Comonads, Applicative Functors, Monads
and other principled things

Tony Morris

NICTA

November 25, 2014

If you've ever googled any of these. . .

You probably got an answer as sensible as this

Goals

- Emphasis on the *practical motivations* for the specific structures.
- This is not about the details of concepts like monads.
- This is about the process of reasoning that leads to their discovery.

Goals

- Emphasis on the *practical motivations* for the specific structures.
- This is not about the details of concepts like monads.
- This is about the process of reasoning that leads to their discovery.

Goals

- Emphasis on the *practical motivations* for the specific structures.
- This is not about the details of concepts like monads.
- This is about the process of reasoning that leads to their discovery.

Goals

Nothing I tell you pertains to any specific programming language.

- Java
- Python
- JavaScript
- *doesn't matter, it still applies*

Goals

There is no emphasis on a specific type of programming.

- Functional
- Dysfunctional
- Object-disoriented
- Dynamically-typed
- Hacking it out like a drunk dog muffin
- *it's all the same*

Principled Things

- What do we mean by a principled thing?
- Principled reasoning gives rise to useful inferences.

$$\begin{array}{l} p \\ \underline{p \to q} \\ \therefore q \end{array}$$

Principled Things

- What do we mean by a principled thing?
- Principled reasoning gives rise to useful inferences.

$$\begin{array}{rl} & p \\ & \underline{p \rightarrow q} \\ \therefore & q \end{array}$$

Principled reasoning is already familiar
using Java/C# syntax

```
enum Order { LT, EQ, GT }

interface Compare<A> {
  Order compare(A a1, A a2);
}
```

We define this interface because

- We can produce data structures to satisfy the interface.
- We can define operations that function on all instances of the interface.

Principled Reasoning

Data structures such as

- integers
- strings
- list of elements where the elements can be compared

Principled Reasoning

Operations such as

- `List#sort`
- `Tree#insert`
- `List#maximum`

We might also define constraints required of instances.

For example

- if `compare(x, y) == LT` then `compare(y, x) == GT`
- if `compare(x, y) == EQ` then `compare(y, x) == EQ`
- if `compare(x, y) == GT` then `compare(y, x) == LT`

We will call these *laws*. Laws enable reasoning on abstract code.

Summary

- a principled interface
- law-abiding instances
- derived operations

Principled Reasoning for Practical Application

- We try to maximise instances and derived operations, however, these two objectives often trade against each other.
- For example, all things that can `compare` can also be tested for equality, but not always the other way around[1].
- Obtaining the best practical outcome requires careful application of *principled reasoning*.

[1] such as complex numbers

Some boring syntax issues

Java

```
enum Order { LT, EQ, GT }

interface Compare<A> {
  Order compare(A a1, A a2);
}
```

Haskell

```
data Order = LT | EQ | GT

class Compare a where
  compare :: a -> a -> Order
```

Mappable

The interface

Java 8/C# with the addition of higher-kinded polymorphism

```
interface Mappable<T> {
  <A, B> T<B> map(Function<A, B> f, T<A> a);
}
```

Haskell

```
class Mappable t where
  map :: (a -> b) -> t a -> t b
```

Identity

```
x.map(z -> z) == x
```

```
map (\z -> z) x == x
```

Composition

```
x.map(z -> f(g(z))) == x.map(g).map(f)
```

```
map (\z -> f (g z)) x == map f (map g x)
```

Mappable

Instances of things that map[2]

List []

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

Reader (e ->)

```
map :: (a -> b) -> (e -> a) -> (e -> b)
```

There are an ***enormous*** *number of instances.*

[2] map is called Select in C#/LINQ.

Map a constant value

```
mapConstant :: Mappable t => a -> t b -> t a
mapConstant a b = fmap (\_ -> a) b
```

Map function application

```
mapApply :: Mappable t => t (a -> b) -> a -> t b
mapApply f a = fmap (\g -> g a) f
```

The set of derived operations is relatively small.

- The more common name for `Mappable` is a *functor*.
- We have seen:
    - The interface for a functor
    - The laws that the functor instances must satisfy
    - The instances of the functor interface
    - The operations derived from functor

?

Make sure we understand Mappable!

Monad
The interface

Java 8/C# with the addition of higher-kinded polymorphism

```
interface Monad<T> {
  <A> T<A> join(T<T<A>> a);
  <X> T<X> unit(X x);
}
```

Haskell

```
class Monad t where
  join :: t (t a) -> t a
  unit :: x -> t x
```

Monad

- The monad interface has laws too.
- The monad interface has strictly stronger requirements than functor.
  - In other words, all structures that are monads, are also functors.
  - However, not all structures that are functors, are also monads.
- Therefore, there are fewer monad instances than functor instances.

