# Verifying Equivalence of Spark Programs

**Shelly Grossman[1], Sara Cohen[2], Shachar Itzhaky[3], Noam Rinetzky[1], and Mooly Sagiv[1]**

1 **School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
`{shellygr,maon,msagiv}@tau.ac.il`

2 **School of Engineering and Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel**
`sara@cs.huji.ac.il`

3 **Computer Science, Massachusetts Institute of Technology, USA**
`shachari@mit.edu`

## Abstract

In this paper, we present a novel approach for verifying equivalence of Spark programs.

**1998 ACM Subject Classification** D.2.4 Software Engineering Software/Program Verification. D.1.3 Programming Techniques Concurrent Programming. F.3.2 Logics and Meanings of Programs Semantics of Programming Languages (Program analysis). H.2.3 Query languages. H.2.4 Distributed databases.

**Keywords and phrases** Spark, Map reduce, Program equivalence

**Digital Object Identifier** 10.4230/LIPIcs...

## Contents

## 1 Introduction

The rise of Cloud computing and Big Data in the last decade allowed the advant of new programming models, with the intention of simplifying the development process for large-scale needs, letting the programmer focus on business-logic and separating it from the technical details of data management over computer clusters, distribution, communication and parallelization. The first model was MapReduce [6], It allowed programmers to define their logic by composing several iterations of map and reduce operations, where the programmer provided the required map and reduce functions in each step, and the framework was responsible for facilitating the dataflow to the provided functions. MapReduce gave a powerful, yet a clean and abstract programming model. Later on, other frameworks were developed, allowing programmers to write procedural code while still retaining the ability to run it on a large computer cluster, without having the programmer to handle distribution and error recovery. One such framework is Apache Spark [13], in which programmers keep writing code in their programming language of choice, (e.g., Scala [3], Java [1], Python [2], or R [9]) but utilize a special object provided by Spark, called *resilient distributed dataset* (*RDD*), providing access to the distributed data itself and to perform transformations on it, using the cloud resources for actual computing.

The architecture of Spark comprises of a single master node, referred to as the *driver*, and *worker* nodes in a clustered computer environment. All the nodes have access to the program code, but the driver orchestrates its execution using the underlying Spark framework, abstracting away communications, error recovery, distribution, data partitioning, and parallelization. In particular, datasets are distributed, meaning that fragments of it, referred to as *partitions*, reside in the files system of some or all nodes [7]. The access to the data is via the *RDD* API. The *RDD* can be thought of as a simple database table, but which provides support for *User Defined Functions* (*UDF*), greatly increasing its expressive power.

Spark programs handle a family of common tasks involving large datasets, such as log parsing, database queries, training algorithms and different numeric computations in various fields. Due to this, many Spark programs share several properties: they are mostly short and relatively simple to read and understand. We believe that thanks to this nature of Spark programs, the problem of verifying a program's properties, or even program equivalence as we focus on in this paper, may become feasible, even decidable in a usable class of Spark programs.

**Main Results.** In this paper we define a simple programming language called $SPARK$, in which operations on a single RDD are abstracted as composite simply typed $\lambda$-calculus expressions. The operations correspond to operations in relational algebra, with additional aggregate operations and unique Spark operations such as *mapPartitions*. We describe the problem of *program equivalence modulo a transform* (*PEMT*), and provide a classification of $SPARK$ programs according to the decidability of the *PEMT* problem for programs in the same class. When *PEMT* is decidable, we show an algorithm for solving it. An immediate corollary is a classification of $SPARK$ programs for the decidability of the *program equialence* (*PE*) problem.

**Overview.** Section 2 provides necessary preliminaries for the rest of the paper. In section 3 we give a complete formalization (syntax and semantics) of $SPARK$. In section 4 we describe the *PEMT* and *PE* problems. In section 5 we discuss the most basic class of $SPARK$ programs, which are programs without aggregate expressions. We continue in Section 6, where we discuss different classes of $SPARK$ contianing an aggregated expression. Section 7 presents our abstraction for Spark's partitioning, which plays a vital role in program optimization and thus presents real-life applications of the solution to $PE$.

## 2 Notations

This section provides necessary notations used throughout this paper.

**Basic notations** We denote the set of natural numbers (including zero) by $\mathbb{N}$, and by $\mathbb{N}^+$ the positive natural numbers. We also denote $\mathbb{Z}$ for integers (positive and non-positive), and $\mathbb{B}$ for booleans: $tt, ff$. The *undefined* value is denoted by $\bot$. We denote the *size* (number of elements) of a set $X$ by $|X|$. We denote a general if-then-else operator (*ite*) as $ite(cond, then\_expr, else\_expr) = \begin{cases} then\_expr & cond \\ else\_expr & otherwise \end{cases}$.

**Tuples** Let $X = X_1 \times \cdots \times X_n$ be a tuple domain of arity $n$. $p_i(X) = X_i$ is defined for $i \in \{1, \ldots, n\}$. Respectively, for an element $x = (x_1, \ldots, x_n) \in X$, $p_i(x) = x_i$ for $i \in \{1, \ldots, n\}$. The common arithmetic operations $(+, -, *)$ over integers are lifted to (vectorial) operations tuples in a point-wise manner.

**Binary Relations** Let $R \subseteq X \times Y$ be a binary relation. We use the following notations: **Projection** $(i \in \{1, 2\})$**:** $p_i(R) = \{p_i(t) \mid t \in R\}$, **Relation restriction:** $\sigma_\varphi(R) = \{t \in R \mid t \in \varphi\}$, **Relation restriction by first element:** $\sigma_Z(R) = \sigma_{\{t | p_1(t) \in Z\}}(R)$, $\sigma_x(R) = \sigma_{\{t | p_1(t) \in \{x\}\}}(R)$, **Restriction image:** $R(Z) = p_2(\sigma_Z(R))$, $R(x) = R(\{x\})$.

**Functions** A function $f$ from $X$ to $Y$ is a relation between $X$ and $Y$ which is single valued $(\forall x \in X. |\sigma_x(f)| \leq 1)$. We write $y = f(x)$ to dente that $(x, y) \in f$. We use $\cdot \hookrightarrow \cdot$ and $\cdot \to \cdot$ to denote *partial* and *total* functions, respectively. We denote the *support* and the *image* of $f$ by $\sup(f) = p_1(f)$ and $\mathrm{img}(f) = p_2(f)$, respectively. We say that a function $f$ is *undefined* at $x$ if $x \notin \sup(f)$, and denote it by $f(x) = \bot$. If the support of $f$ is empty then we write $f = \bot$.

**Bags (multisets)** A *bag* (*multiset*) $m$ over a domain $X$ is a partial function from $X$ to $\mathbb{N}^+$. For $x \in \mathrm{dom}(m)$, we say that $x \in X$ is *in* $m$ $(x \in m)$ if $m(x) \neq \bot$, and refer to $m(x)$ as its *multiplicity* in $m$. Conversely, $x \in X$ is *not in* $m$ $(x \notin m)$ if $m(x) = \bot$. We write $m = \{\!\{Z\}\!\}$, where $Z \subseteq X$, to denote that $Z$ is the support of the bag $m$. To define a bag $m$ we write, similarly to set-comprehension notation, $\{\!\{x\,;\, m(x) \mid x \in \phi\}\!\}$. Elements $x$ for which $x \notin \phi$, do not belong in the bag $(m(x) = \bot)$. Similarly to sets, the *size* of a bag $m$ is denoted $|m|$, and is equal to $\Sigma_{x \in m} m(x)$.

**Bag operators** Figure 1 defined **five operations on bags**:

1. *union* $(\cdot \cup \cdot)$ - accepts two bags as arguments, and the multiplicity of an element is equal to the sum of its multiplicities in all participating bags. We lift the union operator to a sequence $m_1, \cdots, m_n$ of bags, returning the bag whose support is equal to the union of the support sets of all participating bags,
2. *(strong) subtraction* $(\cdot \setminus \cdot)$ accepts two bags as arguments. The resulting bag's support is the equal to the subtraction of support sets, and multiplicities of the surviving element does not change. For example, $\{\!\{(1;7)\}\!\} \setminus \{\!\{(1;3)\}\!\} = \{\!\{\}\!\}$ (and not $\{\!\{(1;4)\}\!\}$).
3. *cartesian products* $(\cdot \times \cdot)$ - accepts two bags as arguments, and returns a bag whose domain is the cartesian product of argument bags' domains, taking all possible pairings from both bags. For $x \in m_1, y \in m_2$, we get that the multiplicity of $(x, y)$ in $(m_1 \times m_2)((x, y)) = m_1(x) \cdot m_2(y)$. By abuse of notation we may sometimes write $(m_1(x) \cdot m_2(y))(x, y)$.

$$
\begin{array}{lcl}
m_1 \cup m_2 & = & \{\!\{x\,;\, ite(m_1(x) \neq \bot, m_1(x), 0) + ite(m_2(x) \neq \bot, m_2(x), 0) \mid x \in m_1 \cup m_2\}\!\} \\
m_1 \setminus m_2 & = & \{\!\{x\,;\, m_1(x) \mid x \in X_1 \wedge x \notin m_2\}\!\} \\
m_1 \times m_2 & = & \{\!\{(x_1, x_2)\,;\, m_1(x) \cdot m_2(x) \mid x_1 \in m_1 \wedge x_2 \in m_2\}\!\} \\
m\restriction_S & = & \sigma_S(m) \\
\pi_i(m) & = & \{\!\{p_i(x)\,;\, \sum_{y \in m \wedge p_i(y) = p_i(x)} m(y) \mid x \in m\}\!\}
\end{array}
$$

**Figure 1** Definition of *bag* operations

$$
\begin{array}{lcl}
P_{m_1 \cup m_2} & = & (p_1(P_{m_1}), \ldots, p_{p_1}(P_{m_1}), p_1(P_{m_2}), \ldots, p_{p_2}(P_{m_2})) \\
P_{m_1 \setminus m_2} & = & (p_1(P_{m_1}) \setminus m_2, \ldots, p_{p_1}(P_{m_1}) \setminus m_2) \\
P_{m_1 \times m_2} & = & (p_1(P_{m_1}) \times p_1(P_{m_2}), \ldots, p_1(P_{m_1}) \times p_{p_2}(P_{m_2}), \\
 & & p_2(P_{m_1}) \times p_1(P_{m_2}), \ldots, p_2(P_{m_1}) \times p_{p_2}(P_{m_2}), \\
 & & \ldots, \\
 & & p_{p_1}(P_{m_1}) \times p_1(P_{m_2}), \ldots, p_{p_1}(P_{m_1}) \times p_{p_2}(P_{m_2})) \\
P_{m\restriction_S} & = & (p_1(P_m)\restriction_S, \ldots, p_p(P_m)\restriction_S) \\
P_{\pi_i(m)} & = & (\pi_i(p_1(P_m)), \ldots, \pi_i(p_p(P_m)))
\end{array}
$$

**Figure 2** Definition of effect of bag operations on partitioning

4. *restriction* $(\cdot\restriction.)$ - accepts a bag and a set $S \subseteq \mathrm{dom}(m)$. It returns a bag where the support is restricted to the set $S$, so for every $x \in m, x \notin S$, $m \restriction_S (x) = \bot$.
5. *projection* $(\pi.(\cdot))$ - accepts a bag as an argument, with a domain $X = X_1 \times \cdots X_n$. It returns a bag whose domain is $p_i(X) = X_i$, handling summation of multiplicities from different elements mapping to the same $i$-th element.

**Partitioned bag and their operators** A *partitioned bag* (*partitioned multiset*) is an extension of the *bag* definition. A *partitioned bag* is a *bag* with some *p-partitioning*. A *p-partitioning* of a bag $m$ is a sequence of $p$ bags: $P_m = (m_1, \ldots, m_p)$ such that $m = \bigcup_{i=1,\ldots,p} m_i$. To select the $i$-th partition of a bag $m$ with partitioning $P_m$ we write: $p_i(P_m) = m_i$. The *dimension* of a partitioning $P_m$ is the number of partitions in it, denoted $\dim(P_m) = p$. The different operations modify the partitioning of a bag, as specified in Figure 2.

SG: Noam: Add discussion on multiplicities and on our model vs. Spark model

# 3 The $SPARK$ language

In this section, we define the syntax of $SPARK$, a simple imperative programming language which allows to use Spark's *resilient distributed datasets* (*RDDs*) [13].

