Zippers

and algebra and stuff

FunctionalConf, 2019

Tony Morris

QFPL

http://qfpl.io/

The term zipper is a colloquial that is used to describe n-hole contexts into a data structure; most often `n=1`.

That is, a data structure that has a hole or pointer focussed on a specific element.

An important property of a zipper is an ability to efficiently traverse to and modify neighbours.

Loosely speaking

Take any data structure and walk to any element (1-hole) on it.

Now look around you. What do you see?

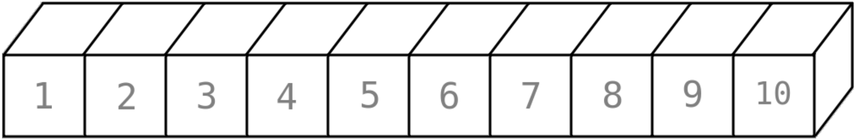
For example

Here is the list one to ten

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

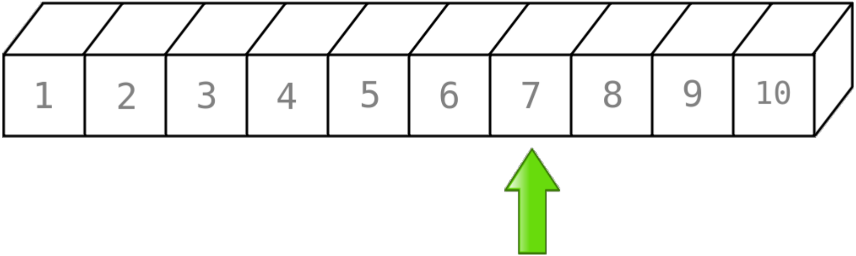
For example

Here is a depiction of a physical list one to ten



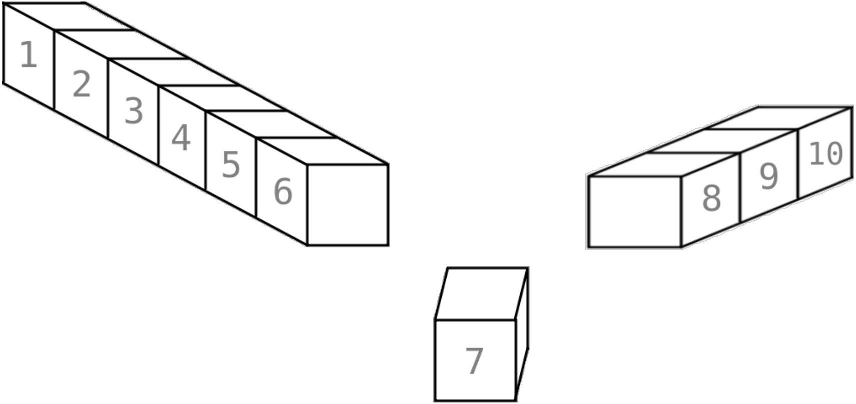
For example

Let's move to the element containing 7



List zipper

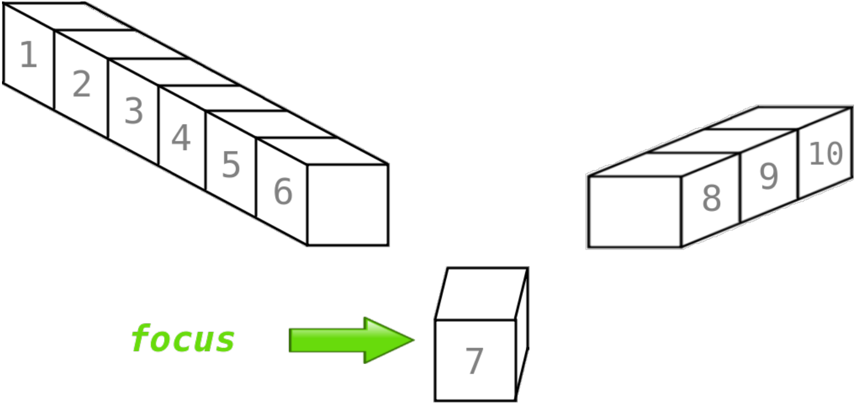
and look around us




```
([6,5,4,3,2,1], 7, [8,9,10])
```

