Hylomorphisms

Or: intermediate structures in a non-awful way

Koz Ross

14th September, 2017

[AUT CSC]

AUT Computer Science Club

Outline

Intermediate structures

Intermediate structures

Definition

We say that a data structure is *intermediate* if it is used in an algorithm, but not given as an input or returned as an output.

Intermediate structures

Definition

We say that a data structure is *intermediate* if it is used in an algorithm, but not given as an input or returned as an output.

Essentially, it's a data structure which we *use*, but never *see*.

An example

Definition

The *Fibonacci numbers* $(F_0, F_1, \ldots)$ are a sequence of natural numbers, such that:

An example

Definition

The *Fibonacci numbers* $(F_0, F_1, \ldots)$ are a sequence of natural numbers, such that:

- $F_0 = F_1 = 1$

An example

Definition

The *Fibonacci numbers* $(F_0, F_1, \ldots)$ are a sequence of natural numbers, such that:

- $F_0 = F_1 = 1$
- For any $n > 1$, $F_n = F_{n-1} + F_{n-2}$

An example

Definition

The *Fibonacci numbers* $(F_0, F_1, \ldots)$ are a sequence of natural numbers, such that:

- $F_0 = F_1 = 1$
- For any $n > 1$, $F_n = F_{n-1} + F_{n-2}$

If we wanted to write an algorithm for computing F_n given an input n, we can convert this definition directly:

An example

Definition

The *Fibonacci numbers* $(F_0, F_1, \ldots)$ are a sequence of natural numbers, such that:

- $F_0 = F_1 = 1$
- For any $n > 1$, $F_n = F_{n-1} + F_{n-2}$

If we wanted to write an algorithm for computing F_n given an input n, we can convert this definition directly:

function fib(n)
    if $n = 0$ or $n = 1$ **then**
        return 1
    else
        return fib($n - 1$) + fib($n - 2$)

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$

Why this is awful

Let's see what happens if we call fib(4):

1. fib$(3) +$ fib(2)
2. fib$(2) +$ fib$(1) +$ fib(2)
3. fib$(1) +$ fib$(0) +$ fib$(1) +$ fib(2)

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$
6. $1 + 1 + 1 + \mathsf{fib}(2)$

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$
6. $1 + 1 + 1 + \mathsf{fib}(2)$
7. $1 + 1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(0)$

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$
6. $1 + 1 + 1 + \mathsf{fib}(2)$
7. $1 + 1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(0)$
8. $1 + 1 + 1 + 1 + \mathsf{fib}(0)$

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$
6. $1 + 1 + 1 + \mathsf{fib}(2)$
7. $1 + 1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(0)$
8. $1 + 1 + 1 + 1 + \mathsf{fib}(0)$
9. $1 + 1 + 1 + 1 + 1$

Why this is awful

Let's see what happens if we call $\text{fib}(4)$:

1. $\text{fib}(3) + \text{fib}(2)$
2. $\text{fib}(2) + \text{fib}(1) + \text{fib}(2)$
3. $\text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(2)$
4. $1 + \text{fib}(0) + \text{fib}(1) + \text{fib}(2)$
5. $1 + 1 + \text{fib}(1) + \text{fib}(2)$
6. $1 + 1 + 1 + \text{fib}(2)$
7. $1 + 1 + 1 + \text{fib}(1) + \text{fib}(0)$
8. $1 + 1 + 1 + 1 + \text{fib}(0)$
9. $1 + 1 + 1 + 1 + 1$
10. 5

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$
6. $1 + 1 + 1 + \mathsf{fib}(2)$
7. $1 + 1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(0)$
8. $1 + 1 + 1 + 1 + \mathsf{fib}(0)$
9. $1 + 1 + 1 + 1 + 1$
10. 5

That's a *lot* of wasted effort — for example, we have to compute $\mathsf{fib}(2)$ twice.

Why this is awful

Let's see what happens if we call $\mathsf{fib}(4)$:

1. $\mathsf{fib}(3) + \mathsf{fib}(2)$
2. $\mathsf{fib}(2) + \mathsf{fib}(1) + \mathsf{fib}(2)$
3. $\mathsf{fib}(1) + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
4. $1 + \mathsf{fib}(0) + \mathsf{fib}(1) + \mathsf{fib}(2)$
5. $1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(2)$
6. $1 + 1 + 1 + \mathsf{fib}(2)$
7. $1 + 1 + 1 + \mathsf{fib}(1) + \mathsf{fib}(0)$
8. $1 + 1 + 1 + 1 + \mathsf{fib}(0)$
9. $1 + 1 + 1 + 1 + 1$
10. 5

That's a *lot* of wasted effort — for example, we have to compute $\mathsf{fib}(2)$ twice. Only gets worse for higher n!

