Verification Problems in SPARK Programs

Bow-Yaw Wang

Academia Sinica, Taiwan

Workshop on Constraints
29 August – 2 September 2016
What is Apache SPARK?

- **SPARK** is a more recent platform for large-scale data parallel processing.
- To reduce I/O overhead, **SPARK** is memory-based.
  - Keep intermediate results in memory;
  - Recompute results upon faults.
  - Google MAPREDUCE is disk-based.
- To avoid unnecessary computation, **SPARK** is lazy.
  - Computation is performed when necessary.
  - Think of lazy evaluation in purely functional languages such as Haskell.
Why Apache SPARK? I

- SPARK is still under (very) active development.
- It has lots of applications.
  - Go to databricks for more information.
- It is an easy-to-use distributed computing environment.
- Many distributed programs are developed and executed.
Programming in SPARK is so simple that novice programmers are writing distributed SPARK programs.

Concurrent programs are easy to have tricky bugs.

I believe this is a good opportunity for verification!
  - We can make impacts on SPARK development.
  - We can verify distributed SPARK programs.
  - We can contribute to big data researches.
Plan

1. SPARK Overview
2. Algebraic Properties
3. More Problems
4. Conclusions
SPARK Architecture

<table>
<thead>
<tr>
<th>Spark SQL</th>
<th>Spark Streaming</th>
<th>MLlib</th>
<th>GraphX</th>
</tr>
</thead>
<tbody>
<tr>
<td>core Spark</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC2, Hadoop YARN</td>
<td>HDFS, Hive</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- SPARK runs on top of distributed process/data clusters.
  - EC2, Hadoop YARN manage distributed processes.
  - HDFS, Hive manage distributed data.
- Core Spark provides basic APIs for distributed data processing.
- On top of core Spark, four libraries are available:
  - Spark SQL: database queries
  - Spark Streaming: streaming data processing
  - MLlib: machine learning library
  - GraphX: pregel-like graph library
A distributed collection of data items is a *Resilient Distributed Dataset (RDD)*.

A SPARK program creates an RDD from raw data stored in the underlying distributed file system.

A SPARK program computes by processing data items in RDDs.
An RDD can be changed into another RDD through *transformation*.

Data in an RDD can be collected through *actions*.

Typically, a SPARK program performs a sequence of transformations then followed by an action on an RDD.
An RDD consists of several *partitions*.

A partition is a basic block of data items.

- It is stored in a *worker* node of the underlying cluster.
- Computation on a partition is local.
A transformation is hence performed locally and distributively.
An action however requires local computation followed by communication with the *master* node.

![Diagram of a distributed data processing system](image)

- **RDD**
  - $Part_0$
  - $Part_1$
  - $\ldots$
  - $Part_n$

- **Sub**
  - $Sub_0$
  - $Sub_1$
  - $\ldots$
  - $Sub_n$

- **Result**
In practice, data flow can be complicated.

SPARK takes care of job scheduling, message passing, and even fault tolerance automatically.

Programmers only specify local computation.
We are interested in functional behaviors of SPARK programs.

To simplify our discussion, an abstract functional model for SPARK will be used.

Our abstract model ignores implementation details (performance, fault tolerance, etc) but focuses on functional behaviors.
A partition is modeled by a list.

For instance, a partition of data items 0, 1, 2 is modeled by the list [0, 1, 2].

In general, a partition consisting of data items of the type $\alpha$ is modeled by a list of the type $[\alpha]$.

Define

\[
\text{type Partition } \alpha = [\alpha]
\]
An RDD is a sequence of partitions.

Hence an RDD is modeled by a list of partitions.

That is, an RDD consisting of data items of the type $\alpha$ is modeled by a list of the type $[\text{Partition } \alpha]$.

Similarly, define

$$\text{type } \text{RDD } \alpha = [\text{Partition } \alpha]$$
A transformation transforms an RDD into another RDD.

Consider the transformation $\text{map}$ as an example.

Given a function from $\alpha$ to $\beta$, $\text{map}$ transforms an RDD whose data items are of the type $\alpha$ to another RDD whose data items are of the type $\beta$.

That is, we have

$$[\text{RDD } \alpha].\text{map} :: (\alpha \to \beta) \to \text{RDD } \beta$$

The functional specification of $\text{map}$ is straightforward for functional programmers.

- Exercise!
An action returns a result by collecting data items in an RDD.

Consider the action `reduce` as an example.