List zipper

and look around us



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

List zipper

We can easily move to our neighbours in O(1) time

```
listz =
  ([6,5,4,3,2,1], 7, [8,9,10])

moveLeft listz =
  ([5,4,3,2,1], 6, [7,8,9,10])
```

List zipper

The zipper for [a] is ([a], a, [a])

```
data ListZipper a =
  ListZipper [a] a [a]
```

List zipper

Some useful operations on a list zipper:

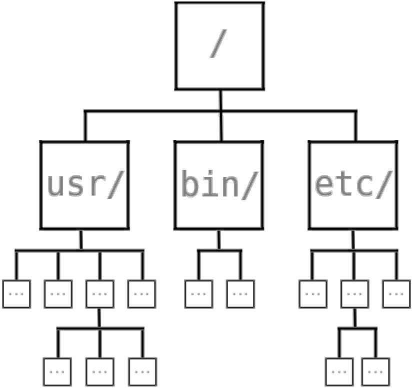
- `moveLeft/Right :: ListZipper a -> Maybe (ListZipper a)`
- `findLeft/Right :: (a -> Bool) -> ListZipper a -> Maybe (ListZipper a)`
- `modify :: (a -> a) -> ListZipper a -> ListZipper a`
- `delete :: ListZipper a -> Maybe (ListZipper a)`

Multi-way Tree

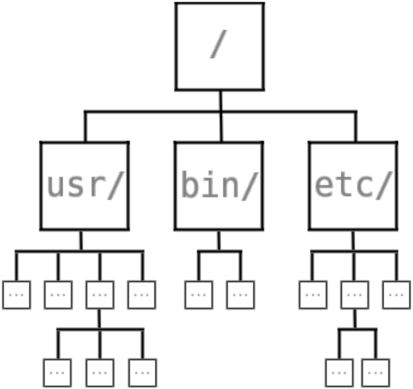
How about a multi-way tree?

```
data Tree a =
  Tree a [Tree a]
```

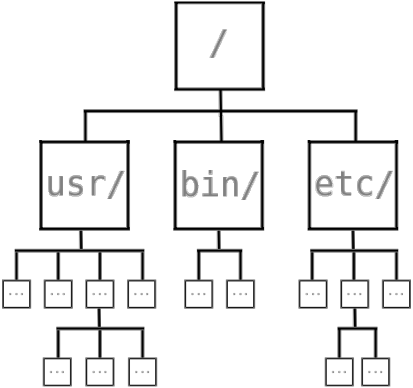
What if we stand on an element and look around?



