An Executable Sequential Specification for SPARK Aggregation

Yu-Fang Chen[1], Chih-Duo Hong[1], **Ondřej Lengál**[1,2], Shin-Cheng Mu[1], Nishant Sinha[3], Bow-Yaw Wang[1]

[1]Academia Sinica, Taiwan
[2]Brno University of Technology, Czech Republic
[3]IBM Research, India

19 May 2017 (NETYS'17)

Current Trends in CAV (Computer-Aided Verification)

Current Trends in CAV (Computer-Aided Verification):

- verification of HW

Current Trends in CAV (Computer-Aided Verification)

Current Trends in CAV (Computer-Aided Verification):

- verification of HW
- verification of sequential programs:
  - w/ integers
  - w/ floats
  - w/ heap manipulation
  - ...

Current Trends in CAV (Computer-Aided Verification)

Current Trends in CAV (Computer-Aided Verification):

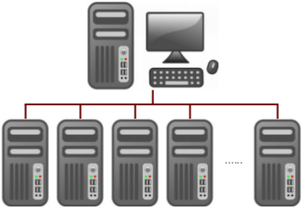
- verification of HW
- verification of sequential programs:
  - w/ integers
  - w/ floats
  - w/ heap manipulation
  - ...
- verification of concurrent programs
  - mutual exclusion protocols
  - concurrent data structures
  - ...

Current Trends in Computer Science:

- machine learning, deep neural networks, IoT, smart-$*$, ...

Current Trends in Computer Science:

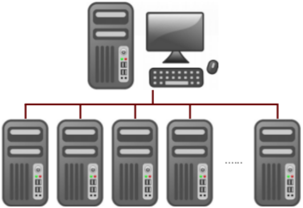
- machine learning, deep neural networks, IoT, smart-$*$, ...
- big data & cluster computing
  - sciences
  - AI
  - advertising analysis
  - data mining
  - biology
  - search engines
  - market value:
    - expected over $50 billion by 2020



Current Trends in Computer Science

Current Trends in Computer Science:

- machine learning, deep neural networks, IoT, smart-$*$, ...
- big data & cluster computing
  - sciences
  - AI
  - advertising analysis
  - data mining
  - biology
  - search engines
  - market value:
    - expected over $50 billion by 2020



- Can verification be applied here, too?

Frameworks for Big Data

Programming for Big Data:

Frameworks for Big Data

Programming for Big Data:

- many repeating tasks
  - distribution of data on nodes

Frameworks for Big Data

Programming for Big Data:

- many repeating tasks
  - distribution of data on nodes
  - collection of computed results

Frameworks for Big Data

Programming for Big Data:

- many repeating tasks
  - distribution of data on nodes
  - collection of computed results
- $\rightsquigarrow$ frameworks that create abstraction over the communication

Frameworks for Big Data

Programming for Big Data:

- many repeating tasks
  - distribution of data on nodes
  - collection of computed results
- $\rightsquigarrow$ frameworks that create abstraction over the communication
- web services providing easy-to-setup computation using these (Amazon, Microsoft, IBM, ...)

Frameworks for Big Data

Programming for Big Data:

- many repeating tasks
  - distribution of data on nodes
  - collection of computed results
- $\rightsquigarrow$ frameworks that create abstraction over the communication
- web services providing easy-to-setup computation using these (Amazon, Microsoft, IBM, ...)
- examples:
  - Hadoop MAPREDUCE
  - PIG
  - HIVE
  - **Apache SPARK**

Frameworks for Big Data

New verification problems:

Frameworks for Big Data

New verification problems:

- verification of correctness of the frameworks

Frameworks for Big Data

New verification problems:

- verification of correctness of the frameworks
- verification of correctness of user programs
  - correctness: checking special properties

Apache SPARK

Apache SPARK:

- *successor* of Hadoop MAPREDUCE
  - claims to be up to 100× faster due to in-memory computation

Apache SPARK

Apache SPARK:

- *successor* of Hadoop MAPREDUCE
  - claims to be up to 100$\times$ faster due to in-memory computation
- a relaxed fault tolerant model
  - sub-results are recomputed upon faults

Apache SPARK

Apache SPARK:

- *successor* of Hadoop MAPREDUCE
  - claims to be up to 100$\times$ faster due to in-memory computation
- a relaxed fault tolerant model
  - sub-results are recomputed upon faults
- lazy evaluation semantics

Apache SPARK

Apache SPARK:

- *successor* of Hadoop MAPREDUCE
  - claims to be up to 100$\times$ faster due to in-memory computation
- a relaxed fault tolerant model
  - sub-results are recomputed upon faults
- lazy evaluation semantics
- contains libraries for
  - processing graphs
  - streaming computation
  - machine learning
  - SQL-based database computation
  - ...

