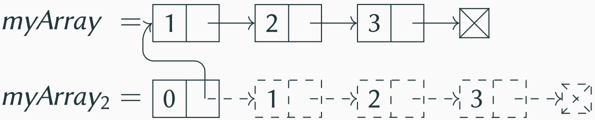
Keeping History Efficiently

To “prepend” an element (i.e. append to front), you might assume we would have to copy again:



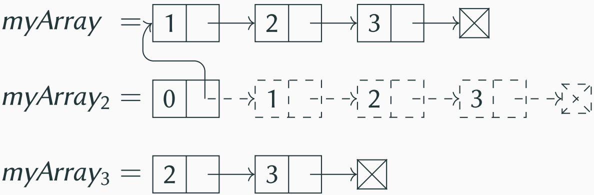
Keeping History Efficiently

However, this is not the case.

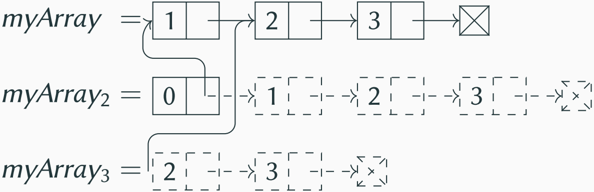


Keeping History Efficiently

The same trick also works with deletion.



Keeping History Efficiently



Persistent Data Structures

Persistent Data Structure

A persistent data structure is a data structure which preserves all versions of itself after modification.

Persistent Data Structures

Persistent Data Structure

A persistent data structure is a data structure which preserves all versions of itself after modification.

An array is "persistent" in some sense, if all operations are implemented by copying. It just isn't very *efficient*.

Persistent Data Structures

Persistent Data Structure

A persistent data structure is a data structure which preserves all versions of itself after modification.

An array is "persistent" in some sense, if all operations are implemented by copying. It just isn't very *efficient*.

A linked list is much better: it can do persistent *cons* and *uncons* in $\mathcal{O}(1)$ time.

Persistent Data Structures

Persistent Data Structure

A persistent data structure is a data structure which preserves all versions of itself after modification.

An array is "persistent" in some sense, if all operations are implemented by copying. It just isn't very *efficient*.

A linked list is much better: it can do persistent *cons* and *uncons* in $\mathcal{O}(1)$ time.

Immutability

While the semantics of languages like Haskell necessitate this property, they also *facilitate* it.

After several additions and deletions onto some linked structure we will be left with a real rat's nest of pointers and references: strong guarantees that no-one will mutate anything is essential for that mess to be manageable.

?

As it happens, all of you have
already been using a persistent data structure!

As it happens, all of you have
already been using a persistent data structure!

Git is perhaps the most widely-used
persistent data structure in the world.

As it happens, all of you have
already been using a persistent data structure!

Git is perhaps the most widely-used
persistent data structure in the world.

It works like a persistent file system: when you
make a change to a file, git *remembers* the old version, instead of
deleting it!

Git

As it happens, all of you have already been using a persistent data structure!

Git is perhaps the most widely-used persistent data structure in the world.

It works like a persistent file system: when you make a change to a file, git *remembers* the old version, instead of deleting it!

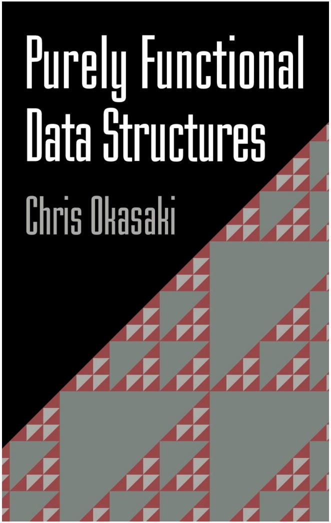
To do this efficiently it doesn't just store a new copy of the repository whenever a change is made, it instead uses some of the tricks and techniques we're going to look at in the rest of this talk.

The Book

Chris Okasaki. *Purely Functional Data Structures.* **Cambridge University Press, June 1999**

Much of the material in this lecture comes directly from this book.

It's also on your reading list for your algorithms course next year.



Arrays

While our linked list can replace a normal array for some applications, in general it's missing some of the key operations we might want.

Indexing in particular is $\mathcal{O}(n)$ on a linked list but $\mathcal{O}(1)$ on an array.

We're going to build a data structure which gets to $\mathcal{O}(\log n)$ indexing in a pure way.

Implementing a Functional Algorithm: Merge Sort

Merge Sort

Merge sort is a classic divide-and-conquer algorithm.

It divides up a list into singleton lists, and then repeatedly merges adjacent sublists until only one is left.

