Haskell Design Patterns for Genericity and Asynchronous Behavior

Sherri Shulman

The Evergreen State College

sherri@evergreen.edu

October 2, 2015

Overview

Introduction

This talk focuses on the development of a series of related Haskell design patterns that support genericity and (ultimately) asynchronous behavior.

My background is primarily in Haskell and C, but in general I teach programming language design and am interested in how different language features and different language paradigms support different problem solutions.

Why Haskell?

- Statically typed
- Strongly typed
- Referentially transparent
- First class functions
- Very expressive type language
- Orthogonal language features
- Clear semantics
- No side effects

How does this impact C++?

C++ vs Haskell

Although C++ is a very different language than Haskell, the language features and design patterns in Haskell can promote ideas about how to structure solutions. In some sense Haskell can become the "design language" and C++ the "implementation language"

Support for C++ Concepts

These patterns and language primitives can also suggest ways in which a language like C++ might evolve to include these language features (C++ concepts!)

Current interest in Haskell in the C++ community

Bartosz Milewski's blog on some Haskell patterns with associated suggested implementations in current C++

Haskell Features

Haskell Features

1. Type classes
2. Algebraic data types
3. Functors
4. Monads
5. Continuations

These features are used to support genericity and high level abstractions that promote reuse and a particular style of computation.

Type Classes

1. What is a type class and how does it support genericity?
2. How does it differ from interfaces or abstract classes?
3. Does it have inheritance? Default definitions? Dependencies?
4. How does it support genericity?

Type Classes

```
>       type List x = [x]
>       class HasEmpty x where
>           empty :: x -> Bool

>       class Hashable x where
>           hash :: x -> Bool

>       class (HasEmpty x, Hashable (Element x)) =>
>         Hashset x where
>           type Element x
>           size :: x -> Int

>       almostFull :: Hashset t => t -> Bool
>       almostFull h = False
```

Instances of Type Classes

```
>       instance Hashable Int where
>           hash x = x

>       instance HasEmpty (IntMap k) where
>           empty = Data.IntMap.null

>       instance Hashable k =>
>         Hashset (IntMap (List k)) where
>           type Element (IntMap (List k)) = k
>           size m = Data.IntMap.size m

>       h :: IntMap (List Int)
>       h = Data.IntMap.empty
>       test = almostFull h
```

Full code: Figure 1

Type class notes

There are three type classes. We restrict the types x that model *Hashset* to those types x that model *HasEmpty* and *Hashabale*.

This data type IntMap (List k)) is still not a concrete type since k is a type parameter but so long as k models *Hashable* we are type correct and type safe.

The example instantiates k with *Int*, resulting in a full implementation. But k can be instantiated with any type with the appropriate constraint, with full type safety.

A C++ Implementation using concepts

```
// concept
concept Hashset<typename X> : HasEmpty<X> {
    typename element;
    requires Hashable<element>;
    int size(X);
}

// modelling
template<Hashable K>
    concept_map Hashset<intmap<list<K>>> {
int size(intmap<list<K>> m) { ... }
}
```

A C++ Implementation using concepts (continued)

```
// algorithm
template<Hashset T>
    bool almostFull(T h) { ... }

// instantiation
intmap<list <int>> h;
bool test = almostFull(H);
```

Here *Hashset* is a subclass of *HasEmpty* with an independent constraint that element be *Hashable*.

C++ comments

The two implementations and their semantics are parallel. Unfortunately concepts are not yet a part of standard C++ implementations, although I hope they will be soon.

They provide an explicit type-level constraint that serves both to explicitly communicate requirements, but also provide more static information.

Type classes do not have inheritance of code, although the context constraints enforce an inheritance of type expectations.

They clearly support genericity and reusability in their ability to construct hash maps from with a variety of types assuming those types meet required capability: there is a checkable predicate that prevents instantiations for which the implementation would not make sense.

Algebraic Data Types

1. What is an Algebraic Data Type?
2. How does it differ from classes and subclasses? Advantages?
3. How does it support genericity? reusability?

An example using Algebraic Data Types in Haskell

For this example we'll use a design and implementation of an evaluator for a lambda calculus like language.

In Haskell we might implement this using an algebraic data type specifying what a term looks like.

