Why Funcitonal Programming Matters

Massyl Nait Mouloud, Architect/Senior Developer/Team Leader @ Sfeir (@nmassyl)

30 Jun 2015

Functional programming history

- The logician Haskell B. Curry, with Alonzo Church, established the theoretical foundations of functional programming (Lambda Calculus)
- First high-level FP language Lisp (McCarthy), developed in the late 1950s
- Lisp the seed for functional languages family
- ML family adding static typing and type inference (SML, OCaml, Haskell...)

Introduction

- Functional programs works with VALUES not STATE. Their tools are EXPRESSIONS not COMMANDS
- Immutability is the backbone of all FP programs.
- The basic building bloc is a `Function`
- `Declarative` programming style
- Reference transparency
- Easy to prove correctness of programs using mathematics tools (induction . . .)

Functional programming basics (Static FP)

The following are the basic building blocks, every FP relies on. In the heart of every FP language there is

Function

- Mapping from Type to another Type

```
show :: a -> String
f :: A -> B
map :: (a-> b) -> [a] -> [b]
```

- Standalone thing
- Functions are values

```
x = 1
add x y = x + y
succ = add 1
(succ 2) + 10
```

Function Composition

- Like UNIX pipe (|) or puzzle pieces
- When domain of one function coincides with Codomain of the other

```
lock :: Fun<TransactionManager, TransactionManager>
read :: Fun<TransactionManager, Optional<Counter>>
lockThenRead :: compose(read, lock);
```

- Allows to build coarse grained operations from thin ones with a disciplined way
- Divide and Conqueror

Types

- Set of related values
- Algebra of Types (Sum, Product, Monoid . . .)

Currying

- Functions with multiple arguments are defined in Haskell using the notion of currying. That is, the arguments are taken one at a time by exploiting the fact that functions can return functions as results.

```
var add = function(a,b){return a+b;}
add(2,5);

var add2 =
     function(a){return function(b){return a+b}};

var succ = function(){ return add2(1);};
```

```
curry::Fun2<A, B, C> -> Function<A, Function<B, C>>
add::Fun2<Integer, Integer, Integer>
add'=curry add::Fun<Integer, Fun<Integer, Integer>>
```

Higher order functions

- Formally speaking, a function that takes a function as an argument or returns a function as a result is called higher-order
- Examples

```
map       -- maps a function on list elements
filter    -- filters elements whith a predicate
takeWhile -- stops when the predicate returns false
dropWhile -- stops when the predicate returns false
```

Fold and Universal Property

- Standard operator that encapsulates a simple pattern of recursion for processing lists

```
fold :: (a -> b -> b) -> b -> [a] -> b
fold f acc [] = acc
fold f acc (x:xs) = f x (fold f acc xs)

sum :: [Int] -> Int
sum = fold (+) 0

product :: [Int] -> Int
product = fold (*) 1

and :: [Bool] -> Bool
and = fold (&&) True
```

Universal Property of fold

```
g []     = v                          <=> g = fold f v
g (x:xs) = f x (g xs)
```

Example of proof using this universal property

```
(+1) . sum = fold (+) 1
```

1. g = (+ 1) . sum , f= (+) , v = 1
2. relace if the definition

```
((+1) . sum) [] = 1
((+1) . sum) (x:xs) = (+) x ((+1) . sum) xs)
```

3. Simplifying using composition and sectioning

```
sum [] + 1 = 1
sum (x:xs) + 1 = x + (sum xs + 1)
```

4. use induction for both cases (sum (x:xs) = x + sum xs)

Object oriented Pattern/Principle

- Open/Closed
- Single Responsibility
- Dependency Inversion
- Interface Segregation
- Factory
- Strategy
- and so on ...

Functional counterpart

- So simple just Function :-)

```
map (Int -> Function) [1..]
```

DDD and FP

A domain model in problem solving and software engineering is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain. It does not describe the solutions to the problem.

