Monads
Technion – Institute of Technology
Software Design (236700)

Author: Gal Lalouche - Technion 2016 ©
Monads - motivation

Consider this simple code:

```java
return getStudent()
    .getCourse("Software Design")
    .getMoedA()
    .getGrade()
```

What happens when we need to handle nulls?

```java
Student s = getStudent();
if (s == null)
    return null;
Course c = s.getCourse("Software Design");
if (c == null)
    return null;
MoedA m = c.getMoedA();
if (m == null)
    return null;
return m.getGrade();
```
We already saw the `Optional<T>` monad last week. If everything returned an `Optional`, we could greatly simplify our code:

```java
class Optional<T> {
    public static <T> Optional<T> of(T t) {
        return new Optional(t); // throws exception on null
    }
    public <S> Optional<S> flatMap(Function<T, Optional<S>> f) {
        return this == Optional.empty() ?
            this : Optional.ofNullable(f.apply(get()));
    }
}
```

```
return getStudent()
    .flatMap(x -> x.getCourse("Software Design"))
    .flatMap(x -> x.getMoedA())
    .flatMap(x -> x.getGrade());
```
What is a Monad?

What?

Monoids: Relationships

A monoid is a category of endofunctors with binary operator as composition and identity being the identity endofunctor.

SAY MONAD

ONE MORE TIME

What's the big deal?

What's the problem?
What is a Monad?

- In **Haskell**, it is defined thusly:

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- sometimes called bind
```

- Okay, and in English (i.e., Java)?

```java
class Monad<T> {
    public static <T> Monad<T> of(T t); // return
    public <S> Monad<S> flatMap(Function<T, Monad<S>> f); // >>=
}
```
What is a Monad? (Cont.)

A proper Monad needs to satisfy the **Three Monadic Laws:**

I. **Left Identity:**
   
   \[
   \text{Monad.of}(x)\text{.flatMap}(f) \equiv f\text{.apply}(x)
   \]

II. **Right Identity:**

   \[
   m\text{.flatMap}(x \rightarrow \text{Monad.of}(x)) \equiv m
   \]

III. **Associativity:**

   \[
   m\text{.flatMap}(f)\text{.flatMap}(g) \equiv m\text{.flatMap}(x \rightarrow f\text{.apply}(x)\text{.flatMap}(g))
   \]

These are the exact laws of a **monoid**, where \text{of} is the **identity element**, and \text{flatMap} is the **binary operation**
But wait, what about `get`, `map` and other higher-order functions?

`flatMap` and `of` is enough to implement everything! We can implement `foreach` and `map`:

```java
class Monad<T> {
    public void foreach(Consumer<T> f) {
        flatMap(t -> {f.accept(t); return this;});
    }
    public <S> Monad<S> map(Function<T, S> f) {
        return flatMap(t -> of(f.apply(t)));
    }
}
```

In the real world, we may implement these differently (for performance, etc.), but the important thing is that `flatMap` and `of` are sufficient for implementing this.
Examples of Monads: `Try<T>`

- Sometimes, `Optional` is not enough
- If `Optional` is explicitly returning `null`, how do we explicitly throw an exception?
- Checked exceptions are too verbose, and work on a `method` level, rather than at a `variable` level

```java
class Try<T> {
    public static <T> Try<T> of(T t) {
        return new Success(t);
    }
    // this isn't the Monad's of!
    public static <T> Try<T> of(Callable<T> c) {
        try {
            return of(c.call());
        } catch (Exception e) {
            return new Failure(e);
        }
    }
    public T get();
}

class Success<T> extends Try<T> { ... }

class Failure<T> extends Try<T> { ... }
```
Examples of Monads (Cont.)

- We’ve seen Optional and Try
- List is also a Monad!
  - Many of the names given to methods make more sense for lists, e.g., foreach on most monads accesses just a single element, but it iterates over all elements in a list
- In Java, Stream is the only monadic implementation of a collection
- In Haskell, IO is a way to encapsulate state changing functions, in a language that has no concept of change (i.e., a pure language)
  - For example, reading input, printing output, side effects including mutability
- Next we will examine Future in depth
So what are Monads good for exactly?

- Optional and Try allow us to encapsulate **failure**
  - Our code may yield nothing
  - Or throw an exception
- List/Stream encapsulates **plurality**
  - Also non-determinism: we don’t know how many elements will be returned; could be none, one, or multiple
- IO encapsulates **side effects**
- Future encapsulates **asynchronicity**
Monads make the context explicit

- Monads let us encapsulate and structure contexts
- The context, assumptions and side effects of computations are made **explicit** rather than **implicit**
- We can transform our data using the same functions

![Diagram showing the relationship between T, S, and U with functions f, g, f', g', map(f), map(g), flatMap(f'), and flatMap(g').]
The Future<T> Monad - motivation

Suppose we have a task that takes a very long time:

```java
List<Picture> getAlbum(String name) {
    Profile p = getProfile(name); // fetches data from remote server
    List<Album> as = getAlbums(p); // fetches data …
    Album a = find(as, "Thailand"); // fetches data …
    return getPictures(a); // fetches data …
}
```

- How should handle this?
  - **Blocking** until all tasks are complete is sometimes **unacceptable**
  - For example, in a **user interface**
  - The **asynchronous** way is to use **callback**:

```java
void getAlbum(String name, Consumer<List<Picture>> callback) {
    new Thread(() -> callback.accept(getAlbum(name))).start();
}
```
The `Future<T>` Monad - motivation (Cont.)

- What if want to compose such callbacks?