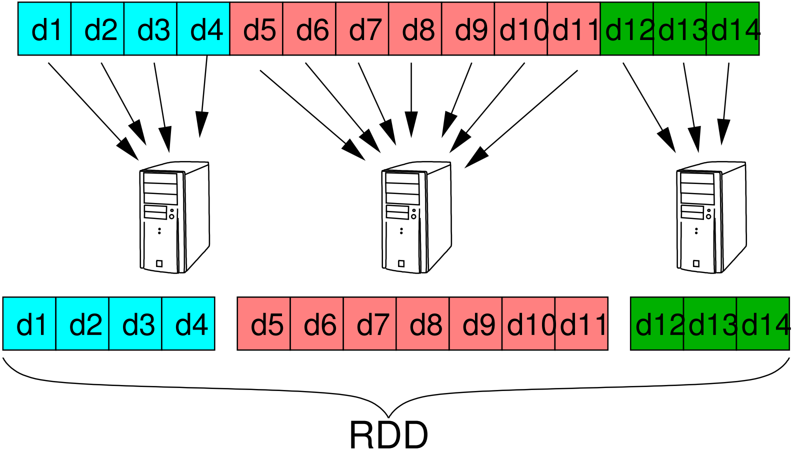
Apache SPARK

RDD—Resilient Distributed Dataset:

- the principal data abstraction

RDD—Resilient Distributed Dataset:

- the principal data abstraction



Computation in SPARK

- map-style
  - **map**, **filter**

```
RDD rdd = ...
RDD newRdd = rdd.map(λ x . x * 2)
```

Computation in SPARK

- map-style
  - **map**, **filter**

```
RDD rdd = ...
RDD newRdd = rdd.map(λ x . x * 2)
```

- aggregation
  - **aggregate**, **reduce**
  - **treeAggregate**, **treeReduce**

```
RDD rdd = ...
int sum = rdd.reduce(λ x y . x + y)
```

Computation in SPARK

- map-style
  - **map**, **filter**

```
RDD rdd = ...
RDD newRdd = rdd.map(λ x . x * 2)
```

- aggregation
  - **aggregate**, **reduce**
  - **treeAggregate**, **treeReduce**

```
RDD rdd = ...
int sum = rdd.reduce(λ x y . x + y)
```

- combined
  - **aggregateByKey**, **reduceByKey**

```
PairRDD rdd = ...
PairRDD sum = rdd.reduceByKey(λ x y . x + y)
```

Formal specification of SPARK aggregation

Formal specification of SPARK aggregation

- SPARK is written in SCALA (bindings to Java, Python, ...)
  - multi-paradigm programming language
  - both imperative and functional code (side-effects)
  - unclear semantics

Formal specification of SPARK aggregation

Formal specification of SPARK aggregation

- SPARK is written in SCALA (bindings to Java, Python, . . .)
  - multi-paradigm programming language
  - both imperative and functional code (side-effects)
  - unclear semantics
- documentation is not clear about requirements of functions

Formal specification of SPARK aggregation

- SPARK is written in SCALA (bindings to Java, Python, ...)
  - multi-paradigm programming language
  - both imperative and functional code (side-effects)
  - unclear semantics
- documentation is not clear about requirements of functions
- our contribution:
  - PURESPARK: a specification of aggregation functions in HASKELL
    - purely functional language
    - executable specification
    - suitable for formal reasoning (e.g. AGDA)

Formal specification of SPARK aggregation

Formal specification of SPARK aggregation

- SPARK is written in SCALA (bindings to Java, Python, . . .)
  - multi-paradigm programming language
  - both imperative and functional code (side-effects)
  - unclear semantics
- documentation is not clear about requirements of functions
- our contribution:
  - PURESPARK: a specification of aggregation functions in HASKELL
    - purely functional language
    - executable specification
    - suitable for formal reasoning (e.g. AGDA)
  - correctness requirements on aggregation functions (next slide)

Formal specification of SPARK aggregation

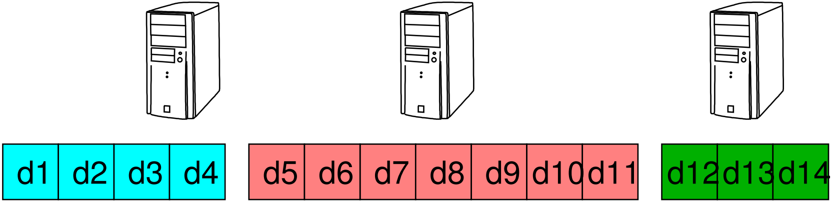
Formal specification of SPARK aggregation

- SPARK is written in SCALA (bindings to Java, Python, . . .)
  - multi-paradigm programming language
  - both imperative and functional code (side-effects)
  - unclear semantics
- documentation is not clear about requirements of functions
- our contribution:
  - PURESPARK: a specification of aggregation functions in HASKELL
    - purely functional language
    - executable specification
    - suitable for formal reasoning (e.g. AGDA)
  - correctness requirements on aggregation functions (next slide)
  - analysis of case studies—finding numeric instability in ML library