Wikipedia (http://en.wikipedia.org/wiki/Domain_model)

Object oriented : Rich domain models

- Domain abstraction through a Class
- Class contains both state and behavior (private field)
- Sometimes(Always) hard to decide what should go inside a class
- We always need a coarse(bloated) service class to combine multiple classes
- We focus on nouns first then design the state of entities with classes
- Add related behavior to the class and put larger one into Service class

Functional approach : Lean (Anemic) domain models

- Domain abstraction through Algebraic Data Type (ADT)
- Contains state (every thing is immutable) , no need for private
- Behaviors are outside ADTs, defined by reusable functions
- Use function composition to build coarse grained services
- We focus on the behavior of the domain first (Function signatures only)
- Then focus on how those functions compose to compose larger functions
- Then use ADT to build entities defined in function signatures

Domain model Elements in OO/FP

- Entities & VO ——> ADT
- Behaviors ——> Functions
- Business rules ——> constraints & validation
- Bounded Context ——> Modules and Functions
- Ubiquitous language ——> Functions and DSL
- (implicit concepts explicit, intention revealing interfaces, Side effect free functions, Declarative design, Specification for validation)

Why we Functional approach

- Ability to reason about your code (parametricity, purity and referential transparency)
- Modularity and reusability (Behavior is separated from state)
- Immutability
- Concurrency and Parallelism

Example

- TradeLot, Bid, Payment, Seller, Purchaser, Account
- findLots from seller
- makeBid for each lot
- payBids

```
account :: [Account]

findLots:: Seller -> [TradeLot]

makeBid :: TradeLot -> [Bid]

payBid :: [Account] -> Bid -> [Payment]
```

TDD as Type driven development

Parametricity and Theoreme for Free

Advanced FP (Functor, Monad, Applicative . . .)

Introduction

- The functional programming community divides into two camps. Pure languages such as Haskell, and impure languages such as ML, Scheme. Pure languages are easier to reason about and may benefit from lazy evaluation. Impure languages in the other hand offer **efficiency** benefits and sometimes make possible a more compact mode of expression

Pure function and explicit data flow

- One of the big advantage of pure languages is making the flow of the data explicit.
- Explicit data flow ensures that the value of an expression depends only on its free variables. Hence substitution of equals to equals(reference transparency) is valid (easy to reason about such programs).
- Explicit data flow also ensures that the order of evaluations are irrelevant, making lazy evaluation possible.
- Sometimes, when managing state for instance, it becomes really painful to thread explicitly the intermediate state.

Monad to rescue

- The concept of monad came from Category theory, and applied by Moggi to structure the denotational semantics of programming languages.

Definitions

- Monads offer a solution to integrate the benefit of pure and impure schools.
- Monads provide a convenient framework for simulating effects found in other languages, such as Global state, exception handling, IO, or non-determinism.
- A monad is a way to structure computations in terms of values and sequences of computations using those values.
- Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations.
- Monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required.

Benefits of monads for programmers

Monads are useful tools for structuring functional programs. They have 4 main properties that make them especially useful:

1. Modularity

They allow computations to be composed from simpler computations and separate the combination strategy from the actual computations being performed.

2. Flexibility

They allow functional programs to be much more adaptable than equivalent programs written without monads. This is because the monad distills the computational strategy into a single place instead of requiring it be distributed throughout the entire program.

3. Isolation

They can be used to create imperative-style computational structures which remain safely isolated from the main body of the functional program. This is useful for incorporating side-effects (such as I/O) and state (which violates referential transparency) into a pure functional language like Haskell.

4. Abstraction

They enforce interface based programming. Monads offer a single interface (bind, inject) for all monad instances. We do not have to take care of the underlying implementation to use monad types.

Monad lows

1. Left identity :

```
bind(inject(a), f) = f a
```

2. Right identity:

```
bind(f, function(value){return inject(value)}) = f
```

3. Associativity :

```
bind(f, function(x){ return bind(g(x), h)}) =
bind(bind(f,g), h)
```

A structure with Left, Right identity and a binary operation that is associative is called a Monoid.

List Monad example

```
var inject = function(value){ return [value];};
var bind   = function(ma, fn){
    return concatMap(fn)(ma);
};


var concatMap = function(fn){
  return function(list){
    if(list.length === 0) return [];
    else{
       var rs =  fn(list[0]);
       var tail=list.slice(1);
     return rs.concat(concatMap(fn)(tail));
    }
  };
};
```

Functor

Conclusion

Why learning FP mainly Haskell is worth your time?

1. Get new toolset for solving complex problems with very clean and elegant solutions
2. Learn new way of thinking about programs.
3. Learn new ways factorizing common patterns (Functor, Monad . . .)
4. Learn to reason about your program and rely havely on types and compiler to express invarients