How awful is it?

Every call to $\mathsf{fib}(n)$ produces two recursive calls: one to $\mathsf{fib}(n-1)$ and another to $\mathsf{fib}(n-2)$. We stop when $n = 0, 1$.

How awful is it?

Every call to fib(n) produces two recursive calls: one to fib$(n-1)$ and another to fib$(n-2)$. We stop when $n = 0, 1$. That means that fib(n) requires $2^{n-2} + 2^{n-3}$ recursive calls.

How awful is it?

Every call to $\mathsf{fib}(n)$ produces two recursive calls: one to $\mathsf{fib}(n-1)$ and another to $\mathsf{fib}(n-2)$. We stop when $n = 0, 1$. That means that $\mathsf{fib}(n)$ requires $2^{n-2} + 2^{n-3}$ recursive calls. This means our algorithm is $O(2^n)$.

How awful is it?

Every call to fib(n) produces two recursive calls: one to fib$(n-1)$ and another to fib$(n-2)$. We stop when $n = 0, 1$. That means that fib(n) requires $2^{n-2} + 2^{n-3}$ recursive calls. This means our algorithm is $O(2^n)$. This is atrocious.

How awful is it?

Every call to fib(n) produces two recursive calls: one to fib$(n-1)$ and another to fib$(n-2)$. We stop when $n = 0, 1$. That means that fib(n) requires $2^{n-2} + 2^{n-3}$ recursive calls. This means our algorithm is $O(2^n)$. This is atrocious.

One reason why it's so bad is because we waste a lot of work — in order to compute fib$(n-1)$, we have to compute fib$(n-2)$, only to throw it away and do it again!

How awful is it?

Every call to $\mathsf{fib}(n)$ produces two recursive calls: one to $\mathsf{fib}(n-1)$ and another to $\mathsf{fib}(n-2)$. We stop when $n = 0, 1$. That means that $\mathsf{fib}(n)$ requires $2^{n-2} + 2^{n-3}$ recursive calls. This means our algorithm is $O(2^n)$. This is atrocious.

One reason why it's so bad is because we waste a lot of work — in order to compute $\mathsf{fib}(n-1)$, we have to compute $\mathsf{fib}(n-2)$, only to throw it away and do it again! If only we kept a hold of $\mathsf{fib}(n-2)$, we could reduce the number of recursive calls required from 2 to 1.

How awful is it?

Every call to $\mathsf{fib}(n)$ produces two recursive calls: one to $\mathsf{fib}(n-1)$ and another to $\mathsf{fib}(n-2)$. We stop when $n = 0, 1$. That means that $\mathsf{fib}(n)$ requires $2^{n-2} + 2^{n-3}$ recursive calls. This means our algorithm is $O(2^n)$. This is atrocious.

One reason why it's so bad is because we waste a lot of work — in order to compute $\mathsf{fib}(n-1)$, we have to compute $\mathsf{fib}(n-2)$, only to throw it away and do it again! If only we kept a hold of $\mathsf{fib}(n-2)$, we could reduce the number of recursive calls required from 2 to 1. Let's try using an intermediate data structure for that...

Take 2

Let's use an array to hold the intermediate computations:

Take 2

Let's use an array to hold the intermediate computations:

function fib$'(n)$
    arr $\leftarrow$ an empty array of length n
    arr$[0] = 1$
    arr$[1] = 1$
    for $i \in 2, 3, \ldots, n-1$ **do**
        arr$[i] \leftarrow$ arr$[i-1] +$ arr$[i-2]$
    return arr$[n-1]$

Take 2

Let's use an array to hold the intermediate computations:

function $\mathsf{fib}'(n)$
  $\mathsf{arr} \leftarrow$ an empty array of length n
  $\mathsf{arr}[0] = 1$
  $\mathsf{arr}[1] = 1$
  for $i \in 2, 3, \ldots, n-1$ **do**
    $\mathsf{arr}[i] \leftarrow \mathsf{arr}[i-1] + \mathsf{arr}[i-2]$
  return $\mathsf{arr}[n-1]$

Since we have a (nearly) n-length loop in $\mathsf{fib}'(n)$, this is clearly $O(n)$ — much better! All thanks to the intermediate array.

