# Towards Modularizing Machine Learning Jobs

Lance Co Ting Keh

Machine Learning @ Box

lance@box.com

Spark SUMMIT EAST

# About me

- Founding member of machine learning team at Box
- Designer and baby-sitter of the ML infrastructure
- MS and BS from Duke University specializing in ML and distributed systems

# Machine Learning at box

1. Security
2. Recommendations
3. Search Relevancy

# Basic Anatomy of an ML job

1. **Read and parse:** enforce type system on input data
2. **Collate:** join and deduplicate datasets
3. **Feature massaging:** transform, augment, drop or append features
4. **Algorithm:** regression, classification, clustering, topic models, sampling, etc
5. **Metrics**: area-under-curve, binary and multi-class, ranking, etc
6. **Format and output results:** write to various data stores and end points

# Sparkles

- Single logical unit of work
- Comprised of a series of Spark transformations and actions
- Enforces:
  - ☑ ***type safety***
  - ☑ ***modularity***
  - ☑ ***testability***
  - ☑ ***interpretable metrics and logging***
  - ☑ ***best Spark-programming practices***

# Usage

```
val finalSparkle =
  for {
    rddA <-  ReadTextFile(textPath)
    rddB <-  ReadSequenceFile(sequencePath)
    rddC <-  JsonDecode[TypeA](rddA)
    rddD <-  DecodeKeyValue[TypeA](rddB)
    rddE <-  JoinByKey(List(rddC, rddD))
    rddF <-  Localize(rddE)
    rddG <-  LabelPropagation(rddF)
    rddH <-  AccuracyMetric(rddG)
    rddI <-  Globalize(rddH)
    _    <-  WriteMetadata(rddI)
    _    <-  WriteHDFS(rddJ)(_.asJson.nospaces)
  } yield rddI
```

# Design

- Want to compose modules of Spark computation

  `flatMap: Sparkle[A] => (A => Sparkle[B]) => Sparkle[B]`

- Nice surface syntax

  for-comprehensions

- Leverage type system to enforce input/output

  It's just a function.. with some cleverness

# Implementation

```
/* Spark job that will produce an A */
trait Sparkle[A] {

}
```

# Implementation

```scala
/* Spark job that will produce an A */
trait Sparkle[A] {
  /* Same as function SparkContext => A */
  def run(sc: SparkContext): A
}
```

# Implementation

```
/* Spark job that will produce an A */
trait Sparkle[A] { parent =>
  /* Same as function SparkContext => A */
  def run(sc: SparkContext): A

  /* Compose with another Sparkle */
  def flatMap[B](f: A => Sparkle[B]): Sparkle[B] =
    new Sparkle[B] {
      def run(sc: SparkContext): B = {
        val parentResult: A = parent.run(sc)
        val nextSparkle: Sparkle[B] = f(parentResult)
        nextSparkle.run(sc)
      }
    }
}
```

# Implementation

```
/* Spark job that will produce an A */
trait Sparkle[A] { parent =>
  /* Same as function SparkContext => A */
  def run(sc: SparkContext): A

  /* Compose with another Sparkle */
  def flatMap[B](f: A => Sparkle[B]): Sparkle[B] =
    new Sparkle[B] {
      def run(sc: SparkContext): B = {
        val parentResult: A = parent.run(sc)
        val nextSparkle: Sparkle[B] = f(parentResult)
        nextSparkle.run(sc)
      }
    }
}
```

# Implementation

```
/* Spark job that will produce an A */
trait Sparkle[A] { parent =>
  /* Same as function SparkContext => A */
  def run(sc: SparkContext): A

  /* Compose with another Sparkle */
  def flatMap[B](f: A => Sparkle[B]): Sparkle[B]

  /* Transform output of Sparkle */
  def map[B](f: A => B): Sparkle[B] =
    new Sparkle[B] {
      def run(sc: SparkContext): B =
        f(parent.run(sc))
    }
}
```

# Implementation

```
trait Sparkle[A] {
  def run(sc: SparkContext): A
  def flatMap[B](f: A => Sparkle[B]): Sparkle[B]
  def map[B](f: A => B): Sparkle[B]
}
```

- A Sparkle is just a function that requires a SparkContext and produces some value A
- We compose Sparkles using flatMap and map
  - This detail is often syntactically hidden via for-comprehensions

# Implementation

```scala
trait Sparkle[A] {
  def run(sc: SparkContext): A
  def flatMap[B](f: A => Sparkle[B]): Sparkle[B]
  def map[B](f: A => B): Sparkle[B]
}
```

- How do Sparkle's take additional arguments?

```scala
case class JsonDecode[T](jsonRDD: RDD[String]) extends Sparkle[RDD[T]]
{
  /* Can reference `jsonRDD` in the function */
  def run(sc: SparkContext): RDD[String] = . . .
}
```

- This is how some languages support "closures"

# Anatomy of a Sparkle

Function arguments

```
case class Foo(a: A, b: B, c: C) extends Sparkle[Z] {
  def run(sc: SparkContext): Z
}
```

Required arguments whose value will be common across all composed Sparkles. Mostly used by Launchers, not Sparkles.