```csharp
callback1(params1, t1 ->
    callback2(params2, t2 ->
        callback3(params3, t3 -> ...)
    ...
} );
```

- This (painful) pattern should be familiar to anyone who programmed using AJAX or similar frameworks
- The problem is that instead of holding T, we’re holding a `Future` of T
- When we want to compose abstractions of generic types, we want Monads!
The `Future<T>` Monad

class `Future<T>` {
    private Thread thread;
    private T value = null;
    public `Future`(`Supplier<T>` s) {
        this.thread = new Thread(() -> {
            this.value = s.get();
        });
        this.thread.start();
    }
    public T get() { thread.join(); return value; }
    // Monadic functions:
    public static<T> `Future<T>` `of`(T t) {
        // if we want, we can optimize this to be threadless
        return new `Future<>`(() -> t);
    }
    public <S> `Future<S>` `flatMap`(Function<T, `Future<S>`> f) {
        return new `Future<>`(() -> f.apply(this.get()).get());
    }
}
The Future<T> Monad - usage

But wait! What about a callback?
- That’s exactly the default foreach implementation
- Reminder:

```java
public void foreach(Consumer<T> f) {
    flatMap(t -> {f.accept(t); return this;});
}
```

- Usage in Future:

```java
new Future<>(() -> {
    long start = System.currentTimeMillis();
    for (long i = 1; i < 1000000000L; i++){}
    return System.currentTimeMillis() - start;
}).foreach(System.out::println);
```
The *Future<T>* Monad – usage (Cont.)

- Our transformed code is much cleaner:

```java
Future<List<Picture>> getAlbum(String name) {
    return getProfile(name)
        .flatMap(p -> getAlbums(p))
        .flatMap(as -> find(as, "Thailand"))
        .flatMap(a -> getPictures(a));
}
```

- What if we want to use transformations that are synchronous?
- We use *map* of course!
Additive Monads – Monads that are also Monoids

Some monads have two more properties:

1. An empty instance
   - e.g., an empty List (Nil, []), Optional.empty(), a Future that throws an exception
   - Will usually be implemented as a singleton (Do we need more than one empty list?)

2. A plus operator
   - e.g., List concatenation (++)
   - Trivial implementation (In the next slide, we will see why it’s valid)

```java
public Monad<T> plus(Monad<T> other) {
    return this == empty ? other : this;
}
```
Additive Monads (Cont.)

the following laws apply
1. m.plus(empty) == m (additive identity)
2. empty.plus(m) == m (additive identity)
3. empty.flatMap(f) == empty (multiplicative identity)
4. m.flatMap(empty) == empty (multiplicative identity)
5. m1.plus(m2).plus(m3) == m1.plus(m2.plus(m3)) (Assoc.)

(But I thought Monads were already monoids!
A monad is a monoid in the category of endofunctors. In layman’s terms, the methods of the monads are the elements in)
Additive Monads – Why?

We can implement `filter` using `empty`:

```
public Monad<T> filter(Predicate<T> p) {
    return flatMap(t -> p.test(t) ? of(t) : empty);
}
```

We can recover from failure with `plus` (not part of Java’s optional)
This is also called null coalescing. In C#, we use `??` for `nulls`.

```
.getOrElse(getOrElse(getOrElse())
.replaceWith(try2())
.replaceWith(try3())
```

Or with Try:

```
try1
  .recoverWith(try2())
  .recoverWith(try3())
```
Further Reading

- cdsmith.wordpress.com/2012/04/18/why-do-monads-matter/ (Language Agnostic / Haskellish)
- wiki.haskell.org/All_About_Monads (Haskell)
- learnyouahaskell.com/a-fistful-of-monads (Haskell)
- www.coursera.org/course/reactive (Scala)
- github.com/ReactiveX/RxJava (Java)
Appendix: Fluent APIs

In object-oriented languages, monadic transformations are chained in the order that they are evaluated:

```java
stream
  .map(x -> x * x)
  .filter(x -> x % 3 == 0)
  .map(x -> x + 8)
  .sum()
```

Examples include Java's Stream, C# LINQ (with IEnumerable), Ruby's Enumerable module, and even bash piping.

Without fluent APIs, we have to read functions in reverse order:

```java
sum(
  map(x -> x + 8,
       filter(x -> x % 3 == 0,
              map(x -> x * x, list))))
```

Languages with non-fluent API: ML, Haskell, Python, C++, Lisp(s)
Fluent APIs (cont.)

- Often enough, any API that returns *this* from method calls is called **Fluent**
- This pattern is especially common with **builder** objects
- But unlike moods, it doesn’t reverse the order of reading:

```java
new Builder()
    .setX(x)
    .setY(y)
    .build();
```

**VS.**

```java
Builder b = new Builder()
b.setX(x)
b.setY(y)
b.build();
```

- It’s possible to transform non-fluent APIs to fluent APIs either with wrapping / decorators, or special functions (e.g., in Haskell/F#: `x >>= f = f x`)
Appendix: Monads and Monoids

• I thought monads were already monoids! Why are “additive monads” special?
• Monads are monoids in the category of endofunctors. In other words, it’s not the monad instances themselves that form a monoid, but their methods.
  • Identity: static of method
  • Binary associative operator: flatMap method
• In a monoid class, the instances of the class are the elements of the monoid:
  • Identity: empty (a special object)
  • Binary associative operator: plus adds two instances to get a new instance
• Another way to think about it is this: Monads are parametric (Monad<T>); monoids aren’t (Integers are monoids but not monads).
Appendix: Monads in Haskell

• In languages with a richer type-system than Java, like Haskell or Scala, Monads are much more interesting/useful
• This is because you can write code that works on a general Monad
• For example, in our java implementation we have to implement map, foreach, filter, etc. for every new Monad class we define
• If we had typeclasses, we could implement it just once, for all monads!