Re-discovering Monads in C++

Bartosz Milewski, Бартош Милевски
Twitter: @BartoszMilewski
Novosibirsk, November 2014

Data in Boxes

- Smart pointers: unique_ptr, shared_ptr
- Optional, expected
- Containers, ranges
- Streams, channels (telemetry, stock quotes)
- Futures
- Database queries

In and Out of Boxes

- Common patterns in data processing
  - Extract data from box
  - Process it
  - Put it back in a box

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

Separate Concerns

- Separate extraction/repacking from processing
- Get rid of unessential variables
- Iterators and algorithms: better but not much

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

Declarative Approach

- Describe “what” rather than “how”
- Example: ranges

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

Functor

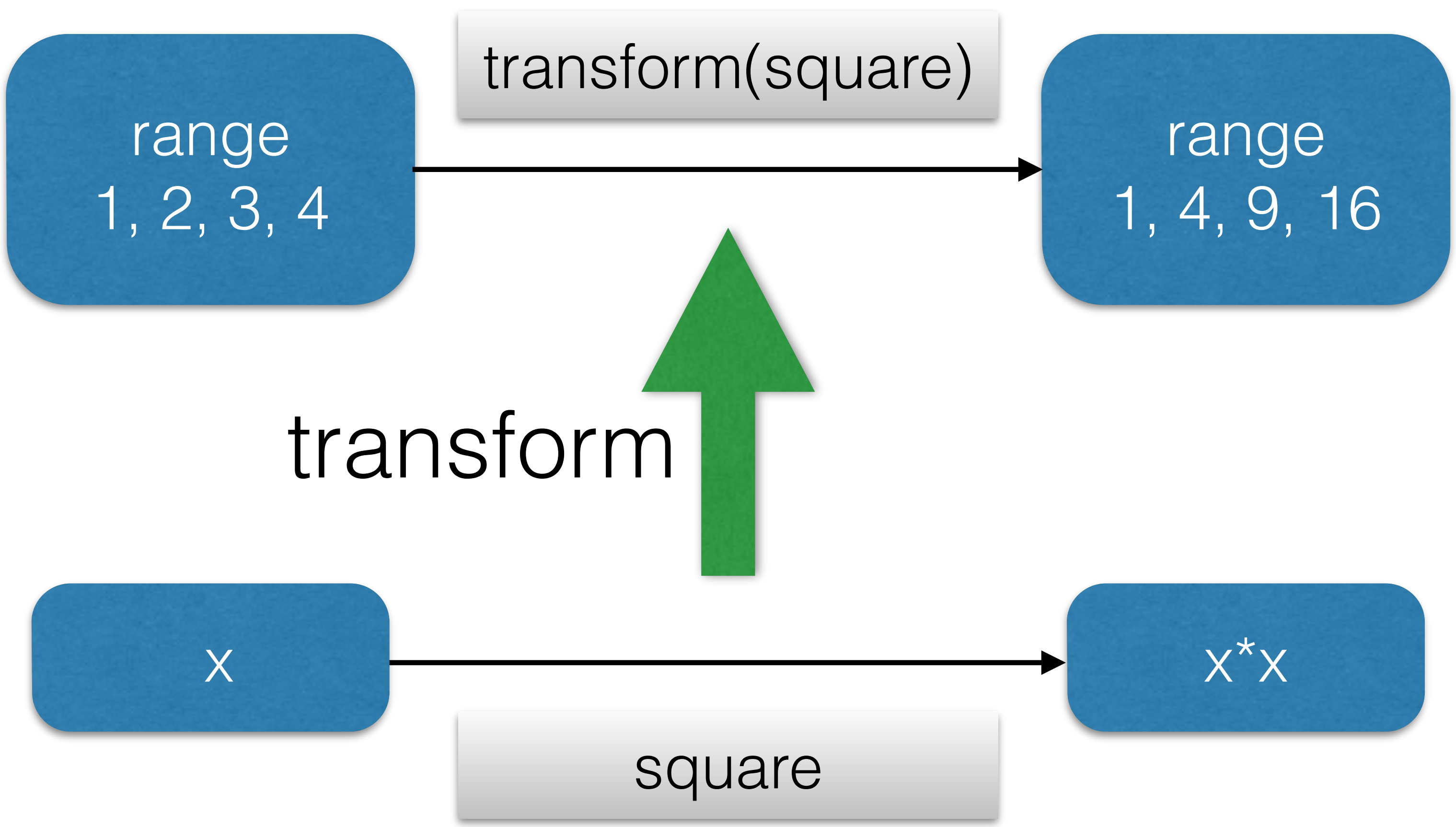
Change of Perspective

- 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, 4, …}
  ```
  view::iota(1) |
  view::transform([](int x){return x*x;})
  ```

- Returns a range of items {1, 4, 9, 16, …}


transform(square)
range
1, 2, 3, 4
range
1, 4, 9, 16
transform
x
x*x
square

Laziness

- Infinite range {1, 2, 3, 4, …}
  `std::iota(1)`
- Can't process it eagerly!
- Transform it lazily, on demand
- Working with data that can't fit in memory
  - infinite ranges
  - big data
  - database queries: LINQ