Intermediate structures are *everywhere*

Intermediate structures are *everywhere*

- Graph processing algorithms (DFS, BFS, Dijkstra's, Prim's, Kruskal's, ...)

Intermediate structures are *everywhere*

- Graph processing algorithms (DFS, BFS, Dijkstra's, Prim's, Kruskal's, ...)
- Graphics pipeline (rasterization, translations, shading, ...)

Intermediate structures are *everywhere*

- Graph processing algorithms (DFS, BFS, Dijkstra's, Prim's, Kruskal's, ...)
- Graphics pipeline (rasterization, translations, shading, ...)
- Compilers (of any sort)

Intermediate structures are *everywhere*

- Graph processing algorithms (DFS, BFS, Dijkstra's, Prim's, Kruskal's, ...)
- Graphics pipeline (rasterization, translations, shading, ...)
- Compilers (of any sort)
- Every time you do recursion (implicitly)

Intermediate structures are *everywhere*

- Graph processing algorithms (DFS, BFS, Dijkstra's, Prim's, Kruskal's, ...)
- Graphics pipeline (rasterization, translations, shading, ...)
- Compilers (of any sort)
- Every time you do recursion (implicitly)
- And so many more...

Intermediate structures are *everywhere*

- Graph processing algorithms (DFS, BFS, Dijkstra's, Prim's, Kruskal's, …)
- Graphics pipeline (rasterization, translations, shading, …)
- Compilers (of any sort)
- Every time you do recursion (implicitly)
- And so many more…

However, it's not all brilliant — intermediate structures have two major issues.

Issue One: ad-hoc-ism

Every time we employ an intermediate data structure, we have to devise the algorithm 'from first principles'. This has several downsides:

Issue One: ad-hoc-ism

Every time we employ an intermediate data structure, we have to devise the algorithm 'from first principles'. This has several downsides:

- Correctness is harder to determine

Issue One: ad-hoc-ism

Every time we employ an intermediate data structure, we have to devise the algorithm 'from first principles'. This has several downsides:

- Correctness is harder to determine
- Structure and content are entangled (no general treatment)

Issue One: ad-hoc-ism

Every time we employ an intermediate data structure, we have to devise the algorithm 'from first principles'. This has several downsides:

- Correctness is harder to determine
- Structure and content are entangled (no general treatment)
- Incidental complexity from the structure adds to the complexity of the algorithm

Issue One: ad-hoc-ism

Every time we employ an intermediate data structure, we have to devise the algorithm ‘from first principles’. This has several downsides:

- Correctness is harder to determine
- Structure and content are entangled (no general treatment)
- Incidental complexity from the structure adds to the complexity of the algorithm

This is mostly because we’re being forced to consider a specific solution every time we want an intermediate structure to be built up, then torn down.

Issue Two: efficiency

Let's look at fib' again:

Issue Two: efficiency

Let's look at fib' again:

function $\mathsf{fib}'(n)$
  $\mathsf{arr} \leftarrow$ an empty array of length n
  $\mathsf{arr}[0] = 1$
  $\mathsf{arr}[1] = 1$
  for $i \in 2, 3, \ldots, n-1$ **do**
    $\mathsf{arr}[i] \leftarrow \mathsf{arr}[i-1] + \mathsf{arr}[i-2]$
  return $\mathsf{arr}[n-1]$

Issue Two: efficiency

Let's look at fib' again:

function $\mathsf{fib}'(n)$
  $\mathsf{arr} \leftarrow$ an empty array of length n
  $\mathsf{arr}[0] = 1$
  $\mathsf{arr}[1] = 1$
  for $i \in 2, 3, \ldots, n-1$ **do**
    $\mathsf{arr}[i] \leftarrow \mathsf{arr}[i-1] + \mathsf{arr}[i-2]$
  return $\mathsf{arr}[n-1]$

At no point in the algorithm are we interested in the whole array — just the last two elements.