Commutativity in SPARK aggregation

Commutativity in SPARK aggregation

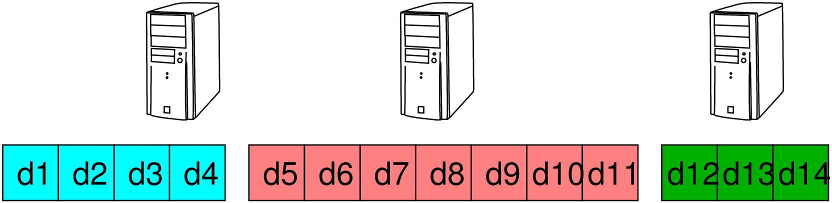
$$\texttt{aggregate}(seq, comb, \perp, rdd)$$



Commutativity in SPARK aggregation

Commutativity in SPARK aggregation

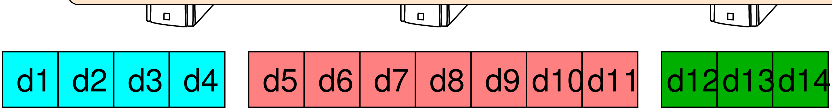
$$\texttt{aggregate}(seq, comb, \bot, rdd)$$



$$\texttt{foldl}(seq, \bot, [d_1, d_2, d_3, d_4]) \quad \rightsquigarrow r_A$$
$$\texttt{foldl}(seq, \bot, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \quad \rightsquigarrow r_B$$
$$\texttt{foldl}(seq, \bot, [d_{12}, d_{13}, d_{14}]) \quad \rightsquigarrow r_C$$

Commu

$$\texttt{foldl} :: (B \to A \to B) \to B \to [A] \to B$$
$$\texttt{foldl}(seq, \bot, [\,]) = \bot$$
$$\texttt{foldl}(seq, \bot, x : xs) = \texttt{foldl}(seq, \underline{seq(\bot, x)}, xs)$$

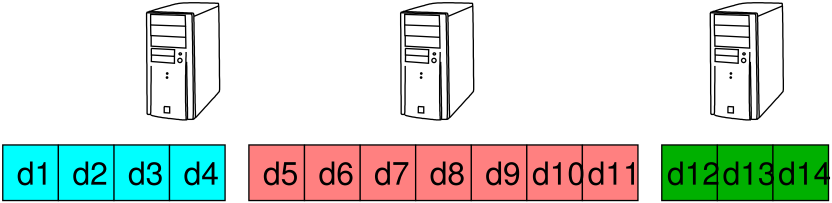


$$\texttt{foldl}(seq, \bot, [d_1, d_2, d_3, d_4]) \quad \rightsquigarrow r_A$$
$$\texttt{foldl}(seq, \bot, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \quad \rightsquigarrow r_B$$
$$\texttt{foldl}(seq, \bot, [d_{12}, d_{13}, d_{14}]) \quad \rightsquigarrow r_C$$

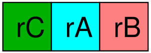
Commutativity in SPARK aggregation

Commutativity in SPARK aggregation

$$\texttt{aggregate}(seq, comb, \perp, rdd)$$

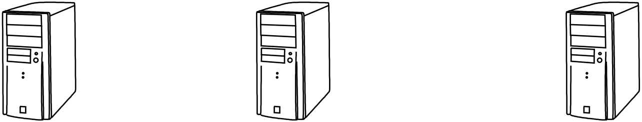


$$\texttt{foldl}(seq, \perp, [d_1, d_2, d_3, d_4]) \quad \rightsquigarrow r_A$$
$$\texttt{foldl}(seq, \perp, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \quad \rightsquigarrow r_B$$
$$\texttt{foldl}(seq, \perp, [d_{12}, d_{13}, d_{14}]) \quad \rightsquigarrow r_C$$

- collected results: 

Commutativity in SPARK aggregation

$$\texttt{aggregate}(seq, comb, \perp, rdd)$$



d1	d2	d3	d4		d5	d6	d7	d8	d9	d10	d11		d12	d13	d14

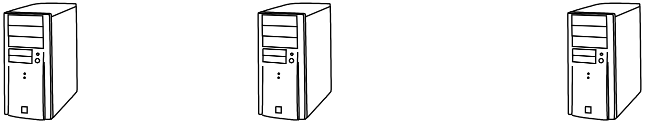
$$\texttt{foldl}(seq, \perp, [d_1, d_2, d_3, d_4]) \rightsquigarrow r_A$$
$$\texttt{foldl}(seq, \perp, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \rightsquigarrow r_B$$
$$\texttt{foldl}(seq, \perp, [d_{12}, d_{13}, d_{14}]) \rightsquigarrow r_C$$

- collected results: | rC | rA | rB |

nondeterministic!