We'll do this in two ways: one using standard algebraic data types and one using GADTs (Generalized Algebraic Datatypes).

Algebraic Data Type

If we want to implement a lambda calculus language (untyped) we might do this directly using an algebraic data type (also known as a discriminated union):

```
>       data Term = Var String | Abs String Term |
>            App Term Term
```

Algebraic Data type-like implementation in C++

This can be implemented similarly in C++ using abstract and derived classes:

```
struct Term {virtual ~Term() {}, };
struct Var : Term {std::string name;};
struct Abs : Term {Var & var, Term & body;};
struct App : Term {Term & lfunc; Term & arg; } ;
```

Pattern Matching

An algebraic data type uses a discriminated union: it enumerates all the ways in which a value in the type can be constructed.

If we write an evaluator for this language in Haskell we have to enumerate all the cases for each kind of element:

```
> eval (Var s) = ...
> eval (Abs v t2) = ...
> eval (App g1 g2) = ...
```

The Oop approach to evaluation

The C++ implementation uses classes and subclasses. Each of the kinds of term is implemented in a subclass. In order to implement an evaluator, each subclass will have its own eval method.

How do these two approaches compare?

What if we decide that we want a new kind of Term? Perhaps a constant.

Extending an algebraic data type

```
>       data Term = Var String |Abs String Term |
>            App Term Term | Cons Int

>       eval (Var s) =   ...
>       eval (Abs v t2) =   ...
>       eval (App t1 t2) =   ...
>       eval (Cons c) =   ...
```

We will have to change eval (and any other functions that are written using the data type Term) to add the new kind of Term.

Extending an Oop design

In C++ we can add the new subclass *Cons*:

```
>        struct Cons : Term {int val;}
```

After this, we will only have to add a new eval function to the Cons class in order for the evaluator to be extended (and similarly for any other behaviors on Terms.)

Adding behaviors

What if we want to add a behavior? As well as eval, perhaps we want a pretty printer. In the algebraic data type design, we only have to add the pretty printer, with all its cases:

```
>       prettypr (Var s) = ...
>       prettypr (Abs v t2) = ...
>       prettypr (App t1 t2)
```

On the other hand, in the OOP design we'll have to add a pretty printer method to each subclass.

This is sometimes called the Row vs Column effect: in both styles we have to change something. It's easier to add behavior to the algebraic data type design and harder to add a new kind of value. It's easier to add a new kind of value in the OOP design but harder to add new behavior.

Adding Discriminated Unions

Can we get some of the features of discriminated unions in C++? The desired features are 1) the ability to enumerate the possible allowable values (or types in OOP) 2) the requirement that the user define behaviors for all possible types of a discriminated union or get a compile time type error.

Discriminated unions in Oop

The "standard" way to implement discriminated unions is to use an enum type for a tag and a union to describe the alternate values. This approach is error prone; it is hard to retain coherence between the tag, the data, and the functions that use them. There is no support in the language to ensure that these components are coherent.

```
struct DiscType {
    enum TypeTag
{typeVar, typeAbs, typeApp, typeEmpty} tag;
    union {
      char* var;
      struct Abs abs;
      struct App app;
    };
};
```

(Note there is a slight abuse of the enum type above to make it similar to the previous example. But see typelists below.)

An Oop solution using typelists

Any function wanting to implement a behavior (eval or pretty printer) would have to inspect the tag, access the appropriate field, etc. Moreover, the type checker couldn't check for type-incorrect access since it really doesn't know that the enum is connected in any way to the union.

A more expressive approach that embodies type safety, allows user defined types, and is more dynamic is described in [Alexandrescu, 2002]:

```
template <class TList>
class Variant {
    ....
};

typedef Variant<
    TYPELIST_4(typeVar, typeAbs, typeApp, typeEmpty)
```

(Assuming that these user-defined types have been defined.)

What must an implementation of Variant do?

The details of the implementation of these discriminated unions are beyond the scope of this presentation. But now the task of ensuring that functions using the discriminated union are completely defined is possible statically.

But such a variant must include:

1. Constructors that accept any type in the typelist.
2. Assignment operators that similarly accept any type.
3. A type safe way to convert to any of the type in the typelist.
4. A way to determine the actual type stored in the variant. (Decoherence)
5. A way to convert/coerce a value appropriately.
6. a way to determine the storage needs (size and alignment).