Issue Two: efficiency

Let's look at fib′ again:

function fib′(n)
  arr $\leftarrow$ an empty array of length n
  arr$[0] = 1$
  arr$[1] = 1$
  for $i \in 2, 3, \ldots, n-1$ **do**
    arr$[i] \leftarrow$ arr$[i-1]$ + arr$[i-2]$
  return arr$[n-1]$

At no point in the algorithm are we interested in the whole array — just the last two elements. However, we can't get rid of any part of the structure until the *very* end, even though it's useless to us!

Can we do better?

What we really want is:

Can we do better?

What we really want is:

- A general way of handling *any* algorithm which involves intermediate structures

Can we do better?

What we really want is:

- A general way of handling *any* algorithm which involves intermediate structures
- A guarantee of *some* correctness properties (from a formal/mathematical point of view)

Can we do better?

What we really want is:

- A general way of handling *any* algorithm which involves intermediate structures
- A guarantee of *some* correctness properties (from a formal/mathematical point of view)
- Some way of automatically only building up as much structure as we need

Can we do better?

What we really want is:

- A general way of handling *any* algorithm which involves intermediate structures
- A guarantee of *some* correctness properties (from a formal/mathematical point of view)
- Some way of automatically only building up as much structure as we need

Sounds impossible, right?

Can we do better?

What we really want is:

- A general way of handling *any* algorithm which involves intermediate structures
- A guarantee of *some* correctness properties (from a formal/mathematical point of view)
- Some way of automatically only building up as much structure as we need

Sounds impossible, right? However: there's a morphism for that!

Where we last left recursion schemes...

Where we last left recursion schemes...

```
-- our knotting hack
data Knot f = Tie { untie :: f (Knot f) }

-- pronounced `then', for easier composition
>>> :: (a -> b) -> (b -> c) -> (a -> c)
f >>> g = g ∘ f

type Algebra f a = f a -> a

-- catamorphism - tears structures down
cata :: Functor f => Algebra f a -> Knot f -> a
cata f = untie >>> map (cata f) >>> f

type Coalgebra f a = a -> f a

-- anamorphism - builds structures up
ana :: Functor f => Coalgebra f a -> a -> Knot f
ana f = f >>> map (ana f) >>> Tie
```

Trying to use recursion schemes

When we use an intermediate structure, we first build it up, then tear it down. What would happen if we tried to write this using recursion schemes?

Trying to use recursion schemes

When we use an intermediate structure, we first build it up, then tear it down. What would happen if we tried to write this using recursion schemes?

```
-- what's the type of this?
huh coalg alg = ana coalg >>> cata alg
```

Trying to use recursion schemes

When we use an intermediate structure, we first build it up, then tear it down. What would happen if we tried to write this using recursion schemes?

```
-- what's the type of this?
huh coalg alg = ana coalg >>> cata alg
huh :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
-- remember, (a -> f a) is Coalgebra f a
-- and (f b -> b) is Algebra f b
```

Trying to use recursion schemes

When we use an intermediate structure, we first build it up, then tear it down. What would happen if we tried to write this using recursion schemes?

```
-- what's the type of this?
huh coalg alg = ana coalg >>> cata alg
huh :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
-- remember, (a -> f a) is Coalgebra f a
-- and (f b -> b) is Algebra f b
```

Now, `huh` is a very silly name. We should really fix our terminology again...

The humble hylomorphism

```
hylo :: Functor f => Coalgebra f a -> Algebra f b -> a -> b
hylo f g = ana f >>> cata g
```

The term ‘hylomorphism’ comes from the Greek root *hylo* (meaning ‘tree’), as its behaviour can be seen as tree-like (building up is like the tree growing, and tearing down is like the tree being turned into matches).

The humble hylomorphism

```
hylo :: Functor f => Coalgebra f a -> Algebra f b -> a -> b
hylo f g = ana f >>> cata g
```

The term ‘hylomorphism’ comes from the Greek root *hylo* (meaning ‘tree’), as its behaviour can be seen as tree-like (building up is like the tree growing, and tearing down is like the tree being turned into matches).

Note: The intermediate structure being operated on by the algebra and coalgebra must be *the same*, but the data can be different.