Commutativity in SPARK aggregation

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd})$$



d1	d2	d3	d4		d5	d6	d7	d8	d9	d10	d11		d12	d13	d14

$$\texttt{foldl}(\mathit{seq}, \perp, [d_1, d_2, d_3, d_4]) \rightsquigarrow r_A$$

$$\texttt{foldl}(\mathit{seq}, \perp, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \rightsquigarrow r_B$$

$$\texttt{foldl}(\mathit{seq}, \perp, [d_{12}, d_{13}, d_{14}]) \rightsquigarrow r_C$$

- collected results:

rC	rA	rB

nondeterministic!

$$\texttt{foldl}(\mathit{comb}, \perp, [r_C, r_A, r_B]) \rightsquigarrow \mathit{result}$$

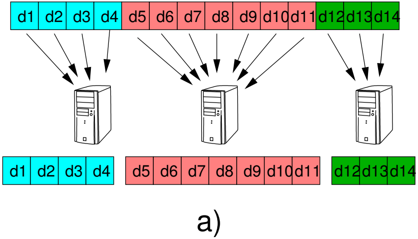
Commutativity in SPARK aggregation

Two sources of nondeterminism:

Commutativity in SPARK aggregation

Two sources of nondeterminism:

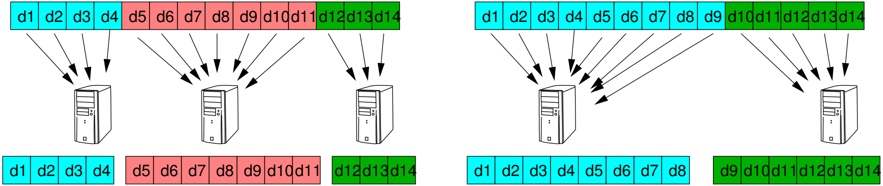
1. Partitioning into RDD



Commutativity in SPARK aggregation

Two sources of nondeterminism:

1. Partitioning into RDD



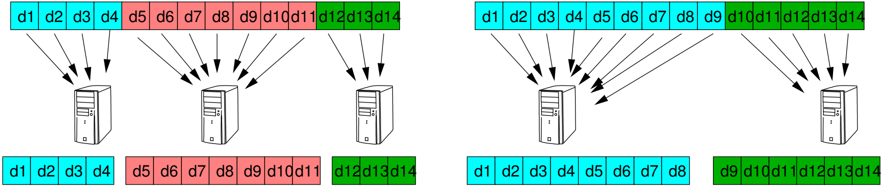
a) b)

2. Order in which nodes send partial results
   - rC rA rB
   - rC rB rA
   - rA rB rC

Commutativity in SPARK aggregation

Two sources of nondeterminism:

1. Partitioning into RDD



a) b)

2. Order in which nodes send partial results
   - rC rA rB
   - rC rB rA
   - rA rB rC

`aggregate` can yield different results!!!

Commutativity in SPARK aggregation

Example of a nondeterministic aggregation

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd})$$

$$\begin{aligned} \mathit{seq}(\mathit{acc}, x) &= \mathit{acc} + x \\ \mathit{comb}(\mathit{lhs}, \mathit{rhs}) &= \mathit{rhs} + \mathit{rhs} \qquad (\mathit{typo}) \\ \perp &= 0 \end{aligned}$$

Commutativity in SPARK aggregation

Example of a nondeterministic aggregation

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd})$$

$$\begin{aligned}
\mathit{seq}(\mathit{acc}, x) &= \mathit{acc} + x \\
\mathit{comb}(\mathit{lhs}, \mathit{rhs}) &= \mathit{rhs} + \mathit{rhs} && (\mathit{typo}) \\
\perp &= 0
\end{aligned}$$

d1	d2	d3	d4		d5	d6	d7	d8	d9	d10	d11		d12	d13	d14

Commutativity in SPARK aggregation

Example of a nondeterministic aggregation

$$\texttt{aggregate}(seq, comb, \perp, rdd)$$

$$\begin{aligned} seq(acc, x) &= acc + x \\ comb(lhs, rhs) &= rhs + rhs \qquad (typo) \\ \perp &= 0 \end{aligned}$$

d1	d2	d3	d4

d5	d6	d7	d8	d9	d10	d11

d12	d13	d14

$\texttt{foldl}(seq, 0, [d_1, d_2, d_3, d_4]) = (((0 + d_1) + d_2) + d_3) + d_4 \quad \rightsquigarrow r_A$

$\texttt{foldl}(seq, 0, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) = \ldots \quad \rightsquigarrow r_B$

$\texttt{foldl}(seq, 0, [d_{12}, d_{13}, d_{14}]) = \ldots \quad \rightsquigarrow r_C$

