Chapter 8: Applicative and traversable functors
Part 2: Their laws and structure

Sergei Winitzki

Academy by the Bay

2018-06-07

Motivation for applicative laws. Naturality for `map2`

Treat `map2` as a replacement for a monadic block with independent effects:

```
for {                          map2 (
  x ← cont1                      cont1,
  y ← cont2                      cont2
} yield g(x, y)                ) { (x, y) ⇒ g(x, y) }
```

- Main idea: Formulate the monad laws in terms of `map2` and `pure`

Naturality laws: Manipulate data in one of the containers

```
for {                          for {
  x ← cont1.map(f)               x ← cont1
  y ← cont2                      y ← cont2
} yield g(x, y)                } yield g(f(x), y)
```

and similarly for `cont2` instead of `cont1`; now rewrite in terms of for `map2`:

- **Left naturality** for `map2`:

```
map2(cont1.map(f), cont2)(g)
  = map2(cont1, cont2){ (x, y) ⇒ g(f(x), y) }
```

- **Right naturality** for `map2`:

```
map2(cont1, cont2.map(f))(g)
  = map2(cont1, cont2){ (x, y) ⇒ g(x, f(y)) }
```

Associativity and identity laws for `map2`

Inline two generators out of three, in two different ways:

```
for {                           for {
  x ← cont1                       (x, y) ← for {
  (y, z) ← for {                          xx ← cont1
          yy ← cont2                      yy ← cont2
          zz ← cont3                    } yield (xx, yy)
        } yield (yy, zz)          z ← cont3
} yield g(x, y, z)              } yield g(x, y, z)
```

Write this in terms of `map2` to obtain the **associativity law** for `map2`:

```
map2(cont1, map2(cont2, cont3)((_,_)){ case(x,(y,z))⇒g(x,y,z)}
  = map2(map2(cont1, cont2)((_,_)), cont3){ case((x,y),z))⇒g(x,y,z)}
```

Empty context precedes a generator, or follows a generator:

```
for { x ← pure(a)               for {
      y ← cont                    y ← cont
} yield g(x, y)                 } yield g(a, y)
```

Write this in terms of `map2` to obtain the **identity laws** for `map2` and `pure`:

```
map2(pure(a), cont)(g) = cont.map { y ⇒ g(a, y) }
map2(cont, pure(b))(g) = cont.map { x ⇒ g(x, b) }
```

Deriving the `ap` operation from `map2`

Can we avoid having to define map_n separately for each n?
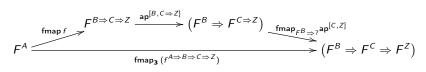
- Use curried arguments, $\text{fmap}_2 : (A \Rightarrow B \Rightarrow Z) \Rightarrow F^A \Rightarrow F^B \Rightarrow F^Z$
- Set $A = B \Rightarrow Z$ and apply fmap_2 to the identity $\text{id}^{(B \Rightarrow Z) \Rightarrow (B \Rightarrow Z)}$: obtain $\text{ap}^{[B,Z]} : F^{B \Rightarrow Z} \Rightarrow F^B \Rightarrow F^Z \equiv \text{fmap}_2 \, (\text{id})$
- The functions `fmap2` and `ap` are computationally equivalent:

$$\text{fmap}_2 \, f^{A \Rightarrow B \Rightarrow Z} = \text{fmap} \, f \circ \text{ap}$$

$$
\begin{array}{ccc}
 & F^{B \Rightarrow Z} & \\
 \nearrow^{\text{fmap} f} & & \searrow^{\text{ap}} \\
F^A \xrightarrow[\text{fmap}_2 \, (f^{A \Rightarrow B \Rightarrow Z})]{} & & (F^B \Rightarrow F^Z)
\end{array}
$$

