Italian C++ Community

www.italiancpp.org

Seeing Monads in C++

Bartosz Milewski

Just another pattern

- Object
- Facade
- Visitor
- Factory
- Bridge
- Strategy
- Observer
- Flyweight

- Functor
- Applicative Functor
- Monad
- Comonad
- Monoid
- Foldable
- Traversable
- Lens

Data processing practice

- Data comes in boxes, crates, or containers
- Data processing
  - Extract an item
  - Process it
  - Re-pack it

```
for (int i = 0; i < len; ++i)
    w.push_back(f(v[i]));
```

Declarative approach

- Separate packaging from data processing
- Describe “what” rather than “how”
- A slightly better approach:

```
transform(begin(v), end(v), back_inserter(w), f);
```

- Still describes unpacking and re-packing

With ranges

```
int total = accumulate(view::iota(1) |
                       view::transform([](int x){return x*x;}) |
                       view::take(10), 0);
```

- No extraction or re-packing!

Change of perspective

- Take a function that acts on items

  `[](int x){return x*x;}`

- Lift it using **`view::transform`**

  `view::transform([](int x){return x*x;})`

- Lifted function acts on ranges of items (1, 2, 3, …)

  `view::iota(1) | view::transform([](int x){return x*x;})`

- Returns a range of items (a stream of squares, 1, 4, 9, …)

Lazy ranges

- Infinite range 1, 2, 3, 4,…

std::iota(1)

- We can’t process it eagerly!
- But we can transform it lazily

- Lazy processing works for data that can’t fit in memory
  - infinite ranges
  - big data
- LINQ

Functor on ranges

1. “Function” on types (a template)
   - Take any type **`T`**, produce the type **`range<T>`**
2. “Function” on functions (**`view::transform`**)
   - Take any function **`f`** from **`t1`** to **`t2`**
   - return a function from **`range<t1>`** to **`range<t2>`**
3. Preserves identity:

   ```
   view::transform(id) = id
   ```
4. Preserves composition (fusion)

Functor

1. "Function" on types (a template)
   - Take any type `T`, produce the type `Functor<T>`
2. "Function" on functions (often called `fmap`)
   - Take any function `f` from `t1` to `t2`
   - return a function from `Functor<t1>` to `Functor<t2>`
3. Preserves identity:

   ```
   fmap(id) = id
   ```
4. Preserves composition (fusion)

   ```
   fmap(compose(f, g)) = compose(fmap(f), fmap(g));
   ```

What else is a functor?

- **`std::expected`** (proposal)
- Contains either a value of type T or an error
- Usage:

```
expected<exception_ptr, double>
  safe_divide(double x, double y);
```

```
auto w = safe_divide(x, y).fmap(square);
```

Future

`std::future<T> fut = std::async(…);`

- a package that may eventually contain a value of type T
- Processing this value:

```
auto x = fut.get(); // may block
auto y = f(x);
```

- Or

```
auto y = fut.next(f).get(); // proposed
```

- **`next`** takes a function and lifts it to futures
- future is a functor

On towards monads

Pointed functors

- Ability to package a value
  - Singleton container: `vector<int>{5}`
  - Lazy range: `single`, `yield`, `repeat`
  - `expected` with a value (rather than error)
  - `make_ready_future` (proposed)

- Returning a value from a function that returns a package
- Terminate recursion

- Standard names: `pure`, `return` (confusing in C-lookalikes)

Applicative functors

- Ability to apply multi-argument functions to multiple packages at once

```
expected<exception_ptr,int> f(int i, int j, int k)
{
  return fmap(plus,
              safe_divide(i, k),
              safe_divide(j, k));
}
```

Applicative range 1

- **`zip_with`** algorithm
  - takes a function of n arguments
  - takes n ranges
  - applies function to n zipped elements at a time
  - returns a range
- “pure” implemented as **`repeat`**

Applicative range 2

- Apply a function to all combinations of arguments
- Could be implemented as a variadic `transform`
- Particularly useful with lazy ranges
- Can be implemented using a monad

Applicative future

- Very useful in parallel computation
- Apply a multi-argument function to multiple futures
- Return a future that becomes ready when all arguments are ready (and the function applied to them)
- Could be implemented using proposed **`when_all`**, but very clunky
- Problem: futures may return exceptions

Applicative functor

1. Is a functor (defines `fmap` or equivalent)
2. Is pointed (defines `pure` or equivalent)
3. Defines action of multi-argument functions (or variadic `fmap`)
4. Must obey a few simple axioms

Monad: Composing package factories

Range factories

```
range<Node> parseXML(…);
range<Property> Node::getProps();
```

- Traditional approach

```
for(Node n: parseXML(…))
{
  auto props = n.getProps();
  for(Property p: props)
     process(p);
```

Flattening

- Range is a functor

```
parseXML(…) | transform(&Node::getProps)
```

- Problem: returns `range<range<Property>>`
- Solution: flatten

```
range<T> flatten(range<range<T>>);
```

```
parseXML(…) | transform(&Node::getProps)
            | flatten | process
```

Bind

- **`fmap`** followed by **`flatten`** = **`bind`**

```
range<T> bind(range<U>, function<range<T>(U)>);
```

- **`bind`** is also called **`for_each`**

```
std::map<int, std::string> m;
m = view::for_each(view::ints(0,4), [&](int i) {
            return yield(std::make_pair(i, to_string(i)));
        });
```

LINQ

- fmap is called **`Select`**
- bind is called **`SelectMany`**

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
      this IEnumerable<TSource> source,
      Func<TSource, IEnumerable<TResult>> selector)
```

Monad

- Applicative functor
  - fmap
  - pure (or return)
- Either flatten (a.k.a. join) or bind
  where bind is a combination of fmap and flatten
- Plus a few axioms

Expected monad

```
// i/k + j/k
expected<exception_ptr,int> f2(int i, int j, int k)
{
  return mbind(safe_divide(i, k) ,[&r](auto s1) {
    return mbind(safe_divide(j, k),[&r](auto s2) {
      return s1 + s2;
    });
  });
}
```

future monad

- Overloading of next
- as fmap

```
future<T> future<U>::next(function<T(U)>)
```

- as bind

```
future<T> future<U>::next(function<future<T>(U)>)
```

Example: asynchronous `file_open` followed by asynchronous `read`.

Problems

- Lack of overall abstraction
- Random naming
  - fmap, transform, next, Select
  - pure, single, yield, await, make_ready_future
  - bind, mbind, for_each, next, then, SelectMany
- Lack of syntactic sugar
  - Haskell do notation
  - Resumable functions?

Conclusions

The same pattern applicable to many problems

- ranges
- lazy ranges
- expected
- future
- LINQ (IEnumerable)
- many more…