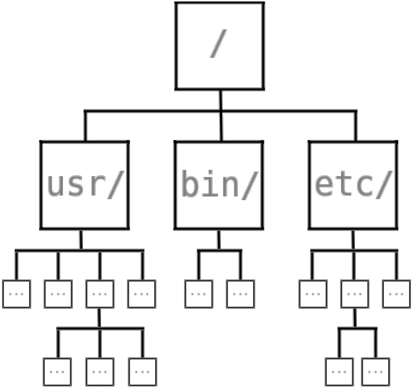
`leftSiblings :: ?`



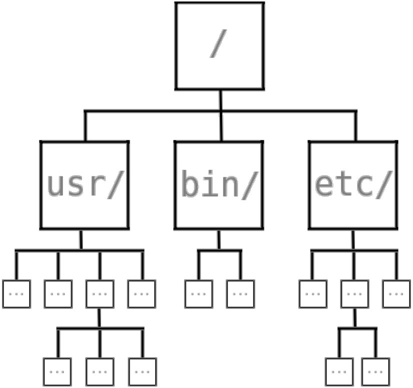
`leftSiblings :: [Tree a]`



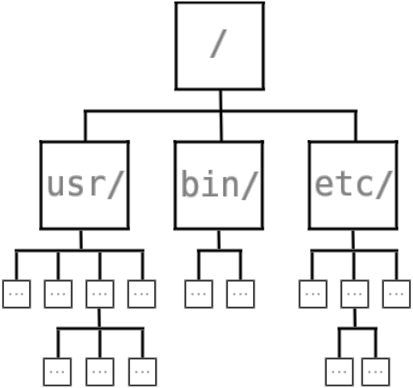
`rightSiblings :: [Tree a]`



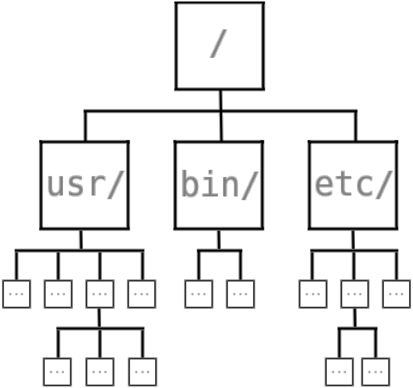
`focus :: ?`



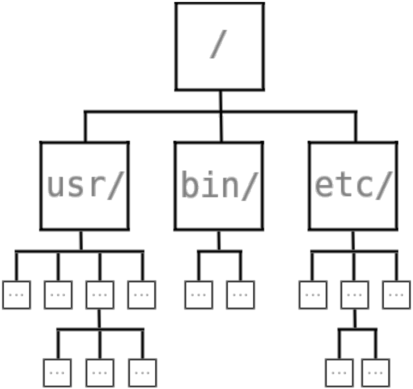
`focus :: a`



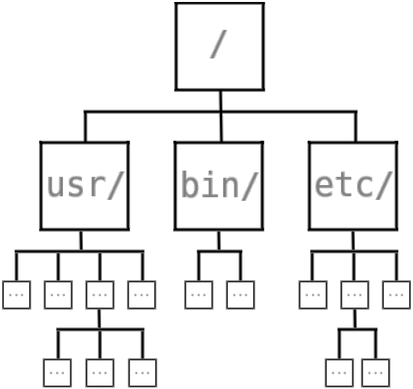
```
children :: ?
```



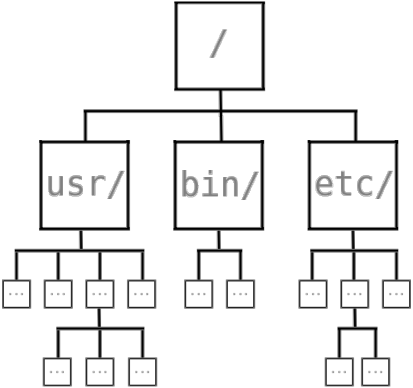
`children :: [Tree a]`



`parents :: ?`



```
parents :: [([Tree a], a, [Tree a])]
```



The zipper for a multi-way tree is

```haskell
data TreeZipper a =
  TreeZipper
    [Tree a]                  -- left siblings
    [Tree a]                  -- right siblings
    a                         -- focus
    [Tree a]                  -- children
    [([Tree a], a, [Tree a])] -- parents
```

Tree zipper

Some useful operations on a tree zipper:

- `moveParent/Child :: TreeZipper a -> Maybe (TreeZipper a)`
- `moveLeft/Right :: TreeZipper a -> Maybe (TreeZipper a)`
- `find :: (a -> Bool) -> TreeZipper a -> Maybe (TreeZipper a)`
- `all :: (a -> Bool) -> TreeZipper a -> Bool`
- `modify :: (a -> a) -> TreeZipper a -> TreeZipper a`
- `modifyTree :: (Tree a -> Tree a) -> TreeZipper a -> TreeZipper a`
- `insertSiblingLeft/Right :: Tree a -> TreeZipper a -> TreeZipper a`

Other zippers

Some other data structures have useful zippers:

- JSON
- CSV
- ASN.1
- text editors
- Pilot logbook
- *many more!*

Speaking of useful operations. . .

