functional programming in c++

1

who am I

- alfons haffmans
- git hub : https://github.com/fons
- linkedin : www.linkedin.com/in/alfonshaffmans/
- about me : http://about.me/alfonshaffmans
- lisp programming for about 5 years
- cl-mongo : mongodb client in common lisp
- cl-twitter : twitter client in common lisp

overview

- oop vs fp
- function signatures and type constructors
- fp techniques
- c++ support for fp techniques
- advanced topics : abstract nonsense
- advanced fp in c++ ?
- conclusions

OOP vs FP


GODZILLA
KING KONG
THE BIGGEST BATTLE OF THE
CENTURY IS ABOUT TO BEGIN

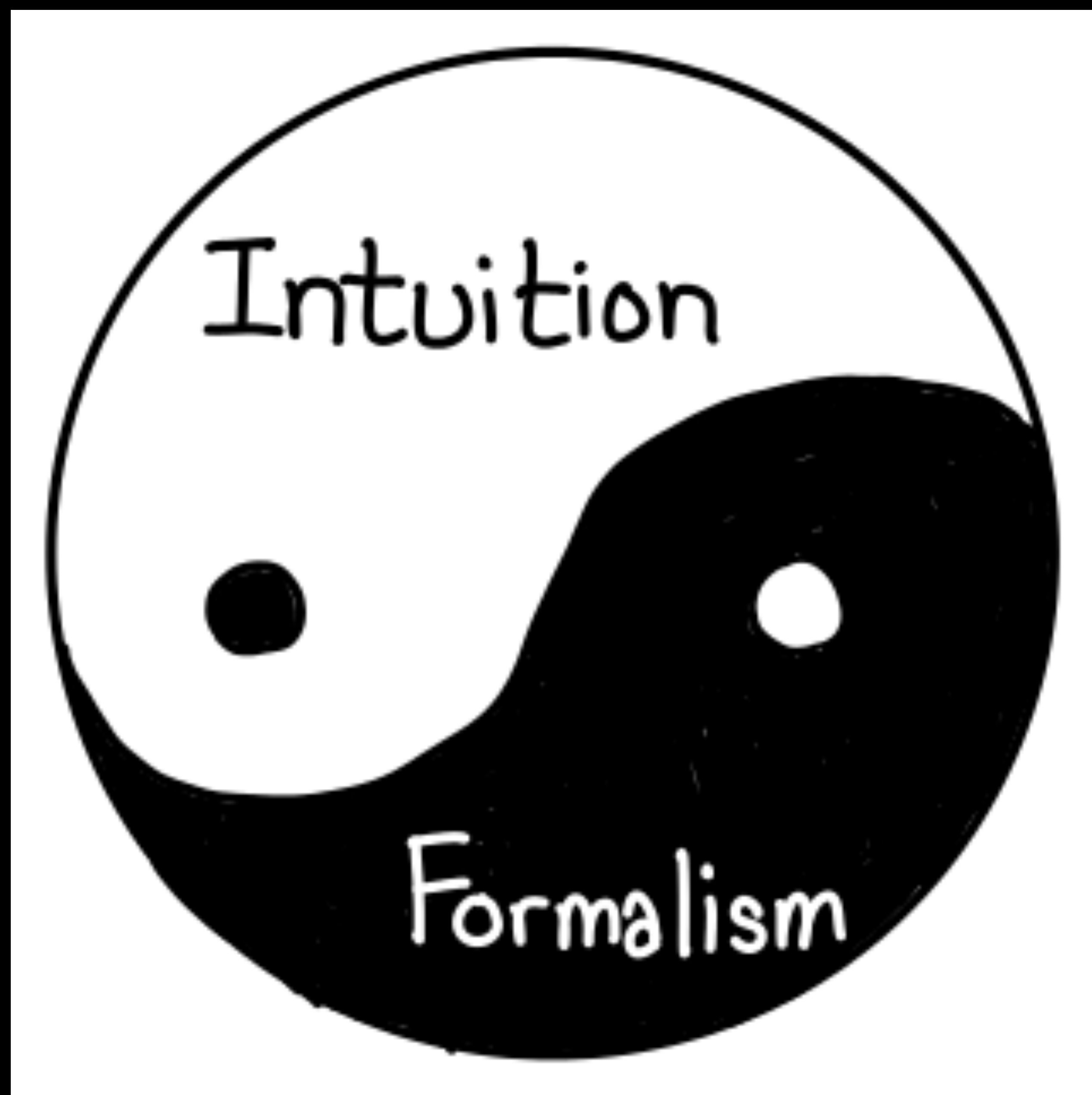
OOP

- everything is an object
- encapsulation : state and methods together
- inheritance : object composition
- polymorphism : different behaviours interface with a common interface.
- imperative programming using statements

FP

- everything is a function / computation
- immutable data
- referential transparency; no side effects
- decouple data from operations on the data
- few data structures; lots of operations
- declarative style : combine functions in expressions

functions and types



7

talking about functions

- need a way to talk about functions
- equivalent of how uml helps us understand the relationship between classes
- use haskell's type notation for function signatures
- f:: (a, b) -> c => function f takes a pair of arguments of type a and type b and returns a result of type c

examples

- id :: a -> a
- id (int x) = x :: int
- id (Person p) = p :: Person

- equal::(a, b) -> c
- equal (Person p, int x) = (x == p.ssn) :: bool
- equal (Person p, Person q) = (p ==q) :: bool

type constructors

- types which require other types to be fully defined
- think c++ templates
- M a : general type constructor
- M a <=> template<typename a> M [...]

examples

- [a] : list type constructor
- [a] <=> std::forward_list<a> or std::list<a>

- (a -> b) : Function type constructor (->)
- (a -> b) <=> [...]std::function < b (a) >

building blocks

- lambda's
- currying