Commutativity in SPARK aggregation

Example of a nondeterministic aggregation

$$\texttt{\textbf{aggregate}}(seq, comb, \perp, rdd)$$

$$\begin{aligned} seq(acc, x) &= acc + x \\ comb(lhs, rhs) &= rhs + rhs \qquad\qquad (typo) \\ \perp &= 0 \end{aligned}$$

d1	d2	d3	d4		d5	d6	d7	d8	d9	d10	d11		d12	d13	d14

$$\texttt{\textbf{foldl}}(seq, 0, [d_1, d_2, d_3, d_4]) = (((0 + d_1) + d_2) + d_3) + d_4 \quad \rightsquigarrow r_A$$

$$\texttt{\textbf{foldl}}(seq, 0, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) = \ldots \quad \rightsquigarrow r_B$$

$$\texttt{\textbf{foldl}}(seq, 0, [d_{12}, d_{13}, d_{14}]) = \ldots \quad \rightsquigarrow r_C$$

Collecting results:

- rC rA rB : $\texttt{\textbf{foldl}}(comb, 0, [r_C, r_A, r_B]) = 2r_B$
- rA rB rC : $\texttt{\textbf{foldl}}(comb, 0, [r_A, r_B, r_C]) = 2r_C$

Example of a nondeterministic aggregation

$$\texttt{aggregate}(seq, comb, \bot, rdd)$$

$$\begin{aligned} seq(acc, x) &= acc + x \\ comb(lhs, rhs) &= rhs + rhs \qquad (typo) \\ \bot &= 0 \end{aligned}$$

d1	d2	d3	d4		d5	d6	d7	d8	d9	d10	d11		d12	d13	d14

$\texttt{foldl}(seq, 0, [d_1, d_2, d_3, d_4]) = (((0 + d_1) + d_2) + d_3) + d_4 \quad \rightsquigarrow r_A$

$\texttt{foldl}(seq, 0, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) = \ldots \quad \rightsquigarrow r_B$

$\texttt{foldl}(seq, 0, [d_{12}, d_{13}, d_{14}]) = \ldots \quad \rightsquigarrow r_C$

Collecting results:

- rC rA rB: $\texttt{foldl}(comb, 0, [r_C, r_A, r_B]) = 2r_B$
- rA rB rC: $\texttt{foldl}(comb, 0, [r_A, r_B, r_C]) = 2r_C$

$$2r_B \neq 2r_C$$

Commutativity in SPARK aggregation

Commutativity of `aggregate`

Definition

A call

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd})$$

is **commutative** iff

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd}(L)) = \texttt{foldl}(\mathit{seq}, \perp, L)$$

for every partitioning $\mathit{rdd}(L)$ of L.

Commutativity in SPARK aggregation

Commutativity of `aggregate`

Definition

A call

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd})$$

is **commutative** iff

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \perp, \mathit{rdd}(L)) = \texttt{foldl}(\mathit{seq}, \perp, L)$$

for every partitioning $\mathit{rdd}(L)$ of L.

- i.e., **`aggregate`** is an implementation of **`foldl`**

Commutativity in SPARK aggregation

Commutativity of `aggregate`

Definition

A call

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \bot, \mathit{rdd})$$

is **commutative** iff

$$\texttt{aggregate}(\mathit{seq}, \mathit{comb}, \bot, \mathit{rdd}(L)) = \texttt{foldl}(\mathit{seq}, \bot, L)$$

for every partitioning $\mathit{rdd}(L)$ of L.

- i.e., **`aggregate`** is an implementation of **`foldl`**
- if a call to **`aggregate`** is commutative:
  - the call is deterministic
  - when analyzing the program, we can assume one partitioning

Commutativity in SPARK aggregation

Conditions for commutative `aggregate`

Theorem

Consider rdd of elements of type $\mathbb{T}$ and $\bot \in \mathbb{R}$. A call

$$\texttt{aggregate}(seq, comb, \bot, rdd)$$

is commutative iff

1. $(img(\texttt{foldl}(seq, \bot)), comb, \bot)$ *is a commutative monoid and*
2. *for all $d \in \mathbb{T}$ and $e \in img(\texttt{foldl}(seq, \bot))$, it holds that*
$$seq(e, d) = comb(e, seq(z, d)).$$

Commutativity in SPARK aggregation

Conditions for commutative `aggregate`

Theorem

Consider rdd of elements of type $\mathbb{T}$ *and* $\perp \in \mathbb{R}$. *A call*

$$\texttt{aggregate}(seq, comb, \perp, rdd)$$

is commutative iff

1. $(img(\texttt{foldl}(seq, \perp)), comb, \perp)$ *is a commutative monoid and*
2. *for all* $d \in \mathbb{T}$ *and* $e \in img(\texttt{foldl}(seq, \perp))$, *it holds that*
$$seq(e, d) = comb(e, seq(z, d)).$$