Example: mergesort

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
  1. An array of length 1 is always sorted

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
    1. An array of length 1 is always sorted
    2. We can merge two sorted arrays into one sorted array

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
  1. An array of length 1 is always sorted
  2. We can merge two sorted arrays into one sorted array
- Relies on a three-step *merge procedure*:

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
  1. An array of length 1 is always sorted
  2. We can merge two sorted arrays into one sorted array
- Relies on a three-step *merge procedure*:
  1. Create an array big enough to hold the contents of both inputs

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
    1. An array of length 1 is always sorted
    2. We can merge two sorted arrays into one sorted array
- Relies on a three-step *merge procedure*:
    1. Create an array big enough to hold the contents of both inputs
    2. Walk over the new array, inserting the smallest element from either input at each stage

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
  1. An array of length 1 is always sorted
  2. We can merge two sorted arrays into one sorted array
- Relies on a three-step *merge procedure*:
  1. Create an array big enough to hold the contents of both inputs
  2. Walk over the new array, inserting the smallest element from either input at each stage
  3. When we've finished, return the result

Example: mergesort

- A well-known *comparison-sorting* algorithm for arrays
- An example of a *divide-and-conquer* algorithm
- Relies on the following two principles:
  1. An array of length 1 is always sorted
  2. We can merge two sorted arrays into one sorted array
- Relies on a three-step *merge procedure*:
  1. Create an array big enough to hold the contents of both inputs
  2. Walk over the new array, inserting the smallest element from either input at each stage
  3. When we've finished, return the result
- Implicitly builds up a binary tree of deferred sort operations

The merge procedure, in detail

function merge(a_1, a_2)
  $n \leftarrow \text{len}(a_1) + \text{len}(a_2)$
  $a \leftarrow$ a new array of length n
  $f_1, f_2 \leftarrow 0$
  for $i \in 0, 1, \ldots, n-1$ **do**
    if $f_1 = \text{len}(a_1)$ **then**
      $a[i] = a_2[f_2]$
      $f_2 \leftarrow f_2 + 1$
    else if $f_2 = \text{len}(a_2)$ **then**
      $a[i] \leftarrow a_1[f_1]$
      $f_1 \leftarrow f_1 + 1$
    else
      $e \leftarrow \min\{a_1[f_1], a_2[f_2]\}$
      $a[i] \leftarrow e$
      Increment corresponding f
  return a

Viewing mergesort as a hylomorphism

- When we do the 'divide' step of mergesort, we build up a binary tree:

Viewing mergesort as a hylomorphism

- When we do the 'divide' step of mergesort, we build up a binary tree:
  - The leaves are singleton arrays

Viewing mergesort as a hylomorphism

- When we do the ‘divide’ step of mergesort, we build up a binary tree:
  - The leaves are singleton arrays
  - The internal nodes are deferred merges of the children

Viewing mergesort as a hylomorphism

- When we do the ‘divide’ step of mergesort, we build up a binary tree:
  - The leaves are singleton arrays
  - The internal nodes are deferred merges of the children
- Once we’ve built up this tree, we can tear it down bottom-up by realizing the deferred computations based on their children

Viewing mergesort as a hylomorphism

- When we do the ‘divide’ step of mergesort, we build up a binary tree:
  - The leaves are singleton arrays
  - The internal nodes are deferred merges of the children
- Once we’ve built up this tree, we can tear it down bottom-up by realizing the deferred computations based on their children
- At the end, we have a single, sorted array, and the tree is gone

Viewing mergesort as a hylomorphism

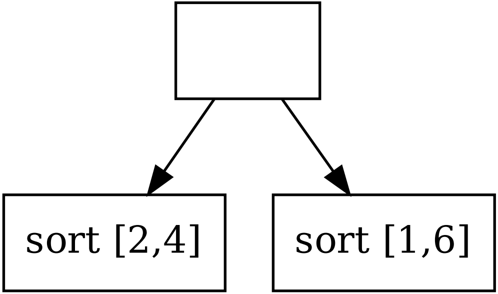
- When we do the 'divide' step of mergesort, we build up a binary tree:
  - The leaves are singleton arrays
  - The internal nodes are deferred merges of the children
- Once we've built up this tree, we can tear it down bottom-up by realizing the deferred computations based on their children
- At the end, we have a single, sorted array, and the tree is gone

Let's see how this works in practice: we will sort the array `[4, 2, 6, 1]`.

