Functor and Applicative

FIT2102 Programming Paradigms

S2 2017 Week 8

Preamble

This week, we attack *real* functional programming: Functors and Applicatives. Think of them of generalisation of higher functions. Functors and applicatives are two of the main building blocks of functional programming theory.

Functor and Applicative are *typeclasses* like we saw last week. That is they are properties you apply on types. Types, by themselves, cannot enforce certain properties it is therefore the programmer's task to implement them.

When we implemented other typeclasses we implicitly used a concept called "minimal complete definition." That is, the minimum definition(s) – or function(s) – that are needed to express a property.[1] Functor and Applicative are no different.

[1] E.g., `Eq` needs only `==`, `Ord` can be expressed with `compare` or `Eq` and `<`, etc.

Functor

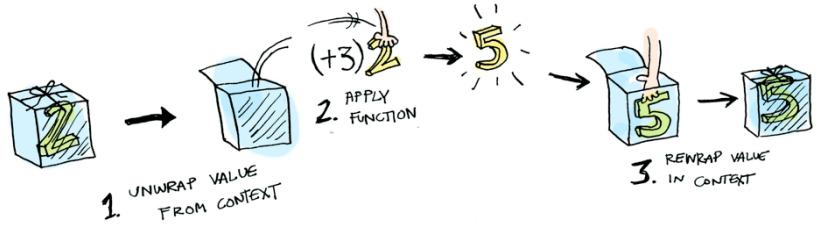

Figure 1: Functor application

Minimal complete definition

The minimal definition of a functor is `<$>` (fmap[2]).

```
(<$>) :: (a -> b) -> f a -> f b
```

[2] Note that `fmap` stands for "functor map" because `map` was already reserved, it is not `flatMap`.

Laws

All instances of the `Functor` type-class must satisfy two laws. These laws are not checked by the compiler. These laws are given as:

1. The law of identity [3]

$$\forall x : (\text{id} <\$> x) \equiv x$$

2. The law of composition

$$\forall f, g, x : (f \circ g <\$> x) \equiv (f <\$> (g <\$> x))$$

So a functor takes a function, an element in a context, applies the function to the element, and returns the result in the context.[4]

[3] Where `id` is the identity function.

[4] If you replace `f` with `[]` you will notice that `fmap` is a general version of `map`; in a way, a type implementing functor is a type over which you can map.

Applicative

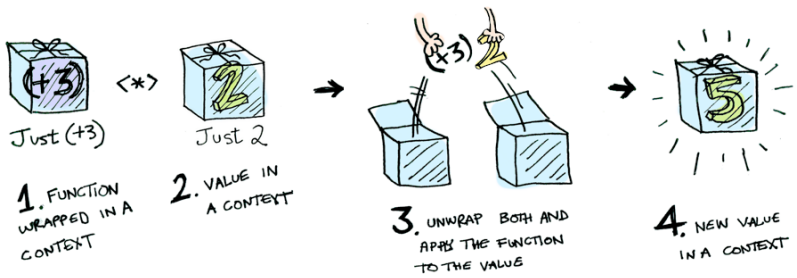

Figure 2: Using apply

Minimal complete definition

The minimal definition of an applicative functor is `pure` and `<*>` (apply):

```
pure  :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Laws

All instances of the `Applicative` type-class must satisfy four laws. These laws are not checked by the compiler.

1. Identity:

```
pure id <*> v = v
```

2. Composition:

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

3. Homomorphism:

```
pure f <*> pure x = pure (f x)
```

4. Interchange:

```
u <*> pure y = pure ($ y) <*> u
```

So, an applicative takes a function within a context, an element within the same context, applies the function, and returns the results in the context.

Exercises

Now that the fundamentals are out of the way, you will implement some functions that leverage the functor and applicative typclasses:

1. `lift`, aka the *cheat function*. `lift` takes a binary function and applies it to two elements wrapped in a context.
2. If you had time, you implemented `flipMaybe` last week. It has one major shortcoming: it is not general. Using `<$>` and `<*>` you can write a general function that takes a list of wrapped elements and returns a wrapped list of elements.[5]
3. `replicate` takes an element in a context and replicates the effect a given number of times.
4. `filtering` is a compound filter function which takes a predicate that also produces an effect.

[5] Think about it as a way to automatically handle failures.

Optional

What does the following code produce?

```
filtering (const $ [True, False])
```

Credits

Images from adit.io in Functors, Applicatives, And Monads In Pictures.