Visualisation of Merge Sort

2	6	10	7	8	1	9	3	4	5

Visualisation of Merge Sort

2	6	10	7	8	1	9	3	4	5

$\boxed{2}$ $\boxed{6}$ $\boxed{10}$ $\boxed{7}$ $\boxed{8}$ $\boxed{1}$ $\boxed{9}$ $\boxed{3}$ $\boxed{4}$ $\boxed{5}$

Visualisation of Merge Sort

2	6	10	7	8	1	9	3	4	5

Visualisation of Merge Sort

2	6	10	7	8	1	9	3	4	5

2	6	7	10	1	8	3	9	4	5

Visualisation of Merge Sort

2	6

7	10

1	8

3	9

4	5

Visualisation of Merge Sort

2	6

7	10

1	8

3	9

4	5

2	6	7	10

1	3	8	9

4	5

Visualisation of Merge Sort

2	6	7	10

1	3	8	9

4	5

Visualisation of Merge Sort

2	6	7	10

1	3	8	9

4	5

1	2	3	6	7	8	9	10

4	5

Visualisation of Merge Sort

1	2	3	6	7	8	9	10

4	5

Visualisation of Merge Sort

1	2	3	6	7	8	9	10

4	5

1	2	3	4	5	6	7	8	9	10

Visualisation of Merge Sort

1	2	3	4	5	6	7	8	9	10

Visualisation of Merge Sort

2 6 10 7 8 1 9 3 4 5

2   6   10   7   8   1   9   3   4   5

2 6   7 10   1 8   3 9   4 5

2 6 7 10   1 3 8 9   4 5

1 2 3 6 7 8 9 10   4 5

1 2 3 4 5 6 7 8 9 10

Just to demonstrate some of the complexity of the algorithm when implemented imperatively, here it is in Python.

Just to demonstrate some of the complexity of the algorithm when implemented imperatively, here it is in Python.

You do not need to understand the following slide!

```python
def merge_sort(arr):
  lsz, tsz, acc = 1, len(arr), []
  while lsz < tsz:
    for ll in range(0, tsz-lsz, lsz*2):
      lu, rl, ru = ll+lsz, ll+lsz, min(tsz, ll+lsz*2)
      while ll < lu and rl < ru:
        if arr[ll] <= arr[rl]:
          acc.append(arr[ll])
          ll += 1
        else:
          acc.append(arr[rl])
          rl += 1
      acc += arr[ll:lu] + arr[rl:ru]
    acc += arr[len(acc):]
    arr, lsz, acc = acc, lsz*2, []
  return arr
```

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.
- We will do away with index arithmetic, instead using pattern-matching.

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.
- We will do away with index arithmetic, instead using pattern-matching.
- We will avoid complex `while` conditions.

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.
- We will do away with index arithmetic, instead using pattern-matching.
- We will avoid complex `while` conditions.
- We won't mutate anything.

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.
- We will do away with index arithmetic, instead using pattern-matching.
- We will avoid complex `while` conditions.
- We won't mutate anything.
- We will add a healthy sprinkle of types.

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.
- We will do away with index arithmetic, instead using pattern-matching.
- We will avoid complex `while` conditions.
- We won't mutate anything.
- We will add a healthy sprinkle of types.

How can we improve it?

Merge sort is actually an algorithm perfectly suited to a functional implementation.

In translating it over to Haskell, we are going to make the following improvements:

- We will abstract out some patterns, like the fold pattern.
- We will do away with index arithmetic, instead using pattern-matching.
- We will avoid complex `while` conditions.
- We won't mutate anything.
- We will add a healthy sprinkle of types.

Granted, all of these improvements could have been made to the Python code, too.

Merge in Haskell

We'll start with a function that merges two sorted lists.

Merge in Haskell

We'll start with a function that merges two sorted lists.

$$
\begin{aligned}
&\mathit{merge} :: \mathrm{Ord}\ a \Rightarrow [a] \to [a] \to [a] \\
&\mathit{merge}\ [\,]\ \ \mathit{ys} = \mathit{ys} \\
&\mathit{merge}\ \mathit{xs}\ [\,] = \mathit{xs} \\
&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) \\
&\quad \mid x \leqslant y \qquad\ \ = x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys}) \\
&\quad \mid \mathit{otherwise} = y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

Merge in Haskell

We'll start with a function that merges two sorted lists.

$$
\begin{aligned}
&\mathit{merge} :: \mathrm{Ord}\ a \Rightarrow [a] \to [a] \to [a]\\
&\mathit{merge}\ [\,]\ \ \mathit{ys} = \mathit{ys}\\
&\mathit{merge}\ \mathit{xs}\ [\,] = \mathit{xs}\\
&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys})\\
&\qquad \mid x \leqslant y \qquad\ \ = x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys})\\
&\qquad \mid \mathit{otherwise} = y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

```
>>> merge [1,8] [3,9]
[1,3,8,9]
```

Using the Merge to Sort

Next: how do we use this merge to sort a list?

