framp.me

Advanced Profunctors

Recap on framp.me/profunctors

a Profunctor is a Bifunctor contravariant in the first argument`

a Profunctor is a container with a `.promap()`

can a function be a Profunctor? Yes

```
const Func = (f) => ({
  value: f,
  // promap :: Func a b -> (c->a) -> (b->d) -> Func c d
  promap: (contra, co) =>
    Func((...args) => co(f(contra(...args))))
})

assert.deepEqual(Func(a => a*2)
        .promap(a => a.length, a => a*4).value('ab'), 16)
```

```
const capitalise = a => a[0].toUpperCase() + a.slice(1)
const capitaliseWords = Func(a => a.map(capitalise))
  .promap(a => a.split(' '), a => a.join(' ')).value

assert.equal(capitaliseWords('tom cat bob cat'),
  'Tom Cat Bob Cat')
```

Example 1

Limits

```
const Limits = (step, check) => ({
  //step :: a -> (b, b)
  //check :: a -> a -> Bool
  step, check,
  // promap :: Lim g h -> (f->g) -> (h->i) -> Lim f i
  promap: (contra, co) =>
    Limits((...args) => step(contra(...args)).map(co),
           (...args) => check(...args.map(contra)))
})

const step1 = Limits(a => [a-1, a+1],
                     (a,b) => Math.abs(a-b)===1)
assert.deepEqual(step1.step(1), [0,2])
assert.equal(step1.check(1,2), true)
assert.equal(step1.check(1,3), false)

const step110 = step1.promap(a => a*10, a => a/10)
assert.deepEqual(step110.step(1), [0.9,1.1])
assert.equal(step110.check(1,2), false)
assert.equal(step110.check(1,1.1), true)
```

```
const Just = (v) => ({ v })
const Nothing = () => { n: true }
//maybe :: b -> (a -> b) -> Maybe a -> b
const maybe = defVal => a => a.n ? defVal : a.v
```

```
const stepMaybe = (defVal) =>
  step1.promap(maybe(defVal), Just)
assert.deepEqual(stepMaybe(0).step({ v:5 }),
  [{ v:4 }, { v:6 }])
assert.deepEqual(stepMaybe(0).step({ n: true }),
  [{ v:-1 }, { v:1 }])
assert.equal(stepMaybe(0).check({ v:1 }, { v:2 }), true)
assert.equal(stepMaybe(0).check({ v:1 }, { v:3 }), false)
```

Example 2

Extra composition with Category

a Category is a container with a `.compose()` and an `C.id`

can a function be a Category?

```
const Func = f => ({
  value: f,
  // compose :: Func a b -> Func b c -> Func a c
  compose: (fn) =>
    Func((...args) => fn.value(f(...args)))
})
Func.id = Func(a => a)

const f = Func(a => a*2)
  .compose(Func(a => a+3))
  .compose(Func(a => a+5))
  .compose(Func.id)
assert.deepEqual(f.value(4), 16)
```

Note: Auto is function which returns a value and a modified version of itself

```
const Auto = step => ({
  //step :: a -> (Auto a c, b)
  step,
  // compose :: Auto b c -> Auto a b -> Auto a c
  compose: a => Auto((v) => {
   const [a1, av] = a.step(v)
   const [b1, bv] = step(av)
   return [b1.compose(a1), bv]
  }),
  //promap :: Auto b c ->  Auto a b -> Auto a c
  promap: (contra, co) => Auto((v) => {
   const [b1, bv] = step(contra(v))
   return [b1, co(bv)]
  })
})

const times2 = Auto(a => [times2, a*2])
assert.equal(times2.step(1)[1], 2)
assert.equal(times2.step(1)[0].step(5), 10)
const add3 = Auto(a => [add3, a+3])
assert.equal(times2.compose(add3).step(1)[1], 8)
assert.equal(
  times2.promap(a => a+1, a => a-1).step(5)[1], 11)
```

```
//run :: Auto a b -> [a] -> [b]
const run = (auto, inputs) => inputs.reduce(([auto, outpu
  const [newAuto, output] = auto.step(input);
  return [newAuto, outputs.concat([output])];
}, [auto, []])[1]
const times2 = Auto(a => [times2, a*2])
assert.deepEqual(run(times2, [1,2,3]), [2,4,6])
```

```
//accumState :: (b -> a -> (c, b)) -> a -> Auto a (c, b)
const accumState = (fn, acc) => Auto(input => {
  const [output, acc1] = fn(acc, input)
  return [accumState(fn, acc1), output]
})
//accum :: (b -> a -> b) -> a -> Auto a b
const accum = (fn, acc) => accumState((v) =>
  [fn(v), fn(v)], acc)

//total :: Number -> Auto Number Number
const total = accum((a,b)=>a+b, 0)
assert.equal(total.step(3)[0].step(8)[1], 11)
//length :: Number -> Auto Number Number
const length = accum((a)=>a+1, 0)
assert.equal(length.step(3)[0].step(8)[1], 2)
assert.equal(run(length, [42,9,11]), [1,2,3])
```

```
//how can I return both accumulated values together?
assert.deepEqual(run(times2.compose(length), [42,9,11]),
  [2,4,6]) // nope
```

Example 3

The need for Strength

a Strong Profunctor is a container with a `.first()`

```
const Auto = step => ({
  step,
  compose: a => Auto((v) => {
   const [a1, av] = a.step(v)
   const [b1, bv] = step(av)
   return [b1.compose(a1), bv]
  }),
  promap: (contra, co) => Auto((v) => {
   const [b1, bv] = step(contra(v))
   return [b1, co(bv)]
  }),
  //first :: Auto a b -> Auto (a, c) (b, c)
  first: () => Auto(([v, x]) => {
    const [b1, bv] = step(v)
    return [b1, [bv, x]]
  }),
  second: () => Auto(([x, v]) => {
    const [b1, bv] = step(v)
    return [b1, [x, bv]]
  })
})
```

note: second can be derived from first easily

```
const times2 = Auto(a => [times2, a*2])
assert.deepEqual(
  times2.first().step([1,"lol"])[1], [2, "lol"])
```

```
//split :: Auto a b -> Auto c d -> Auto (a, c) (b, d)
const split = (a, b) => Auto(([v, u]) => {
  const [a1, av] = a.step(v)
  const [b1, bu] = b.step(u)
  return [split(a1, b1), [av, bu]]
})
assert.deepEqual(
  split(length, times2).step([6,6]), [1,12])
//fanout :: Auto a b -> Auto a c -> Auto a (b, c)
const fanout = (a, b) => Auto((v) => {
  const [a1, output] = split(a, b).step([v, v])
  return [Auto(v => a1.step([v, v])), output]
})
assert.deepEqual(
  split(length, times2).step(6), [1,12])

run(fanout(length, times2), [32,45,87,21])
```

Fin.

slides on framp.me/profunctors/2