Functor Pattern

- Function on types (type template)
  - Take any type **`T`** and produce **`Functor<T>`**
- Function on functions
  - Take any function **`f`** from **`t1`** to **`t2`**
  - Return a function from **`Functor<t1>`** to **`Functor<t2>`**
- Preserve identity
- Preserve composition

Range as Functor

- Function on types: **`T`** to **`range<T>`**
- Function on functions: **`view::transform`**

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

future as Functor

- A box that may eventually contain a value of type **`T`**
  `std::future<T> fut = std::async(…);`

- Processing this value:
  ```
  auto x = fut.get(); // may block
  auto y = f(x);
  ```

- Lifting a function using **`next`**:
  ```
  auto y = fut.next(f).get(); // proposed
  ```

- **`fut.next(f).next(g).get();`**

Pointed Functor

Lifting Values

- A pointed functor is a functor
  - Function on types
  - Function on functions (lifting functions)
- Lifting a single value
  - A template function from **`T`** to **`Functor<T>`**

Examples

- Containers: Creating a singleton container **`vector<double> {3.14}`**
- Lazy range: **`view::single`**
- Future: **`make_ready_future`** (proposed)
- Usage:
  - return value when **`Functor`** expected
  - terminate recursion

Applicative Functor

Lifting Multi-Arg Functions

- Applicative functor is a pointed functor:
  - Function on types
  - Lifting functions and values
- Lifting multi-argument functions to functions taking multiple Functor arguments (of different types)

Lazy Applicative Range (1)

- Example: Apply 2-arg function **`plus`** to two ranges **`zip_with(plus, r1, r2);`**
- Variadic template **`zip_with`**
  - Takes a function **`f`** of n-arguments
  - Takes n ranges of corresponding types
  - Applies **`f`** to corresponding elements (until one of the ranges runs out)
- Value lifting through **`repeat`** (infinite range)

Lazy Applicative Range (2)

- Apply n-argument function to all combinations of values from n-ranges
- Could be an overload of **`view::transform`**?
- Implemented as nested loop
- Value lifting through **`single`**

Applicative Law

- Lifting a 1-argument function
  - As function: **`transform(f, r)`**
  - As value (1): **`repeat(f)`**
    - Applying it to range: **`zip_with(apply, repeat(f), r)`**
  - As value (2): **`single(f)`**
    - Applying it to range: **`transform(apply, single(f), r)`**

Applicative future

- Apply multi-argument function to multiple futures
- Not planned, instead when_all
- To apply the function, all arguments must be ready
- Problem: futures may return exceptions, so it’s a combination of applicatives

Monad

Functor Factories

- Library writer provides functions that return **`Functor`**s
- User wants to define their own
- How do you chain Functor factories?

```
Functor<t2> makeT2(t1);
Functor<t3> makeT3(t2);
```

Future example

- Composing asynchronous calls
```
future<HANDLE> async_open(string &);
future<Buffer> async_read(HANDLE fh);
```

- Option 1: call **`get()`** twice

- Option 2: call **`next`**. Problem: Double future.
```
future<future<Buffer>> ffBuf =
async_open("foo").next(&async_read);
```

- Solution: Collapse double future using **`unwrap()`**
```
async_open("foo").next(&async_read).unwrap()
                 .next(&async_process);
```

- Better solution (proposed): Overload **`next`** for functions returning futures

Range Example

- Functions returning ranges: Range factories
- Applying range factory to range factory using transform results in a range of ranges
- Collapsing using **`flatten`**
- Combination of **`transform`** and **`flatten`** called **`for_each`**

Pythagorean Triples

```
auto triples =
  for_each(ints(1), [](int z) {
    return for_each(ints(1, z), [=](int x) {
      return for_each(ints(x, z), [=](int y) {
        return yield_if(x*x + y*y == z*z,
            std::make_tuple(x, y, z));
      });
    });
  });
```

- **`yield`** is the same as **`single`** inside **`for_each`**
- **`yield_if`** is conditional **`yield`** (monad plus)

Monad

- Monad is an applicative functor
  - function on types
  - function on functions: function lifting
  - value lifting
  - multi-argument function lifting
- Flatten (reduce double Functor to single Functor)
- Or Bind (combination of lifting and flatten)
- And some monad laws

Current Problems

- Lack of an overall abstraction (a monad template)
- Random naming
  - fmap, transform, next, then, Select (LINQ)
  - pure, single, yield, await, make_ready_future
  - bind, for_each, next, then, SelectMany
- Need syntactic sugar, like Haskell **`do`** notation
  - Resumable functions (proposed)

Conclusion

- The same pattern fits many problems
  - ranges, lazy ranges
  - optional, expected
  - futures
  - LINQ (IEnumerable)
  - state, continuation, input, output
  - many more…