Using the Merge to Sort

Next: how do we use this merge to sort a list?

We know how to combine 2 sorted lists, and that combine function has an *identity*, so how do we use it to combine n sorted lists?

$$merge\ xs\ [] = xs$$

Using the Merge to Sort

Next: how do we use this merge to sort a list?

We know how to combine 2 sorted lists, and that combine function has an *identity*, so how do we use it to combine *n* sorted lists?

$$merge\ xs\ [\,] = xs$$

foldr?

The Problem with *foldr*

$$sort :: \mathrm{Ord}\ a \Rightarrow [a] \to [a]$$
$$sort\ xs = foldr\ merge\ [\,]\ [[x] \mid x \leftarrow xs]$$

The Problem with *foldr*

$$\begin{aligned}&\mathit{sort} :: \mathrm{Ord}\ a \Rightarrow [a] \to [a]\\&\mathit{sort}\ \mathit{xs} = \mathit{foldr}\ \mathit{merge}\ []\ [[x] \mid x \leftarrow \mathit{xs}]\end{aligned}$$

Unfortunately, this is actually insertion sort!

The Problem with *foldr*

$$\begin{aligned} &\mathit{sort} :: \mathrm{Ord}\ a \Rightarrow [a] \rightarrow [a] \\ &\mathit{sort}\ \mathit{xs} = \mathit{foldr}\ \mathit{merge}\ [\,]\ [[x] \mid x \leftarrow \mathit{xs}] \end{aligned}$$

Unfortunately, this is actually insertion sort!

$$\mathit{merge}\ [x]\ \mathit{ys} = \mathit{insert}\ x\ \mathit{ys}$$

The Problem with *foldr*

$$\begin{aligned}&sort :: \mathrm{Ord}\ a \Rightarrow [a] \rightarrow [a]\\&sort\ xs = foldr\ merge\ [\,]\ [[x] \mid x \leftarrow xs]\end{aligned}$$

Unfortunately, this is actually insertion sort!

$$merge\ [x]\ ys = insert\ x\ ys$$

The problem is that *foldr* is too unbalanced.

$$\begin{aligned}&foldr\ (\oplus)\ \emptyset\ [1\,..\,5] =\\&\quad 1 \oplus (2 \oplus (3 \oplus (4 \oplus (5 \oplus \emptyset))))\end{aligned}$$

The Problem with *foldr*

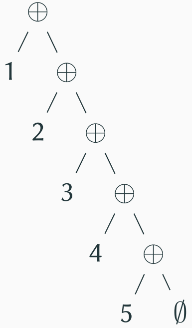
$$sort :: \mathrm{Ord}\ a \Rightarrow [a] \rightarrow [a]$$
$$sort\ xs = foldr\ merge\ [\,]\ [[x] \mid x \leftarrow xs]$$

Unfortunately, this is actually insertion sort!

$$merge\ [x]\ ys = insert\ x\ ys$$

The problem is that *foldr* is too unbalanced.

$$foldr\ (\oplus)\ \emptyset\ [1\,..\,5] =$$
$$1 \oplus (2 \oplus (3 \oplus (4 \oplus (5 \oplus \emptyset))))$$



The Problem with *foldr*

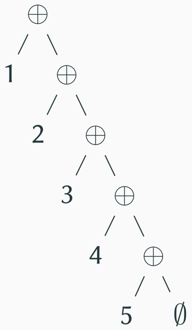
$$\mathit{sort} :: \mathrm{Ord}\ a \Rightarrow [a] \to [a]$$
$$\mathit{sort}\ \mathit{xs} = \mathit{foldr}\ \mathit{merge}\ [\,]\ [[x] \mid x \leftarrow \mathit{xs}]$$

Unfortunately, this is actually insertion sort!

$$\mathit{merge}\ [x]\ \mathit{ys} = \mathit{insert}\ x\ \mathit{ys}$$

The problem is that *foldr* is too unbalanced.

$$\mathit{foldr}\ (\oplus)\ \emptyset\ [1\,..\,5] =$$
$$1 \oplus (2 \oplus (3 \oplus (4 \oplus (5 \oplus \emptyset))))$$



Merge sort crucially divides the work in a balanced way!