## 3.1 Data Model

**Basic types.** $SPARK$ supports two primitive types: integers (`Int`) and booleans (`Boolean`). On top of this, the user can define types which are cartesian products of primitive types In the following we use $\boldsymbol{c}$ to range over integer numerals (constants), $\boldsymbol{b} \in \{\mathtt{true}, \mathtt{false}\}$ to range over boolean constants, and $\tau$ to range over basic types and record types.

**RDDs.** In addition, $SPARK$ allows the user to define $RDD$s. RDDs are bags of *records*, all of the same type. Hence, $RDD_{\boldsymbol{\tau}}$ denotes bags containing records of type $\boldsymbol{\tau}$.

**Semantic Domains.** We interpret the integers and boolean primitive types as *integers* ($\mathbb{Z}$) and *booleans* ($\mathbb{B}$), respectively. The interpretation of both primitive types is denoted $\mathrm{T} = \mathbb{Z} \cup \mathbb{B}$. The interpretation of all possible types (including mixed cartesian products of the primitive types) is denoted by $\mathcal{T} = \bigcup_n \mathrm{T}^n$.

The $RDD$ type is interpreted as a *bag*. Therefore, $\boldsymbol{r}$ ranging over $RDD_{\boldsymbol{\tau}}$ is interpreted as a bag of type $[\![\boldsymbol{\tau}]\!]$, $[\![r]\!] = r \in ([\![\boldsymbol{\tau}]\!] \hookrightarrow \mathbb{N})$. We let $RDD = \bigcup_{\tau \in \mathcal{T}} RDD_{\tau}$, the semantic domain of RDDs over all possible record types $\tau \in \mathcal{T}$.

**Interpretation operator** We use $[\![\cdot]\!]$ (semantic brackets) to denote the mathematical interpretation of an expression. We shall see in the next subsections that it may be a function of an *environment* when the expressions contain variables. The exact meaning of environments and semantic interpretation of syntactic strings under environments is fully explained in Section 3.4.

## 3.2 Functional Model

**Operations** $RDD$s are analogous to database tables and as such the methods to query the $RDD$s are inspired by both *Relational Algebra* ($RA$) [] and Spark. $RA$ has 5 basic operators, which are *Select*, *Project*, *Cartesian*, *Union* and *Subtract*. The *Select* operator is analogous to *filter* in $SPARK$, and *Project* is analogous to *map*. The expressive power of $SPARK$'s *map* and *filter* is greater than their analogous $RA$ operations thanks to the UDFs (see next), which allow *extended projection* as well as greater flexibility in executing complex operations on elements of different types.

**UDFs.** A special property of Spark is in allowing some of its standard operations to be *higher order functions* - to receive a function and to apply it on an RDD in a method defined by the operation. For example, we can *fold* an RDD containing integer numbers by providing a sum function of two integers to the *fold* transform. Such a function is called a *"User-Defined Function"*, or simply *UDF*, for short. The *signature* of an UDF contains information on the return type and the arguments types. The syntax of the *body* of an UDF is the same as that of first-order simply typed lambda expressions. For example: $sumMod10\colon \mathtt{Int} \times \mathtt{Int} \to \mathtt{Int}$ is the signature of the following function: $sumMod10 = \lambda x, y.x\%10 + y\%10$. The types are omitted from the body of the function for brevity, but when the types are not clear from the context, we may also write it as: $sumMod10 = \lambda x\colon \mathtt{Int}, y\colon \mathtt{Int}.x\%10 + y\%10\colon \mathtt{Int}$. Note

$$\text{Let:} \quad \begin{aligned} isOdd &= \lambda x{:}\,\texttt{Int}.\neg x\%2 = 0 \\ doubleAndAdd &= \lambda c{:}\,\texttt{Int}.\lambda x{:}\,\texttt{Int}.2 * x + c \\ sumFlatPair &= \lambda A{:}\,\texttt{Int}, (x,y){:}\,\texttt{Int} \times \texttt{Int}.A + x + y \end{aligned}$$

```
P1(R_0: RDD_Int, R_1: RDD_Int):
A = filter(isOdd)(R_0)
B = map(doubleAndAdd(1))(A)
C = cartesian(B, R_1)
v = fold(0, sumFlatPair)(C)
return v
```

**Figure 3** Example *SPARK* program

that $sumMod10$ has two arguments, but we could also write it as a function with single argument having the following signature: $sumPairMod10{:}\,(\texttt{Int} \times \texttt{Int}) \to \texttt{Int}$, with body: $sumPairMod10 = \lambda(x,y).x\%10 + y\%10$.

We allow the definition of these functions to be parametric, meaning that there are free variables in the lambda expressions, which we wrap with an additional $\lambda$. For example, we could define a function that adds 1 to an integer: $\boldsymbol{f} = \lambda x{:}\,\texttt{Int}.x + 1$, but we could also make it more generic and flexible by writing $\boldsymbol{g} = \lambda a{:}\,\texttt{Int}.\lambda x{:}\,\texttt{Int}.x + a$. This is an example of a *parametric function*. Parametric functions can be transformed to regular functions by beta-reducing the first lambda abstraction: $\boldsymbol{g}(1)$ which is identical to $\boldsymbol{f}$.

## 3.3 Syntax

The syntax of *SPARK* language is defined in Figure 4.

**Syntactic Categories** We assume variables to be an infinite syntactic category, ranged over by $\boldsymbol{v}, \boldsymbol{b}, \boldsymbol{r} \in \texttt{Vars}$. Expressions range over $\boldsymbol{e}$. An integer constant is denoted $c$. Operations are divided to 4 categories: Relational, Mapping, Grouping, and Aggregating. Some of the operations require arguments, which may be either a primitive expression, an *RDD*, or a function. Functions range over $\boldsymbol{f}, \boldsymbol{F} \in \texttt{LambdaExpressions}$. Parametric functions are denoted by capital meta-variables ($\boldsymbol{F}$ as opposed to $\boldsymbol{f}$ for regular functions) and must always be given the list of parameters when passed to an operation.

**Program structure** The header of a program contains function definitions. Loops are not allowed in the body of a program. Variable declarations are in *SSA* (*Static Single Assignment*) form [5]. Variables are immutable by this construction. Programs have no side effects, do not change the inputs, and always return a value. The *program signature* will consist of its name, its input types and return type: $\boldsymbol{P}(\overline{\mathcal{T}_i}, \overline{RDD_i}){:}\,\mathcal{T}_o$

**Example program** Consider the example *SPARK* program in Figure 3.

From the example program we can see the general structure of *SPARK* programs: First, the functions that are used as UDFs in the program are declared and defined: $isOdd, sumFlatPair$ defined as *Fdef*, and *doubleAndAdd* defined as a *PFdef*. Then, the name of the program ($\boldsymbol{P} = P1$) is announced with a list of input RDDs $(R_0, R_1)$ (*Prog* rule). Instead of writing *Let* $l_1$ *in Let* $l_2$ *in* ..., we use syntactic sugar, where each line of code contains a single '*Let*' definition, and the target expression (which may be a '*Let*' expression itself) follows. Then, 3 variables of RDD type are defined $(A, B, C)$ and one integer variable

| | | | |
|---|---|---|---|
| **Basic Types** | $\tau$ | ::= | $\texttt{int} \mid \texttt{bool} \mid \tau \times \ldots \times \tau$ |
| **RDDs** | $RDD$ | ::= | $\texttt{RDD}_\tau$ |
| **Variables** | $x$ | ::= | $\boldsymbol{v} \mid \boldsymbol{r}$ |
| **Arithmetic Exp.** | $ae$ | ::= | $c \mid ae + ae \mid -ae \mid c * ae \mid ae \,/\, c \mid ae \,\%\, c$ |
| **Boolean Exp.** | $be$ | ::= | $\texttt{true} \mid \texttt{false} \mid e = e \mid ae < ae \mid \neg be \mid be \wedge be \mid be \vee be$ |
| **General Basic Exp.** | $e$ | ::= | $ae \mid be \mid \boldsymbol{v} \mid (e\,, e) \mid \boldsymbol{p}_i(e) \mid \texttt{if } (b) \texttt{ then } e \texttt{ else } e$ |
| **Functions** | $Fdef$ | ::= | $\texttt{def } \boldsymbol{f} = \lambda\overline{\boldsymbol{y}:\tau}\;\; e:\tau$ |
| **Parametric Functions** | $PFdef$ | ::= | $\texttt{def } \boldsymbol{F} = \lambda\overline{\boldsymbol{x}:\tau}.\lambda\overline{\boldsymbol{y}:\tau}\;\; e:\tau$ |
| **RDD Exp.** | $re$ | ::= | $\texttt{union}(\boldsymbol{r},\boldsymbol{r}) \mid \texttt{cartesian}(\boldsymbol{r},\boldsymbol{r}) \mid \texttt{subtract}(\boldsymbol{r},\boldsymbol{r})$ |
| | | | $\texttt{map}(\boldsymbol{f})(\boldsymbol{r},\boldsymbol{r}) \mid \texttt{filter}(\boldsymbol{f})(\boldsymbol{r},\boldsymbol{r})$ |
| | | | $\texttt{foldByKey}(\boldsymbol{e},\boldsymbol{f})(\boldsymbol{r})$ |
| **RDD Aggregation Exp.** | $ge$ | ::= | $\texttt{fold}(\boldsymbol{e},\boldsymbol{f})(\boldsymbol{r})$ |
| **General Exp.** | $\eta$ | ::= | $e \mid re \mid ge$ |
| **Program Body** | $E$ | ::= | $Let\ \boldsymbol{x} = \eta\ in\ E \mid \eta$ |
| **Program** | $Prog$ | ::= | $\boldsymbol{P}(\overline{\boldsymbol{r}:RDD_\tau}, \overline{\boldsymbol{v}:\tau}) = \overline{Fdef}\;\; \overline{PFdef}\;\; E$ |

**Figure 4** Syntax for *SPARK*

$(v)$. We can see in the definition of $A$ an application of the *filter* operation, accepting the RDD $R_0$ and the function *isOdd*, For $B$'s definition we apply the *map* operation with a parametric function *doubleAndAdd* with the parameter 1, or simply the function $\lambda x.2 * x + 1$. $C$ is the cartesian product of $B$ and input RDD $R_1$. We apply an aggregation using *fold* on the RDD $C$, with an initial value 0 and the function *sumFlatPair*, which 'flattens' elements of tuples in $C$ by taking their sum, and summing all those vectorial sums to a single value stored in the variable $v$. As we omit '*Let*' expressions from the example, we use a the `return` keyword to return a value. The returned value is the integer variable $v$. The program's signature is $P1(RDD_{\texttt{Int}}, RDD_{\texttt{Int}}):\texttt{Int}$.

## 3.4 Semantics

**Program Environment** We define a unified semantic domain $\mathcal{D} = \mathcal{T} \cup RDD$ for all types in *SPARK*. The *program environment* type:

$$\mathcal{E} = \texttt{Vars} \rightarrow \mathcal{D}$$

is a mapping from each variable in `Vars` to its value, according to type. A variable's type does not change during the program's run, nor does its value.