Comonad is

Any functor `F` supporting:

- `extract :: F a -> a`
- `duplicate :: F a -> F (F a)`
- *satisfying laws of identity and associativity*

Comonad is

Any functor `F` supporting:

- `extract :: F a -> a`
- `duplicate :: F a -> F (F a)`
- *satisfying laws of identity and associativity*

Comonad is

Any functor `F` supporting:

- `extract :: F a -> a`
- `duplicate :: F a -> F (F a)`
- *satisfying laws of identity and associativity*

Does a `ListZipper` satisfy the requirements for a comonad?

Yes

```
fmap :: (a -> b) -> ListZipper a -> ListZipper b
extract :: ListZipper x -> x
duplicate :: ListZipper w -> ListZipper (ListZipper w)
```

What about a `TreeZipper`?

Yes

```
fmap :: (a -> b) -> TreeZipper a -> TreeZipper b
extract :: TreeZipper x -> x
duplicate :: TreeZipper w -> TreeZipper (TreeZipper w)
```

Actually

All zippers are comonads *(Uustalu, 2005)*

Here is a utility function

```
-- Do any (max: 2) adjacent focii of the list zipper
-- satisfy the given predicate?
adjacentFociiSatisfy ::
  (a -> Bool)
  -> ListZipper a
  -> Bool
adjacentFociiSatisfy p z =
  let mvs k = any p (focus <$> k z)
  in  mvs moveLeft || mvs moveRight
```

Requirement

Find all zippers with a focus adjacent to a given value

Find all zippers with a focus adjacent to a given value

- `duplicate`
  we now have `ListZipper (ListZipper a)`
- `toList`
  we now have `[ListZipper a]`
- filter with `adjacentFociiSatisfy`
  we still have `[ListZipper a]`

Find all zippers with a focus adjacent to a given value

- `duplicate`
  we now have `ListZipper (ListZipper a)`
- `toList`
  we now have `[ListZipper a]`
- filter with `adjacentFociiSatisfy`
  we still have `[ListZipper a]`

Find all zippers with a focus adjacent to a given value

- `duplicate`
  we now have `ListZipper (ListZipper a)`
- `toList`
  we now have `[ListZipper a]`
- `filter` with `adjacentFociiSatisfy`
  we still have `[ListZipper a]`

allWithAdjacent

```
allWithAdjacent ::
  Eq a =>
  a
  -> ListZipper a
  -> [ListZipper a]
allWithAdjacent n =
  filter (adjacentFociiSatisfy (==n)) . toList . duplicate
```

allWithAdjacent

```
> allWithAdjacent 3 (ListZipper [3,2,1] 4 [5..10])
[
  ListZipper {
    lefts = [1]
  , focus = 2
  , rights = [3,4,5,6,7,8,9,10]
  }
, ListZipper {
    lefts = [3,2,1]
  , focus = 4
  , rights = [5,6,7,8,9,10]
  }
]
```