Visualisation of Merge Sort

2	6	10	7	8	1	9	3	4	5

[2] [6] [10] [7] [8] [1] [9] [3] [4] [5]

[2 | 6] [7 | 10] [1 | 8] [3 | 9] [4 | 5]

[2 | 6 | 7 | 10] [1 | 3 | 8 | 9] [4 | 5]

[1 | 2 | 3 | 6 | 7 | 8 | 9 | 10] [4 | 5]

1	2	3	4	5	6	7	8	9	10

A More Balanced Fold

A More Balanced Fold

$$
\begin{aligned}
&\mathit{treeFold} :: (a \to a \to a) \to [a] \to a \\
&\mathit{treeFold}\ (\oplus)\ [x] = x \\
&\mathit{treeFold}\ (\oplus)\ \mathit{xs}\ \ = \mathit{treeFold}\ (\oplus)\ (\mathit{pairMap}\ \mathit{xs}) \\
&\quad \textbf{where} \\
&\qquad \mathit{pairMap}\ (x_1 : x_2 : \mathit{xs}) = x_1 \oplus x_2 : \mathit{pairMap}\ \mathit{xs} \\
&\qquad \mathit{pairMap}\ \mathit{xs} \qquad\qquad\quad\ \ = \mathit{xs}
\end{aligned}
$$

A More Balanced Fold

$$
\begin{aligned}
&\mathit{treeFold} :: (a \to a \to a) \to [a] \to a \\
&\mathit{treeFold}\ (\oplus)\ [x] = x \\
&\mathit{treeFold}\ (\oplus)\ \mathit{xs}\ \ = \mathit{treeFold}\ (\oplus)\ (\mathit{pairMap}\ \mathit{xs}) \\
&\quad \textbf{where} \\
&\qquad \mathit{pairMap}\ (x_1 : x_2 : \mathit{xs}) = x_1 \oplus x_2 : \mathit{pairMap}\ \mathit{xs} \\
&\qquad \mathit{pairMap}\ \mathit{xs} \qquad\qquad\quad\ = \mathit{xs}
\end{aligned}
$$

This can be used quite similarly to how you might use *foldl* or *foldr*:

$$
\mathit{sum} = \mathit{treeFold}\ (+)
$$

A More Balanced Fold

$$
\begin{aligned}
&\mathit{treeFold} :: (a \to a \to a) \to [a] \to a \\
&\mathit{treeFold}\ (\oplus)\ [x] = x \\
&\mathit{treeFold}\ (\oplus)\ xs\ \ = \mathit{treeFold}\ (\oplus)\ (\mathit{pairMap}\ xs) \\
&\quad \mathbf{where} \\
&\qquad \mathit{pairMap}\ (x_1 : x_2 : xs) = x_1 \oplus x_2 : \mathit{pairMap}\ xs \\
&\qquad \mathit{pairMap}\ xs \qquad\qquad\quad\ = xs
\end{aligned}
$$

This can be used quite similarly to how you might use *foldl* or *foldr*:

$$\mathit{sum} = \mathit{treeFold}\ (+)$$

(although we would probably change the definition a little to catch the empty list, but we won't look at that here)

A More Balanced Fold

$$
\begin{aligned}
&\mathit{treeFold} :: (a \to a \to a) \to [a] \to a \\
&\mathit{treeFold}\ (\oplus)\ [x] = x \\
&\mathit{treeFold}\ (\oplus)\ \mathit{xs}\ \ = \mathit{treeFold}\ (\oplus)\ (\mathit{pairMap}\ \mathit{xs}) \\
&\quad \textbf{where} \\
&\qquad \mathit{pairMap}\ (x_1 : x_2 : \mathit{xs}) = x_1 \oplus x_2 : \mathit{pairMap}\ \mathit{xs} \\
&\qquad \mathit{pairMap}\ \mathit{xs} \qquad\qquad\quad\ = \mathit{xs}
\end{aligned}
$$

This can be used quite similarly to how you might use *foldl* or *foldr*:

$$\mathit{sum} = \mathit{treeFold}\ (+)$$

(although we would probably change the definition a little to catch the empty list, but we won't look at that here)

The fundamental difference between this fold and, say, *foldr* is that it's *balanced*, which is extremely important for merge sort.