**Data flow** The *environment function* $\rho \in \mathcal{E}$ denotes the environment of the program. The environment function is initialized as $\rho = \perp$, and filled for input variables according to the inputs: $\boldsymbol{v}_1^{in}, \ldots, \boldsymbol{v}_k^{in}$ at initial environment: $\rho(\boldsymbol{v}_j^{in}) = [\![\boldsymbol{v}_j^{in}]\!]$, where $[\![\cdot]\!]$ is used to express the interpretation of the input in the semantic domain. We denote $[\![\boldsymbol{v}]\!](\rho) = \rho(\boldsymbol{v})$ as the *interpreted value* of the variable $\boldsymbol{v}$ of type $\mathcal{D}$. The semantics of expressions are straightforward and we provide the semantics with the current environment $\rho$ (using $[\![\cdot]\!](\cdot)$), see Figure 5 for details. In Figure 6 we specify the behavior of $[\![\cdot]\!](\cdot)$ on the body of the program.

| | | | |
|---|---|---|---|
| **Constants** | $[\![c]\!](\rho)$ | $=$ | $c$ |
| **Variables** | $[\![\boldsymbol{x}]\!](\rho)$ | $=$ | $\rho(v)$ |
| **Unary operations** | $[\![\texttt{uOp}\, e]\!](\rho)$ | $=$ | $uOp\, [\![e]\!](\rho)$ |
| **Binary operations** | $[\![e\, \texttt{binOp}\, e]\!](\rho)$ | $=$ | $[\![e]\!](\rho)\, binOp\, [\![e]\!](\rho)$ |
| **Ternary operations** | $[\![\texttt{if } e \texttt{ then } e \texttt{ else } e]\!](\rho)$ | $=$ | $ite\,([\![e]\!](\rho), [\![e]\!](\rho), [\![e]\!](\rho))$ |

**Figure 5** Basic Expression semantics (for $e$) - `uOp`, `binOp`, and `terOp` are taken from Figure 4 : `uOp` $\in \{-, \neg, \boldsymbol{\pi}_i\}$, `binOp` $\in \{+, *, /, \%, =, <, \wedge, \vee, (,)\}$

| | | | |
|---|---|---|---|
| **Program** | $[\![Prog]\!]$ | $=$ | $[\![E]\!](\rho)$ |
| **Assignment** | $[\![\boldsymbol{x} = \eta]\!](\rho)$ | $=$ | $\rho[\boldsymbol{x} \mapsto [\![\eta]\!](\rho)]$ |
| **Sequence** | $[\![Let\ \ \boldsymbol{x} = \eta\ in\ E]\!](\rho)$ | $=$ | $[\![E]\!]([\![\boldsymbol{x} = \eta]\!](\rho))$ |

**Figure 6** Semantics for $SPARK$ programs in structural induction. Semantics of $\eta$ are described in Figures 5 and 8

**Function and UDF semantics** For UDFs, which are based on a restricted fragment of the *simply typed lambda calculus* [], we assume the syntax and semantics are the same as in the $\lambda$-calculus. Note however that the syntax does not allow passing higher order functions as UDFs, and forces any higher order function to be reduced to a first-order function beforehand. In addition, all parameters passed to UDF which are based on higher-order functions are read-only. The semantics of functions are defined in Figure 7.

**Semantics of operations** In Figure 8 the semantics of all RDD expressions, including aggregation, are explicitly stated.

**Notes** In the *bag semantics*:

- In *map*, if several elements $y$ map to $x$ by $f$, then the multiplicity of $x$ is the sum of multiplicities of all $y$ elements. In other words, if an element $x$ appears $n$ times, we apply the map UDF on it $n$ times.
- fold *is well defined:* It should be noted that the Spark specification requires UDFs passed to aggregate operations to be **commutative** and **associative** for the value to be uniquely defined. By the assumed commutativity of $\boldsymbol{f}$, the order in which elements from the bag are chosen in the *fold* operation does not affect the result.
- In *foldByKey*, each key $k$ is guaranteed to appear exactly once, with the folded value of all the values $v$ such that $(k, v) \in r$. Therefore the result of *foldByKey* can be seen as a set, too.
- The semantics of *subtract* determine that if an element $x$ appears in both $r_1$ and $r_2$ with positive multiplicity, then it is not be included in the result, even if $[\![r_1]\!](\rho)(x) > [\![r_2]\!](\rho)(x)$.

| | | | |
|---|---|---|---|
| **Simple functions** | $[\![\boldsymbol{f}]\!]$ | $=$ | $\lambda\overline{y\!:\!\tau}.\ [\![e]\!]$ |
| **Parametric functions** | $[\![\boldsymbol{F}]\!]$ | $=$ | $\lambda\overline{x\!:\!\tau}, \overline{y\!:\!\tau}.\ [\![e]\!]$ |
| **Parametric functions with parameters** | $[\![\boldsymbol{F}(\overline{e_p})]\!]$ | $=$ | $\lambda\overline{y\!:\!\tau}.\ [\![e[\overline{\boldsymbol{x}} \mapsto \overline{e_p}]]\!]$ |

**Figure 7** Semantics for functions

$$
\begin{array}{llcl}
\textbf{Application:} & [\![\mathtt{op(args)(rdds)}]\!] & = & [\![\mathtt{op}]\!]([\![\mathtt{args}]\!])([\![\mathtt{rdds}]\!]) \\
\\
\textbf{Map:} & [\![\mathtt{map}]\!](f)(r) & = & \pi_2(\{\!\{(x, f(x)); r(x) \mid x \in r\}\!\}) \\
\textbf{Filter:} & [\![\mathtt{filter}]\!](b)(r) & = & r\restriction_{\{x \mid b(x)\}} \\
\textbf{Cartesian:} & [\![\mathtt{cartesian}]\!](r_1, r_2) & = & r_1 \times r_2 \\
\textbf{Union:} & [\![\mathtt{union}]\!](r_1, r_2) & = & r_1 \cup r_2 \\
\textbf{Subtract:} & [\![\mathtt{subtract}]\!](r_1, r_2) & = & r_1 \setminus r_2 \\
\textbf{Fold:} & [\![\mathtt{fold}]\!](v, f)(r) & = & \begin{cases} f([\![\mathtt{fold}]\!](v, f)(r'), a) & r = r' \cup \{\!\{a; 1\}\!\} \\ v & r = \bot \end{cases} \\
\textbf{FoldByKey:} & [\![\mathtt{foldByKey}]\!](a_0, f)(r) & = & \{\!\{(k, v); 1 \mid (k, \_) \in r \land v = [\![\mathtt{fold}]\!](a_0, f)(\pi_2(r\restriction_{(k,\_)}))\}\!\}
\end{array}
$$

**Figure 8** Semantics for RDD expressions $(re, ge)$

**Example** For the example program Figure 3, suppose we were given the following input: $R_0 = \{\!\{(1;7),(2;1)\}\!\}, R_1 = \{\!\{(3;4),(5;2)\}\!\}$. Then: $\rho(A) = \{\!\{(1;7)\}\!\}$, $\rho(B) = \{\!\{(3;7)\}\!\}$, $\rho(C) = \{\!\{((3,3);28),((3,5);14)\}\!\}$, $\rho(v) = 28*(3+3)+14*(3+5)$ and the program returns $\rho(v) = 280$.

## 4 The *PEMT* and *PE* problem definition

Next we describe the main decision problems which we try to solve for different classes of $SPARK$ programs.

**The Program Equivalence (*PE*) problem** Let $P_1$ and $P_2$ be $SPARK$ programs, with signature $P_i(\overline{T}, \overline{RDD_T})\colon\tau$ for $i \in \{1,2\}$. We use $[\![P_i]\!]([\![\overline{v}]\!], [\![\overline{r}]\!])$ to denote the result of $P_i$. We say that $P_1$ and $P_2$ are *equivalent*, if for all input values $\overline{v}$ and RDDs $\overline{r}$, it holds that $[\![P_1]\!]([\![\overline{v}]\!], [\![\overline{r}]\!]) = [\![P_2]\!]([\![\overline{v}]\!], [\![\overline{r}]\!])$.

**The Program Eqvuivalence Modulo Transforms (*PEMT*) problem** Let $P_1$ and $P_2$ be $SPARK$ programs as before. Let $T_1$ and $T_2$ be $SPARK$ programs, with an input parameter of type $\sigma_1$ and $\sigma_2$ respectively, as well as $k'$ RDDs and $m'$, returning a value of type $\sigma$.

We use $[\![T_i]\!]([\![P_i]\!]([\![r_1]\!], \ldots, [\![r_k]\!], [\![v_1]\!], \ldots, [\![v_m]\!]), [\![r'_1]\!], \ldots, [\![r'_{k'}]\!], [\![v'_1]\!], \ldots, [\![v'_{m'}]\!])$ to denote the result of applying $T_i$ to the result of $P_i$, given input RDDs $r_1, \ldots, r_k$. We say that the programs $P_1$ and $P_2$ are *equivalent modulo $T_1$ and $T_2$*, if for any $k+k'$ RDDs $r_1, \ldots, r_k, r'_1, \ldots, r'_{k'}$, and $m+m'$ values $m_1, \ldots, m_k, m'_1, \ldots, m'_{k'}$ it holds that

$$[\![T_1]\!]([\![P_1]\!]([\![r_1]\!], ..., [\![r_k]\!], [\![v_1]\!], \ldots, [\![v_m]\!]), [\![r'_1]\!], \ldots, [\![r'_{k'}]\!], [\![v'_1]\!], \ldots, [\![v'_{m'}]\!]) =$$
$$[\![T_2]\!]([\![P_2]\!]([\![r_1]\!], ..., [\![r_k]\!], [\![v_1]\!], \ldots, [\![v_m]\!]), [\![r'_1]\!], \ldots, [\![r'_{k'}]\!], [\![v'_1]\!], \ldots, [\![v'_{m'}]\!])$$

The *PEMT problem* is, given $P_1, P_2$ and $T_1, T_2$, determine whether the programs $P_1$ and $P_2$ are equivalent modulo $T_1$ and $T_2$.

# 5 Basic decidability results for $SPARK$

Throughout the rest of this paper, we shall ignore the *subtract* operation entirely, as it is known to be a source of undecidability of RA []. move up

## 5.1 An equivalency checking framework

**Introduction and intuition** We will describe an alternative, equivalent semantics for $SPARK$ where the program is interpreted as a term in first order logic over the theory of integer arithmetic. This term is called the *program term* and denoted $\Phi(P)$ for program $P$. The variables of the term are taken from the input RDDs.
For example:

$$\text{Let:} \quad \begin{array}{l} f = \lambda(x,y).x + 2 * y + 1 \\ g = \lambda(x,y).x + y + 1 \end{array}$$

| | P1($R_0$: $RDD_{\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}}$): | P2($R_0$: $RDD_{\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | `return map`$(f)$(`cartesian`$(R_0, R_1)$) | `return map`$(g)$(`cartesian`$(R_0,$ `map`$(\lambda x.2 * x)(R_1)$)) |

In $P1$, the term for `cartesian`$(R_0, R_1)$ will be $(\mathbf{x}_{R_0}, \mathbf{x}_{R_1})$. When we apply the map using the UDF $f = \lambda(x,y).x + 2 * y + 1$ we get $f(\mathbf{x}_{R_0}, \mathbf{x}_{R_1}) = \mathbf{x}_{R_0} + 2 * \mathbf{x}_{R_1} + 1$.
In $P2$, we apply the cartesian product on $R_0$ and a mapping of $R_1$ and get $(\mathbf{x}_{R_0}, 2{*}\mathbf{x}_{R_1})$. When we apply the map using the UDF $g = \lambda(x,y).x{+}y{+}1$ we get again $g(\mathbf{x}_{R_0}, \mathbf{x}_{R_1}) = \mathbf{x}_{R_0}{+}2{*}\mathbf{x}_{R_1}{+}1$. We see that the program term of both $P1$ and $P2$ is $\mathbf{x}_{R_0} + 2 * \mathbf{x}_{R_1} + 1$, and the programs are trivially equivalent.

In non-trivial cases, we will use a solver to verify the validity of the equivalence of two program terms.

**Representative elements of RDDs** To build the program terms as done above, we use variables that are based on the input RDDs. Such variables are called *representative elements*. Suppose there is a program that receives as an input an RDD $r^i$. We denote the representative element of $r^i$ as: $\mathbf{x}_{r^i}$. The set of possible valuations of that variable is equal to the bag defined by $r^i$, and an additional 'undefined' value ($\perp$), for the empty RDD. Therefore $\mathbf{x}_{r^i}$ ranges over $\sup(r^i) \cup \{\perp\}$.
By abuse of notation, the term for an RDD computed in a $SPARK$ program is also called a representative element.

**Multiplicity terms of $SPARK$ programs** Program terms are built upon bags, and thus are affected by the multiplicity of the elements in the bags. For example:

| | P1($R_0$: $RDD_{\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}}$): | P2($R_0$: $RDD_{\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | `return map`$(\lambda x.1)(R_0)$ | `return map`$(\lambda x.1)(R_1)$ |

We see that $P1$ and $P2$ have the same program term (1), but the multiplicity of that element in the output bag is different. In $P1$, its multiplicity is the same as the size of $R_0$, and in $P2$ it's the same as the size of $R_1$. $P1$ and $P2$ are therefore not equivalent, because we can provide inputs $R_0, R_1$ of different sizes.
For each program we will define a *multiplicity term*, denoted $\mathcal{M}(P)$ for program $P$. The multiplicity term is describing the multiplicity of an arbitrary element, i.e. how many times a certain valuation should be applied to receive that element. Multiplicity terms will have variables based on the input RDDs, denoted with $\mu_R$. In the example, the multiplicity term of $P1$ is $\mathcal{M}(P1) = \mu_{R_0}$, and of $P2$ it is $\mathcal{M}(P2) = \mu_{R_1}$.