In addition, the availability of pattern matching would be a plus. The OOP way to do pattern matching can be implemented via the visitor pattern, which is not nearly as direct.

GADTs

We can step up a bit in abstraction and consider how to encapsulate specific type information (generalized algebraic data types). The best way to describe this is with an example. Suppose that we want a term language that evaluates expressions to a value domain that contain more than one kind of value (for instance booleans and ints).

```
>       data Term = Lit Int | Succ Term | IsZero Term |
>            If Term Term Term | Pr Term Term
```

The problem is that an evaluator would return a value that is either an Int or a Boolean or a Pair. There is no information that connects these values: they can't be unified.

The usual way to do this is to introduce a new data type for the values.

Gadts

Our evaluator would have to do the following:

```
>       eval (IsZero t) = eval t == 0
>       eval (Succ t) = 1 + eval t
>       ...
```

This won't type since in one case it returns a Boolean and the other an integer. In order to unify these two disparate types, we can introduce a value domain:

```
>       data Val = VInt Int | VBool Bool | Vpr Val Val
```

Now we can write an evaluator that always returns a *Val*. This can be implemented in OOP using a class and subclass (it's another algebraic type). But even with this approach it is possible to construct types that are correctly formed syntactically but incorrectly typed. For instance the following is type correct according to Haskell but is still meaningless:

```
>       t1 = Succ (IsZero (Lit 0))
```

t1 is syntactically correct and will be passed by the Haskell type checker.

Gadts

Generalized algebraic data types introduce a new type variable (an existential) that asserts for any term there is a type that represents its value:

```
>       data Term a where
>         Lit    :: Int -> Term Int
>         Succ   :: Term Int -> Term Int
>         IsZero :: Term Int -> Term Bool
>         If     :: Term Bool -> Term a -> Term a ->
>                   Term a
>         Pair   :: Term a -> Term b -> Term (a,b)
```

This new Term introduces a type variable. Now we can refine each of the different kinds of terms to the kind of value it embodies. In this new term we would not be able to construct the previous term because IsZero will be constrained to apply only to Ints. Implementing this kind of functionality is explored in [Kennedy, Russo, 2005]

Algebraic Data Types vs Oop

The advantages and disadvantages of algebraic datatypes vs classes/subclasses are not quite as clear as one might like. Algebraic data types have more precision (there is no need to consider subtype polymorphism, we don't have to query what type an object actually is). Algebraic data types can enforce that each function covers all cases and inform the programmer about any missed cases statically.

On the other hand, algebraic data types are less dynamic: we have to consider all the kinds of values that may occur. Adding new kinds of values is easier in a subclass hierarchy (potentially at increased cost in adding behavior).

However, generalized algebraic data types add to the expressivity of the type system and allow more information to be captured in the type allowing more type errors to be detected statically.

Functors

Moving up from type classes and algebraic data types, Haskell Functors capture the idea that something is mappable. A mappable "thing" is a generalization of the map function over lists with the type

```
>       map :: (a -> b) -> [a] -> [b]
```

So map is a function that takes a function mapping values in type *a* to values in type *b*. It then takes a list of values of type *a* and produces a list of values of type *b*, each converted by applying the given function.

A Functor then is something that defines a mapping function (named *fmap*) defined as a type class:

```
>       class Functor f where
>         fmap :: (a -> b) -> f a -> f b
```

If *f* is a Functor, then it has a function *fmap* that takes a mapping function and a Functor holding a value (or values) of type *a*, and produces a Functor holding a value (or values) of type *b*.

Functors

You can think (very generally) of a Functor as something that holds a value and provides this mapping function. A list is an instance of a Functor:

```
>       instance Functor [] where
>         fmap = map
```

Here is a common type in Haskell that embodies the idea that a computation may fail:

```
>       data Maybe a = Just a | Nothing
```

(A computation either computes a value of type *a* or Nothing). The Maybe type is a functor:

```
>       instance Functor Maybe where
>          fmap f (Just x) = Just (f x)
>          fmap f Nothing = Nothing
```

Functors

So far functors don't seem to do much other than create a uniform way to look at things that can mapped (lists, trees, pairs, and so on.) There is a limitation here: we expect the Functor to take one argument. However we can use partial application to bridge this constraint.