Visualisation of *treeFold*

$$\textit{treeFold}\ (\oplus)\ [1\mathinner{..}10] =$$
$$\quad \textit{treeFold}\ (\oplus)\ [1,2,3,4,5,6,7,8,9,10]$$

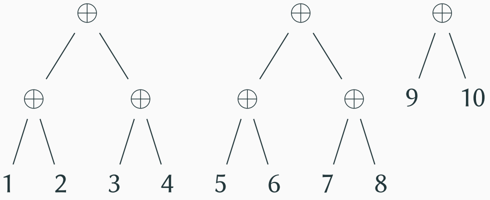
1 2 3 4 5 6 7 8 9 10

Visualisation of *treeFold*

$$\begin{aligned}&\mathit{treeFold}\ (\oplus)\ [1\mathinner{\ldotp\ldotp}10] = \\ &\quad \mathit{treeFold}\ (\oplus)\ [1 \oplus 2, 3 \oplus 4, 5 \oplus 6, 7 \oplus 8, 9 \oplus 10]\end{aligned}$$

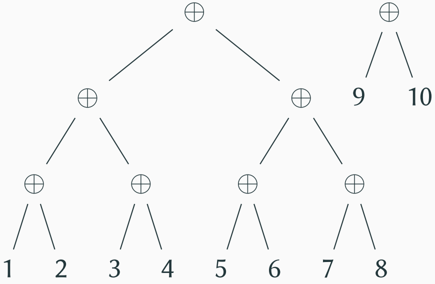
Visualisation of *treeFold*

$$\begin{aligned}&\mathit{treeFold}\ (\oplus)\ [1\mathinner{..}10] = \\ &\quad \mathit{treeFold}\ (\oplus)\ [(1 \oplus 2) \oplus (3 \oplus 4), (5 \oplus 6) \oplus (7 \oplus 8), 9 \oplus 10]\end{aligned}$$



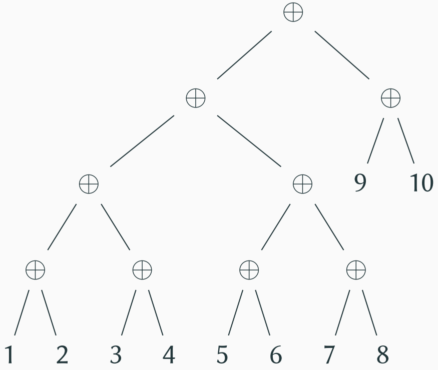
Visualisation of *treeFold*

$$\begin{aligned}&\mathit{treeFold}\ (\oplus)\ [1\,..\,10] = \\ &\quad \mathit{treeFold}\ (\oplus)\ [((1 \oplus 2) \oplus (3 \oplus 4)) \oplus ((5 \oplus 6) \oplus (7 \oplus 8)), 9 \oplus 10]\end{aligned}$$



Visualisation of *treeFold*

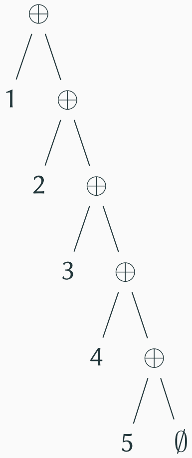
$$\mathit{treeFold}\ (\oplus)\ [1\,..\,10] =$$
$$(((1 \oplus 2) \oplus (3 \oplus 4)) \oplus ((5 \oplus 6) \oplus (7 \oplus 8))) \oplus (9 \oplus 10)$$



Visualisation of *foldr*

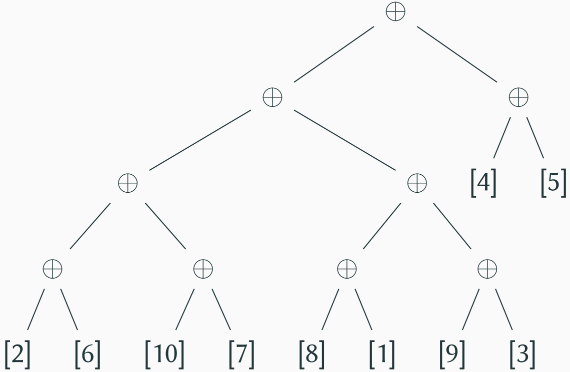
Compare to *foldr*:

$$
\begin{aligned}
&\mathit{foldr}\ (\oplus)\ \emptyset\ [1\mathinner{..}5] = \\
&\quad 1 \oplus (2 \oplus (3 \oplus (4 \oplus (5 \oplus \emptyset))))
\end{aligned}
$$