Given a function from \((\alpha \times \alpha)\) to \(\alpha\), `reduce` reduces an RDD whose data items are of the type \(\alpha\) to a value of the type \(\alpha\).

That is,

\[
[RDD \ \alpha].reduce :: (\alpha \times \alpha \to \alpha) \to \alpha
\]

Again, the functional specification is straightforward.

- Another exercise!
We have the functional specification of SPARK transformations and actions.

Our specification is in Haskell and Scala.

We use the executable specification to develop a distributed randomized graph coloring SPARK program.
Writing distributed programs is easy in SPARK.

Consider the SPARK program for word counts from its tutorial.

```scala
val textFile = sc.textFile("README.md")
val wordCounts =
    textFile.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey((a, b) => a + b)
```
val textFile = sc.textFile("README.md")
val wordCounts =
    textFile.flatMap(line => line.split("_"))
    .map(word => (word, 1))
    .reduceByKey((a, b) => a + b)

- `sc.textFile` reads a text file and returns an RDD whose data items are strings.
- `flatMap`, `map`, and `reduceByKey` are transformations.
val textFile = sc.textFile("README.md")
val wordCounts =
    textFile.flatMap(line => line.split("_"))
    .map(word => (word, 1))
    .reduceByKey((a, b) => a + b)

Let us read the code by types in our functional specification.

textFile :: String → RDD String

[RDD α].flatMap :: (α → [β]) → RDD β

textFile is an RDD of strings. Each string is a line in the file.
flatMap maps a line to a list of strings and then flattens all lists.
We have an RDD of strings. Each string is a word in the file.
val textFile = sc.textFile("README.md")
val wordCounts =
  textFile.flatMap(line => line.split("_"))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)

• For each string in the RDD, we map it to a pair of the string and 1.

[RDD $\alpha$].map :: $(\alpha \rightarrow \beta) \rightarrow$ RDD $\beta$

[RDD $(\alpha \times \beta)$].reduceByKey :: $(\beta \times \beta \rightarrow \beta) \rightarrow$

RDD $(\alpha \times \beta)$

• For each string appeared in a pair, we reduce the associated integers by summation.
val textFile = sc.textFile("README.md")
val wordCounts =
  textFile.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)

- Finally, we have

  wordCounts :: RDD (String × Int)

- Each string is associated with exactly an integer.
- The associated integer is the number of occurrences of the string in the file.
Lazy Evaluation

- Notice that a sequence of transformations are performed in word counts without any action.
- By lazy evaluation, SPARK will not perform any computation. This means SPARK returns immediately after the program is entered.
- The computation will be carried out only when concrete values are needed.
- For example, we can count the number of words by
  \[
  \text{wordCounts.reduce}(( (_, u), (_, v)) \mapsto ("\), u+v)) \quad .\_2
  \]
  \[
  // (a, b).\_2 = b
  \]
- Or, we can ask SPARK to compute an array of data items by
  \[
  \text{wordCounts.collect}()
  \]
Several libraries are built on top of core SPARK.

The SPARK GraphX library is a framework for implementing distributed graph algorithms.

In GraphX, a graph is represented by two RDDs:
- a vertex RDD contains vertices;
- an edge RDD contains edges.

It also has a pregel-like interface for distributed iterative computation over graphs.

We have interns to implement a handful of distributed graphs algorithms in GraphX during this summer.
- Correctness?
Plan

1. SPARK Overview
2. Algebraic Properties
3. More Problems
4. Conclusions
val textFile = sc.textFile("README.md")
val wordCounts =
    textFile.flatMap(line => line.split("_"))
    .map(word => (word, 1))
    .reduceByKey((a, b) => a + b)
wordCounts.reduce((((_, u), (_, v)) => ("", u+v))._2

- Observe that SPARK programmers only specify local computation in transformations and actions.
- It is easy to write distributed SPARK programs.
- So what can go wrong?
Similar to **MAPREDUCE**, concurrency makes computation non-deterministic.
Due to concurrency and efficiency, SPARK computation is inherently non-deterministic.

What we look for is deterministic outcomes.

That is, the results will be the same for all non-deterministic computation.
To ensure deterministic outcomes, SPARK requires certain algebraic properties on local computation.

For instance, the documentation of the action `reduce` says:

“Reduces the elements of this RDD using the specified commutative and associative binary operator.”