But still a *very large* amount

- `List`
- Reader `((->) e)`
- `State s`
- `Continuation r`
- `Maybe/Nullable`
- `Exception`
- `Writer w`
- `Free f`

and lots of operations too

- `sequence :: [t a] -> t [a]`
- `filterM :: (a -> t Bool) -> [a] -> t [a]`
- `findM :: (a -> t Bool) -> [a] -> Maybe [a]`

This is what monad is for.

- A lawful interface.
- Satisfied by lots of instances.
- Gives rise to lots of useful operations.

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in $SPECIFIC_PROGRAMMING_LANGUAGE.~~
- ~~*blah blah* relating to $SPECIFIC_MONAD_INSTANCE.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

[a] yes, seriously, this is a thing.

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in $SPECIFIC_PROGRAMMING_LANGUAGE.~~
- ~~*blah blah* relating to $SPECIFIC_MONAD_INSTANCE.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

[a] yes, seriously, this is a thing.

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in `$SPECIFIC_PROGRAMMING_LANGUAGE`.~~
- ~~*blah blah* relating to `$SPECIFIC_MONAD_INSTANCE`.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

[a] yes, seriously, this is a thing.

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in $SPECIFIC_PROGRAMMING_LANGUAGE.~~
- ~~*blah blah* relating to $SPECIFIC_MONAD_INSTANCE.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

[a] yes, seriously, this is a thing.

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in `$SPECIFIC_PROGRAMMING_LANGUAGE`.~~
- ~~*blah blah* relating to `$SPECIFIC_MONAD_INSTANCE`.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

[a] yes, seriously, this is a thing.

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in $SPECIFIC_PROGRAMMING_LANGUAGE.~~
- ~~*blah blah* relating to $SPECIFIC_MONAD_INSTANCE.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

[a] yes, seriously, this is a thing.

Monad
Some mythbusting

Monad

- ~~for controlling side-effects.~~
- ~~make my program impure.~~
- ~~*something blah something* IO.~~
- ~~*blah blah* in `$SPECIFIC_PROGRAMMING_LANGUAGE`.~~
- ~~*blah blah* relating to `$SPECIFIC_MONAD_INSTANCE`.~~
- ~~*Monads Might Not Matter, so use Actors instead*~~[a]
- Too much bullshizzles to continue enumerating.

---

[a] yes, seriously, this is a thing.

Java 8 with the addition of higher-kinded polymorphism

```
interface Comonad<T> {
  <A> T<T<A>> duplicate(T<A> a);
  <X> X extract(T<X> x);
}
```

Haskell

```
class Comonad t where
  duplicate :: t a -> t (t a)
  extract :: t x -> x
```

Comonad

Like monad, comonad is

- Another interface, with laws, instances and operations.
- The *co* prefix denotes *categorical dual*.
- Like monad, is strictly stronger than functor.
- All comonads are functors.

Java 8/C# with the addition of higher-kinded polymorphism

```
interface Applicative<T> {
  <A, B> T<B> apply(T<Function<A, B>> f, T<A> a);
  <X> T<X> unit(X x);
}
```

Haskell

```
class Applicative t where
  apply :: t (a -> b) -> t a -> t b
  unit :: x -> t x
```

Applicative Functor

Well blimey mate. Guess what?

- It's just another interface, with laws, instances and operations.
- An applicative functor is
  - strictly stronger than functor. All applicatives are functors.
  - strictly weaker than monad. All monads are applicative.

Let's take a step back

Summary

Monads, Comonads, Applicative Functors . . .

All just the names of common interfaces.

- with many distinct and disparate instances.
- with many derived operations.

Each making different trade-offs for differences in utility.

Utility

When might I use any of these interfaces?

The same reason we already use interfaces.

Begin with a simple principle and exploit its diversity *to abstract away code repetition*.

Ubiquity

If these interfaces are so useful, why aren't they used everywhere?

- familiarity
- expressibility

Turning a list of potentially `null` into a potentially `null` list

```
args(list)
  result = new List;
  foreach el in list
    if(el == null)
      return null;
    else
      result.add(el);
  return result;
```

Applying a list of functions to a single value

```
args(list, t)
  result = new List;
  foreach el in list
    result.add(el(t));
  return result;
```

These expressions share structure

```
List (MaybeNull a) -> MaybeNull (List a)
List ((t ->)     a) -> (t ->)    (List a)
List (m          a) -> m         (List a)
```

Commonly called `sequence`.