**Compiling** $SPARK$ **to logical terms using representative elements** A program $P$ is compiled to a logical term using representative element variables and functions based on UDFs and the $SPARK$ operations. We use $\vec{r^i}$ as standard notation for the inputs of some $SPARK$ program, and $r^o$ as standard notation for its output.
The term $\phi_P$ is called the *program term of* $P$ and it is defined by structural induction according to the notations in Figure 9.
The term $\mathcal{M}(P)$ is called the *multiplicity term of* $P$, also defined by structural induction, according to figure 10.
The SR semantics of a program that returns an RDD-type output is a pair consisting of a program term and a multiplicity term:

$$SR(P)(\vec{r^i}) = (\{\!\{\Phi(P)[\vec{\mathbf{x}_{r^i}} \mapsto \vec{x}]; \mathcal{M}(P)[\vec{\mu_{r^i}} \mapsto r^i(\vec{x})] \mid \Phi(P)(\vec{x}) \neq \bot \wedge \vec{x} \in \vec{r^i}\}\!\}, \mathcal{M}(P))$$

Assigning a concrete valuation to the variables of $\phi$ returns an element in the output RDD. By taking all possible valuations to the term by elements from $\vec{r^i}$, we get the bag equal to the output RDD.

We elaborate on programs that return an aggregated expression (*fold* ,*foldByKey* ) in section 6, where the definition of SR() semantics differ.

▸ Proposition 5.1. The semantics defined in section 3.4 and the SR(P) semantics are equivalent.

**Proof.** Let $P : \boldsymbol{P}(\overline{\boldsymbol{r}}, \overline{\boldsymbol{v}}) = \overline{\boldsymbol{F}}\, \overline{\boldsymbol{f}}\, E$ be a program with input $\vec{r^i}$, $[\![P]\!]$ be the interpretation of its output according to the defined semantics, and $SR(P)$ by the symbolic representation semantics of $P$ as described earlier. Then:

$$p_1(SR(P)(\vec{r^i})) = [\![P]\!]([\![\vec{r^i}]\!])$$

The proof follows by structural induction on the available operations in the $SPARK$ program $P$ (The syntactic term $E$). We also assume the RDD given as arguments to the operations are input RDDs. In addition, multiplicity is expressed using the choice of a representative element: $\mathcal{M}(r) = ite(\mathbf{x}_r \in r, r(\mathbf{x}_r), 0)$ and $\mathcal{M}(P)$ is calculated accordingly (see Figure 10).

- *map* : Suppose $E = \mathtt{map}(f)(r)$. We have

  $$p_1(SR(P)(r)) = \{\!\{\Phi(P)(\mathbf{x}_r); r(\mathbf{x}_r) \mid \Phi(P)(\mathbf{x}_r) \neq \bot \wedge \mathbf{x}_r \in r\}\!\} = \{\!\{f(\mathbf{x}_r)); r(\mathbf{x}_r) \mid f(\mathbf{x}_r) \neq \bot \wedge \mathbf{x}_r \in r\}\!\}$$

  While in the original semantics:

  $$[\![P]\!] = [\![\mathtt{map}]\!](f)(r) = \pi_2(\{\!\{(x, f(x)); r(x) \mid x \in r\}\!\})$$

  Recall that $[\![\mathtt{map}]\!]$ returns a bag, so by taking some $y$ such that $y \in [\![\mathtt{map}]\!](f)(r)$, we know how to calculate its multiplicity in the bag:

  $$([\![\mathtt{map}]\!](f)(r))(y) = \sum_{(x,f(x)) \in \{\!\{(x,f(x));r(x) \mid x \in r\}\!\} \wedge f(x)=y} \{\!\{(x, f(x)); r(x) \mid x \in r\}\!\}(x, f(x)) = \sum_{x \in r \wedge f(x)=y} r(x)$$

  which is the canonical representation of the bag defined by $SR(P)(r)$ - the multiplicity of $f(\mathbf{x}_r)$ is equal to the sum of all possible preimages $r(\mathbf{x}_r)$ in $r$.
- *filter* : Suppose $E = \mathtt{filter}(f)(r)$.

  $$\begin{aligned} p_1(SR(P)(r)) &= \{\!\{\Phi(P)(\mathbf{x}_r); r(\mathbf{x}_r) \mid \Phi(P)(\mathbf{x}_r) \neq \bot \wedge \mathbf{x}_r \in r\}\!\} \\ &= \{\!\{ite(f(\mathbf{x}_r) = tt, (\mathbf{x}_r; r(\mathbf{x}_r)), \bot) \mid f(\mathbf{x}_r) = tt \wedge \mathbf{x}_r \in r\}\!\} \\ &= \{\!\{(\mathbf{x}_r; r(\mathbf{x}_r)) \mid f(\mathbf{x}_r) = tt \wedge \mathbf{x}_r \in r\}\!\} \end{aligned}$$

While:

$$[\![\texttt{filter}]\!](f)(r) = r\restriction_{\{x|f(x)\}} = \sigma_{x\in\{x|f(x)\}}(r) = \{\!\{x; r(x) \mid x \in \{x \mid f(x)\}\}\!\}$$

And the equality of the bags follows.

- *cartesian*: Suppose $E = \texttt{cartesian}(r_1, r_2)$. We have:

$$\begin{aligned} p_1(SR(P)(r_1,r_2)) &= \{\!\{\Phi(P)(\mathbf{x}_{r_1},\mathbf{x}_{r_2}); r_1(\mathbf{x}_{r_1})r_2(\mathbf{x}_{r_2}) \mid \Phi(P)(\mathbf{x}_{r_1},\mathbf{x}_{r_2}) \neq \bot \wedge \mathbf{x}_{r_1} \in r_1 \wedge \mathbf{x}_{r_2} \in r_2\}\!\} \\ &= \{\!\{(\mathbf{x}_{r_1},\mathbf{x}_{r_2}); r_1(\mathbf{x}_{r_1})r_2(\mathbf{x}_{r_2}) \mid (\mathbf{x}_{r_1},\mathbf{x}_{r_2}) \neq \bot \wedge (x,y) \in (r_1,r_2)\}\!\} \\ &= \{\!\{(\mathbf{x}_{r_1},\mathbf{x}_{r_2}); r_1(\mathbf{x}_{r_1})r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_1} \in r_1 \wedge \mathbf{x}_{r_2} \in r_2\}\!\} \end{aligned}$$

In the standard semantics, we have:

$$[\![\texttt{cartesian}]\!](r_1,r_2) = r_1 \times r_2 = \{\!\{(x_1,x_2)\,;\, r_1(x_1)\cdot r_2(x_2) \mid x_1 \in r_1 \wedge x_2 \in r_2\}\!\}$$

And the equality is straightforward. For self-cartesian-products, we will have two unique names in $SR(P)$: $\mathbf{x}_r^{(1)}, \mathbf{x}_r^{(2)}$, so again equality will follow immediately.

- *union*: Suppose $E = \texttt{union}(r_1, r_2)$. We have: still cleaning it up

$$\begin{aligned} p_1(SR(P)(r_1,r_2)) &= \{\!\{\Phi(P)(\mathbf{x}_{r_1},\mathbf{x}_{r_2}); \mathcal{M}(r_1,r_2)(\mathbf{x}_{r_1}, r_2(\mathbf{x}_{r_2}) \mid \Phi(P)(\mathbf{x}_{r_1},\mathbf{x}_{r_2}) \neq \bot \wedge (\mathbf{x}_{r_1},\mathbf{x}_{r_2}) \in (r_1,r_2)\}\!\} \\ &=^{def} \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\} \cup \{\!\{\mathbf{x}_{r_2}; r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_2} \in r_2\}\!\} \end{aligned}$$

while:

$$[\![\texttt{union}]\!](r_1,r_2) = r_1 \cup r_2 = \{\!\{x\,;\, ite(r_1(x) \neq \bot, r_1(x), 0) + ite(r_2(x) \neq \bot, r_2(x), 0) \mid x \in r_1 \cup r_2\}\!\}$$

We prove containment of both sides:

  - $p_1(SR(P)(r_1,r_2)) \subset [\![\texttt{union}]\!](r_1,r_2)$: Let $a \in p_1(SR(P)(r_1,r_2))$. Then $a \in \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\}$ or $a \in \{\!\{\mathbf{x}_{r_2}; r_2[\mathbf{x}_{r_2} \mid \mathbf{x}_{r_2} \in r_2\}\!\}$. Suppose w.l.o.g. (symmetry) that $a \in \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\}$. Then $a$ has multiplicity $r_1(a)$ in $r_1$. If we also have $a \in \{\!\{\mathbf{x}_{r_2}; r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_2} \in r_2\}\!\}$ then $a$ has multiplicity $r_2(a)$ in $r_2$ and we get that $a \in r_1 \cup r_2$ with multiplicity $r_1(a) + r_2(a)$. Therefore $a \in \{\!\{x\,;\, ite(r_1(x) \neq \bot, r_1(x), 0) + ite(r_2(x) \neq \bot, r_2(x), 0) \mid x \in r_1 \cup r_2\}\!\} = [\![\texttt{union}]\!](r_1,r_2)$ as required. If $a \notin \{\!\{\mathbf{x}_{r_2}; r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_2} \in r_2\}\!\}$ then $r_2(a) = \bot$. Still, $a \in r_1 \cup r_2$ but with multiplicity $r_1(a)$, thus: $a \in \{\!\{x\,;\, ite(r_1(x) \neq \bot, r_1(x), 0) + ite(r_2(x) \neq \bot, r_2(x), 0) \mid x \in r_1 \cup r_2\}\!\} = [\![\texttt{union}]\!](r_1,r_2)$ with multiplicity $r_1(a) + 0$, as required.
  - $[\![\texttt{union}]\!](r_1,r_2) \subset p_1(SR(P)(r_1,r_2))$: Let $a \in [\![\texttt{union}]\!](r_1,r_2)$. Then $a \in r_1 \cup r_2$ and from the condition we know that at least one of $r_1(a)$ or $r_2(a)$ are not $\bot$. Assume w.l.o.g. (symmetry) that $r_1(a) \neq \bot$. Then $a \in \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\} \subset p_1(SR(P)(r_1,r_2))$ as required.

- *fold*: Suppose $E = \texttt{fold}(e,f)(r)$. We have: $p_1(SR(P)(r)) =^{def} [\mathbf{x}_r]_{e,f} =^{def} [\![\texttt{fold}]\!](e,f)(r)$ pending aggregate completion
- *foldByKey*: Suppose $E = \texttt{foldByKey}(e,f)(r)$. We have: pending by-key completion

$$p_1(SR(P)(r)) =^{def} \{\!\{\langle \mathbf{x}_r\rangle_{e,f}; 1 \mid \mathbf{x}_r \in r\}\!\} = \{\!\{(p_1(\mathbf{x}_r), [p_2(\mathbf{x}_r)]_{e,f}; 1 \mid \mathbf{x}_r \in r\}\!\}$$

By proving: $[p_2(\mathbf{x}_r)]_{e,f} = [\![\texttt{fold}]\!](e,f)(\pi_2(r\restriction_{(\mathbf{x}_r^k,\_)}))$ we get $p_1(SR(P)(r)) = [\![\texttt{foldByKey}]\!](e,f)(r)$, as required.

◂

**Program equivalence problem formalization using representative elements** Given two programs $P, Q$ receiving as input a series of RDDs $\vec{r^i} = (r^i_1, \ldots, r^i_k)$. The $PE$ problem becomes the problem of proving the following formulas:

$$(*) \quad \mathcal{M}(P) = \mathcal{M}(Q)$$
$$(**) \quad \neg(\exists \vec{x} \in \overrightarrow{r^i \cup \{\bot\}}.\ \Phi(P)[\overrightarrow{\mathbf{x}_{r^i}} \mapsto \vec{x}] \neq \Phi(Q)[\overrightarrow{\mathbf{x}_{r^i}} \mapsto \vec{x}])$$

where the choice of $r^i$ is arbitrary.

Note, that in the $PE$ problem, we had to check equivalence of functions $\mathcal{D}^n \to \mathcal{D}^m$, while here we check equivalence of functions $\mathcal{T} \cup \{\bot\} \to \mathcal{T} \cup \{\bot\}$ (Reminder: $\mathcal{T}$ is the semantic domain of all products of primitive types `Int`, `Boolean`; $\mathcal{D}$ is the semantic domain including both $\mathcal{T}$ and $RDD$).