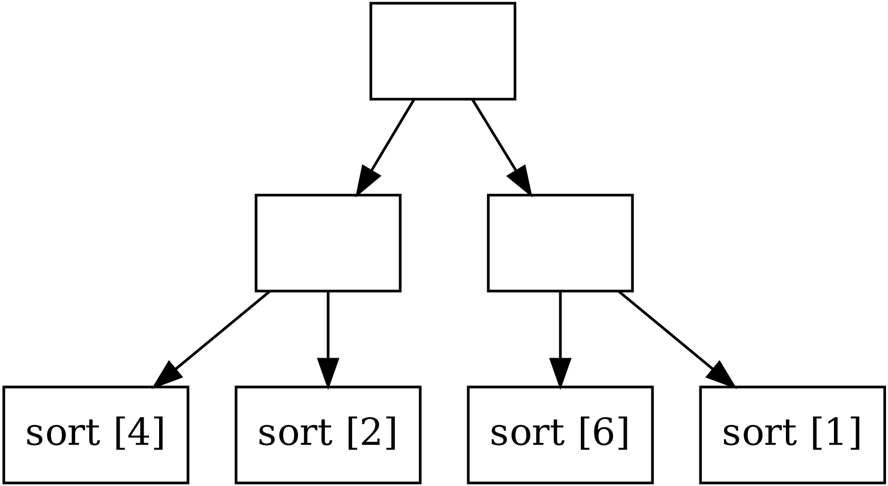
Worked example

$\boxed{\text{sort } [4,2,6,1]}$

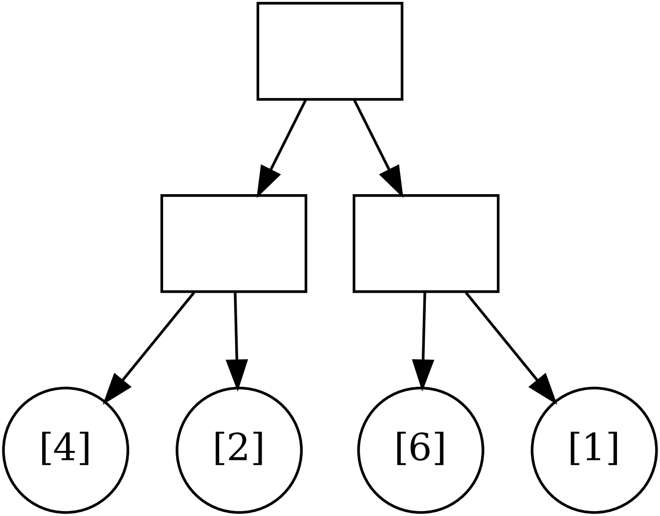
Worked example



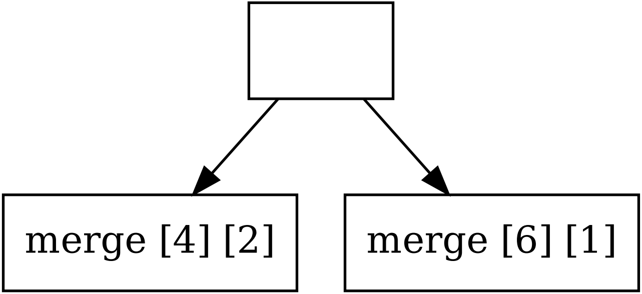
Worked example



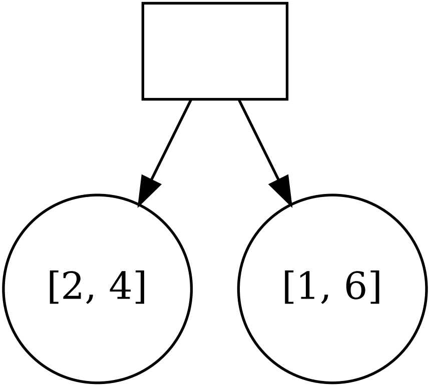
Worked example



Worked example



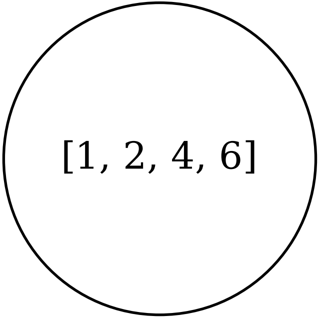
Worked example



Worked example

$$\boxed{\text{merge } [2, 4]\ [1, 6]}$$

Worked example



Putting it into a hylo

First, we will need an intermediate tree structure:

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

```
data ITree a s = Leaf a | Internal s s
```

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

```
data ITree a s = Leaf a | Internal s s
```

Lastly, we need a `Functor` instance for it. This can be derived automatically (both in theory *and* practice).