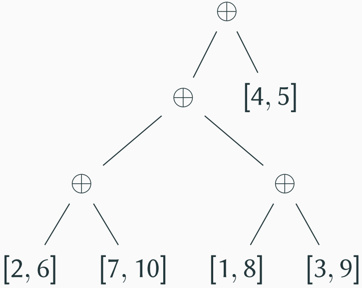
Visualisation of Merge Sort in Haskell

$$\mathit{treeFold}\ \mathit{merge}\ [2, 6, 10, 7, 8, 1, 9, 3, 4, 5] =$$



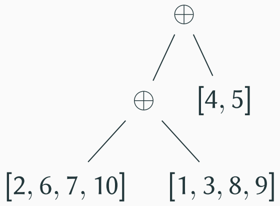
Visualisation of Merge Sort in Haskell

$$\mathit{treeFold}\ \mathit{merge}\ [2, 6, 10, 7, 8, 1, 9, 3, 4, 5] =$$



Visualisation of Merge Sort in Haskell

$$\textit{treeFold merge}\ [2, 6, 10, 7, 8, 1, 9, 3, 4, 5] =$$



Visualisation of Merge Sort in Haskell

$$\mathit{treeFold}\ \mathit{merge}\ [2, 6, 10, 7, 8, 1, 9, 3, 4, 5] =$$

$$\oplus$$

$$[1, 2, 3, 6, 7, 8, 9, 10] \qquad [4, 5]$$

Visualisation of Merge Sort in Haskell

$$\mathit{treeFold}\ \mathit{merge}\ [2,6,10,7,8,1,9,3,4,5] = [1,2,3,4,5,6,7,8,9,10]$$

Sort Algorithm

$$
\begin{aligned}
&\mathit{sort} :: \mathrm{Ord}\ a \Rightarrow [a] \rightarrow [a] \\
&\mathit{sort}\ [\,] = [\,] \\
&\mathit{sort}\ \mathit{xs} = \mathit{treeFold}\ \mathit{merge}\ [[x] \mid x \leftarrow \mathit{xs}]
\end{aligned}
$$

So Why Is This Algorithm Fast?

It's down to the pattern of the fold itself.

Because it splits the input evenly, the full algorithm is $\mathcal{O}(n \log n)$ time.

If we had just used *foldr*, we would have defined insertion sort, which is $\mathcal{O}(n^2)$.

Monoids

Monoids

class Monoid a **where**

$\epsilon :: a$

$(\bullet) :: a \to a \to a$

Monoid

A monoid is a set with a neutral element ϵ, and a binary operator $\bullet$, such that:

$$(x \bullet y) \bullet z = x \bullet (y \bullet z)$$

$$x \bullet \epsilon = x$$

$$\epsilon \bullet x = x$$

Examples of Monoids

- $\mathbb{N}$, under either $+$ or $\times$.
- Lists:

$$
\begin{aligned}
&\textbf{instance}\ \mathrm{Monoid}\ [a]\ \textbf{where}\\
&\quad \epsilon = [\,]\\
&\quad (\bullet) = (+\!\!+)
\end{aligned}
$$

- *Ordered* lists, with *merge*.

Let's Rewrite *treeFold* to use Monoids

$$
\begin{aligned}
&\mathit{treeFold} :: \mathsf{Monoid}\ a \Rightarrow [a] \rightarrow a \\
&\mathit{treeFold}\ [\,] \quad = \epsilon \\
&\mathit{treeFold}\ [x] = x \\
&\mathit{treeFold}\ \mathit{xs} \quad = \mathit{treeFold}\ (\mathit{pairMap}\ \mathit{xs}) \\
&\quad \mathbf{where} \\
&\qquad \mathit{pairMap}\ (x_1 : x_2 : \mathit{xs}) = (x_1 \bullet x_2) : \mathit{pairMap}\ \mathit{xs} \\
&\qquad \mathit{pairMap}\ \mathit{xs} = \mathit{xs}
\end{aligned}
$$

We can actually prove that this version returns the same results as *foldr*, as long as the monoid laws are followed.

It just performs the fold in a more efficient way.

We've already seen one monoid we can use this fold with: ordered lists.

Another is floating-point numbers under summation. Using *foldr* or *foldl* will give you $\mathcal{O}(n)$ error growth, whereas using *treeFold* will give you $\mathcal{O}(\log n)$.

Let’s Make It Incremental

treeFold currently processes the input in one big operation.

However, if we were able to process the input incrementally, with useful intermediate results, there are some other applications we can use the fold for.

A Binary Data Structure

We’re going to build a data structure based on the binary numbers.

A Binary Data Structure

We're going to build a data structure based on the binary numbers.

For, say, 10 elements, we have the following binary number:

```
I O I O
```

A Binary Data Structure

We're going to build a data structure based on the binary numbers.

For, say, 10 elements, we have the following binary number:

$$\mathtt{I}_8\mathtt{O}_4\mathtt{I}_2\mathtt{O}_1$$

(With each bit annotated with its significance)

A Binary Data Structure

We're going to build a data structure based on the binary numbers.