- higher order functions

lambda's

- lambda : anonymous function.
- a function you can create on the fly and use
- bind a lambda to a variable or argument
- let f = \x-> x + 7

currying

- currying replaces a function which takes a tuple of parameters with a chain of functions
- f :: (a , b) -> c => f:: a -> b -> c
- curried plus : let cplus = \x -> \y -> x+y
- cplus(5) = \y-> 5 + y
- cplus(5)(6) = 11

higher order functions

- functions as first class objects
- functions take functions as arguments
- functions which return functions
- apply:: (a->b) -> a -> b
- after::(a->b)->(r->a)->(r->b)

basic hof’s

- map :: (a -> b) -> [a] -> [b]
- maps a function over a list and returns a list
- (dual of) the visitor pattern
- (a->b) : functional (combination of) business rules
- reduce a.k.a foldl :: (a->b->a)->a->[b]->a
- accumulates a result across a list of values
- catamorphism (yeah google this..)
- foldr :: (a->b->b)->b->[a]->b
- recursion over a list of values

functional techniques in c++

- lambda’s
- tuples
- bind
- stl

λ expressions in c++

- [...] (...) -> rettype { }
- auto f = [=] (int x) { return 2*x;};
- (...) : arguments; cannot be templated
- -> rettype : return type optional
- [...] : capture specifier

λ capture specifiers

- captures create a closure which provides access to the enclosing environment.

- [] : doesn't capture the environment
- [=] : capture by value
- [&] : capture by reference

- int y = 5
- auto l = [=] (int x) { x + y}
- l (10) = 15

λ example

```
int lambda_1()
{
  int x = 0;
  int y = 42;
  std::cerr << "hello  world" << std::endl;
  auto func = [x, &y] () { std::cout << "Hello world from lambda : " << x << "," << y << std::endl; };
  auto inc = [&y] () { y++; };
  auto inc_alt = [y]  () mutable { y++; };
  auto inc_alt_alt = [&]  () { y++; x++; };
  func();

  y = 900;
  func();
  inc();
  func();
  inc_alt();
  func();
  inc_alt_alt();
  func();//notice what gets outputted here !
  std::cout << " x :" << x << "; y :" << y << std::endl;
  return 0;

}
```