An advantage of this uniform way to look at mappable things is that we can compose them or combine them because we know they have this uniform structure. Another functor that may seem unusual is the arrow constructor: $((\rightarrow)r)$. This takes some getting used to, but $\rightarrow$ is the constructor for a function type, here written in prefix form.

A function type is $r \rightarrow b$ for arbitrary types r and b. If we "partially" apply the type to the first argument, we get a constructor of one argument as required for functors. (For instance $((\rightarrow)Char)$ is the type of a function that takes a Char to a value of some other type. Similarly $((\rightarrow)Int)$ is the type of a function that takes an Int to a value of some other type.

Functors cont

Here is the definition of the functor for arrows:

```
>       instance Functor ((->) r) where
>         fmap f g = (\x -> f (g x))
```

So the fmap takes two functions: g is a function that maps an r to some other type, let's call it a: $((\rightarrow)r\ a)$. f is a function of type $a \rightarrow b$. We return a new function that takes an r (which we know since g is applied to the x), and returns an a. We then apply f to get a b. so $\lambda x \rightarrow f(gx)$ has the type $((\rightarrow r)b)$. So it converts a function of type $r \rightarrow a$ to one of type $r \rightarrow b$. As a matter of fact, this is just function composition (as the definition indicates). So we could just say:

```
>       instance Functor ((->) r) where
>         fmap = (.)
```

Functor laws

Functors obey some laws:

1. fmap id = id
2. fmap (f . g) = fmap f . fmap g or equivalently fmap (f.g) F = fmap f (fmap g F)

Ignoring Nothing for brevity, we can demonstrate this for the Maybe type:

```
fmap id (Just x) = Just (id x) = Just x
fmap (f.g) (Just x) = Just ((f.g) x) = Just (f (g x))
(fmap f . fmap g) (Just x) = fmap f (fmap g (Just x))
                           = fmap f (Just (g x)) = Just (f (g x))
```

Functors cont

Functors generalize a certain kind of behavior that we can use to build further abstractions. The laws themselves allow us (and the compiler) to prove properties, and simplify code, particularly in the presence of pure functions with no side-effects (which is not necessarily true in C++.)

What might a Functor look like in C++? The following sketch is due to Bartosz Milewski in [Bartosz Milewski. Jan 2015] and [Bartosz Milewski. Sep 2012].

```
template<template<class> F, class A, class B>
F<B> fmap(std::function<B(A)>, F<A>

template<class A, class B>
optional<B> fmap(std::function<B(A)> f,
    optional<A> opt) {
if (!opt.isValid()) return optional<B>{}
else return optional<B>{ f(opt.val()) };
}
```

Functors cont

A note on partial application and curried vs uncurried functions. The function fmap defined for a Functor f with type $(a \to b) \to fa \to fb$ expects to get a function of one argument and a Functor holding a value of type a. If I apply fmap to a function g (fmap g) we get a function of type $fa \to fb$. So in a sense fmap has transformed a function g of type $a \to b$ to a function of type $fa \to fb$. Examples:

```
fmap length [[1,2], [3,4,5], [5,6,7]]
fmap Just [1,2,3,4]
fmap chr [101, 102, 103]
fmap ord "abcd"
```

In order to do partial application we require first-order functions and the function must be curried form $(a \to b \to c)$ rather than uncurried form $((a, b) \to c$. Using higher order functions we can move back and forth between the curried and uncurried forms.

```
curry f = \a -> \b -> f(a,)
uncurry f = \ (a,b) -> f a b
```

Functor types

Function	**fmap'd function**
length :: $[a] \rightarrow Int$	fmap length :: $Functor\ f \Rightarrow fa \rightarrow fInt$
Just :: $a \rightarrow Maybea$	fmap Just :: $Functor\ f \Rightarrow fa \rightarrow f(Maybea)$
chr :: $Int \rightarrow Char$	fmap chr :: $Functor\ f \Rightarrow fInt \rightarrow fChar$
ord :: $Char \rightarrow Int$	fmap ord :: $Functorlf \Rightarrow fChar \rightarrow fInt$

Using fmap we've been able to use a mapping function without knowing any of the detail of its implementation so there would be no need to break encapsulation, similar to the motivation for using an iterator pattern as opposed to a loop. In addition the functor laws guarantee composition in a very general way.