allWithAdjacent

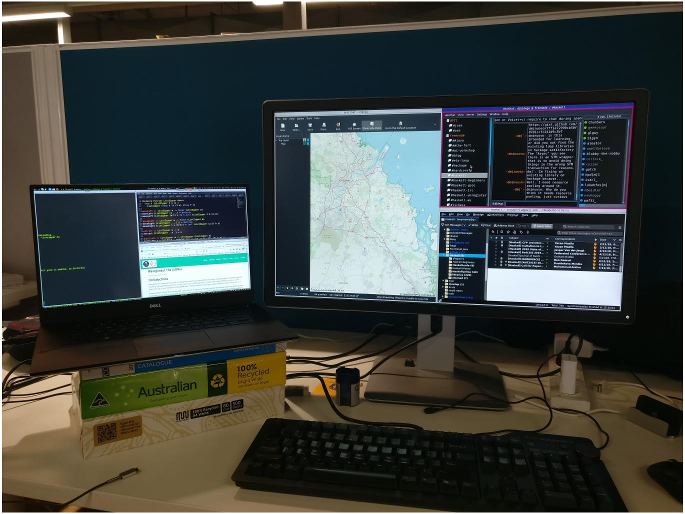
```
> allWithAdjacent 3 (ListZipper [3,2,1] 4 [7,2,6,3])
[
  ListZipper {
    lefts = [1]
  , focus = 2
  , rights = [3,4,7,2,6,3]
  }
, ListZipper {
    lefts = [3,2,1]
  , focus = 4
  , rights = [7,2,6,3]
  }
, ListZipper {
    lefts = [2,7,4,3,2,1]
  , focus = 6
  , rights = [3]
  }
]
```

Who's wanted that in their text editor before?

What were you editing? JSON? Your pilot logbook?

What about **a programming language**?

other uses of zippers



OK, but...

Why zipper?

If I wanted to modify the element of a tree, why wouldn't I use a lens (or traversal)?

OK, but. . .

```
immediateChildren :: Traversal (Tree a) (Tree a)
focus :: Lens (Tree a) a
```

Zipper vs Lens

While lens gives you nice compositional properties, zipper does *context-dependent* updates

Zipper vs Lens

- **lens**

  view one hole in a data structure, then operate on it

- **traversal**

  view many holes in a data structure, then operate on it

- **zipper**

  *view one hole in a data structure, then **depend** on it to move efficiently to another hole (and so on)*

Zipper vs Lens

- **lens**

  view and operate on y *in* `(x, y, z)`

- **traversal**

  view and operate on all of the y *in* `(x, y, z, y, [y])`

- **zipper**

  view and operate on a specific y *in* `(y, y, y)` *and depending on the operation outcome, move to a different* y *(and so on)*

Algebra

Algebraic Data Types can be thought of in terms of regular algebraic equations

Some examples include

- **sum types**
  `Either A B` or "A or B" corresponds to the equation `A + B`
- **product types**
  `(A, B)` or "A and B" corresponds to the equation `A * B`
- **exponentiation**
  `A -> B` corresponds to the equation B^A
- **unit**
  given `data Unit = Unit`,
  the `Unit` data type corresponds to the value 1
- **void**
  given `data Void`,
  the `Void` data type corresponds to the value 0

Some examples include

- **sum types**
  `Either A B` or "A or B" corresponds to the equation `A + B`
- **product types**
  `(A, B)` or "A and B" corresponds to the equation `A * B`
- **exponentiation**
  `A -> B` corresponds to the equation B^A
- **unit**
  given `data Unit = Unit`,
  the `Unit` data type corresponds to the value 1
- **void**
  given `data Void`,
  the `Void` data type corresponds to the value 0

Some examples include

- **sum types**
  `Either A B` or "A or B" corresponds to the equation `A + B`
- **product types**
  `(A, B)` or "A and B" corresponds to the equation `A * B`
- **exponentiation**
  `A -> B` corresponds to the equation $\texttt{B}^{\texttt{A}}$
- **unit**
  given `data Unit = Unit`,
  the `Unit` data type corresponds to the value 1
- **void**
  given `data Void`,
  the `Void` data type corresponds to the value 0

Some examples include

- **sum types**
  `Either A B` or "A or B" corresponds to the equation `A + B`
- **product types**
  `(A, B)` or "A and B" corresponds to the equation `A * B`
- **exponentiation**
  `A -> B` corresponds to the equation B^{A}
- **unit**
  given `data Unit = Unit`,
  the `Unit` data type corresponds to the value 1
- **void**
  given `data Void`,
  the `Void` data type corresponds to the value 0