Keep elements of a list matching a predicate with potential `null`

```
args(pred, list)
  result = new List;
  foreach el in list
    ans = pred(el);
    if(ans == null)
      return null;
    else if(ans)
      result.add(el);
  return result;
```

Keep elements of a list matching a predicate with argument passing

```
args(pred, list, t)
  result = new List;
  foreach el in list
    if(pred(el, t))
      result.add(el);
  return result;
```

Familiarity

Identifying the pattern again

These expressions share structure

```
(a -> MaybeNull Bool) -> List a -> MaybeNull (List a)
(a -> (t ->)    Bool) -> List a -> (t ->)    (List a)
(a -> m         Bool) -> List a -> m         (List a)
```

Commonly called `filter`.

Find the first element matching a predicate with potential `null`

```
args(pred, list)
  result = new List;
  foreach el in list
    ans = pred(el);
    if(ans == null)
      return null;
    else if(ans)
      return a;
  return null;
```

Find the first element matching a predicate with argument passing

```
args(pred, list, t)
  foreach el in list
    ans = pred(el, t);
    if(ans)
      return true;
  return false;
```

Familiarity

Identifying the pattern, once again

These expressions share structure

```
(a -> MaybeNull Bool) -> List a -> MaybeNull Bool
(a -> (t ->)     Bool) -> List a -> (t ->)     Bool
(a -> m          Bool) -> List a -> m          Bool
```

Commonly called `find`.

Turn a list of lists into a list

```
args(list)
  result = new List;
  foreach el in list
    result.append(el);
  return result;
```

Familiarity
Identifying the pattern, last time

Turn a potential `null` of potential `null` into a potential `null`

```
args(value)
  if(value == null)
    return null;
  else
    return value.get;
```

Familiarity

Identifying the pattern, last time

Apply to the argument, then apply to the argument

```
args(f, t)
  return f(t, t);
```

Familiarity

Identifying the pattern

These expressions share structure

```
List        (List a)       -> List      a
MaybeNull (MaybeNull a) -> MaybeNull a
(t ->)      ((t ->)     a) -> (t ->)    a
m           (m          a) -> m         a
```

Commonly called `join`.

Type systems and Expressibility Limits

Some type systems **limit expression** of abstraction.

- Java
- C#
- F#

Type systems and Expressibility Limits

These type systems are limited in the kinds of interfaces that they can describe.

The missing type system feature is called *higher-kinded polymorphism*.

Type systems and Expressibility Limits

Some type systems render abstraction **humanly intractable**

a

- JavaScript
- Ruby
- Python

[a] though some brave souls have tried

Type systems and Expressibility Limits

The likelihood of correctly utilising abstraction at the level of these interfaces approaches zero very quickly.

Type systems and Expressibility Limits

So we enter this feedback loop

The programmer is limited by tools, and then the tools limit the creative potential of the programmer.

The Parable of the listreverse project

Imagine, for a minute, a programming language that did not allow the programmer to generalise on list element types . . .

The Parable of the listreverse project

. . . and if you wanted to reverse a list of bananas, you would solve that problem specific to bananas.

The Parable of the listreverse project

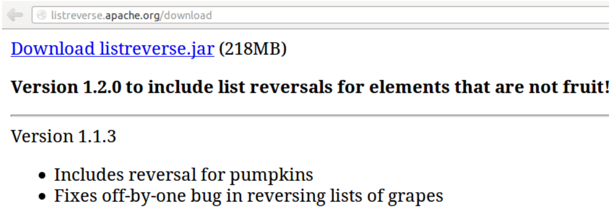
- But what if we then had to also reverse a list of oranges?
- Well, we would copy and paste the previous code :)

The Parable of the listreverse project

- But what if we then had to also reverse a list of oranges?
- Well, we would copy and paste the previous code :)

The Parable of the listreverse project

Soon enough, there would be a listreverse project and contributors, with all the different list reversals.


listreverse.apache.org/download

Download listreverse.jar (218MB)

Version 1.2.0 to include list reversals for elements that are not fruit!

Version 1.1.3

- Includes reversal for pumpkins
- Fixes off-by-one bug in reversing lists of grapes

The Parable of the listreverse project

So, you asked. . .

Why don't we use a programming environment that supports reversal on *any* element type?

The Parable of the listreverse project

and you were told. . .

> *The listreverse project is doing just fine and is used in many enterprise projects and has many contributors successfully incorporating it into their solutions.*

The Parable of the listreverse project

The reason

These interfaces are not exploited is due to *unfamiliarity* and tool support that discourages exploitation providing the perception of progress.

Mission

It is my mission is to change this and to help others exploit useful programming concepts, so please ask me more about it!