In the word counts example, the binary operator is both associative and commutative.

\[(\text{"", } u + v) \]
Algebraic Properties I

- Let $\oplus : \alpha \times \alpha \rightarrow \alpha$ be a binary operator.
- $\oplus$ is associative if for every $x, y, z$
  \[(x \oplus y) \oplus z = x \oplus (y \oplus z).\]
- $\oplus$ is commutative if for every $x, y$
  \[x \oplus y = y \oplus x.\]
- 0 is a neutral element of $\oplus$ if for every $x$
  \[x \oplus 0 = 0 \oplus x = x.\]
- These algebraic properties are required for various SPARK actions.
More generally, consider any universally quantified equations over functions and constants.

\[ t ::= x \mid c \mid f(t, \ldots, t) \]
\[ \forall x. t = t \]

- **Distributive law**

\[ \forall x, y, z. x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z) \]

- **Idempotent unary function**

\[ \forall x. f(f(x)) = f(x) \]

- **Can we prove such algebraic properties hold for SPARK operators?**
Abstractly, algebraic properties are easy to specify.

Their concrete interpretations necessarily depend on SPARK operators in practice.

We divide SPARK operators into three classes:

- numerical operators;
- abstract data operators;
- arbitrary user-defined operators.
• Algebraic properties over numerical SPARK operators are straightforward to interpret.
• They are just different from mathematical operators.
• Mathematical addition and multiplication are both associative and commutative.
• SPARK programs rarely use exact mathematical addition and multiplication.
• For bounded integers, are \( + \) and \( \ast \) still associative and commutative?
• For floating-point numbers, are \( + \) and \( \ast \) still associative and commutative?
Abstract data types are employed in SPARK programs.

Their operations are often used in transformations and actions.

Consider the union operator $\cup$ for the data type $\text{Set}[\alpha]$.

We know the set-theoretic union $\cup$ is associative and commutative.

Do we really have, say, $A \cup B = B \cup A$?

Probably not. The concrete data structure for $A \cup B$ is likely to be different from $B \cup A$.

What do we mean (and show) that $\cup$ is commutative?
For user-defined SPARK operators, interpretations of algebraic properties are even less clear.

They may not correspond to any abstract objects.

What do we mean (and show) that such operators satisfy certain algebraic properties?
When we prove the correctness of data type implementations, similar problems arise.

Let $U$ be an implementation of the set-theoretic union $\cup$.

How do we show that $U$ is correct?

Tony Hoare came up with a solution back in 1972.
Let $A$ be a *representation function* mapping any concrete set $S$ to an abstract set $S$.

To show $U$ is correct, it is to show

$$A(A \cup B) = A(A) \cup A(B)$$

for every concrete sets $A$ and $B$. 
The representation function lifts concrete structures to abstract objects.

Using a representation function, it is clear how to verify algebraic properties.

For example, \( U \) is commutative if for every \( A \) and \( B \)

\[
\mathcal{A}(A \cup B) = \mathcal{A}(B \cup A).
\]
In other words, we can prove algebraic properties of SPARK operators after establishing correctness of abstract data types.

This looks good on paper.

But it is an overkill.

We only want to show algebraic properties, not correct data type implementations.

What about arbitrary user-defined SPARK operators?

Representation functions can be obscure.
To find a simpler solution, let us take another look at algebraic properties.

Let $U$ be an implementation of the set-theoretic union $\cup$.

Since the concrete structure of $A \cup B$ is different from those of $B \cup A$, we have

$$A \cup B \neq B \cup A.$$  

We interpret the equality $=$ too narrowly.
Define the binary relation \( \approx \) over concrete structures:

\[
\approx = \{(S, T) : S \text{ and } T \text{ denote the same set.}\}
\]

Then we have for every \( A \) and \( B \)

\[
A \cup B \approx B \cup A.
\]

This is the intended interpretation of algebraic properties.
Our interpretation in fact coincides with Hoare’s formulation.

The binary relation $\approx$ can simply be

$$\approx = \{(S, T) : A(S) = A(T)\}.$$ 

Then for every $A$ and $B$ $A \cup B \approx B \cup A$ is a paraphrase of

for every $A$ and $B$ $A(A \cup B) = A(B \cup A)$.

This particular definition of $\approx$ is fine but unsatisfactory.

- It needs the representation function $A$. 
Intended interpretations of algebraic properties depend on the interpretation of equality.
  - That is, the binary relation $\approx$.

Two problems remain:
- What are the criteria for the soundness of $\approx$?
  - Taking $\approx$ to be the universal binary relation is not sound. Why?
- How to define such $\approx$ without representation functions?
  - Can it be computed automatically?
Let us try to formulate $\approx$.