std::function

- function type wrapper
- generalizes the function pointer
- std::function

std::function

```
int lambda_7 ()
{
  // as opposed to [=] or [] or [l]
  std::function<int (int)> l = [&l] (int x) ->int {
    std::cout << x << ",";
    if (x < 0) return 0;
    x--;
    std::cout << x << ",";
    l(x);
  };
  l(25);
  return 0;
}
```

curry

```
curry :: (T, U) -> T -> U -> R

template <typename T, typename U, typename R>
std::function<std::function<int (U)> (T)> curry (std::function<R (T,U)> op)
{
  return [=] (T x) { return [=] (U y) {return op(x, y);};};
}

int lambda_6 ()
{
  auto l = curry<int,int, int> ([](int x, int y) { return (5 + x) * y;});
  std::cout << l(1)(1) << std::endl;
  return 0;
}
```

std::bind

- std::bind and std::placeholders::_..
- bind(op,args,...)
- bind can be used to create functions by combining other functions

```
int bind_1()
{

  auto l = std::bind(plus<int>(), _1, 10);
  auto r = l(902);
  cout << l(902) << endl;
  return 0;
}

int bind_4()
{
  auto l1     = [](int x, int y) { return x*y;};
  auto repeat = [](int n, int x) { int y = 1;
                                   while (n-- > 0) y *= x;
                                   return y;};

  auto rpl    = std::bind (repeat,
                           std::placeholders::_1,
                           std::bind(l1,
                                     std::placeholders::_2,
                                     std::placeholders::_3));

  std::cout << " l1 (1,2) : " << l1(1,2) << std::endl;
  std::cout << " repeat (2,2) : " << repeat(2,2) << std::endl;
  auto val = rpl(4, 1, 2); //( 1 * 2) ^ 4
  std::cout << " result : " << val << std::endl;

}
```

std::forward_list

- std::forward_list
- singly linked list
- adding to the head of the list is fast

map

- (a->b)->[a]->[b]
- std::for_each : basic looping; in place updates. Not very functional.
- std::transform : non-destructive looping

std::transform/map

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1, OutputIt d_first, UnaryOperation unary_op );
```

- non-destructive
- unary_op <=> (a->b)
- (first1, last1) <=> [a]
- (d_first) <=> [b]

map with std::transform

```
template<typename A, typename B>
std::forward_list<B> map (std::function<B(A)> f, const std::forward_list<A>& L)
{
  std::forward_list<B> H;
  std::transform(L.begin(), L.end(), std::front_inserter(H), f);
  H.reverse();
  return H;
}
```

```
int trans_5()
{
  std::forward_list<int> L = {1,67,89,23,45,1,3,99,-90};
  std::function<int(int)> show = [] (int v) { std::cout << v << ","; return v;};
  map(show, L);
  std::function<int(int)> op = [] (int y) {return (y + 79) % 45;};
  std::forward_list<int> H2 = map (op, L);

  std::cout << std::endl << "--------------" << std::endl;
  map(show, H2);
  return 0;
}
```

std::transform / zipWith

```
template< class InputIt1, class InputIt2, class OutputIt, class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt d_first, BinaryOperation binary_op );
```

- two lists as input : (first1, last1, (first2, last2)
- zipWith::(a->b->c) ->[a]->[b]->[c]
- zip :: [a]->[b]->[(a,b)]

zip with std::transform

```
template<typename A, typename B>
std::forward_list<std::tuple<A,B>> zip (const std::forward_list<A>& L, const std::forward_list<B>& M)
{
  std::forward_list<std::tuple<A,B>> H;
  auto zipper = [] (const A& a, const B& b) {return std::make_tuple(a,b);};
  std::transform(L.begin(), L.end(), M.begin(), std::front_inserter(H), zipper);
  H.reverse();
  return H;
}

int trans_6()
{
  std::forward_list<int>  L = {1,67,89,23,45,1,3,99,-90};
  std::forward_list<char> M = {'a','b','l','u','t','v','r','6','h'};

  std::function<std::tuple<int, char> (std::tuple<int,char> v)> show = [] (std::tuple<int,char> v) {
                                                          std::cout << "(" << std::get<0>(v);
                                                                        std::cout << "," << std::get<1>(v) << "),";
                                                                        return v;};
  std::forward_list<std::tuple<int, char>> H2 = zip (L, M);
  map(show,H2);
  std::cout << std::endl;
  return 0;
}
```

foldl/reduce

- foldl::(a->b->a)->a->[b]->a
- std::accumulate (2nd form)
- move a binary operation over a list and return a result.
- min/max/avg/...

std::accumulate