In addition, $(**)$ can be written as:

$$\neg(\exists \vec{x}. \Phi(P)[\overrightarrow{\mathbf{x}_{r^i}} \mapsto \vec{x}] \neq \Phi(Q)[\overrightarrow{\mathbf{x}_{r^i}} \mapsto \vec{x}])$$

The change from $(**)$ is that we do not need to be aware of the input RDDs themselves, only of their domain. This is possible because we are ignoring the *subtract* operation.

locality was here originally.

**A remark on variable renaming** $\phi$ is guaranteed to return a fresh variable for an input RDDs each time it is called. We could have one program term over the variable $\mathbf{x}^{(1)}_R$, and for the second program to have a program term over the variable $\mathbf{x}^{(2)}_R$, Both being representative elements of the same input RDD, $R$. To handle this in the general case, where 2 programs receive as input $k$ input RDDs $R_i$, and for each $i$ its representative element has $n_i$ instances in the program term. Because we already verified $\mathcal{M}(P) = \mathcal{M}(Q)$, $n_i$ is well defined and equal in both program terms. Suppose that the set of representative elements of $R_i$ in $P$ is called $S^i_P$, and $S^i_Q$ in $Q$. We denote $S_P = \cup_{i=1}^k S^i_P, S_Q = \cup_{i=1}^k S^i_Q$ as the set of variable names for all RDDs in $P, Q$. Thus, we can find an *isomorphism* $\mathcal{S}$ between $S_P$ and $S_Q$, such that $\forall i. \sigma_{S^i_P}(\mathcal{S})$ is injective on $S^i_Q$.

## 5.2 Proof technique

### 5.2.1 Equivalency without RDDs

The following proposition follows directly from the decidability of Presburger arithmetic [].

▸ Proposition 5.2. Given two $SPARK$ programs $P$ and $Q$ which use only integer or boolean basic types and no RDDs, $PE$ is decidable.

**Proof.** *sketch.* Let the signatures of $P$ and $Q$ be $P(\overline{T}){:}\,\tau$, $Q(\overline{T}){:}\,\tau$. Expression and functions in $SPARK$ belong to an extension of the Presburger arithmetic to integer numbers, which is decidable [4]. If the return types $\tau$ are tuples, then on per-element basis, equivalency is decidable: For each element of the returned tuple we get an equation of functions applied to the variables in the programs, definable in the Presburger arithmetic, so the problem of program equivalence reduces to solving the Presburger formula. It is decidable by using a decision procedure such as *Cooper's algorithm* []. ◂

*A remark on complexity: Cooper's algorithm has an upper bound of $2^{2^{2^{pn}}}$ for some $p > 0$ and where $n$ is the number of symbols in the formula [12]. In practice, our experiments show that Cooper's algorithm on non-trivial formulas returns almost instantly, even on commodity hardware.*

$$
\begin{array}{llcl}
\textbf{Program:} & \phi_P(E,k) & = & \begin{cases} (t_{E'}[x \mapsto t_\eta], m),\ \textit{where} & \\ (t_{E'}, n) = \phi_P(E', k), (t_\eta, m) = \phi_P(\eta, n) & E = \textit{Let } x = \eta \textit{ in } E' \\ \phi_P(\eta, k) & E = \eta \end{cases} \\
\textbf{Basic exps.:} & \phi_P(e,k) & = & (e,k) \\
\textbf{RDD exprs.:} & & & \\
\textbf{Map:} & \phi_P(\mathtt{map}(f)(r),k) & = & (f(t), m),\ \textit{where } (t,m) = \phi_P(r,k) \\
\textbf{Filter:} & \phi_P(\mathtt{filter}(f)(r)) & = & \textit{ite}(f(t) = tt, (t,m), \bot),\ \textit{where } (t,m) = \phi_P(r,k) \\
\textbf{Cartesian:} & \phi_P(\mathtt{cartesian}(r_1,r_2),k) & = & ((t_{r_1}, t_{r_2}), m),\ \textit{where } (t_{r_1}, n) = \phi_P(r_1,k), (t_{r_2}, m) = \phi_P(r_2, n) \\
\textbf{Union:} & \phi_P(\mathtt{union}(r_1,r_2),k) & = & ((t_{r_1} \cup t_{r_2}), m),\ \textit{where } (t_{r_1}, n) = \phi_P(r_1,k), (t_{r_2}, m) = \phi_P(r_2, n) \\
 & & & \\
\textbf{Input RDDs:} & \phi_P(r,k) & = & \begin{cases} (\mathbf{x}_r^{(k)}, k+1) & r \in \overline{r} \\ (r,k) & \textit{otherwise} \end{cases}
\end{array}
$$

$$
\textit{Let } P : \boldsymbol{P}(\overline{\boldsymbol{r}}, \overline{\boldsymbol{v}}) = \overline{\boldsymbol{F}}\,\overline{\boldsymbol{f}}\,E
$$
$$
\Phi(P) = t,\ \textit{where } \phi_P(E, 0) = (t, \_)
$$

**Figure 9** Compiling *SPARK* expressions to logical terms ($\phi$).

$$
\textit{Let } P : \boldsymbol{P}(\overline{\boldsymbol{r}}, \overline{\boldsymbol{v}}) = \overline{\boldsymbol{F}}\,\overline{\boldsymbol{f}}\,E
$$

$$
\begin{array}{llcl}
\textbf{Program:} & \mathcal{M}(P) & = & \mathcal{M}(E) \\
 & \mathcal{M}(E) & = & \begin{cases} \mathcal{M}(E[x \mapsto \eta]) & E = \textit{Let } \boldsymbol{x} = \eta \textit{ in } E \\ e & E = \textit{return } e \end{cases} \\
\textbf{Basic exprs.:} & \mathcal{M}(e) & = & 1 \\
\textbf{RDD expressions:} & & & \\
\textbf{Map:} & \mathcal{M}(\mathtt{map}(f)(r)) & = & \mathcal{M}(r) \\
\textbf{Filter:} & \mathcal{M}(\mathtt{filter}(f)(r)) & = & \mathcal{M}(r) \\
\textbf{Cartesian:} & \mathcal{M}(\mathtt{cartesian}(r_1,r_2)) & = & \mathcal{M}(r_1) \cdot \mathcal{M}(r_2) \\
\textbf{Union:} & \mathcal{M}(\mathtt{union}(r_1,r_2)) & = & \mathcal{M}(r_1) + \mathcal{M}(r_2) \\
 & & & \\
\textbf{Input RDDs} & \mathcal{M}(r) & = & \mu_r
\end{array}
$$

**Figure 10** Compiling *SPARK* expressions to multiplicity terms ($\mathcal{M}$).

1. Compare the program signatures to see input types match. If not, return **not equivalent**.
2. Using typing rules (see appendix 10), check if the output RDDs' types match. If the types do not match, return **not equivalent**.
3. Apply for both $P$ and $Q$ the SR() semantics: build *program terms* (denoted $\Phi(P), \Phi(Q)$) and *multiplicity terms* (denoted $\mathcal{M}(P), \mathcal{M}(Q)$). Construction of the program and multiplicity terms is done by structural induction according to the rules appearing in Figures 9 and 10.
4. Verify that:
$$\mathcal{M}(P) = \mathcal{M}(Q)$$
If not, output **not equivalent**.
5. a. Choose an isomorphism $\mathcal{S}$ of the representative elements of the input RDDs in both $P, Q$, and rewrite $\phi_P$ accordingly by replacing all original names with the names from $Q$ returned by $\mathcal{S}$.
   b. Given an arbitrary vector $\vec{v}$ of concrete values to the input RDDs $\vec{r^i}$, we check the following formula is satisfiable:
$$\Phi(P)[\vec{\mathbf{x}_{r^i}} \mapsto \vec{v}] \neq \Phi(Q)[\vec{\mathbf{x}_{r^i}} \mapsto \vec{v}]$$
   c. If it is satisfiable, go back to (a) and repeat until finding an unsatisfiable formula, or all possible isomorphisms were exhausted.
   d. If the formula is unsatisfiable, return **equivalent**.
   e. If all isomorphisms were exhausted without finding an unsatisfiable formula, then return **not equivalent**.

**Figure 11** An algorithm for solving *PE*

### 5.2.2 Basic operations: Map and Filter

**Lemma 1** (Decidability for basic programs: *map* and *filter* operations)**.** *Given two SPARK programs P and Q which use only basic types coming from T (where all integer expressions are expressible using the* Presburger arithmetic*), only* map *and* filter *are allowed operations for RDDs,* PE *is decidable.*

**Proof.** For non-RDD return types we already proved in Proposition 5.2 - without aggregate operations, basic types can not be influenced by operations performed on RDDs. Thus, we can remove all RDD operations from the program and get an equivalent program in the setting of Proposition 5.2. For RDDs we provide an algorithm in Figure 11, which is a decision procedure. The correctness of the algorithm follows from the equivalency of the SR() semantics and the semantics defined in 3.4. It checks that two SR() expressions are equal, and the process terminates:

- Termination follows from having no loops in *SPARK*, and a finite number of RDDs giving a finite number of isomorphisms to check.
- Each step involving verifying a formula is decidable because the language of *SPARK* is limited to expressions in the decidable Presburger arithmetic theory.

◀

We proceed with examples.
**Note:** All examples use syntactic sugar for '*Let*' expressions. For brevity, instead of applying $\phi$ on the underlying '*Let*' expressions, we apply it line-by-line from the top-down. Finding

the isomorphism $\mathcal{S}$ between variable names in both programs is done automatically in all programs, but formally it is part of the decision procedure. In addition, we assume that in programs returning an RDD-type, the RDD is named $r^o$, and the programs always end with `return` $r^o$. Thus, $\Phi(P) = \phi_P(r^o), \mathcal{M}(P) = \mathcal{M}_P(r^o)$. Variable renaming is done automatically and not detailed in the examples.

▸ Example 5.1 (Basic optimization - operator pushback). This example shows a common optimization of pushing the filter/selection operator backward, to decrease the size of the dataset.

| | $\text{P1}(R\colon RDD_{\texttt{Int}})$: | $\text{P2}(R\colon RDD_{\texttt{Int}})$: |
|---|---|---|
| 1 | $R' = \texttt{map}(\lambda x.2 * x)(R)$ | $R' = \texttt{filter}(\lambda x.x < 7)(R)$ |
| 2 | $\texttt{return filter}(\lambda x.x < 14)(R')$ | $\texttt{return map}(\lambda x.x + x)(R')$ |

**RDD return type:** Both programs use integer operations only, and return an RDD of integers.

**Multiplicity terms:** We have $\mathcal{M}(P1) = \mathcal{M}_{P1}(R') = \mathcal{M}_{P1}(R) = \mu_R$, same for $P2$. Thus, multiplicity terms are equal.

**Analysis of representative elements:**
$\phi_{P1}(R') = 2 * \mathbf{x}_R$, and $\phi_{P1}(r^o) = \begin{cases} \varphi & \varphi < 14 \land \varphi = \phi_{P1}(R') \\ \bot & otherwise \end{cases} = \begin{cases} 2 * \mathbf{x}_R & 2 * \mathbf{x}_R < 14 \\ \bot & otherwise \end{cases}$.
$\phi_{P1}(R') = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases}$, and $\phi_{P2}(r^o) = (\lambda x.x + x)(\phi_{P1}(R')) = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases} + \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases}$.

We need to verify that:

$$\forall \mathbf{x}_R. \begin{cases} 2 * \mathbf{x}_R & 2 * \mathbf{x}_R < 14 \\ \bot & otherwise \end{cases} = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases} + \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases}$$

To prove this, we need to encode the cased expressions in Presburger arithmetic. Undefined ($\bot$) values indicate 'don't care' and are not part of the Presburger arithmetic. However, they can be handled by assuming the 'if' condition is satisfied, and verifying that the condition is indeed satisfied equally for all inputs. The first condition, therefore, is that both 'if' conditions agree on all possible values. The second condition is that the resulting expressions are equivalent.