Safe approximation:

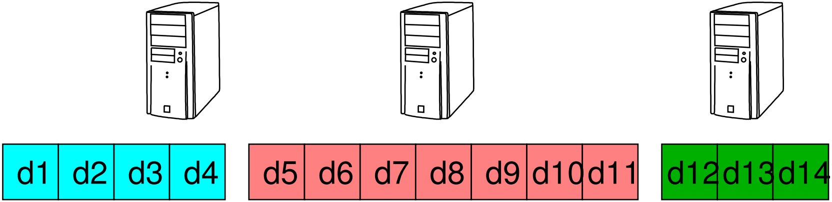
…is commutative if

1. $(\mathbb{R}, comb, \perp)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in \mathbb{R}$, it holds that
$$seq(e, d) = comb(e, seq(z, d)).$$

Commutativity in SPARK aggregation

SPARK `reduce`

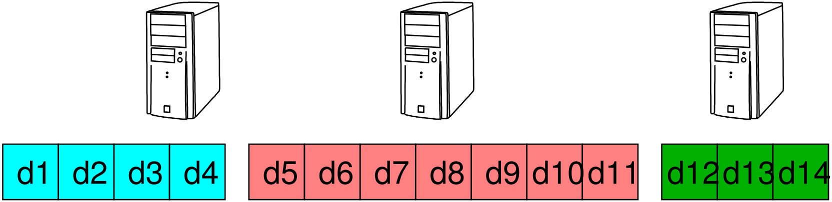
$$\texttt{reduce}(\mathit{comb}, \mathit{rdd})$$



Commutativity in SPARK aggregation

SPARK reduce

reduce(*comb*, *rdd*)



$\texttt{reducel}(comb, [d_1, d_2, d_3, d_4]) \rightsquigarrow r_A$

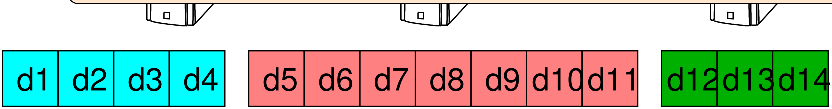
$\texttt{reducel}(comb, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \rightsquigarrow r_B$

$\texttt{reducel}(comb, [d_{12}, d_{13}, d_{14}]) \rightsquigarrow r_C$

Commutativity in SPARK aggregation

SPARK r

$$\texttt{reducel} :: (A \to A \to A) \to [A] \to A$$
$$\texttt{reducel}(comb, x : xs) = \texttt{foldl}(comb, x, xs)$$

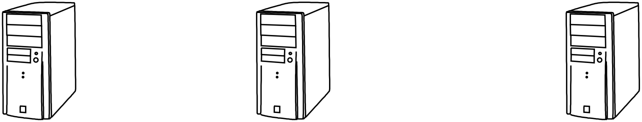


$$\texttt{reducel}(comb, [d_1, d_2, d_3, d_4]) \leadsto r_A$$
$$\texttt{reducel}(comb, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \leadsto r_B$$
$$\texttt{reducel}(comb, [d_{12}, d_{13}, d_{14}]) \leadsto r_C$$

Commutativity in SPARK aggregation

SPARK `reduce`

$$\texttt{reduce}(comb, rdd)$$



d1	d2	d3	d4		d5	d6	d7	d8	d9	d10	d11		d12	d13	d14

$$\texttt{reducel}(comb, [d_1, d_2, d_3, d_4]) \quad \rightsquigarrow r_A$$

$$\texttt{reducel}(comb, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \quad \rightsquigarrow r_B$$

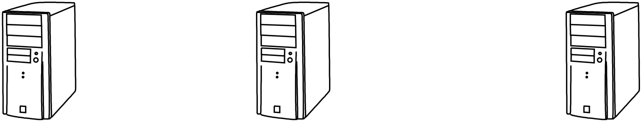
$$\texttt{reducel}(comb, [d_{12}, d_{13}, d_{14}]) \quad \rightsquigarrow r_C$$

- collected results: | rC | rA | rB |

Commutativity in SPARK aggregation

SPARK reduce

reduce(*comb*, *rdd*)



d1	d2	d3	d4

d5	d6	d7	d8	d9	d10	d11

d12	d13	d14

reduceI$(comb, [d_1, d_2, d_3, d_4]) \rightsquigarrow r_A$

reduceI$(comb, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \rightsquigarrow r_B$

reduceI$(comb, [d_{12}, d_{13}, d_{14}]) \rightsquigarrow r_C$

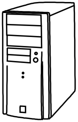
- collected results: | rC | rA | rB |

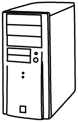
nondeterministic!