```
template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
          BinaryOperation op );
```

- (first,last) <=> [b]
- T init <=> a
- BinaryOp <=> (a->b->a)
- returns the result

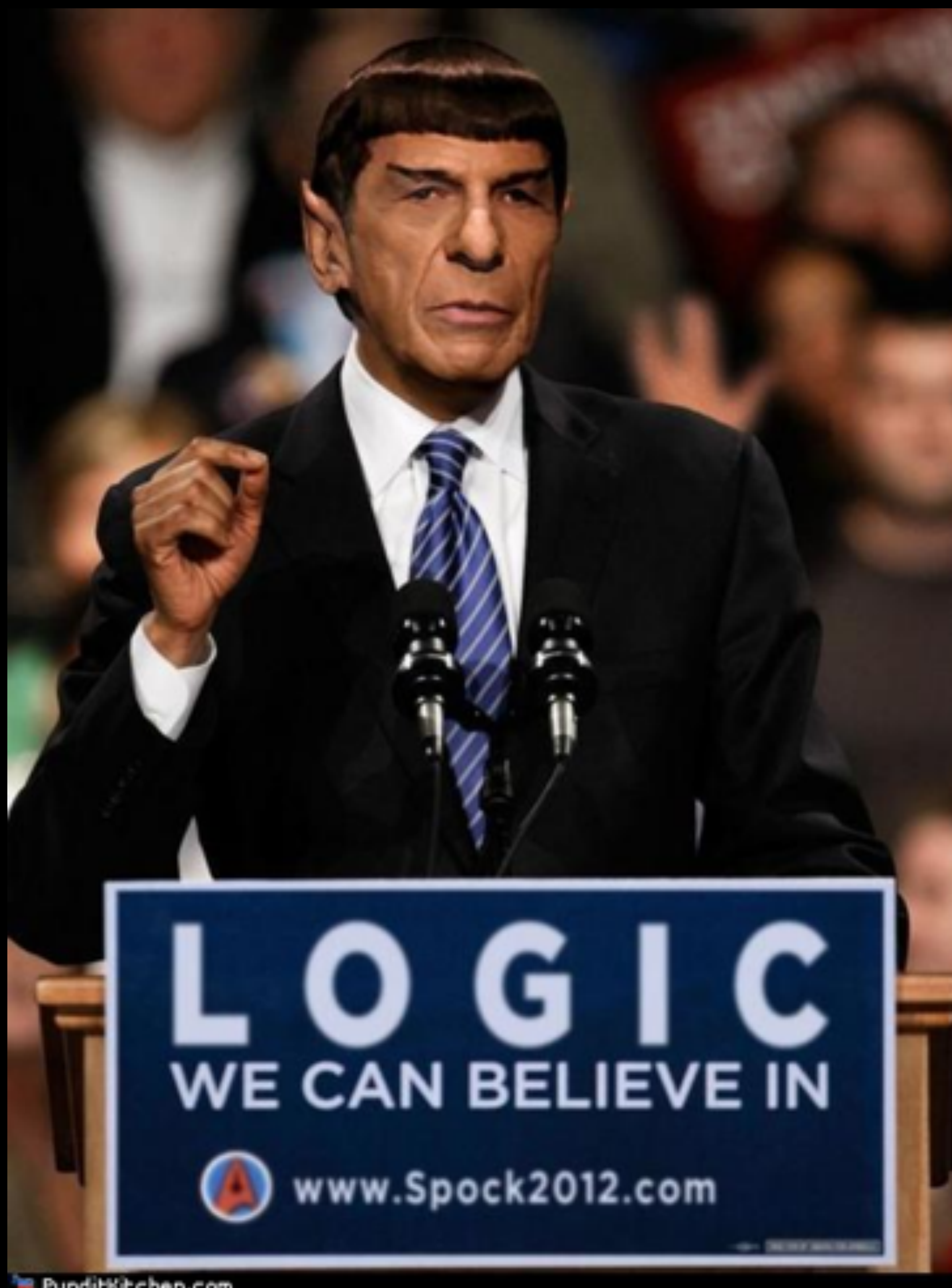
```
static char digits[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };

static char irc (int i)
{
  return digits[ abs(i) % 10 ];
}

int trans_1()
{
  std::forward_list<int> L = {1,2,3,4,5,67,12};
  auto v = irc (2345);
  std::cout << " v : " << v << std::endl;
  std::function <std::string (std::string&, int)> op = [] (std::string& a, int v) { a.append(1, irc(v)); return a;};
  std::string res = std::accumulate(L.begin(), L.end(), std::string(""), op);
  std::cout << " res : " << res << std::endl;
}


int trans_2()
{
  std::forward_list<int> L = {1,-6,23,78,45,13};
  std::function<int (int, int) > max = [] (int x, int y) { return (x > y) ? x : y;};
  auto res = std::accumulate(L.begin(), L.end(), std::numeric_limits<int>::min(), max);
  std::cout << res << std::endl;
  return 0;
}
```

```
int trans_3()
{
  typedef std::forward_list<int> list_t;
  list_t L = {1,-6,23,78,45,13};
  list_t K;
  std::function<list_t (list_t, int) > mappy = [] (list_t L, int y) { L.push_front( 2*y + 1); return L;};

  auto res = std::accumulate(L.begin(), L.end(), K, mappy);

  for (auto& v: res) {std::cout << v << std::endl;}
  return 0;
}
```

```
int trans_4()
{
  typedef std::forward_list<int> list_t;
  list_t L = {1,-6,23,78,45,13};
  //a->[a]
  std::function<list_t (int)> l1 = [] (int y) {list_t L; L.push_front(2*y +1); return L;};
     // ([a],[a])->[a]
  std::function<list_t (list_t, list_t)> concat = [] (list_t A, list_t B) { A.splice_after(A.before_begin(), B); return A;};
  // ([a]->[a]->[a])->(a->[a])->([a]->a->[a])
  auto op = std::bind(concat, std::placeholders::_1, std::bind(l1, std::placeholders::_2));
  //[b] ->(a->[b])
  auto res1 = std::accumulate(L.begin(), L.end(), list_t(), op);
  for (auto& v: res1) {std::cout << v << std::endl;}
  return 0;
}
```

```
// [a] -> (a ->[a]) ->[a]
template<typename A>
std::forward_list<A> bind_list (std::forward_list<A> L, std::function<std::forward_list<A> (A)> f)
{
  std::function<std::forward_list<A> (std::forward_list<A>, std::forward_list<A>)> concat = [] (std::forward_list<A> L,
std::forward_list<A> R) { L.splice_after(L.before_begin(), R); return L;};
  auto op = std::bind(concat, std::placeholders::_1, std::bind(f, std::placeholders::_2));
  return std::accumulate(L.begin(), L.end(), std::forward_list<A>(), op);;
}