▸ Proposition 5.3 (Schemes for converting conditionals to a normal form). We write a series of universally true schemes for translating the *filter* cased expression to Presburger arithemtic when appearing in an equivalence formula:

1. The following useful identity for applying functions on a conditional is true:

$$f\left(\begin{cases} e & cond \\ \bot & otherwise \end{cases}\right) = \begin{cases} f(e) & cond \\ \bot & otherwise \end{cases}$$

2. Equivalence of functions of conditionals:

$$f\left(\begin{cases} e & cond \\ \bot & otherwise \end{cases}\right) = g\left(\begin{cases} e' & cond' \\ \bot & otherwise \end{cases}\right) \iff (cond \iff cond') \land (cond \implies f(e) = g(e'))$$

3. Equivalence of a function of a conditional and an arbitrary expression:

$$f\left(\begin{cases} e & cond \\ \perp & otherwise \end{cases}\right) = e' \iff cond \wedge f(e) = e'$$

4. Applying a function with multiple arguments on multiple conditionals (a function receiving $\perp$ input as one of its arguments returns a $\perp$):

$$f\left(\begin{cases} e & cond \\ \perp & otherwise \end{cases}, \begin{cases} e' & cond' \\ \perp & otherwise \end{cases}\right) = \begin{cases} f(e,e') & cond \wedge cond' \\ \perp & otherwise \end{cases},$$

5. Applying a function with multiple arguments on a conditional and a general expression:

$$f\left(\begin{cases} e & cond \\ \perp & otherwise \end{cases}, e'\right) = \begin{cases} f(e,e') & cond \\ \perp & otherwise \end{cases},$$

6. The two last rules define the base case for functions with more than 2 arguments where at least one of the arguments is a conditional.
7. Unnseting of nested conditionals

$$\begin{cases} \begin{cases} e & c_{int} \\ \perp & otherwise \end{cases} & c_{ext} \\ \perp & otherwise \end{cases} = \begin{cases} e & c_{int} \wedge c_{ext} \\ \perp & otherwise \end{cases}$$

SG: We still need to prove the schemes. It seems an SMT solver can handle this for general theories.

Using Cooper's algorithm and the above schemes, we can prove the equivalence formula is true. See Figure 12 for an implementation. ■

```
integer_ qelim
<<forall x.
(2*x<14 <=> x<7) /\ (2*x<14 ==> 2*x = x+x)>>;;
- :  fol formula = <<true>>
```

**Figure 12** Output of Cooper ML implementation proving this example

### 5.2.3 Cartesian Product and Join

We see that lemma 1 can be extended naturally to cartesian products. We begin with simple examples not requiring cardinality check and handling of self cartesian products. We shall define the *join* operator based on the *cartesian* and *filter* and *map* operators, and then proceed with examples proving properties of the join.

▸ Example 5.2 (Natural Join). This example shows an implementation of the *natural join* using the existing operations. In general, $R: RDD_{K\times V}, R': RDD_{K\times V'}$, but we choose $K = V = V' =$ `Int`.

| | P1($R: RDD_{\texttt{Int}\times\texttt{Int}}, R': RDD_{\texttt{Int}\times\texttt{Int}}$): | P2($R: RDD_{\texttt{Int}\times\texttt{Int}}, R': RDD_{\texttt{Int}\times\texttt{Int}}$): |
|---|---|---|
| 1 | `return join`$(R, R')$ | $C =$ `cartesian`$(R, R')$ |
| 2 | | $J =$ `filter`$(\lambda x.p_1(p_1(x)) == p_1(p_2(x)))(C)$ |
| 3 | | `return map`$(\lambda x.(p_1(p_1(x)),(p_2(p_1(x)),p_2(p_2(x)))))(J)$ |

Let us mark the representative elements of the variables of $P2$:

1. Inputs: $\mathbf{x}_R, \mathbf{x}_{R'}$
2. $\phi_{P2}(C) = (\mathbf{x}_R, \mathbf{x}_{R'})$
3. $\phi_{P2}(J) = \begin{cases} \phi_{P2}(C) & p_1(p_1(\phi_{P2}(C))) = p_1(p_2(\phi_{P2}(C))) \\ \bot & otherwise \end{cases} = \begin{cases} (\mathbf{x}_R, \mathbf{x}_{R'}) & p_1(\mathbf{x}_R) = p_1(\mathbf{x}_{R'}) \\ \bot & otherwise \end{cases}$
4. $\phi_{P2} = \begin{cases} (p_1(\mathbf{x}_R), (p_2(\mathbf{x}_R), p_2(\mathbf{x}_{R'})) & p_1(\mathbf{x}_R) = p_1(\mathbf{x}_{R'}) \\ \bot & otherwise \end{cases}$

We see that the representative element of the returned RDD expression is describing exactly the expected semantics of the natural join. *Remark:* In Spark, however, the *join* transform yields a different partitioning than *cartesian* + *filter* + *map*. Number of partitions after *join* operation is smaller than the product of number of partitions as in *cartesian*. See Appendix.. for an example.

SG: TODO

We can proceed and use *join* in our examples as syntactic sugar for a combination of *cartesian*, *filter* and *map* in the method that fits the arity of the tuple types automatically, similarly to the example.

The representative element for the *join* of RDDs $R\colon RDD_{\tau_k \times \tau_v}, R'\colon RDD_{\tau_k \times \tau_w}$ is:

$$\phi(R \bowtie R') = \begin{cases} (p_1(\varphi), (p_2(\varphi), p_2(\varphi')) & p_1(\varphi) = p_1(\varphi') \land \varphi = \phi(R), \varphi' = \phi(R') \\ \bot & otherwise \end{cases}$$

▸ **Example 5.3** (*join* distributivity with *map*). This example shows a case where *join* is distributive with respect to *map*.

| | P1($R_0$: $RDD_{\texttt{Int}\times\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}\times\texttt{Int}}$): | P2($R_0$: $RDD_{\texttt{Int}\times\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}\times\texttt{Int}}$): |
|---|---|---|
| 1 | `return map`$(\lambda x.2*x)$(`join`$(R_0, R_1)$) | $S_0 =$ `map`$(\lambda x.2*x)(R_0)$ |
| 2 | | $S_1 =$ `map`$(\lambda x.2*x)(R_1)$ |
| 3 | | `return join`$(S_0, S_1)$ |

For $P1$:

$$\phi_{P1} = \begin{cases} 2*((p_1(\mathbf{x}_{R_0}), (p_2(\mathbf{x}_{R_0}), p_2(\mathbf{x}_{R_1}))) & p_1(\mathbf{x}_{R_0}) = p_1(\mathbf{x}_{R_1}) \\ \bot & otherwise \end{cases}$$

For $P2$:

$$\phi_{P2}(S_0) = 2*\mathbf{x}_{R_0}$$

$$\phi_{P2}(S_1) = 2*\mathbf{x}_{R_1}$$

$$\begin{aligned} \phi_{P2} &= \begin{cases} (p_1(\phi_{P2}(S_0)), (p_2(\phi_{P2}(S_0)), p_2(\phi_{P2}(S_1))) & p_1(\phi_{P2}(S_0)) = p_1(\phi_{P2}(S_1)) \\ \bot & otherwise \end{cases} \\ &= \begin{cases} (p_1(2*\mathbf{x}_{R_0}), (p_2(2*\mathbf{x}_{R_0}), p_2(2*\mathbf{x}_{R_1})) & p_1(2*\mathbf{x}_{R_0}) = p_1(2*\mathbf{x}_{R_1}) \\ \bot & otherwise \end{cases} \end{aligned}$$

The equivalence formula $\forall \mathbf{x}_{R_0}, \mathbf{x}_{R_1}.\phi_{P1} = \phi_{P2}$ can be proven in the Presburger arithmetic using the schemes defined in proposition 5.3. ◼

▸ **Example 5.4** (A counterexample - no *join* distributivity with *map* when key mappings do not agree). This example shows when *join* is not distributive with respect to *map*.

| P1($R_0$: $RDD_{\texttt{Int}\times\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}\times\texttt{Int}}$): | P2($R_0$: $RDD_{\texttt{Int}\times\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}\times\texttt{Int}}$): |
|---|---|
| `return map`$(\lambda(x,y).(1,y))$(`join`$(R_0,R_1)$) | $S_0$ = `map`$(\lambda(x,y).(1,y))(R_0)$ |
| | $S_1$ = `map`$(\lambda(x,y).(1,y))(R_1)$ |
| | `return join`$(S_0,S_1)$ |

Fpr $P1$:

$$\phi_{P1} = \begin{cases} (1,(p_2(\mathbf{x}_{R_0}),p_2(\mathbf{x}_{R_1}))) & p_1(\mathbf{x}_{R_0}) = p_1(\mathbf{x}_{R_1}) \\ \bot & otherwise \end{cases}$$

For $P2$:

$$\phi_{P2}(S_0) = (1, p_2(\mathbf{x}_{R_0}))$$
$$\phi_{P2}(S_1) = (1, p_2(\mathbf{x}_{R_1}))$$

$$\begin{aligned} \phi_{P2} &= \begin{cases} (p_1(\phi_{P2}(S_0)),(p_2(\phi_{P2}(S_0)),p_2(\phi_{P2}(S_1))) & p_1(\phi_{P2}(S_0)) = p_1(\phi_{P2}(S_1)) \\ \bot & otherwise \end{cases} \\ &= \begin{cases} (1,(p_2(\mathbf{x}_{R_0}),p_2(\mathbf{x}_{R_1}))) & 1 = 1 \\ \bot & otherwise \end{cases} \end{aligned}$$

The two program terms are equivalent if and only if:

$$\forall \mathbf{x}_{R_0}, \mathbf{x}_{R_1}. p_1(\mathbf{x}_{R_0}) = p_1(\mathbf{x}_{R_1})$$

Or in other words, if the keys of the two input RDDs always agree. This is of course wrong, so there is no equivalency in this case.

▸ **Corollary 2.** *Let $R, R'$ be RDDs with a pair type, the first type in the pair being $K$ and second type being $\tau_1, \tau_2$ respectively. Let $f$ be a function $f : K \times \tau_1 \to K' \times \sigma_1$, and $g$ be a function $g : K \times \tau_2 \to K' \times \sigma_2$. If $f$ and $g$ agree on $K$ (namely, $\forall x,y. p_1(x) = p_1(y) \iff p_1(f(x)) = p_1(g(y))$), then the following two programs are equivalent:*

$$\begin{array}{ll} (1) & \texttt{join}(\texttt{map}(f)(R), \texttt{map}(g)(R')) \\ (2) & \texttt{map}(\mathcal{F})(\texttt{join}(R,R')) \end{array}$$

*where* $\mathcal{F} = \lambda(k,(v,w).p_1(f((k,v))),(p_2(f((k,v))),p_2(g((k,w))))$

**Proof.** We know that $\forall x,y. p_1(x) = p_1(y) \iff p_1(f(x)) = p_1(g(y))$, and proceed to compare representative elements.

$$\phi_{\texttt{map}(f)(R)} = f(\mathbf{x}_R)$$
$$\phi_{\texttt{map}(g)(R')} = g(\mathbf{x}_{R'})$$

$$\phi_{\texttt{join}(\texttt{map}(f)(R),\texttt{map}(g)(R'))} = \begin{cases} (p_1(f(\mathbf{x}_R)),(p_2(f(\mathbf{x}_R)),p_2(g(\mathbf{x}_{R'})))) & p_1(f(\mathbf{x}_R)) = p_1(g(\mathbf{x}_{R'})) \\ \bot & otherwise \end{cases}$$

$$\phi_{\texttt{join}(R,R')} = \begin{cases} (p_1(\mathbf{x}_R),(p_2(\mathbf{x}_R),p_2(\mathbf{x}_{R'})) & p_1(\mathbf{x}_R)) = p_1(\mathbf{x}_{R'}) \\ \bot & otherwise \end{cases}$$

$$\begin{aligned} \phi_{\texttt{map}(\mathcal{F})(\texttt{join}(R,R'))} &= \begin{cases} \mathcal{F}((p_1(\mathbf{x}_R),(p_2(\mathbf{x}_R),p_2(\mathbf{x}_{R'}))) & p_1(\mathbf{x}_R)) = p_1(\mathbf{x}_{R'}) \\ \bot & otherwise \end{cases} \\ &= \begin{cases} (p_1(f(\mathbf{x}_R)),(p_2(f(\mathbf{x}_R)),p_2(g(\mathbf{x}_{R'})))) & p_1(\mathbf{x}_R)) = p_1(\mathbf{x}_{R'}) \\ \bot & otherwise \end{cases} \end{aligned}$$

All that is left to prove is the equivalence of the conditions:

$$\forall \mathbf{x}_R, \mathbf{x}_{R'}. p_1(f(\mathbf{x}_R)) = p_1(g(\mathbf{x}_{R'})) \iff p_1(\mathbf{x}_R)) = p_1(\mathbf{x}_{R'})$$

Which is proved directly from the known property of $f, g$ of agreeing on $K$. ◂

**Self cartesian products and self joins** When handling self-joins or cartesian product on the same RDD, it is crucial to rename the free variables of the representative elements. For example:

| | | |
|---|---|---|
| | P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
| 1 | `return cartesian`$(R, R)$ | `return map`$(\lambda x.(x, x))(R)$ |

The two programs are obviously not equivalent, but sloppy handling of the generation of the representative elements may lead to mistakes. The program term of $P1$ is $(\phi_{P1}(R), \phi_{P1}(R))$, while the program term of $P2$ is: $(\mathbf{x}_R, \mathbf{x}_R)$. If we take $\phi_{P1}(R) = \mathbf{x}_R$ for any application of $\phi_{P1}(R)$ we get: $\phi_{P1} = (\mathbf{x}_R, \mathbf{x}_R) = \phi_{P2}$ which is a mistake. We therefore need to have $\phi$ generate *fresh* variables for any application done in the context of a cartesian product.