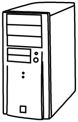
Commutativity in SPARK aggregation

SPARK `reduce`

reduce(*comb*, *rdd*)







d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14

$\texttt{reducel}(comb, [d_1, d_2, d_3, d_4]) \rightsquigarrow r_A$

$\texttt{reducel}(comb, [d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}]) \rightsquigarrow r_B$

$\texttt{reducel}(comb, [d_{12}, d_{13}, d_{14}]) \rightsquigarrow r_C$

- collected results: | rC | rA | rB |

nondeterministic!

$\texttt{reducel}(comb, [r_C, r_A, r_B]) \rightsquigarrow$ *result*

Commutativity in SPARK aggregation

Conditions for commutative `reduce`

$$\texttt{reduce}(\mathit{comb}, \mathit{rdd})$$

Commutativity in SPARK aggregation

Conditions for commutative `reduce`

$$\texttt{reduce}(comb, rdd)$$

- via reduction to **`aggregate`** (using the Maybe monad):

$$\texttt{aggregate}(seq_2, comb_2, \textbf{Nothing}, rdd)$$

$seq_2(x, y) = \textbf{case } x \textbf{ of}$
  $\textbf{Nothing} \rightarrow \textbf{Just } y$
  $\textbf{Just } v \rightarrow \textbf{Just } comb(v, y)$

$comb_2(x, y) = \textbf{case } (x, y) \textbf{ of}$
  $(\textbf{Nothing}, y') \rightarrow y'$
  $(x', \textbf{Nothing}) \rightarrow x'$
  $(\textbf{Just } v_1, \textbf{Just } v_2) \rightarrow \textbf{Just } comb(v_1, v_2)$

Commutativity in SPARK aggregation

Conditions for commutative `reduce`

$$\texttt{reduce}(comb, rdd)$$

- via reduction to **`aggregate`** (using the Maybe monad):

$$\texttt{aggregate}(seq_2, comb_2, \textbf{Nothing}, rdd)$$

$$\begin{aligned} seq_2(x, y) = \textbf{case } x \textbf{ of} \\ \quad \textbf{Nothing} \to \textbf{Just } y \\ \quad \textbf{Just } v \to \textbf{Just } comb(v, y) \end{aligned}$$

$$\begin{aligned} comb_2(x, y) = \textbf{case } (x, y) \textbf{ of} \\ \quad (\textbf{Nothing}, y') \to y' \\ \quad (x', \textbf{Nothing}) \to x' \\ \quad (\textbf{Just } v_1, \textbf{Just } v_2) \to \textbf{Just } comb(v_1, v_2) \end{aligned}$$

Theorem

Consider rdd of elements of type $\mathbb{T}$. A call

$$\texttt{reduce}(comb, rdd)$$

is commutative iff $(\mathbb{T}, comb)$ is a commutative semigroup.

Commutativity in SPARK aggregation

SPARK `treeAggregate` and `treeReduce`

- first stage same as **`aggregate`** and **`reduce`**

Commutativity in SPARK aggregation

SPARK `treeAggregate` and `treeReduce`

- first stage same as **`aggregate`** and **`reduce`**
- second stage is peformed concurrently in a binary tree structure

Commutativity in SPARK aggregation

SPARK `treeAggregate` and `treeReduce`

- first stage same as **`aggregate`** and **`reduce`**
- second stage is peformed concurrently in a binary tree structure

Theorem

- ***`treeAggregate`***$(seq, comb, \perp, rdd)$ *is commutative iff* ***`aggregate`***$(seq, comb, \perp, rdd)$ *is commutative.*
- ***`treeReduce`***$(comb, rdd)$ *is commutative iff* ***`reduce`***$(comb, rdd)$ *is commutative.*

Commutativity in SPARK aggregation

SPARK `aggregateByKey` and `reduceByKey`

- work on PairRDDs: elements are (k, v)

Commutativity in SPARK aggregation

SPARK `aggregateByKey` and `reduceByKey`

- work on PairRDDs: elements are (k, v)
- produce (again) a PairRDD
  - for every k, at most one pair (k, result)
  - result = the output of **`aggregate`** on elements with the key k

Commutativity in SPARK aggregation

SPARK `aggregateByKey` and `reduceByKey`

- work on PairRDDs: elements are (k, v)
- produce (again) a PairRDD
  - for every k, at most one pair $(k, result)$
  - $result =$ the output of **`aggregate`** on elements with the key k

Theorem

- ***`aggregateByKey`***$(seq, comb, \perp, rdd)$ *is commutative iff* ***`aggregate`***$(seq, comb, \perp, rdd)$ *is commutative.*
- ***`reduceByKey`***$(comb, rdd)$ *is commutative iff* ***`reduce`***$(comb, rdd)$ *is commutative.*

Discussion

Conditions for deterministic aggregation:

1. $(\mathit{img}(\mathtt{foldl}(\mathit{seq}, \perp)), \mathit{comb}, \perp)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in \mathit{img}(\mathtt{foldl}(\mathit{seq}, \perp))$, it holds that

$$\mathit{seq}(e, d) = \mathit{comb}(e, \mathit{seq}(z, d)).$$

Discussion

Conditions for deterministic aggregation:

1. $(\mathit{img}(\mathtt{foldl}(\mathit{seq}, \bot)), \mathit{comb}, \bot)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in \mathit{img}(\mathtt{foldl}(\mathit{seq}, \bot))$, it holds that
$$\mathit{seq}(e, d) = \mathit{comb}(e, \mathit{seq}(z, d)).$$

are general:

- apart from scalar data (e.g. integers), they also work for non-scalar (e.g. lists, sets)

Discussion

Conditions for deterministic aggregation:

1. $(img(\texttt{foldl}(seq, \perp)), comb, \perp)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in img(\texttt{foldl}(seq, \perp))$, it holds that
$$seq(e, d) = comb(e, seq(z, d)).$$

are general:

- apart from scalar data (e.g. integers), they also work for non-scalar (e.g. lists, sets)

issues:

- $img(\texttt{foldl}(seq, \perp))$ can be infinite, in general not computable

Discussion

Conditions for deterministic aggregation:

1. $(img(\texttt{foldl}(seq, \bot)), comb, \bot)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in img(\texttt{foldl}(seq, \bot))$, it holds that
$$seq(e, d) = comb(e, seq(z, d)).$$

are general:

- apart from scalar data (e.g. integers), they also work for non-scalar (e.g. lists, sets)

issues:

- $img(\texttt{foldl}(seq, \bot))$ can be infinite, in general not computable
- *seq* and *comb* can be general functions $\rightsquigarrow$ may not terminate

Discussion

Conditions for deterministic aggregation:

1. $(img(\mathtt{foldl}(seq, \perp)), comb, \perp)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in img(\mathtt{foldl}(seq, \perp))$, it holds that
$$seq(e, d) = comb(e, seq(z, d)).$$

are general:

- apart from scalar data (e.g. integers), they also work for non-scalar (e.g. lists, sets)

issues:

- $img(\mathtt{foldl}(seq, \perp))$ can be infinite, in general not computable
- *seq* and *comb* can be general functions $\rightsquigarrow$ may not terminate
- testing the universal equality may be undecidable

Discussion

Conditions for deterministic aggregation:

1. $(\mathit{img}(\texttt{foldl}(\mathit{seq}, \bot)), \mathit{comb}, \bot)$ is a commutative monoid and
2. for all $d \in \mathbb{T}$ and $e \in \mathit{img}(\texttt{foldl}(\mathit{seq}, \bot))$, it holds that
$$\mathit{seq}(e, d) = \mathit{comb}(e, \mathit{seq}(z, d)).$$

are general:

- apart from scalar data (e.g. integers), they also work for non-scalar (e.g. lists, sets)

issues:

- $\mathit{img}(\texttt{foldl}(\mathit{seq}, \bot))$ can be infinite, in general not computable
- *seq* and *comb* can be general functions $\leadsto$ may not terminate
- testing the universal equality may be undecidable
- result for MapReduce [Chen, Hong, Sinha, Wang; TACAS'15]
  - $\mathbb{N}, +, \times$, control(loop-free): undecidable

Commutativity in SPARK aggregation

Case studies:

- manual evaluation of SPARK ML library

Commutativity in SPARK aggregation

Case studies:

- manual evaluation of SPARK ML library
- many functions use **`floats`**

Commutativity in SPARK aggregation

Case studies:

- manual evaluation of SPARK ML library
- many functions use `floats`
- found a redundancy in the SPARK Graph library

Commutativity in SPARK aggregation

Our contribution:

- PURESPARK: a specification of aggregation functions in HASKELL
  - purely functional language
  - executable specification
  - suitable for formal reasoning (e.g. AGDA)

Commutativity in SPARK aggregation

Our contribution:

- PURESPARK: a specification of aggregation functions in HASKELL
  - purely functional language
  - executable specification
  - suitable for formal reasoning (e.g. AGDA)
- correctness requirements on aggregation functions

Commutativity in SPARK aggregation

Our contribution:

- PURESPARK: a specification of aggregation functions in HASKELL
  - purely functional language
  - executable specification
  - suitable for formal reasoning (e.g. AGDA)
- correctness requirements on aggregation functions
- analysis of case studies—finding numeric instability and redundancy in ML library

Commutativity in SPARK aggregation

Our contribution:

- PURESPARK: a specification of aggregation functions in HASKELL
  - purely functional language
  - executable specification
  - suitable for formal reasoning (e.g. AGDA)
- correctness requirements on aggregation functions
- analysis of case studies—finding numeric instability and redundancy in ML library
- also extended to **`aggregate`** over graphs