For, say, 10 elements, we have the following binary number:

$$\mathtt{I}_8\mathtt{O}_4\mathtt{I}_2\mathtt{O}_1$$

This number tells us how to arrange 10 elements into perfect trees.

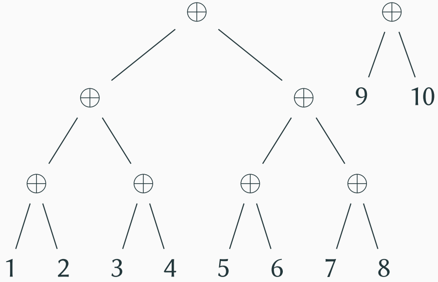
A Binary Data Structure

We're going to build a data structure based on the binary numbers.

For, say, 10 elements, we have the following binary number:

$$\mathtt{I}_8\mathtt{O}_4\mathtt{I}_2\mathtt{O}_1$$

This number tells us how to arrange 10 elements into perfect trees.



The Incremental Type

We can write this as a datatype:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Incremental}\ a = [(\mathsf{Int}, a)] \\
&\mathit{cons} :: (a \to a \to a) \to a \to \mathsf{Incremental}\ a \to \mathsf{Incremental}\ a \\
&\mathit{cons}\ f = \mathit{go}\ 0 \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ i\ x\ [\,] = [(i, x)] \\
&\qquad \mathit{go}\ i\ x\ ((0, y) : \mathit{ys}) = (i + 1, f\ x\ y) : \mathit{ys} \\
&\qquad \mathit{go}\ i\ x\ ((j, y) : \mathit{ys}) = (i, x) : (j - 1, y) : \mathit{ys} \\
&\mathit{run} :: (a \to a \to a) \to \mathsf{Incremental}\ a \to a \\
&\mathit{run}\ f = \mathit{foldr1}\ f \circ \mathit{map}\ \mathit{snd}
\end{aligned}
$$

And we can even implement *treeFold* using it:

$$
\begin{aligned}
&\mathit{treeFold} :: (a \to a \to a) \to [a] \to a \\
&\mathit{treeFold}\ f = \mathit{run}\ f \circ \mathit{foldr}\ (\mathit{cons}\ f)\ [\,]
\end{aligned}
$$

We can now use the function incrementally.

$$treeScanl\ f = map\ (run\ f) \circ tail \circ scanl\ (flip\ (cons\ f))\ [\,]$$
$$treeScanr\ f = map\ (run\ f) \circ init \circ scanr\ (cons\ f)\ [\,]$$

We can now use the function incrementally.

$$\begin{aligned}
treeScanl\ f &= map\ (run\ f) \circ tail \circ scanl\ (flip\ (cons\ f))\ [\,] \\
treeScanr\ f &= map\ (run\ f) \circ init \circ scanr\ (cons\ f)\ [\,]
\end{aligned}$$

We could, for instance, sort all of the tails of a list efficiently in this way. (although I'm not sure why you'd want to!)

$$\begin{aligned}
&treeScanr\ merge \\
&\quad (map\ pure\ [2,6,1,3,4,5]) \equiv \\
&\qquad [[1,2,3,4,5,6] \\
&\qquad ,[1,3,4,5,6] \\
&\qquad ,[1,3,4,5] \\
&\qquad ,[3,4,5] \\
&\qquad ,[4,5] \\
&\qquad ,[5]]
\end{aligned}$$

We can now use the function incrementally.

$$
\begin{aligned}
\mathit{treeScanl}\ f &= \mathit{map}\ (\mathit{run}\ f) \circ \mathit{tail} \circ \mathit{scanl}\ (\mathit{flip}\ (\mathit{cons}\ f))\ [\,] \\
\mathit{treeScanr}\ f &= \mathit{map}\ (\mathit{run}\ f) \circ \mathit{init} \circ \mathit{scanr}\ (\mathit{cons}\ f)\ [\,]
\end{aligned}
$$

We could, for instance, sort all of the tails of a list efficiently in this way. (although I'm not sure why you'd want to!)

$$
\begin{aligned}
&\mathit{treeScanr}\ \mathit{merge} \\
&\quad (\mathit{map}\ \mathit{pure}\ [2,6,1,3,4,5]) \equiv \\
&\qquad [[1,2,3,4,5,6] \\
&\qquad ,[1,3,4,5,6] \\
&\qquad ,[1,3,4,5] \\
&\qquad ,[3,4,5] \\
&\qquad ,[4,5] \\
&\qquad ,[5]]
\end{aligned}
$$

A more practical use is to extract the *k* smallest elements from a list, which can be achieved with a variant on this fold.