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

```
data ITree a s = Leaf a | Internal s s
```

Lastly, we need a `Functor` instance for it. This can be derived automatically (both in theory *and* practice).

```
instance Functor (ITree a) where
```

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

```
data ITree a s = Leaf a | Internal s s
```

Lastly, we need a `Functor` instance for it. This can be derived automatically (both in theory *and* practice).

```
instance Functor (ITree a) where
  map :: (s -> t) -> ITree a s -> ITree a t
```

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

```
data ITree a s = Leaf a | Internal s s
```

Lastly, we need a `Functor` instance for it. This can be derived automatically (both in theory *and* practice).

```
instance Functor (ITree a) where
  map :: (s -> t) -> ITree a s -> ITree a t
  map _ (Leaf a) = Leaf a -- nothing to do
```

Putting it into a hylo

First, we will need an intermediate tree structure:

```
data Tree a = Leaf a | Internal (Tree a) (Tree a)
```

It then needs a slight change so we can use it with recursion schemes:

```
data ITree a s = Leaf a | Internal s s
```

Lastly, we need a `Functor` instance for it. This can be derived automatically (both in theory *and* practice).

```
instance Functor (ITree a) where
  map :: (s -> t) -> ITree a s -> ITree a t
  map _ (Leaf a) = Leaf a -- nothing to do
  map f (Internal l r) = Internal (f l) (f r)
```

A suitable coalgebra

Our coalgebra will represent one 'step' in the dividing process:

A suitable coalgebra

Our coalgebra will represent one ‘step’ in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it

A suitable coalgebra

Our coalgebra will represent one 'step' in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it
- Otherwise, we make an `Internal`, and send half of the array into each child

A suitable coalgebra

Our coalgebra will represent one 'step' in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it
- Otherwise, we make an `Internal`, and send half of the array into each child

```
-- Vector a is an array of a
breakDown :: Coalgebra (ITree (Vector a)) (Vector a)
```

A suitable coalgebra

Our coalgebra will represent one 'step' in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it
- Otherwise, we make an `Internal`, and send half of the array into each child

```
-- Vector a is an array of a
breakDown :: Coalgebra (ITree (Vector a)) (Vector a)
-- breakDown :: Vector a -> ITree (Vector a)
```

A suitable coalgebra

Our coalgebra will represent one 'step' in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it
- Otherwise, we make an `Internal`, and send half of the array into each child

```
-- Vector a is an array of a
breakDown :: Coalgebra (ITree (Vector a)) (Vector a)
-- breakDown :: Vector a -> ITree (Vector a)
breakDown v = case (length v) of
```

A suitable coalgebra

Our coalgebra will represent one ‘step’ in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it
- Otherwise, we make an `Internal`, and send half of the array into each child

```
-- Vector a is an array of a
breakDown :: Coalgebra (ITree (Vector a)) (Vector a)
-- breakDown :: Vector a -> ITree (Vector a)
breakDown v = case (length v) of
  1 -> Leaf v
```

A suitable coalgebra

Our coalgebra will represent one 'step' in the dividing process:

- If we get given an array of length 1, we just make a `Leaf` storing it
- Otherwise, we make an `Internal`, and send half of the array into each child

```
-- Vector a is an array of a
breakDown :: Coalgebra (ITree (Vector a)) (Vector a)
-- breakDown :: Vector a -> ITree (Vector a)
breakDown v = case (length v) of
  1 -> Leaf v
  x -> let half = x `div` 2 in
    Internal (slice v 0 half) (slice v half (x - half))
```

A suitable algebra

Our algebra will represent one ‘step’ in the conquering process:

A suitable algebra

Our algebra will represent one ‘step’ in the conquering process:

- If we have a `Leaf`, we just return the array as-is

A suitable algebra

Our algebra will represent one 'step' in the conquering process:

- If we have a `Leaf`, we just return the array as-is
- If we have an `Internal`, merge the arrays in its left and right children

A suitable algebra

Our algebra will represent one ‘step’ in the conquering process:

- If we have a `Leaf`, we just return the array as-is
- If we have an `Internal`, merge the arrays in its left and right children

We will assume we have `merge` defined appropriately — its details don’t matter for this.