Some examples include

- **sum types**
  `Either A B` or "A or B" corresponds to the equation `A + B`
- **product types**
  `(A, B)` or "A and B" corresponds to the equation `A * B`
- **exponentiation**
  `A -> B` corresponds to the equation B^A
- **unit**
  given `data Unit = Unit`,
  the `Unit` data type corresponds to the value 1
- **void**
  given `data Void`,
  the `Void` data type corresponds to the value 0

Let's look at Bool

```
data Bool = True | False
```

- The `True` constructor has no arguments, which is equivalent to carrying `Unit`
- The `False` constructor has no arguments, which is equivalent to carrying `Unit`
- The whole data type carries 1 or 1
- `Bool ~ 1 + 1 ~ 2`

Let's look at `Bool`

```
data Bool = True | False
```

- The `True` constructor has no arguments, which is equivalent to carrying `Unit`
- The `False` constructor has no arguments, which is equivalent to carrying `Unit`
- The whole data type carries `1` or `1`
- `Bool ~ 1 + 1 ~ 2`

How about `Maybe a`

```
data Maybe a = Nothing | Just a
```

- The `Nothing` constructor has no arguments, which is equivalent to carrying `Unit`
- The `Just` constructor has an argument a
- The whole data type carries 1 or a
- `Maybe a ~ 1 + a`

How about `Maybe a`

```
data Maybe a = Nothing | Just a
```

- The `Nothing` constructor has no arguments, which is equivalent to carrying `Unit`
- The `Just` constructor has an argument `a`
- The whole data type carries `1` or `a`
- `Maybe a ~ 1 + a`

Another one `Either Void a`

- The `Left` constructor carries 0
- The `Right` constructor has an argument `a`
- The whole data type carries 0 or `a`
- `Either Void a ~ 0 + a ~ a`

Another one `Either Void a`

- The `Left` constructor carries 0
- The `Right` constructor has an argument `a`
- The whole data type carries 0 or `a`
- `Either Void a ~ 0 + a ~ a`

and another `(Void, a)`

- The whole data type carries `0` and `a`
- `(Void, a) ~ 0 * a ~ 0`

and another `(Void, a)`

- The whole data type carries `0` and `a`
- `(Void, a) ~ 0 * a ~ 0`

lots of `Bool`

- `(Bool, Bool)`
- `Either Bool Bool`
- `Bool -> Bool`
- $2 * 2$
- $2 + 2$
- 2^2
- These are all 4

lots of `Bool`

- `(Bool, Bool)`
- `Either Bool Bool`
- `Bool -> Bool`
- `2 * 2`
- `2 + 2`
- 2^2
- These are all 4

lots of `Bool`

- `(Bool, Bool)`
- `Either Bool Bool`
- `Bool -> Bool`
- `2 * 2`
- `2 + 2`
- 2^2
- These are all 4

Inhabitants

The resulting algebraic equation gives us the number of *inhabitants*.
Or, the number of values with that type.

Inhabitants

- `Maybe (Bool -> Maybe Bool)`
- has `1 + (1 + 2)`^{2} inhabitants
- 9 inhabitants
- `(Either Bool (Maybe Bool), Bool, (Unit, Bool), Either Void Bool)`
- has `(2 + (1 + 2)) * 2 * (1 * 2) * (0 + 2)` inhabitants
- 40

Inhabitants

- `Maybe (Bool -> Maybe Bool)`
- has `1 + (1 + 2)`^{2} inhabitants
- 9 inhabitants
- `(Either Bool (Maybe Bool), Bool, (Unit, Bool), Either Void Bool)`
- has `(2 + (1 + 2)) * 2 * (1 * 2) * (0 + 2)` inhabitants
- 40

Algebraically

What is `[a]`?