Applicative Functors

We can build on the Functor typeclass to introduce applicative functors. Functors express a regularity associated with mappable things, including function composition.

An applicative Functor typeclass is defined by:

```
>       class (Functor f) => Applicative f where
>           pure :: a -> f a
>           (<*>) :: f (a->b) -> f a -> f b
```

If *f* is a functor, then f is an Applicative functor IF it provides a definition of pure and $\langle * \rangle$. Pure is just a way to wrap something of type *a* in a Functor. $\langle * \rangle$ is an explicit apply at the level of Functors: a Functor holding a computation. Since f must be a Functor, we know we can use fmap.

Applicative Functors cont

pure takes a value of type a and returns an applicative functor holding a value of type a.

$\langle * \rangle$ takes an applicative functor holding a function of type $a \rightarrow b$, and a Functor holding an a, and returns a Functor holding a b (by extracting the function from the Functor and the value of type a and applying the function to the value to get the value of type b).

A simple example with the Maybe type:

```
>       instance Applicative Maybe where
>         pure = Just
>         Nothing <*> _ = Nothing
>         (Just f) <*> something = fmap f something
```

What do Applicative Functors do for us?

We want to incrementally build a function out of pre-existing functions (possibly using partial application). As a simple example, $(+3)$ is a partially applied function that adds 3 to what it is applied to. So (Just $(+3)$) holds a function of type $Int \rightarrow Int$ which adds 3 to its argument. If we do (Just $(+3)$) $\langle * \rangle$ (Just 9) we get (Just 12);

The operator $\langle * \rangle$ is really just an apply operator for Functors. Thinking in types again,

```
concat :: [[a]] -> [a]
((<*>) (Just concat)) :: Maybe [[a]] -> Maybe [a]
```

In order to move from a function $a \rightarrow b$ to a function $f\ a \rightarrow f\ b$ we can use the function pure (it will lift any value, and wrap it in a functor, functions included.)

Example Applicative

```
>     instance Applicative ((->) r) where
>         pure x = (\_ -> x)
>         f <*> g = \x -> f x (g x)

>     instance Applicative [] where
>       pure x = [x]
>       fs <*> xs = [f x | f <- fs, x <- xs]
```

Applicative Functors in C++

We need a function to apply a function over functors (which require unwrapping the functor to get to the function it holds) and then to wrap a function over values in a functor. Here is pure:

```
template <class A>
unique_ptr<A> pure(A a) {
return unique_ptr<A> (new A(a));
}
```

And then the apply:

```
template<class A, class B>
unique_pt<B> apply (unique_ptr<function<B(A)>> f,
unique_ptr<A> p) {
    unique_ptr<B> result;
    if (f && p) result.reset(new B((*f)(*p)));
    return result;
}
```

Applicative axioms

These are somewhat less satisfying than the Haskell because they do expose some of the underlying structure and require an explicit reference to the wrapping effect.

Applicative Functors also have laws:

```
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure (\$ y) <*> u
```

But we won't prove these here.

Applicative Functions to Monads

One thing the applicative functions do for us is allow us to create chains of computations:

$$[(+), (*)] \ <*> \ [1, 2] \ <*> \ [3, 4]$$

First this does

$$[(+), (*)] \ <*> \ [1, 2]$$

resulting in $[(1+), (2+), (1*), (2*)]$. Then each of these functions is applied to $[3, 4]$. This chain of computations is possible since $\langle * \rangle$ expects a Functor and returns a Functor. The limitation is that we often want to do a chain of computations that involves conditional computation: we need to make a decision based on a previous result. This leads to monads which allow us to extract a value from a computation, make a decision, and stuff the value back in. We won't cover monads in detail today but move on to Continuations.

What is a monad?

A monad is a type class that identifies how to return a function and how to sequence chain, having access to the result of a computation.

```
>       class Monad m where
>         return :: a -> m a
>         (>>=) :: m a -> (a -> m b) -> m b
>         (>>) :: m a -> m b -> m b
>         x >> y = x >>= \_ y
>         fail :: String -> m a
>         fail msg = error msg
```

There are two default definitions, so usually we only have to define return and >>= (called bind). The constraint that m be Applicative is not explicit but is so. Intuitively, >>= allows us to run a computation, extract the result (something of type a), and convert to something of type b, and return the value (to be used in further computation). Because we have the result of the computation available in a sequence, we can take actions (make decisions) based on the result. Return is just like pure.

