Functional programming: functors and monads

Michał Dominiak
Wrocław University of Technology
Nokia Networks
griwes@griwes.info

Outline

- Noticing patters
- A quick and mostly wrong analogy
- Functors
- A quick shortcut without mentioning Applicative
- Monads
- do-notation (a joke included)

Noticing patterns and turning them into useful abstractions

- *Straight from my previous talk.*
- Some types have common operations. Same example as before: smart pointers, optional, future – all wrap values.
- It’s useful to be able to do similar things in the same way for similar types – genericity.
- People tend to design abstractions and force them upon types – this happens with most Gang of Four patterns.
- Actually useful patterns and abstractions are not *designed* or *invented*; they are *noticed*, or *discovered*.

A quick and mostly wrong analogy

- A functor is a box over a value.
- Or over no value.
- Or over multiple values.
- Do the multiple values have the same type?

- The box allows you to look into it and call a function on the value inside.
- Or not call it when it’s empty.
- Or call it multiple times.
- Or call different functions because the values are actually of different types.

"A monad is a burrito"

- People are trying to "explain" monads with analogies.
- I found this extremely unhelpful.
- The one tutorial that actually explained something in a sensible way *(and not in Haskell; Learn You A Haskell actually has a pretty good explanation of the idea, but in Haskell)* was a series of 10 blog posts...
- ...that went through examples.

- A monad is not a burrito.
- A functor isn't really a box.

optional<T>

- Contains zero or one value.
- You can check whether the value is there or not.
- You can create both an empty optional and an optional containing a value.
- You can call a function on the value inside... kind of (best wrap that in a function):

```
auto transformed = opt
    ? boost::make_optional(f(*opt))
    : boost::none;
```

vector<T>

- Contains zero or more values.
- You can check how many elements are there.
- You can create a vector of arbitrary (sans memory limits) number of elements.
- You can call a function on all the values (and get results!)... though not in an exactly optimal (from semantics point of view) way (best wrap this into a function):

```
std::vector<decltype(f(*vec.begin()))> transformed;
transformed.reserve(vec.size());
std::transform(vec.begin(), vec.end(), std::back_inserter(transformed), f);
```

future<T>

- "Contains" a value that will appear in the future (...or not at all).
- You can check whether the value is there or not.
- You can create a ready future with with promises (and soon with make_ready_future); you can similarly create an exceptional one.
- You can call a function on the value (assuming having .then() already; but otherwise you also can, by creating a thread and waiting on the value – not recommended) – best wrap this in a function:

```
auto transformed = future.then([&f](auto fut) { return f(fut.get()); });
```

Functors

- All those types (and numerous others) are what's known as *functors*.
- *We are consciously not touching mathematical definitions here, which is why this is the only slide where the term "category theory" appears.*
- A functor is a type, wrapping other type(s), that exposes a function called *map* (or *fmap* – historical reasons in the case of Haskell). To avoid possible confusion with std::map, I'm going to keep calling it fmap.
- The type of fmap is, in Haskell notation:

```
fmap :: (a -> b) -> f a -> f b
```

A generic way to write that in C++ is not particularly pleasant.

Functors pt. 2

- In fact, we’ve already seen fmaps! The little snippets of code on previous slides mostly implemented fmap for the types we talked about on each respective slide.
- Variant can also be seen as a vector, since the function object passed to fmap can have an overloaded call operator.
- This isn’t possible (at least not in a generic way) in Haskell.
- Haskell has a Bifunctor typeclass, with its fmap variant, called bimap, looking like this:

```
bimap :: (a -> b) -> (c -> d) -> f a c -> f b d
```

Functors pt. 3

- fmap is great; it follows value semantics, it allows easy function calling on values contained in a functor.
- fmap defined this way returns a functor of the same kind (*in the same category; sorry for lying before*). This means that those functors are *endofunctors* – that is, functors mapping to themselves *(this will come in handy later)*.

A quick shortcut without mentioning [...]

- *I lied again.*
- Applicative functors allow you to wrap the function itself in a functor.
- This could be useful for example when you had an optional function object and an optional value; the function to apply it would be equivalent to:

```
(function && value) ? boost::make_optional(*function(*value)) : boost::none;
```

- From the perspective of this talk, they are not interesting enough.

Monads: `join`

- In the case of some functors, it’s possible to "flatten" *(<- this is a yet another mostly wrong analogy)* them – that’s called *join*. For example, for optional:

```
template<typename T>
boost::optional<T> join(boost::optional<boost::optional<T>> opt)
{
    return opt ? std::move(*opt) : boost::none;
}
```

Monads: `join`

- For vector:

```
template<typename T>
std::vector<T> join(std::vector<std::vector<T>> vec)
{
    std::vector<T> ret;
    // reserve enough space for all elements
    for (auto && v : vec)
    {
        std::move(v.begin(), v.end(), std::back_inserter(ret));
    }
    return ret;
}
```

Monads: `join`

- In general:

```
join :: m (m a) -> m a
```

- This does different things for different functors, and doesn't make much sense for some of them.
- For those functors where this makes sense, we can define a function that calls fmap, then join. For example, there's a *computation step* that takes a value and returns an optional, and an optional from previous steps. fmap in this case would return optional<optional<T>>.
- If you join that, you get optional<T>, which can be simply fmapped on again.

Monads: `mbind`

- A function that does that, in Haskell, is spelled >>=:

```
(>>=) :: m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

- In C++ >>= has wrong associativity for chaining mbinds: `a >>= b >>= c` does a wrong thing in C++ (it's `a >>= (b >>= c)` instead of `(a >>= b) >>= c`).
- Suggestions for which operator would be best in this case?
- C++ mbind is pretty much identical.
- Sometimes it's better to implement mbind, not join; in that case, join becomes an mbind of the identity function.

Monads

- A monad is a functor.
- A monad is used to represent a sequence of steps of computation.
- Monads are "programmable semicolons".

do-notation

- do-notation is a shorthand notation for monadic bind.
- For example:

```
do
    something <- foo
    bar
    baz something
```

becomes

```
foo >>= \something -> bar >>= \_ -> baz something
```

- For example, with the monad in question being Maybe (akin to C++'s optional); if foo is successful, call bar; if both are successful, call baz.
- Wait a second, that looks familiar!

do-notation (and a joke!)

- That is kind of familiar to…

```
try {
    auto something = foo();
    bar();
    baz(something);
}
catch ( /* ... */ ) { /* ... */ }
```

- *Do or do not; there is no try.*