▸ Example 5.5 (Self joins). In this example, given an RDD of integers, we want all pairs of elements in the cartesian product whose sum is greater than 100. We can filter out all pairs where both elements are not larger than 50.

| | | |
|---|---|---|
| | P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
| 1 | $C$ = `cartesian`$(R, R)$ | $C$ = `cartesian`$(R, R)$ |
| 2 | `return filter`$(\lambda x, y.x + y > 100)(C)$ | $C'$ = `filter`$(\lambda x, y.x > 50 \vee y > 50)(C)$ |
| 3 | | `return filter`$(\lambda x, y.x + y > 100)(C')$ |

The representative elements of $P1$ (note the automatic renaming of variables for $R$):

$$\mathbf{x}_C = (\phi_{P1}(R), \phi_{P1}(R)) = (\mathbf{x}_R^{(1)}, \mathbf{x}_R^{(2)})$$

$$\phi_{P1} = \begin{cases} (\mathbf{x}_R^{(1)}, \mathbf{x}_R^{(2)}) & \mathbf{x}_R^{(1)} + \mathbf{x}_R^{(2)} > 100 \\ \bot & \mathit{otherwise} \end{cases}$$

The representative elements of $P2$:

$$\phi_{P2}(C) = (\mathbf{x}_R^{(1)}, \mathbf{x}_R^{(2)})$$

$$\phi_{P2}(C') = \begin{cases} (\mathbf{x}_R^{(1)}, \mathbf{x}_R^{(2)}) & \mathbf{x}_R^{(1)} > 50 \vee \mathbf{x}_R^{(2)} > 50 \\ \bot & \mathit{otherwise} \end{cases}$$

$$\phi_{P2} = \begin{cases} \begin{cases} (\mathbf{x}_R^{(1)}, \mathbf{x}_R^{(2)}) & \mathbf{x}_R^{(1)} > 50 \vee \mathbf{x}_R^{(2)} > 50 \\ \bot & \mathit{otherwise} \end{cases} & \mathbf{x}_R^{(1)} + \mathbf{x}_R^{(2)} > 100 \\ \bot & \mathit{otherwise} \end{cases}$$

We can show equivalency if we prove:

$$((\mathbf{x}_R^{(1)} > 50 \vee \mathbf{x}_R^{(2)} > 50) \wedge \mathbf{x}_R^{(1)} + \mathbf{x}_R^{(2)} > 100) \iff \mathbf{x}_R^{(1)} + \mathbf{x}_R^{(2)} > 100$$

A proof is implemented in Figure 13. ∎

```
integer_ qelim
<<forall x,y.
((x>50 \/ y>50) /\ x+y>100)<=>(x+y>100)>>;;
- :  fol formula = <<true>>
```

**Figure 13** Output of Cooper ML implementation proving this example

### 5.2.4 Union

▸ Example 5.6 (Union). Union return an representative element for two RDDs. We show how equivalence is proved for it by demonstrating it with an exmaple first.

| | $\mathrm{P1}(R_0\colon RDD_{\mathtt{Int}}, R_1\colon RDD_{\mathtt{Int}})$: | $\mathrm{P2}(R_0\colon RDD_{\mathtt{Int}}, R_1\colon RDD_{\mathtt{Int}})$: |
|---|---|---|
| 1 | $R' = \mathtt{union}(R_0, R_1)$ | $R'_0 = \mathtt{filter}(\lambda x.x < 3)(R_0)$ |
| 2 | $\mathtt{return}\ \mathtt{filter}(\lambda x.x < 3)(R')$ | $R'_1 = \mathtt{filter}(\lambda x.x < 3)(R_1)$ |
| 3 | | $\mathtt{return}\ \mathtt{union}(R'_0, R'_1)$ |

We have: $\phi_{P1}(R') = (\phi_{P1}(\mathbf{x}_{R_0}) \cup \phi_{P1}(\mathbf{x}_{R_1})) = (\mathbf{x}_{R_0} \cup \mathbf{x}_{R_1})$.

When applying *map* or *filter* on a representative element of a union, it is always on the expression inside it. For example, $\phi_{P1} = \left(\begin{cases}\mathbf{x}_{R_0} & \mathbf{x}_{R_0} < 3 \\ \bot & otherwise\end{cases} \cup \begin{cases}\mathbf{x}_{R_1} & \mathbf{x}_{R_1} < 3 \\ \bot & otherwise\end{cases}\right)$.

So we need to prove that: $\phi_{P1}(r^o) = (\phi_{P2}(\mathbf{x}_{R'_0}) \cup \phi_{P2}(\mathbf{x}_{R'_1}))$, or in other words:

$$\forall \mathbf{x}_{R_0}, \mathbf{x}_{R_1}. \left(\begin{cases}\mathbf{x}_{R_0} & \mathbf{x}_{R_0} < 3 \\ \bot & otherwise\end{cases} \cup \begin{cases}\mathbf{x}_{R_1} & \mathbf{x}_{R_1} < 3 \\ \bot & otherwise\end{cases}\right) = (\mathbf{x}_{R'_0} \cup \mathbf{x}_{R'_1})$$

which is straight forward from the definition of $R'_0, R'_1$, by comparing respective elements in the union, and the equal cardinalities. ■

The next lemma provides the additions to the decision procedure described in lemma 1 in order to solve *PE* for programs with union operations.

▸ Lemma 3. *Let $u_1 = (\phi(R_1) \cup \phi(R_2))$ and $u_2 = (\phi(R'_1) \cup \phi(R'_2))$ be two union expressions, where $R_1, R_2, R'_1, R'_2$ are all created using the same input RDDs. $SR(u_1) = SR(u_2)$ if and only if $(\phi(R_1) = \phi(R'_1) \wedge \phi(R_2) = \phi(R'_2)) \vee (\phi(R_1) = \phi(R'_2) \wedge \phi(R_2) = \phi(R'_1))$ and $\mathcal{M}_{R_1} + \mathcal{M}_{R_2} = \mathcal{M}_{R'_1} + \mathcal{M}_{R'_2}$.*

**Proof.** Let $\vec{x}$ be a valuation to the input RDDs of both $u_1, u_2$.
**If:** Suppose w.l.o.g. $\phi(R_1) = \phi(R'_1) = \varphi(\vec{x})$ and $\phi(R_2) = \phi(R'_2) = \psi(\vec{x})$, and also that $\mathcal{M}_{R_1} + \mathcal{M}_{R_2} = \mathcal{M}_{R'_1} + \mathcal{M}_{R'_2}$. Then the bags defined by the union expressions are both equal to: $\{\!\{\varphi(\vec{x}) | \vec{x} \in (\mathrm{dom}(\vec{r^i}))\}\!\} \cup \{\!\{\psi(\vec{x}) | \vec{x} \in (\mathrm{dom}(\vec{r^i}))\}\!\}$
**Only if:** Suppose towards a contradiction that there is $\vec{x} \in (\mathrm{dom}(\vec{r^i}))$ such that $(\phi(R_1)(\vec{x}) \neq \phi(R'_1)(\vec{x}) \vee \phi(R_2)(\vec{x}) \neq \phi(R'_2)(\vec{x})) \wedge (\phi(R_1)(\vec{x}) \neq \phi(R'_2)(\vec{x}) \vee \phi(R_2)(\vec{x}) \neq \phi(R'_1)(\vec{x}))$. We consider the bags $SR(u_1) = \{\!\{\phi(R_1)(\vec{r}) | \vec{r} \in (\mathrm{dom}(\vec{r^i}))\}\!\} \cup \{\!\{\phi(R_2)(\vec{r}) | \vec{r} \in (\mathrm{dom}(\vec{r^i}))\}\!\}$ and $SR(u_2) = \{\!\{\phi(R'_1)(\vec{r}) | \vec{r} \in (\mathrm{dom}(\vec{r^i}))\}\!\} \cup \{\!\{\phi(R'_2)(\vec{r}) | \vec{r} \in (\mathrm{dom}(\vec{r^i}))\}\!\}$, which are by assumption equal. From the equality we know:

1. if w.l.o.g. (symmetry) $\phi(R_1)(\vec{x}) = \bot$ then $u_1(\vec{x}) = (\phi(R_1) \cup \phi(R_2))(\vec{x}) = \bot$ by some *non-deterministic* choice of taking an element of $R_1$ from the union and not an element of $R_2$. Thus, $\bot \in u_2$. By limiting our input RDDs to contain only $x$, we get that $u_2(\vec{x}) = (\phi(R'_1) \cup \phi(R'_2))(\vec{x}) = \bot$, which is possible only if $\phi(R'_1)(\vec{x}) = \bot$ or $\phi(R'_2)(\vec{x}) = \bot$ (because $u_1 = u_2$ for any input RDDs $\vec{r^i}$). Then $(\phi(R_1)(\vec{x}) = \phi(R'_1)(\vec{x}) = \bot \vee \phi(R_1)(\vec{x}) = \phi(R'_2)(\vec{x}) = \bot)$.
2. if w.l.o.g. (symmetry) $\phi(R_1)(\vec{x}) \neq \bot$ then $u_1(\vec{x}) = \phi(R_1)(\vec{x})$ by some *non-deterministic* choice. Again we limit the input RDDs to contain only $\vec{x}$. Because $u_1 = u_2$ for any input RDDs $\vec{r^i}$ we have $u_2(\vec{x}) = \phi(R_1)(\vec{x})$. By definition of $u_2$ we get that $\phi(R_1)(\vec{x}) = \phi(R'_1)(\vec{x}) \vee \phi(R_1)(\vec{x}) = \phi(R'_2)(\vec{x})$.

3. if $\phi(R_1)(\vec{x}) = \phi(R_2)(\vec{x})$ then $u_1(\vec{x}) = \phi(R_1)(\vec{x}) = \phi(R_2)(\vec{x})$ by a *deterministic* choice. Therefore, as $u_1 = u_2$ for any choice of input RDDs $\vec{r^i}$, $u_2(\vec{x})$ is also deterministically chosen, and equal to $\phi(R_1)(\vec{x})$. Thus $\phi(R_1')(\vec{x}) = \phi(R_2')(\vec{x} = \phi(R_1)(\vec{x}) = \phi(R_2)(\vec{x})$, contradiction.

Because of symmetry we can have the same conclusions on $\phi(R_2)$ for both cases in which $\phi(R_2)(\vec{x}) = \perp$ and $\phi(R_2)(\vec{x}) \neq \perp$. To summarize we get that: $(*)(\phi(R_1)(\vec{x}) = \phi(R_1')(\vec{x}) \vee \phi(R_1)(\vec{x}) = \phi(R_2')(\vec{x})) \wedge (\phi(R_2)(\vec{x}) = \phi(R_1')(\vec{x}) \vee \phi(R_2)(\vec{x}) = \phi(R_2')(\vec{x}))$. From the third item we know that for $\phi(R_1)(\vec{x}) = \phi(R_2)(\vec{x})$ case the proof is completed, so we only need to prove for the case when $\phi(R_1)(\vec{x} \neq \phi(R_2)(\vec{x}$. From $(*)$ we assume w.l.o.g. (symmetry) that $\phi(R_1)(\vec{x}) = \phi(R_2)(\vec{x})$. Therefore, $\phi(R_1)(\vec{x}) \neq \phi(R_2)(\vec{x}) \implies^{(*)} \phi(R_2)(\vec{x}) = \phi(R_2')(\vec{x})$. But this is a contradiction.