Simple Example

A simple example:

```
>       instance Monad Maybe where
>         return x = Just x
>         Nothing >>= f = Nothing
>         Just x >>= f = f x
>         fail _ = Nothing
```

So in the Maybe monad we can write a function f that examines the contents of the Maybe monad does some computation, and then rewraps it for further computation.

Continuations

A continuation is a future computation. We can use the Monad structure to capture this idea:

```
newtype Cont r a = Cont { runCont :: (a->r) -> r}
```

```
>       instance Monad (Cont r) where
>          return a = Cont $ \k -> k a
>          (Cont c) >>= f = Cont $ \k ->
>              c (\a -> runCont (f a) k)
```

The return function creates the continuation that will pass the value on. The bind operator adds the bound function f into the computation (continuation) chain.

We can expand on the Monad with a MonadCont that has a mechanism to escape the continuation (abort the current computation.)

Continuations cont

We can introduce a callCC to do the abort:

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

>       instance MonadCont (Cont r) where
>          callCC f = Cont $ \k -> runCont
>                        (f (\a -> Cont $ \_ k a)) k
```

callCC calls a function with the current continuation. The lambda expression gives the continuation a name. Calling the named continuation anywhere in its scope escapes the computation, regardless of the nesting level. (This discussion is based on the information in the Haskell wiki: All About Monads(The Continuation monad).)

Conclusions

There is so much information here on generic programming using Haskell design patterns (and I haven't covered them all and certainly have not gone into sufficient detail on a number of the patterns)!
Without suggesting that Haskell is the ultimate in programming languages, examining the kinds of patterns that it defines and manipulates can serve both to structure how we write programs in C++ and to also suggest how we might proceed in expanding the language to cover these kinds of solutions more effectively and expressively.

In particular thinking about ways to package delayed computations so that we can manipulate them appropriately seems a natural outgrowth of Monads and their ability to sequence operations and wrap computations.

Citations

The Hashmap examples in this presentation were derived from [Bernady, Jansson, Zalewski, Schupp, Prisnitz , 2008].

The Evaluator example in C++ was derived from [Solodkyy, Reis, Stroustrup, 2013].

The Evaluator example in Haskell and GADTs was derived from [Ralf Hinze, 2004].

The discussion of discriminated unions in C++ was motivated by [Alexandrescu, 2002] and [Alexandrescu, 2002].

The comments on generalized algebraic data types in OOP were based on [Kennedy, Russo, 2005].

The C++ implementation sketches of Functors and Applicative functors were from [Bartosz Milewski. Jan 2015] and [Bartosz Milewski. Sep 2012].

References

Bernady, Jansson, Zalewski, Schupp, Prisnitz (2008)

A comparison of C++ concepts and Haskell type classes

Workshop on Generic Programming, WGP 2008 WGP 2008.

Solodkyy, Reis, Stroustrup (2013)

Open Pattern Matching for C++

GPCE'13 GPCE'13 Oct 27-28 2013.

Ralf Hinze (2003)

Fun With Phantom Types

Institut fur Informatik III, Universitat Bonn (March 2003).

Andrei Alexandrescu (2002)

Discriminated Unions

Discriminated Unions (April 2002).

Andrei Alexandrescu (2002)

An Implementation of Discriminated Unions in C++

An Implementaiton of Discriminated Unions in C++ (Aug 2002)

References continued

Andrew Kennedy and Claudio Russo (2005)

Generalized Algebraic Data Types and Object-Oriented Programming

OOPSLA'05 OOPS'05 Oct 16-20

Bartosz Milewski (2015)

Functors

Bartosz Milewski (2012)

Functional Patterns in C++

https://www.fpcomplete.com/blog/2012/09/functional-patterns-in-c

Links to papers

1. A comparison of C++ concepts and Haskell type classes
2. Open Pattern Matching for C++
3. Fun with Phantom Types
4. An Implementation of Discriminated Unions in C++
5. Discriminated Unions
6. Generalized Algebraic Data types and Object-Oriented Programming
7. Functors
8. Functional-Patterns

The End