- The functions `fmap3`, `fmap4` etc. can be defined similarly:

$$\text{fmap}_3 \, f^{A \Rightarrow B \Rightarrow C \Rightarrow Z} = \text{fmap} \, f \circ \text{ap} \circ \text{fmap}_{F^B \Rightarrow ?} \text{ap}$$

$$
\begin{array}{ccc}
 & F^{B \Rightarrow C \Rightarrow Z} \xrightarrow{\text{ap}^{[B, C \Rightarrow Z]}} (F^B \Rightarrow F^{C \Rightarrow Z}) & \\
 \nearrow^{\text{fmap} f} & & \searrow^{\text{fmap}_{F^B \Rightarrow ?} \text{ap}^{[C,Z]}} \\
F^A \xrightarrow[\text{fmap}_3 \, (f^{A \Rightarrow B \Rightarrow C \Rightarrow Z})]{} & & (F^B \Rightarrow F^C \Rightarrow F^Z)
\end{array}
$$

Intuition: the `zip` operation on lists

- Note: Function types $A \Rightarrow B \Rightarrow C$ and $A \times B \Rightarrow C$ are equivalent
- Uncurry fmap_2 to fmap2 : $(A \times B \Rightarrow C) \Rightarrow F^A \times F^B \Rightarrow F^C$
- Compute $\text{fmap2}(f)$ with $f = \text{id}^{A \times B \Rightarrow A \times B}$, expecting to obtain a simpler natural transformation:

$$\text{zip} : F^A \times F^B \Rightarrow F^{A \times B}$$

- This is quite similar to `zip` for lists:
  `List(1, 2).zip(List(10, 20)) = List((1, 10), (2, 20))`
- The functions `zip` and `fmap2` are computationally equivalent:

$$\text{zip} = \text{fmap2}(\text{id})$$

$$\text{fmap2}(f^{A \times B \Rightarrow C}) = \text{zip} \circ \text{fmap} f$$

- The functor F is "zippable" if such a `zip` exists

Deriving the `ap` operation from `zip`

- Set $A \equiv B \Rightarrow C$, get $\text{zip}^{[B \Rightarrow C, B]} : F^{B \Rightarrow C} \times F^B \Rightarrow F^{(B \Rightarrow C) \times B}$
- Use `eval` : $(B \Rightarrow C) \times B \Rightarrow C$ and $\text{fmap}(\text{eval}) : F^{(B \Rightarrow C) \times B} \Rightarrow F^C$
- Uncurry: $\text{app}^{[B, C]} : F^{B \Rightarrow C} \times F^B \Rightarrow F^C \equiv \text{zip} \circ \text{fmap}(\text{eval})$
- The functions `zip` and `app` are computationally equivalent:
  - use pair : $(A \Rightarrow B \Rightarrow A \times B) = a^A \Rightarrow b^B \Rightarrow a \times b$
  - use $\text{fmap}(\text{pair}) \equiv \text{pair}^{\uparrow}$ on an fa^{F^A}, get $(\text{pair}^{\uparrow} fa) : F^{B \Rightarrow A \times B}$; then

$$\text{zip}(fa \times fb) = \text{app}\left((\text{pair}^{\uparrow} fa) \times fb \right)$$

$$\text{app}^{[B \Rightarrow C, B]} = \text{zip}^{[B \Rightarrow C, B]} \circ \text{fmap}(\text{eval})$$

- Rewrite this using curried arguments: $\text{fzip}^{[A, B]} : F^A \Rightarrow F^B \Rightarrow F^{A \times B}$; $\text{ap}^{[B, C]} : F^{B \Rightarrow C} \Rightarrow F^B \Rightarrow F^C$; then $\text{ap } f = \text{fzip } f \circ \text{fmap}(\text{eval})$.
- Now $\text{fzip } p^{F^A} q^{F^B} = \text{ap}(\text{pair}^{\uparrow} p) q$, hence we may omit the argument q: $\text{fzip} = \text{pair}^{\uparrow} \circ \text{ap}$. With explicit types: $\text{fzip}^{[A, B]} = \text{pair}^{\uparrow} \circ \text{ap}^{[B, A \Rightarrow B]}$.