Lists

- [a] is either zero a or one a or two a ...
- $a^0 + a^1 + a^2 \ldots$
- using algebraic rules, this simplifies to 1 + a * [a]
- 1 or (a and [a])
- The [] *(carrying Unit)* or (:) constructor

Lists

- [a] is either zero a or one a or two a ...
- $a^0 + a^1 + a^2 \ldots$
- using algebraic rules, this simplifies to 1 + a * [a]
- 1 or (a and [a])
- The [] *(carrying Unit)* or (:) constructor

Lists

- `[a]` is either zero a or one a or two a ...
- `a`^{0} `+ a`^{1} `+ a`^{2} ...
- using algebraic rules, this simplifies to `1 + a * [a]`
- `1` or `(a and [a])`
- The `[]` *(carrying `Unit`)* or `(:)` constructor

Remember calculus?

Me neither :)

Here is a data type:

`Either x (x, x)`

Algebraically:

`x + (x * x)`

Differentiate

$$\frac{\partial}{\partial x}\ \texttt{(x + (x * x))}$$

$$\frac{\partial}{\partial x}\ \texttt{(x + (x * x))}$$

(sum rule)

$$= \frac{\partial}{\partial x}\ \texttt{(x} + \frac{\partial}{\partial x}\ \texttt{(x * x))}$$

(power rule, line rule)

$$= \texttt{1 + (2 * x)}$$

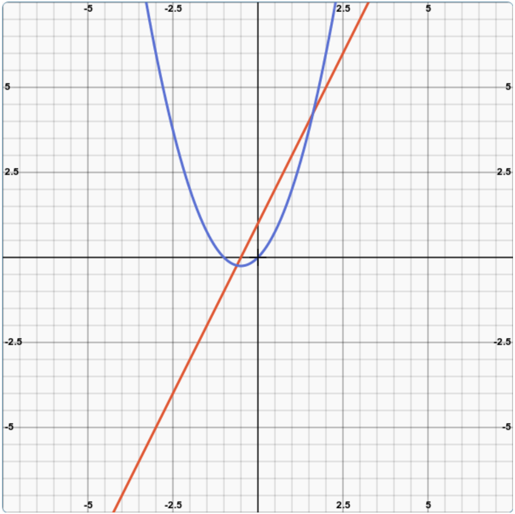
$$= \texttt{Maybe (Bool, x)}$$

$\therefore$

$$\frac{\partial}{\partial x}\,\texttt{Either x (x, x)}$$

$$= \texttt{Maybe (Bool, x)}$$

$\frac{\partial}{\partial x}$ Either x (x, x)



We'll do another one

`(Either x x, Either x x)`

Algebraically:

`(x + x) * (x + x)`

Differentiate

$\frac{\partial}{\partial x}$ `((x + x) * (x + x))`

$$\frac{\partial}{\partial x}\ \texttt{((x + x) * (x + x))}$$

$$= \frac{\partial}{\partial x}\ \texttt{4 * x}^2$$

(power rule)

$$= \texttt{4 * } \frac{\partial}{\partial x}\ \texttt{x}^2$$

(power rule)

$$= \texttt{4 * 2 * x}$$

$$= \texttt{8 * x}$$

$\therefore$

$$\frac{\partial}{\partial x}\ \texttt{(Either x x, Either x x)}$$

$$= \texttt{(Maybe Bool -> Bool, x)}$$

A simpler one

`(x, x, x)`

Algebraically:

```
x * x * x
```

Differentiate

$\frac{\partial}{\partial x}$ `x * x * x`

$$\frac{\partial}{\partial x}\ \mathtt{x * x * x}$$

$$= \frac{\partial}{\partial x}\ \mathtt{x}^3$$

(power rule)

$$= 3 * \mathtt{x}^2$$

$\therefore$

$$\frac{\partial}{\partial x}\ \texttt{(x, x, x)}$$

$$= \texttt{(Maybe Bool, x, x)}$$

Summary

- $\frac{\partial}{\partial x}$ `Either x (x, x) = Maybe (Bool, x)`
- $\frac{\partial}{\partial x}$ `(Either x x, Either x x) = (Maybe Bool -> Bool, x)`
- $\frac{\partial}{\partial x}$ `(x, x, x) = (Maybe Bool, x, x)`