int m_1()
{
  typedef std::forward_list<int> list_t;
  list_t L = {1,-6,23,78,45,13};
  std::function<list_t (int)> l1 = [] (int y) {list_t L; L.push_front(2*y +1); return L;};
  auto res1 = bind_list(L, l1);
  std::cout << "results : " << std::endl;
  for (auto& v: res1) {
    std::cout << v << std::endl;
  }
}
```

advanced topics



- non-deterministic computations
- abstract nonsense
- functors and friends
- apply to c++

is everything a function ?

- single valued functions : same input returns the same output.
- what if things fail ? what if we have side-effects ? what if ..... ?

38

failure is an option

- wwacpd (*)
- let's keep it simple
- f :: a -> [b, bool]
- bool : true -> error occurred.

(*) what would a cobol programmer do ?

other maybe's

- f :: a -> [-a,a] (a is a number).
- f:: a-> [a, rangen] : random number generator
- input gives several outputs
- non-deterministic computations
- f :: a -> M b where M could be [] (a list)

abstract nonsense

- how do we use single valued functions when the results can be non-deterministic ?
- we don’t want to change the interface to our functions.
- things need to remain transparent.
- category theory and abstract algebra
- we just need to lift these functions to an other domain.

say what ?

- f :: a->b <=> single valued function
- [bool, a] : M a <=> non-deterministic value
- How do we apply f to non-deterministic values ? Use a functor !
- fmap :: (a->b)-> M a -> M b
- fmap id = id
- fmap f.g = fmap f . fmap g

I'm intrigued; tell me more

- applicative functor
- enhance the functor :
  - *fmap :: (a->b) -> M a -> M b*
  - pure :: a -> M a
  - apply :: M (a->b) -> M a -> M b

I’m flabbergasted

- enhance applicative to form a monad
- bind :: M a -> (a -> M b) -> M b
- (applicative) functors handle single valued functions.
- monads handle non-deterministic computations.

can it (*) be done in c++ ?

(*) advanced functional programming with functors, applicatives, monads ?

functor in c++

```
template <template<typename T1> class F>
struct functor {
  template<typename A, typename B>
  static std::function < F<B> (F<A>)> fmap(std::function <B (A)> f);;
};
```

- generalizes the map
- class F contains things of type T
- F is a type constructor
- fmap :: (A->B)->F A->F B

std::forward_list as functor

std::forward_list template takes two type parameters

```
template <typename T>
struct proxy_list {
  proxy_list(const std::forward_list<T>& L) : L(L) {}
  proxy_list(const proxy_list& obj) : L(obj.L) {}
  void operator=(const proxy_list& obj) = delete;
  const std::forward_list<T> L; //cannot be reference because it may hold ref to stack
};

template<> struct
functor<proxy_list> {
  template<typename A, typename B>
  static std::function < proxy_list<B> (proxy_list<A>)> fmap(std::function <B (A)> f) {
    return [=] (proxy_list<A> L) {
      return map<A,B>(f,L.L);
    };
  };
};
```

basically fmap=map

std::shared_ptr as functor

```
template <>
struct functor<std::shared_ptr> {
  template<typename A, typename B>
  static std::function<std::shared_ptr<B> (std::shared_ptr<A>)>  fmap (std::function<B(A)> f) {
    return [=](std::shared_ptr<A> v) {
      if (v) {
         return std::make_shared<B>(f(*v));
      }
      return  std::shared_ptr<B>(nullptr);
    };
  }
};
```

applicative functor in c++

```
template <template<typename T1> class F>
struct applicative_functor : public functor <F>
{

  template <typename A>
  static F<A> pure(A val);

  template<typename A, typename B>
  static std::function < F<B> (F<A>)> apply(F <std::function<B(A)>> f );
};
```

- pure : lift value into the functor
- apply : apply a lifted function f to a lifted value

std::forward_list as applicative functor

```
template<> struct
applicative_functor<proxy_list> {
  template<typename A>
  static proxy_list<A> pure(A v) {
    std::forward_list<A> L;
    L.push_front(v);
    return L;
  }

  template<typename A, typename B>
  static std::function< proxy_list<B> (proxy_list<A>)> apply(proxy_list<std::function<B(A)>> f) {
      return [=](proxy_list<A> v) {
         std::forward_list<B> acc;
         const std::forward_list<std::function<B(A)>>& F = f.L;
         const std::forward_list<A>& L = v.L;
         for (auto& func : F) {
           for (auto& arg : L) {
             acc.push_front(func(arg));
           }
          }
         acc.reverse();
         return acc;
      };
    };
};
```