A suitable algebra

Our algebra will represent one 'step' in the conquering process:

- If we have a `Leaf`, we just return the array as-is
- If we have an `Internal`, merge the arrays in its left and right children

We will assume we have `merge` defined appropriately — its details don't matter for this.

```
mergeAlg :: Ord a => Algebra (ITree (Vector a)) (Vector a)
```

A suitable algebra

Our algebra will represent one 'step' in the conquering process:

- If we have a `Leaf`, we just return the array as-is
- If we have an `Internal`, merge the arrays in its left and right children

We will assume we have `merge` defined appropriately — its details don't matter for this.

```
mergeAlg :: Ord a => Algebra (ITree (Vector a)) (Vector a)
-- mergeAlg :: Ord a => ITree (Vector a) -> Vector a
```

A suitable algebra

Our algebra will represent one ‘step’ in the conquering process:

- If we have a `Leaf`, we just return the array as-is
- If we have an `Internal`, merge the arrays in its left and right children

We will assume we have `merge` defined appropriately — its details don't matter for this.

```
mergeAlg :: Ord a => Algebra (ITree (Vector a)) (Vector a)
-- mergeAlg :: Ord a => ITree (Vector a) -> Vector a
mergeAlg (Leaf v) = v
```

A suitable algebra

Our algebra will represent one 'step' in the conquering process:

- If we have a `Leaf`, we just return the array as-is
- If we have an `Internal`, merge the arrays in its left and right children

We will assume we have `merge` defined appropriately — its details don't matter for this.

```
mergeAlg :: Ord a => Algebra (ITree (Vector a)) (Vector a)
-- mergeAlg :: Ord a => ITree (Vector a) -> Vector a
mergeAlg (Leaf v) = v
mergeAlg (Internal v1 v2) = merge v1 v2
```

Putting it all together

```
mergeSort :: Ord a => Vector a -> Vector a
mergeSort v
```

Putting it all together

```
mergeSort :: Ord a => Vector a -> Vector a
mergeSort v
  | length v == 0 = v -- no point sorting an empty array
```

Putting it all together

```
mergeSort :: Ord a => Vector a -> Vector a
mergeSort v
  | length v == 0 = v -- no point sorting an empty array
  | otherwise = hylo breakDown mergeAlg v
```

Putting it all together

```haskell
mergeSort :: Ord a => Vector a -> Vector a
mergeSort v
  | length v == 0 = v -- no point sorting an empty array
  | otherwise = hylo breakDown mergeAlg v
```

Yay!

Reviewing our goals

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown
  - No edge-cases for 'lop-sided' structures

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown
  - No edge-cases for 'lop-sided' structures
  - No 'funny stuff' — no side effects, no structural wrecking we didn't ask for

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown
  - No edge-cases for 'lop-sided' structures
  - No 'funny stuff' — no side effects, no structural wrecking we didn't ask for
- Efficiency still a problem — until we finish `ana f` we can't start on `cata g`!

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown
  - No edge-cases for 'lop-sided' structures
  - No 'funny stuff' — no side effects, no structural wrecking we didn't ask for
- Efficiency still a problem — until we finish `ana f` we can't start on `cata g`!

Luckily for us, Meijer and Hutton solved that last one in their 'Bananas in Space' paper in 1995:

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown
  - No edge-cases for 'lop-sided' structures
  - No 'funny stuff' — no side effects, no structural wrecking we didn't ask for
- Efficiency still a problem — until we finish `ana f` we can't start on `cata g`!

Luckily for us, Meijer and Hutton solved that last one in their 'Bananas in Space' paper in 1995:

```
hylo' :: Functor f => Coalgebra f a -> Algebra f b -> a -> b
hylo' f g = f >>> map (hylo' f g) >>> g
```

Reviewing our goals

- Generalization of all divide-and-conquer algorithms (and almost all intermediate structure work!)
- Strong correctness guarantees:
  - Won't miss any items in the teardown
  - No edge-cases for 'lop-sided' structures
  - No 'funny stuff' — no side effects, no structural wrecking we didn't ask for
- Efficiency still a problem — until we finish `ana f` we can't start on `cata g`!

Luckily for us, Meijer and Hutton solved that last one in their 'Bananas in Space' paper in 1995:

```
hylo' :: Functor f => Coalgebra f a -> Algebra f b -> a -> b
hylo' f g = f >>> map (hylo' f g) >>> g
```

Result: We have everything we wanted, and recursion schemes still rule!

Questions?