Functional Programming

The universality of fold

Chris Constable

Review

- higher-order functions
- `map`, `filter`, and `reduce`
- recursive functions can replace loops

Review

```
[1, 2, 3].reduce(0, (acc, x) => acc + x)

(((0 + 1) + 2) + 3)

6
```

Review

```
[1, 2, 3].reduce(0, (acc, x) => acc + x)

(((0 + 1) + 2) + 3)

6
```

Review

```
[1, 2, 3].reduce(0, (acc, x) => acc + x)

(((0 + 1) + 2) + 3)

6
```

Review

```
[1, 2, 3].reduce(0, (acc, x) => acc + x)

(((0 + 1) + 2) + 3)

6
```

Review

```
[1, 2, 3].reduce(0, (acc, x) => acc + x)

(((0 + 1) + 2) + 3)

6
```

Review

```
[1, 2, 3].reduce(0, (acc, x) => acc + x)

(((0 + 1) + 2) + 3)

6
```

3 challenges

> Challenge 1

The universality of fold: Using only `reduce`, implement `map` and `filter`.

> Challenge 1

The universality of fold: Using only `reduce`, implement `map` and `filter`.

If that seems a bit too difficult, try implementing `map` or `filter` on your own without `reduce`.

> Challenge 1

The universality of fold: Using only `reduce`, implement `map` and `filter`.

If that seems a bit too difficult, try implementing `map` or `filter` on your own without `reduce`.

If that still seems a bit too much pair up with someone!

```
function map(f, list) {
    return list.reduce(
        (acc, x) => acc.concat([f(x)]),
        []
    )
}

function filter(pred, list) {
    return list.reduce(
        (acc, x) => pred(x) ? acc.concat([x]) : acc,
        []
    )
}
```

When we use reduce it goes from left-to-right:

```
------------>
(((i a) b) c)
```

When we use reduce it goes from left-to-right:

```
------------>
(((i a) b) c)
```

What if we want to reduce right-to-left?

```
<------------
(a (b (c i)))
```

When we use reduce it goes from left-to-right:

```
------------>
(((i a) b) c)
```

What if we want to reduce right-to-left?

```
<------------
(a (b (c i)))
```

Who cares?

foldl

```
------------>
(((i a) b) c)
```

foldr

```
<------------
(a (b (c i)))
```

foldl (`reduce` in Javascript)

```
------------>
(((i a) b) c)
```

foldr

```
<------------
(a (b (c i)))
```

```haskell
// Haskell
foldl (++) "a" ["b", "c"]                    // "abc"

// Scala
Array("b", "c").foldLeft("a")(_ + _)         // "abc"

// JS
["b", "c"].reduce((acc, x) => acc + x, "a")  // "abc"
```

```haskell
// Haskell
foldr (++) "a" ["b", "c"]                    // "bca"

// Scala
Array("b", "c").foldRight("a")(_ + _)        // "bca"
```

```
// Haskell
foldr (++) "a" ["b", "c"]                          // "bca"

// Scala
Array("b", "c").foldRight("a")(_ + _)              // "bca"

// JS
["b", "c"].reduceRight((acc, x) => acc + x, "a")   // "acb"
```

```
ghci> foldr (-) 52 [12, 30, 21]
-49

node> [12, 30, 21].reduceRight((acc, x) => acc - x, 52)
-11
```

> Challenge 2

Show that Javascript's `reduceRight` (aka `foldr`) violates the *3rd duality theorem* of fold which states that:

$$foldr\ (\oplus)\ a\ xs = foldl\ (\widetilde{\oplus})\ a\ xs^{reversed}$$

$$\text{where} \quad a \oplus b = b \mathbin{\widetilde{\oplus}} a$$

```
const initial   = 52
const list      = [12, 30, 21]
const listR     = list.reverse()

const f = (acc, x) => acc - x
const g = (x, acc) => acc - x

list.reduce(f, initial) == listR.reduceRight(g, initial)

// false :(
```

> Challenge 3

The universality of foldr: Write a version of `reduceRight` (aka `foldr`) that obeys the *3rd duality theorem*. Using only the `foldr` function, implement `foldl`.

> Challenge 3

The universality of foldr: Write a version of `reduceRight` (aka `foldr`) that obeys the *3rd duality theorem*. Using only the `foldr` function, implement `foldl`.

I'm not going to discuss the answer to this because it's pretty mind-bending. Graham Hutton published a paper on it called "A tutorial on the universality and expressiveness of fold". I've written a `Javascript solution` in the source repo for these slides.

Summary

- folds encapsulates recursion
- using folds alone it is possible to implement an astounding number of useful functions
- `foldr` can be used to implement `foldl`
- don't write cryptic folds that nobody will understand in a week
- check out the `source` for the solution to challenge 3