But, as we saw already, the only required element here is the *Monoid*.

If we remember back to the $(\mathbb{N}, 0, +)$ monoid, we can build now a collection which tracks the number of elements it has.

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Tree}\ a \\
&\quad = \mathsf{Leaf}\ \{ \mathit{size} :: \mathsf{Int}, \mathit{val} :: a \} \\
&\quad \mid \mathsf{Node}\ \{ \mathit{size} :: \mathsf{Int}, \mathit{lchild} :: \mathsf{Tree}\ a, \mathit{rchild} :: \mathsf{Tree}\ a \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{leaf} :: a \to \mathsf{Tree}\ a \\
&\mathit{leaf}\ x = \mathsf{Leaf}\ 1\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{node} :: \mathsf{Tree}\ a \to \mathsf{Tree}\ a \to \mathsf{Tree}\ a \\
&\mathit{node}\ \mathit{xs}\ \mathit{ys} = \mathsf{Node}\ (\mathit{size}\ \mathit{xs} + \mathit{size}\ \mathit{ys})\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

Not so useful, no, but remember that we have a way to build this type *incrementally*, in a *balanced* way.

$$\textbf{type}\ \mathrm{Array}\ a = \mathrm{Incremental}\ (\mathrm{Tree}\ a)$$

Insertion is $\mathcal{O}(\log n)$:

$$
\begin{aligned}
&\mathit{insert} :: a \to \mathrm{Array}\ a \to \mathrm{Array}\ a\\
&\mathit{insert}\ x = \mathit{cons}\ \mathit{node}\ (\mathit{leaf}\ x)\\
&\mathit{fromList} :: [a] \to \mathrm{Array}\ a\\
&\mathit{fromList} = \mathit{foldr}\ \mathit{insert}\ [\,]
\end{aligned}
$$

And finally lookup, the key feature missing from our persistent implementation of arrays, is *also* $\mathcal{O}(\log n)$:

$$
\begin{aligned}
&\mathit{lookupTree} :: \mathsf{Int} \to \mathsf{Tree}\ a \to a \\
&\mathit{lookupTree}\ \_\ (\mathsf{Leaf}\ \_\ x) = x \\
&\mathit{lookupTree}\ i\ (\mathsf{Node}\ \_\ \mathit{xs}\ \mathit{ys}) \\
&\quad \mid i < \mathit{size}\ \mathit{xs} = \mathit{lookupTree}\ i\ \mathit{xs} \\
&\quad \mid \mathit{otherwise} = \mathit{lookupTree}\ (i - \mathit{size}\ \mathit{xs})\ \mathit{ys} \\
&\mathit{lookup} :: \mathsf{Int} \to \mathsf{Array}\ a \to \mathsf{Maybe}\ a \\
&\mathit{lookup} = \mathit{flip}\ (\mathit{foldr}\ f\ b) \\
&\quad \mathbf{where} \\
&\quad b\ \_ = \mathsf{Nothing} \\
&\quad f\ (\_, x)\ \mathit{xs}\ i \\
&\qquad \mid i < \mathit{size}\ x = \mathsf{Just}\ (\mathit{lookupTree}\ i\ x) \\
&\qquad \mid \mathit{otherwise} = \mathit{xs}\ (i - \mathit{size}\ x)
\end{aligned}
$$

Finger Trees

So we have seen a number of techniques today:

- Using pointers and sharing to make a data structure persistent.
- Using monoids to describe folding operations.
- Using *balanced* folding operations to take an $\mathcal{O}(n)$ operation to a $\mathcal{O}(\log n)$ one. (in terms of time and other things like error growth)
- Using a number-based data structure to incrementalise some of those folds.
- Using that incremental structure to implement things like lookup.

There is a single data structure which does pretty much all of this, and more: the Finger Tree.

Finger Trees

Ralf Hinze and Ross Paterson. Finger Trees: A Simple General-purpose Data Structure.

***Journal of Functional Programming*, 16(2):197–217, 2006**

A monoid-based tree-like structure, much like our "Incremental" type.

However, much more general.

Supports insertion, deletion, but also *concatenation*.

Also our lookup function is more generally described by the "split" operation.

All based around some monoid.

Uses for Finger Trees

Just by switching out the monoid for something else we can get an almost entirely different data structure.

- Priority Queues
- Search Trees
- Priority Search Queues (think: Dijkstra's Algorithm)
- Prefix Sum Trees
- Array-like random-access lists: this is precisely what's done in Haskell's Data.Sequence.