Insight

The derivative of *any* data structure, **is its zipper!**[AAMG05] [a]

[a] *without the 1-hole context value*

Let's do list

`List a = 1 + a +` a^2 `+` a^3 `+ ...`

List

```
List a = 1 + a + a² + a³ + ...
```

```
let K a = a + a² + a³ + ...
List a = 1 + K a
```

multiply list by a

```
K a = a * List a
```

$\therefore$ List a $=$ 1 $+$ a * List a

this makes sense if we think of `List` in terms of its constructors

List

```
List a = 1 + a * List a
```

---

subtract `(a * List a)` *both sides*

```
List a - (a * List a) = 1
```

multiply `List a` *by* `1`

```
(1 * List a) - (a * List a) = 1
```

apply distributive law of multiplication

```
List a * (1 - a) = 1
```

divide both sides by `1 - a`

```
List a = 1 / (1 - a)
```

---

apply exponent rule

$\therefore$ `List a = (1 - a)`$^{-1}$

List

$$\texttt{List a} = \texttt{(1 - a)}^{-1}$$

$$\texttt{let u = 1 - a}$$

apply chain rule

$$\frac{\partial}{\partial a}\ \texttt{(1 - a)}^{-1} = \frac{\partial}{\partial u}\ \texttt{u}^{-1} * \frac{\partial}{\partial a}\ \texttt{1 - a}$$

differentiate u^{-1}

$$\frac{\partial}{\partial u}\ \texttt{u}^{-1} = -1\ /\ \texttt{u}^{2}$$

differentiate $1 - a$

$$\frac{\partial}{\partial a}\ \texttt{1 - a} = -1$$

$$\therefore \frac{\partial}{\partial a}\ \texttt{(1 - a)}^{-1} = (-1\ /\ \texttt{u}^{2}) * -1$$

List

$$\frac{\partial}{\partial a}\ \texttt{(1 - a)}^{\texttt{-1}} = \texttt{(-1 / u}^{\texttt{2}}\texttt{) * -1}$$

---

substitute back u *= 1 - a*

$$\frac{\partial}{\partial a}\ \texttt{(1 - a)}^{\texttt{-1}} = \texttt{(-1 / (1 - a)}^{\texttt{2}}\texttt{) * -1}$$

simplify by multiplying right side by -1

$$\frac{\partial}{\partial a}\ \texttt{(1 - a)}^{\texttt{-1}} = \texttt{1 / (1 - a)}^{\texttt{2}}$$

apply exponent rule

$$\frac{\partial}{\partial a}\ \texttt{(1 - a)}^{\texttt{-1}} = \texttt{(1 / 1 - a)}^{\texttt{2}}$$

$$\frac{\partial}{\partial a}\ \texttt{(1 - a)}^{\texttt{-1}} = \texttt{((1 - a)}^{\texttt{-1}}\texttt{)}^{\texttt{2}}$$

$\therefore$ The derivative of a `List` is a pair of `List`

List derivative

```
-- 1 + (a * List a)
List a ~ Nil | Cons a (List a)

-- (1 + (a * List a)) * (1   + (a * List a))
ListDerivative ~ (List a, List a)

-- add the hole back to the derivative
-- (1 + (a * List a)) * a * (1   + (a * List a))
ListZipper ~ (List a, a, List a)
```

```
data ListZipper a = ListZipper [a] a [a]
```

The End

References

Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani, ∂ *for data: Differentiating data structures*, Fundamenta Informaticae **65** (2005), no. 1-2, 1–28.