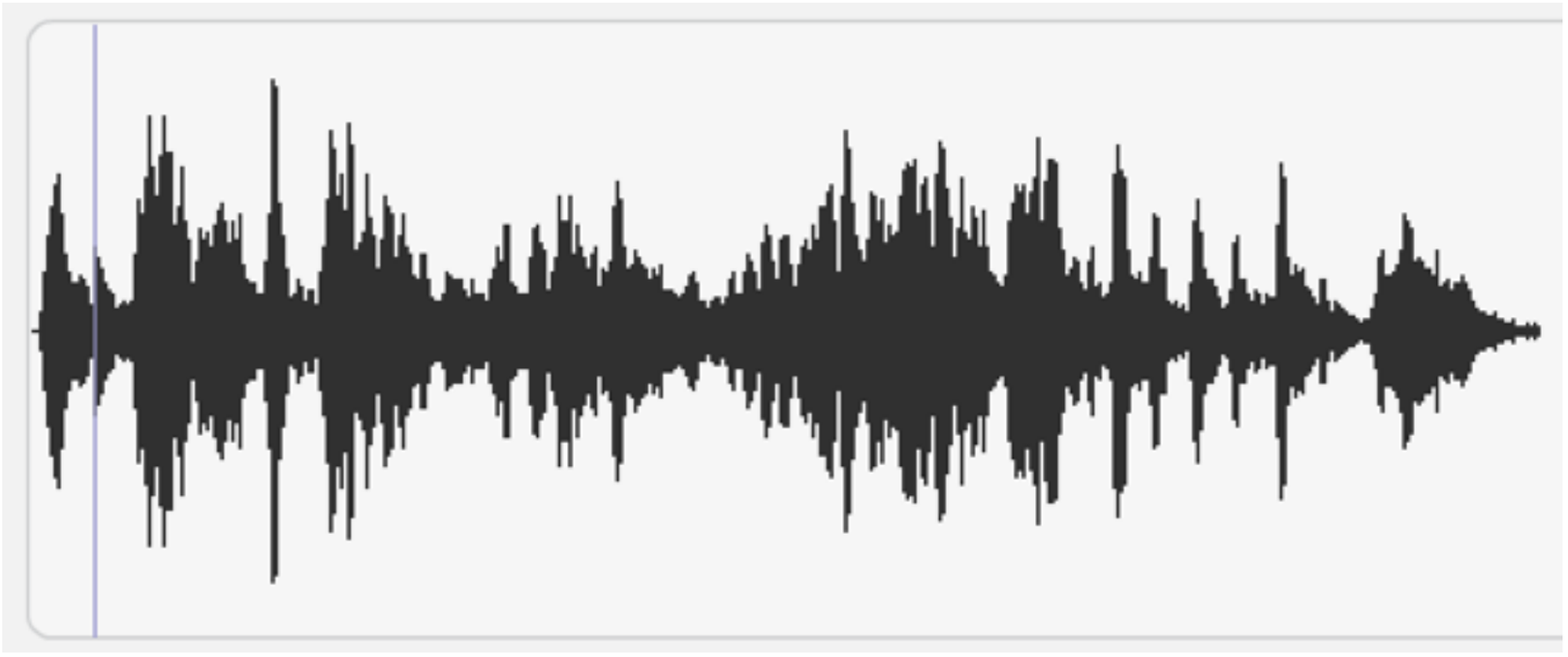
Fun with Categories

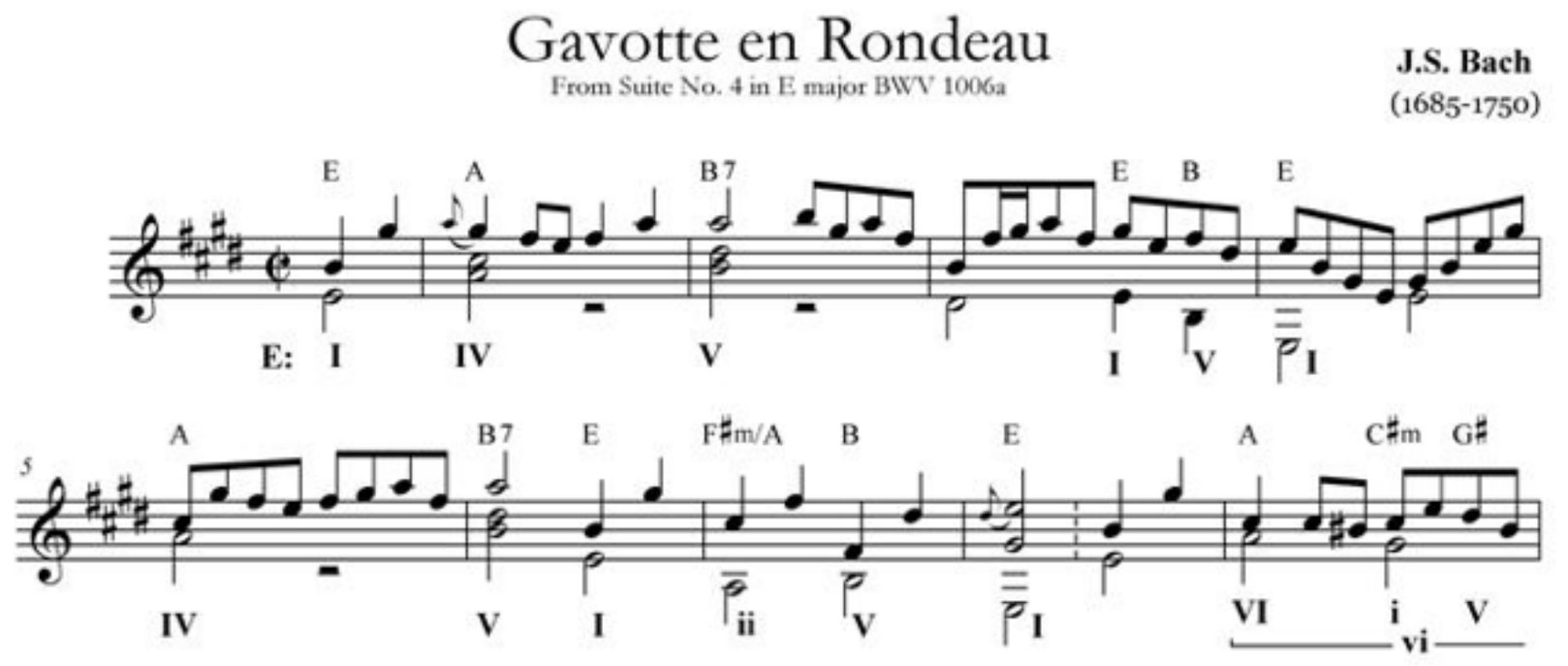
Bartosz Milewski
Bologna, March 2015

The Essence of Programming

- Composition
- Problem Solving: Decomposition
- Coding: Re-composition

Low Level/High Level

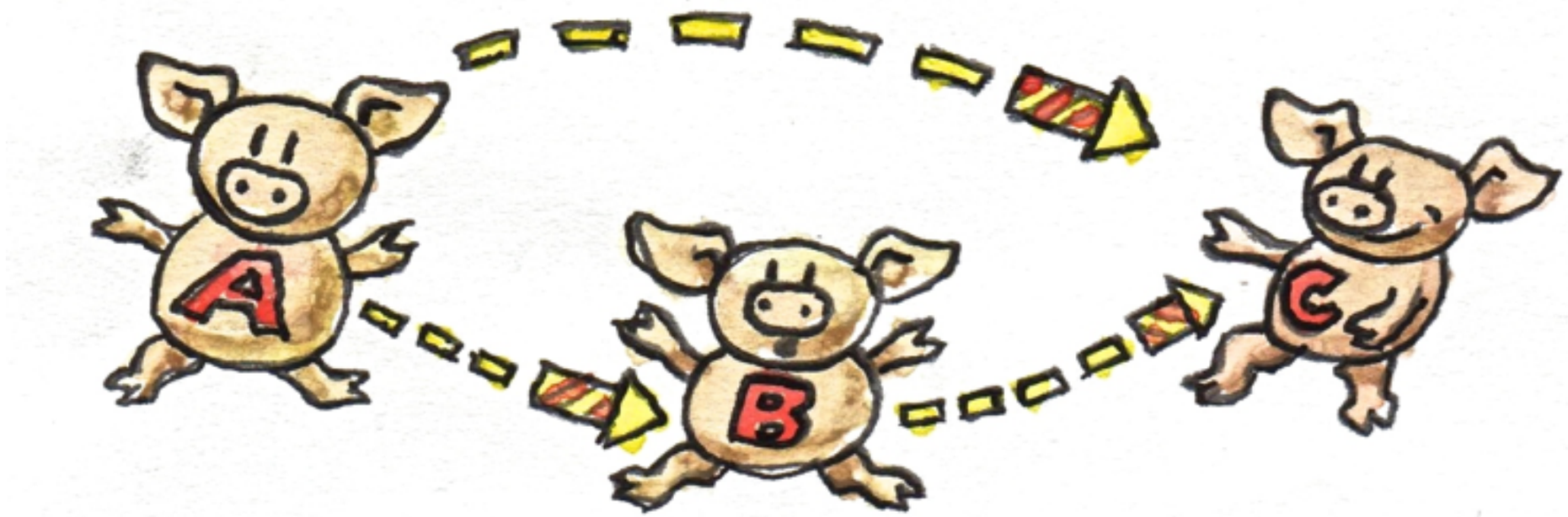




Category

- Embarrassingly simple concept
- Objects
- Arrows between objects

Composition



- Arrow from A to B. `f :: A → B`
- Arrow from B to C. `g :: B → C`
- Composition, g after f. `g ∘ f :: A → C`
- Associativity. `(f ∘ g) ∘ h = f ∘(g ∘ h)`

Identity

- For every object A
- `id :: A → A`
- `f :: A → B`

  `f ∘ id = f`
- `g :: B → A`

  `id ∘ g = g`

Monoid

- Embarrassingly simple concept
- One object M
- Arrows `f :: M → M`
- All arrows composable
- One identity arrow