What is a proper intended interpretation of equality?

Intuitively, $\approx$ relates equivalent concrete structures.

But how do we define “equivalence?”

The process algebra community has solved this problem.

Thanks to Robin Milner.
Interpretation by Bisimulation II

- Informally, two processes are equivalent if no action can differentiate one from the other.
- In our context, we have
  
  two concrete structures are equivalent if no operation can differentiate one from the other.
- For instance, a good definition of \( \approx \) must satisfy
  
  For every \( A, A', B, B' \)

  \[ A \approx A' \text{ and } B \approx B' \text{ imply } A \cup B \approx A' \cup B' \]

  among other operations of interests.
Interpretation by Bisimulation III

- Similar to bisimulation, $\approx$ is relative to operations.
  - If we only insert elements to sets, any two sets are “equal.”
  - Insertion cannot differentiate one set from another.
- Depending on operations of interests, different definitions of $\approx$ are available.
  - Representation functions are no longer needed.
- $\approx$ does not verify correctness of data type implementations.
  - Just like bisimulation, it only relates equivalent concrete structures.
  - It is easier to check and not an overkill.
For SPARK, concrete structures often have higher-order operations.
  - Operations take functions as parameters.

Our theory also covers such concrete structures.

We borrow from logical relations in λ-calculus.
Verifying Algebraic Properties

- Algebraic properties can now be formally verified as follows.
  - Specify $\approx$.
  - Check if $\approx$ is a bisimulation (or logical relation).
  - Check algebraic properties by interpreting equality with $\approx$.
- No abstract objects are referred in this process.
- Verification can be done in automated software verification tools.
  - without proof assistants, that is.
- We have case studies in LEON.
Plan

1. SPARK Overview
2. Algebraic Properties
3. More Problems
4. Conclusions
Interested?

- There are more problems to be addressed in SPARK programming.
- I will briefly describe two (unsolved) problems.
  - Stability problem in numerical computation
  - SQL query optimization
Recall that associative and commutative SPARK operators are required for transformations and actions.

But floating-point operators do not have such algebraic properties.

Particularly, floating-point addition is not associative.

Let \( x = 1.0 \), \( y = -1.0 \), and \( z = 1e-16 \).

\[ 1e-16 = (x + y) + z \neq x + (y + z) = 0.0 \]

No sound definition of \( \approx \) can relate \( 1e-16 \) and \( 0.0 \).

Floating-point addition should not be used in SPARK transformations and actions.

But they are widely used!
Non-deterministic outcomes can in fact be observed from a private function `AreaUnderCurve.of` in MLlib. Given an RDD of sample points of a curve, `AreaUnderCurve.of` performs the numerical integration by summing up areas of trapezoids defined by the curve. Let us consider the following integration

\[
\int_{-2}^{2} x^{73} \, dx
\]

However `AreaUnderCurve.of` returns different answers in 50 runs on the same evenly distributed sample points on \(x^{73}\). The outcomes are not deterministic.
Non-deterministic outcomes are perhaps tolerable if they are not very different.

With 8 worker nodes, we observe values from $-8192.0$ to $12288.0$ in 50 runs.

Recall the integration

$$\int_{-2}^{2} x^{73} \, dx = 0$$

since $x^{73}$ is an odd function.

Outcomes are (very) different from the mathematical result.
Unstable floating-point computation is always a problem in numerical methods.

To have stable computation, a sequence of floating-point operations need to be performed in a certain order.
  ▶ Mathematically equivalent sequences of such operations can have different stability.

For SPARK, the problem is even more uncontrollable.

Floating-point operations are not executed in any fixed-order due to concurrency.

Is there a way to ensure stability in SPARK computation?
SPARK SQL provides an interface to write database queries.
In SPARK, databases are modeled by RDDs.
SQL queries are thus translated to SPARK transformations and actions.
SPARK SQL goes even further to optimize the translation.
More than 30 local optimizations.
  ▶ Data flow analysis, static evaluation, etc.

Divided into 11 batched optimizations.
  ▶ batched optimizations are iteratively performed.

Correctness?
  ▶ stability, algebraic properties of user-defined queries?

Performance?
  ▶ a performance model for SPARK?
Conclusions

- SPARK is now a popular and simple programming model for distributed computing.
- Unlike classical distributed computing, concurrency is restricted in SPARK computation.
- We have shown an abstract functional model and a proof technique for SPARK programs.
- We believe there are many opportunities for the verification community to offer.