```
int functor_3()
{
  std::forward_list<int> K = {2, 5, 10};
  std::forward_list<int> L = {8, 10, 11};
  std::function<int(int)> show = [=](int v) {
    std::cout << v << ",";
    return v;
  };
  //plus :: a->a->a
  std::function < std::function < int (int) > (int)> plus = [] (int x) {
    return [=] (int y) {
      return x + y;
    };
  };

  auto ls  = applicative_functor<proxy_list>::pure(show);
  auto lp  = applicative_functor<proxy_list>::pure(plus);
  auto kl  = applicative_functor<proxy_list>::apply(lp)(K);
  auto M   = applicative_functor<proxy_list>::apply(kl)(L);

  applicative_functor<proxy_list>::apply(ls)(K);
  std::cout << endl;
  applicative_functor<proxy_list>::apply(ls)(L);
  std::cout << endl;
  applicative_functor<proxy_list>::apply(ls)(M);
  std::cout << endl;
}
```

example explained

- lift the plus function into the list functor
- auto lp = applicative_functor<proxy_list>::pure(plus);

- create a set of partial applied plus functions
- [(2 +), (5 +), (10 +)]
- auto kl = applicative_functor<proxy_list>::apply(lp)(K);

- apply the partially applied plus functions to the elements of L
- auto M   = applicative_functor<proxy_list>::apply(kl)(L);

std::shared_ptr as applicative functor

```
template<>
struct applicative_functor <std::shared_ptr> : public functor<std::shared_ptr>
{

  template<typename A>
  static std::shared_ptr<A> pure(A val) {
    return std::make_shared<A>(val);
  }

  template<typename A, typename B>
  static std::function< std::shared_ptr<B> (std::shared_ptr<A> v)>
apply(std::shared_ptr<std::function<B(A)>> f) {
      return [=](std::shared_ptr<A> v) {
         if (v && f) {
           auto F = *f;
           return pure (F(*v));
         }
         return  std::shared_ptr<B>(nullptr);
      };
    }

};
```

```
struct W {

  explicit W(int v, const std::string& s) : v(v), s(s){}
  W(const W& o) : v(o.v), s(o.s){}
  void operator=(const W& o) = delete;

  int ssn() const { return v;}
  std::string name() const {return s;}
  std::ostream& pp(std::ostream& os) const {
    return os << "name : " << s << " ssn :" << v << " ";
  }
private :
  int v;
  std::string s;
};


int functor_1()
{

  std::function<int (W)> show =[](W w) { w.pp(std::cerr) << std::endl; return w.ssn();};

  auto p = applicative_functor<std::shared_ptr>::pure(W(1090867, "hello_kitty"));
  auto s = applicative_functor<std::shared_ptr>::pure(show);
  std::cout << p << std::endl;
  applicative_functor<std::shared_ptr>::fmap(show)(p);
  applicative_functor<std::shared_ptr>::apply(s)(p);

}
```

Monad in c++

```
template <template<typename T1> class F>
struct monad : public applicative_functor <F>
{

  template<typename A, typename B>
  static std::function < F<B> (std::function< F<B> (A) > ) > bind(F<A> val);