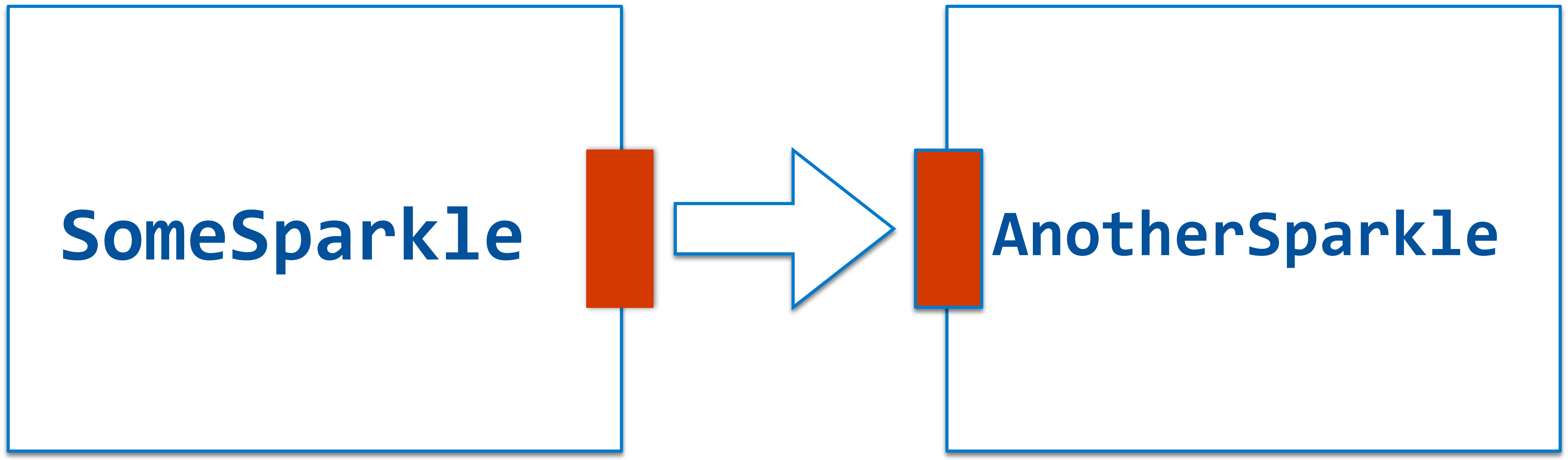
Function output type

# Type Safety with Sparkles

```
val finalSparkle =
  for {
    rddA <-  SomeSparkle(param)
    rddB <-  AnotherSparkle(rddA)
    rddC <-  YetAnotherSparkle(rddB)
    …
  } yield res
```

# Type Safety with Sparkles

```
val finalSparkle =
  for {
    rddA <-  SomeSparkle(param)
    rddB <-  AnotherSparkle(rddA)
    rddC <-  YetAnotherSparkle(rddB)
    …
  } yield res
```



# Type Safety with Sparkles

```
val finalSparkle =
  for {
    rddA <-  SomeSparkle(param)
    rddB <-  AnotherSparkle(rddA)
    rddC <-  YetAnotherSparkle(rddB)
    …
  } yield res
```

# Modularity with Sparkles

# Testing Sparkles

mock[RDD]

```
case class Foo(a: A, b: B, c: C) extends
Sparkle[Z] {

  def run(sc: SparkContext): Z = {
  …
  }
}    mock[SparkContext]
```

# Metrics and Logging with Sparkles

```
case class Foo(a: A, b: B, c: C) extends Sparkle[Z] {

  def run(sc: SparkContext): Z


  def runVerbose(sc: SparkContext): Z = {
   logUsefulThings()
   collectUsefulMetrics()

   run(sc)

   collectUsefulMetrics()
   logUsefulThings()
  }
}
```

# Good Code with Sparkles

# Summary

- Sparkles provide a **natural way** of thinking of Spark work
- **Increase velocity** with modularity and DRY-ness
- **Reduce programmatic** errors with type safety and testability
- Helps enforce best Spark **programming practices**
- **Standardize** metrics and logging