Examples

- Addition + 0
- Multiplication * 1
- String concatenation + “”
- Logging, Gathering Data, Auditing

Types and Functions

- **Set**: category of sets and functions
- Objects: Types = Sets of values
- Arrows: Functions from one type to another
- Composition `g . f`

```
C g_after_f(A x) {
    B y = f(x);
    return g(y);
}
```

```
g_after_f :: A -> C
g_after_f x =
    let y = f x
    in g y
```

Pure Functions

- No side effects
- When called with same arguments, returns same values (referential transparency)
- Can be memoized
- The only dependencies in code are through composition

Side Effects

Auditing

- Simplest example: auditing
- Sequence of functions
  - `getKey(password)->key`
  - `withdraw(key)->money`
- Each function leaves audit trail

Global Auditor

```
string audit;

int logIn(string passwd){
   audit += passwd;
   return 42;
}

double withdraw(int key){
    audit += "withdrawing ";
    return 100.0;
}
```

- Poor scaling, maintenance, flexibility

In/Out Auditor

```
pair<int, string>
logIn(string passwd, string audit){
  return make_pair(42, audit + passwd);
}

pair<double, string>
withdraw(int key, string audit){
  return make_pair(100.0
                   , audit + “withdrawing ”);
}
```

- Doesn’t memoize well
- Each function has access to full log and must know how to accumulate data

Out Auditor

```
pair<int, string>
logIn(string passwd){
  return make_pair(42, passwd);
}

pair<double, string>
withdraw(int key){
  return make_pair(100.0
                   ,"withdrawing ");
}
```

- Each function responsible only for its data
- But how to compose such functions?

Writer

```
template<class A>
using Writer = pair<A, string>;

Writer<int> logIn(string passwd){
  return make_pair(42, passwd);
}

Writer<double> withdraw(int key){
  return make_pair(100.0
                   ,"withdrawing ");
}
```

- How to compose?

Composition

```
Writer<double> transact(string passwd){
  auto p1 logIn(passwd);
  auto p2 withdraw(p1.first);
  return make_pair(p2.first
                , p1.second + p2.second);
}
```

- Audit trail accumulated “between” calls

Abstracting Composition

```
template<class A, class B, class C>
function<Writer<C>(A)> compose(function<Writer<B>(A)> f
                               ,function<Writer<C>(B)> g)
{
    return [f, g](A x) {
        auto p1 = f(x);
        auto p2 = g(p1.first);
        return make_pair(p2.first
                         , p1.second + p2.second);
    };
}
```

- Composing two functions using a higher order function

Using Composition

```
Writer<double> transact(string passwd){
  return compose<string, int, double>
           (logIn, withdraw)(passwd);
}
```

- Between calls
- Explicit type annotations

Type Inference

```
auto const compose = [](auto f, auto g) {
    return [f, g](auto x) {
        auto p1 = f(x);
        auto p2 = g(p1.first);
        return make_pair(p2.first
                         , p1.second + p2.second);
    };
};
```

- C++14 generalized lambdas with return type deduction

Back to Categories

Objects, Arrows, Composition

```
C g_after_f(A x) {
    B y = f(x);
    return g(y);
}
```

- Objects: Data Types

- Arrows: (Pure) Functions

- Composition: result of one function = argument of another function

Category of Embellished Functions

- Objects = types like A, B, C
- Arrow from A to B = function from A to some type that depends on B (*Embellished* type)
  - For instance: `pair<B, string>` (or `Writer<B>`)
- Composition:
  - `f :: A → Writer<B>`
  - `g :: B → Writer<C>`
  - `g ∘ f :: A → Writer<C>`

Example of Kleisli Category

```
auto const compose = [](auto f, auto g) {
    return [f, g](auto x) {
        auto p1 = f(x);
        auto p2 = g(p1.first);
        return make_pair(p2.first
                         , p1.second + p2.second);
    };
};
```

```
template<class A>
Writer<A> identity(A x) {
    return make_pair(x, "");
}
```

Controlling Side Effects

- Pure functions
- Side effects through composition
- Write your code using embellished functions
- Glue it using composition combinators
- Writer generalizes to any **monoid**

Haskell

Writer in Haskell

```
type Writer a = (a, String)
```

```
(>=>) :: (a -> Writer b) -> (b -> Writer c)
      -> (a -> Writer c)
```

```
f >=> g = \x -> let (y, s1) = f x
                    (z, s2) = g y
                in  (z, s1 ++ s2)
```

```
identity :: a -> Writer a
identity x = (x, "")
```

```
transact :: String -> Writer Double
transact = logIn >=> withdraw
```

List in Haskell

```
(>=>) :: (a->[b]) -> (b->[c]) -> (a->[c])

f >=> g = concat . map g . f
```

```
identity :: a -> [a]
identity x = [x]
```

- Lots of other examples

Kleisli in Haskell

- Every monad gives rise to a Kleisli category

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c)
                 -> (a -> m c)

f >=> g = \x -> f x >>= g
```

```
identity = return
```

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Conclusion

- Programming is about composition
- Category theory is about composition
- Composing side effects: Kleisli category
- Kleisli category = monad