};
```

- extend the applicative functor to handle non-deterministic calculations

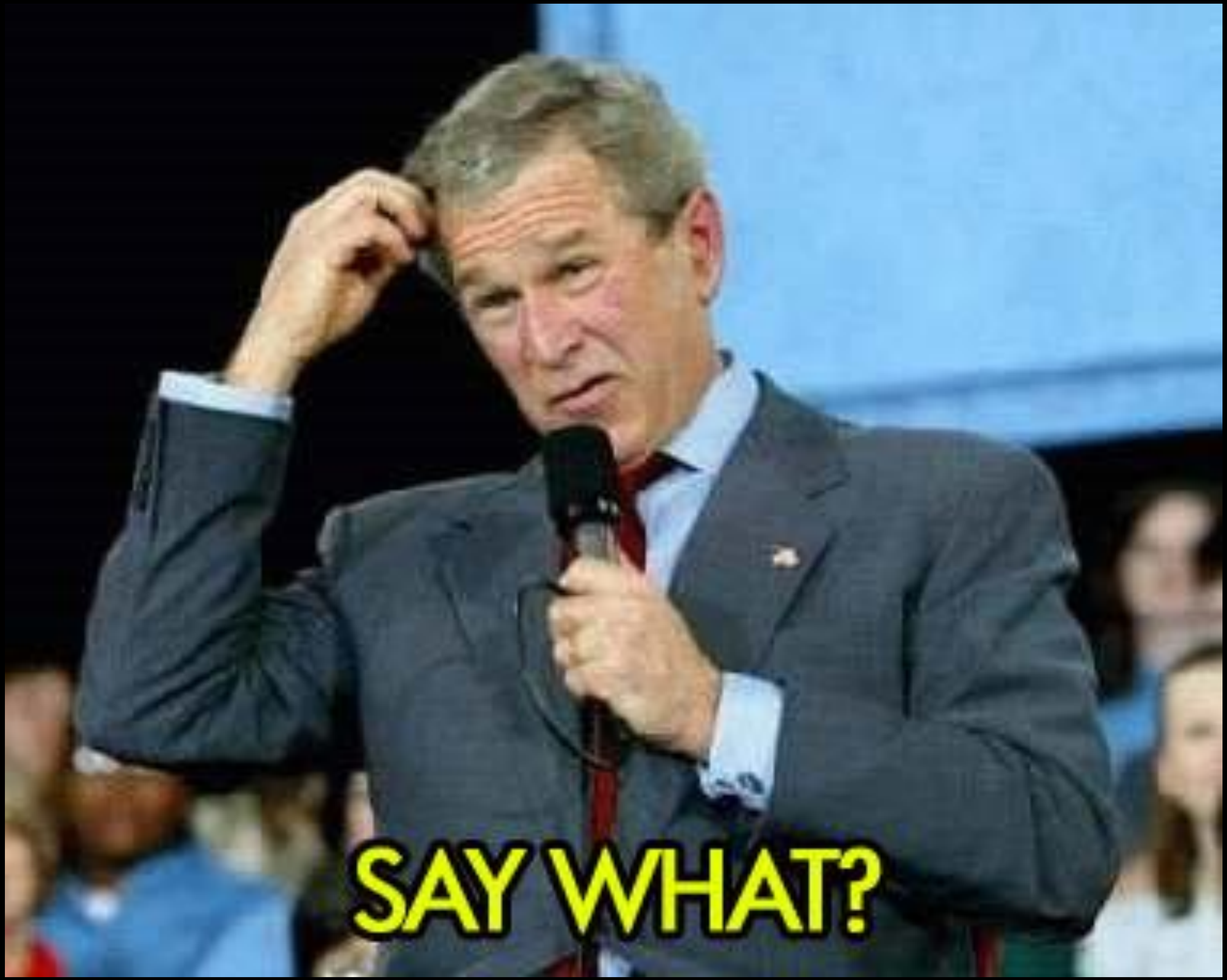
std::forward_list as monad

```
template<> struct monad<proxy_list> : public applicative_functor<proxy_list> {

  template<typename A, typename B>
  static std::function < proxy_list<B> (std::function< proxy_list<B> (A) > ) > bind(proxy_list<A> xs) {
    return [=](std::function<proxy_list<B> (A)> f) {
      std::forward_list<B> R;
      std::forward_list<A> M = xs.L;
      std::forward_list<proxy_list<B>> res = map(f, M);
      for (auto& list : res) {
         std::forward_list<B>& l = list.L;
         R.splice_after(R.before_begin(), list.L);//concatenate
      }
      return R;
    };
  }

};
```

monad example

```
int m_2()
{
  std::forward_list<int> L = {1,3,45,78};
  auto op = [=](int x) {
    std::forward_list<int> R = {x , -x};
    return R;
  };
  proxy_list<int> r = monad<proxy_list>::bind<int,int>(L)(op);
  for (auto& v: r.L) {
     std::cout<< v << "," ;
  }
  std::cout << std::endl;
}
```

conclusions



the good

- not everything needs to be an object
- functional programming style supported through lambda’s, std::bind, stl, shared pointers,....
- building blocks are a simple set of abstractions.
- immutable data is supported

but....

- tradeoff of referential transparency vs. scalability and performance.
- no support for immutable data structures like lists.
- line noise obscures the constructs.
- weak type system.
- does this approach scale ?

The End