Regarding the second claim, that $\mathcal{M}_{R_1} + \mathcal{M}_{R_2} = \mathcal{M}_{R_1'} + \mathcal{M}_{R_2'}$, it follows from the equality of the multisets $SR(u_1) = SR(u_2)$ ◂

Here came a section that showed how we can use an SMT solver instead of Cooper to prove equivalence of two programs with uninterpreted functions, with map, filter and join. Removed.

▸ **Theorem 4.** *The process described in the proof of lemmas 1, 3 and the comments in section 5.2.3 is a decision procedure for the* PE *problem in SPARK programs without the* subtract *transform and aggregations.*

SG: Solve for self-unions - i.e. filter(even)(R) union filter(odd)(R) == R

# DEMONS START HERE

| | |
|---|---|
| **The Fold operator:** | $[\varphi]_{init,f}$ |
| **The Fold By Key operator:** | $\langle\varphi\rangle_{init,f}$ |

**Figure 14** Aggregate operators for *SPARK* programs with aggregations

$$\begin{array}{llcl} \textbf{Fold:} & \phi_P([\![\texttt{fold}]\!](f,e)(r),k) & = & ([t]_{e,f},m),\ \textit{where}\ (t,m)=\phi_P(r,k) \\ & \mathcal{M}([\![\texttt{fold}]\!](f,e)(r)) & = & 1 \\ \textbf{FoldByKey:} & \phi_P([\![\texttt{foldByKey}]\!](f,e)(r),k) & = & (\langle t\rangle_{e,f},m),\ \textit{where}\ (t,m)=\phi_P(r,k) \\ & \mathcal{M}([\![\texttt{foldByKey}]\!](f,e)(r)) & = & 1 \end{array}$$

**Figure 15** Compiling *SPARK* semantic interpretation to representative elements and cardinality terms of RDDs, for aggregate operations. Program subscript is omitted.

# 6 Aggregate expressions

In the following section we discuss how the existing framework can be extended to prove equivalence of *SPARK* programs containing aggregate expressions.

First, it is noteworthy to discuss the many different terms for aggregate operations and their semantic meaning in Spark. The most generic aggregate operation is called simply *aggregate*. It accepts an initial argument, a fold function, and a combiner of folded values. It is very similar to the *fold* operation, except that *fold* does not expect a combiner function (to merge two elements of the result type). In that sense, the iteration of *fold* on the RDD's elements is linear. Another variation is *reduce* , for which we only specify the behavior on two source elements (thus the reduced type is equal to the original element types). *reduce* does not specify the behavior on a set of size 0, thus it is undefined for empty RDDs.

relevant only if we discuss partitioned RDDs

**Extending SR(P) with aggregate expressions** The formulas for aggregate operations are given using special operators (Figure 14).Those operators are used to build the program term of *SPARK* programs with aggregate expressions, and we extend the definitions of $\phi$ and $\mathcal{M}$ accordingly in Figure 15.The $[\varphi]_{\cdot,\cdot}$ operator *binds* all variables in $\varphi$. The $\langle\varphi\rangle_{\cdot,\cdot}$ operator is syntactic sugar for $(p_1(\varphi),[p_2(\varphi)]_{\cdot,\cdot})$ which *binds* all variables in $p_2(\varphi)$ which are not in $p_1(\varphi)$. That is, The variables of $p_1(\varphi)$ are still free. The motivation behind this will be explained in 6.6.

**Organization** The rest of this section is organized as follows: First, in 6.1, we discuss equivalence of programs having a single aggregate (*fold* ) expression which is the returned as an output. We present a sound and complete method for solving *PE* for such programs, allowing *map* , *filter* and *cartesian* operations, without self-cartesian-products, and a sound and complete extension to the method for handling self-cartseian-products as well. Section 6.2 proceeds with a generalization allowing to have programs with multiple aggregate (*fold* ) expressions which are independent of each other (i.e. no aggregate expression is computed using the result of a previous aggregate expression). Section 6.3 shows the previous results on classes of *SPARK* programs for which *PE* is decidable are tight, with a reduction of *Hilbert's 10'th problem* [] to *PE* . Then, we present the requirements for handling *union* operations as well (section 6.4).Concluding the section are chapters on the handling of nested aggregations (section 6.5 and *by-key* operations, i.e. *foldByKey* (section 6.6).

TBD

## 6.1 Single aggregate as the final expression

The simplest case of programs in which an aggregation operator appears, is one where a single aggregate operation is performed and it is the last RDD operation.

▸ Example 6.1 (Maximum and minimum). Below is a simple example of 2 equivalent programs representing the simple case of single aggregate operation in the end of the program:

Let: $max = \lambda M, x.\mathtt{if}(x > M)\,\mathtt{then}\,\{x\}\,\mathtt{else}\,\{M\}$
$min = \lambda M, x.\mathtt{if}(x < M)\,\mathtt{then}\,\{x\}\,\mathtt{else}\,\{M\}$

| | P1($R : RDD_{\mathtt{Int}}$): | P2($R : RDD_{\mathtt{Int}}$): |
|---|---|---|
| 1 | `return fold`$(\bot, max)(R)$ | $R' = \mathtt{map}(\lambda x.0 - x)(R)$ |
| 2 | | `return` 0- `fold`$(\bot, min)(R')$ |

In the above example we compute the maximum element of a numeric RDD in two different methods, in the first program by getting the maximum directly, and in the second by getting the additive inverse of the minimum of the additive inverses of the elements. The equivalence formula is:

$$[\mathbf{x}_R]_{\bot,max} = 0 - [0 - \mathbf{x}_R]_{\bot,min} = -[-\mathbf{x}_R]_{\bot,min}$$

To prove that the two reduced results are equal we use an inductive claim:

$$\forall x, A, A'.A = -A' \Rightarrow max(A, x) = -min(A', -x)$$

$$\begin{array}{lcl}
max(A,x) & =^? & -min(A',-x) \\
\begin{cases} A & A > x \\ x & otherwise \end{cases} & =^? & -\begin{cases} A' & A' < -x \\ -x & otherwise \end{cases} \\
 & = & \begin{cases} -A' & A' < -x \\ x & otherwise \end{cases} \\
 & =^{A=-A'} & \begin{cases} A & -A < -x \\ x & otherwise \end{cases} \\
 & = & \begin{cases} A & A > x \\ x & otherwise \end{cases}
\end{array}$$

Indeed, by replacing $A' = -A$ we get equal expressions. ■

### 6.1.1 Basic proof method

The inductive claim is generalized for the class of programs discussed here, of programs performing a single *aggregate* operation after a series of *map* , *filter* and *cartesian* operations (without variable renaming).

Those definitions lead to the following lemma:

▸ **Lemma 5.** *Let $R_0 \in RDD_{\sigma_0}, R_1 \in RDD_{\sigma_1}$, and denote their representative elements $\varphi_0, \varphi_1$ respectively. We assume $\varphi_0, \varphi_1$ were composed on* map , filter *and cartesian product on RDDs with disjoin set of variables, and that $\mathcal{M}(R_0) = \mathcal{M}(R_1)$. Let there be two* fold *functions $f_0 : \xi_0 \times \sigma_0 \to \xi_0, f_1 : \xi_1 \times \sigma_1 \to \xi_1$, two* initial *values $init_0, init_1{:}\xi$, and two functions $g : \xi_0 \to \xi, g' : \xi_1 \to \xi$. We denote a vector $\vec{v} = FV(\varphi_0, \varphi_1)$ of the free variables in both RDDs' terms,. We have: if*

$$g(init_0) = g'(init_1) \tag{1}$$

$$\begin{array}{rcl} \forall \vec{v}, A_{\varphi_0} : \xi_0, A_{\varphi_1} : \xi_1. \qquad g(A_{\varphi_0}) & = & g'(A_{\varphi_1}) \implies \\ g(f_0(A_{\varphi_0}, \varphi_0[FV(\varphi_0) \mapsto \vec{v}])) & = & g'(f_1(A_{\varphi_1}, \varphi_1[FV(\varphi_1) \mapsto \vec{v}])) \end{array} \tag{2}$$

*then* $g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1})$

**Proof.** First we recall the semantics of the `fold` operation on some RDD $R$, which is a bag. We choose an arbitrary element $a \in R$ and apply the fold function recursively on $a$ and on $R$ with a single instance of $a$ removed. We then write a sequence of elements in the order they are chosen by `fold`: $\langle a_1, \dots, a_n \rangle$, where $n$ is the sum of all multiplicities in the bag $R$. We also know that a requirement of aggregating operations' UDFs is that they are *commutative* and *associative*, so the order of elements chosen does not change the final result. We also extend $f_i$ to $\xi_i \times (\sigma_i \cup \{\bot\})$ by setting $f_i(A, \bot) = A$ ($\bot$ is defined to behave as the neutral element for $f_i$).

To prove $g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1})$, it is necessary to prove that

$$g([\![\texttt{fold}]\!](f_0, init_0)(R_0)) = g'([\![\texttt{fold}]\!](f_1, init_1)(R_1))$$

We set $A_{\varphi_j,0} = init_j$ for $j \in \{0,1\}$. Each element of $R_0, R_1$ is expressible by providing a concrete valuation to the free variables of $\varphi_0, \varphi_1$, namely the vector $\vec{v}$. We choose an arbitrary sequence of valuations to $\vec{v}$, denoted $\langle \vec{a}_1, \dots, \vec{a}_n \rangle$, and plug them into the *fold* operation for both $R_0, R_1$. The result is 2 sequences of *intermediate values* $\langle A_{\varphi_0,1}, \dots, A_{\varphi_0,n} \rangle$ and $\langle A_{\varphi_1,1}, \dots, A_{\varphi_1,n} \rangle$. We have that $A_{\varphi_j,i} = f_j(A_{\varphi_j,i-1}, \varphi_j[FV(\varphi_j) \mapsto \vec{a}_i])$ for $j \in \{0,1\}$, from the semantics of `fold`. Our goal is to show $g(A_{\varphi_0,n}) = g'(A_{\varphi_1,n})$ for all $n$. We prove the equality by induction on the *size* of the sequence of possible valuations of $\vec{v}$, denoted $n$. In each step $i$, we show $g(A_{\varphi_0,i}) = g'(A_{\varphi_1,i})$.

**Case $n = 0$:** $R_0 = R_1 = \bot$, so $[\![\texttt{fold}]\!](f_0, init_0)(R_0) = init_0$ and $[\![\texttt{fold}]\!](f_1, init_1)(R_1) = init_1$. From Equation (1), $g(init_0) = g'(init_1)$, as required.

**Case $n = i$, assuming correct for $n \le i - 1$:** By assumption, we know that the sequence of intermediate values up to $i-1$ is equal up to application of $g, g'$, and specifically $g(A_{\varphi_0,i-1}) = g'(A_{\varphi_1,i-1})$. We are given the $i$'th concrete valuation of $\vec{v}$, denoted $\vec{a}_i$. We need to show $A_{\varphi_0,i} = A_{\varphi_1,i}$, so we use the formula for calculating the next intermediate value:

$$\begin{array}{rcl} A_{\varphi_0,i} & = & f_0(A_{\varphi_0,i-1}, \varphi_0[FV(\varphi_0) \mapsto \vec{a}_i]) \\ A_{\varphi_1,i} & = & f_1(A_{\varphi_1,i-1}, \varphi_1[FV(\varphi_1) \mapsto \vec{a}_i]) \end{array}$$

We use Equation (2), plugging in $\vec{v} = \vec{a}_i$, $A_{\varphi_0} = A_{\varphi_0,i-1}$, and $A_{\varphi_1} = A_{\varphi_1,i-1}$. By the induction assumption, $g(A_{\varphi_0,i-1}) = g'(A_{\varphi_1,i-1})$, therefore $g(A_{\varphi_0}) = g'(A_{\varphi_1})$, so Equation (2) yields $g(f_0(A_{\varphi_0}, \varphi_0[FV(\varphi_0) \mapsto \vec{a}_i])) = g'(f_1(A_{\varphi_1}, \varphi_1[FV(\varphi_1) \mapsto \vec{a}_i]))$. By substituting back $A_{\varphi_j}$ and the formula for the next intermediate value, we get: $g(A_{\varphi_0,i}) = g'(A_{\varphi_1,i})$ as required. ◂

**Induction length and relation to size of bags** The proof of lemma 5 depends on the size of participating RDDs to be equal, or to be more precise, the set of possible valuations to the RDDs to be equal (therefore the condition $\mathcal{M}(R_0) = \mathcal{M}(R_1)$). Otherwise, induction termination is not well defined. As a workaround, we can use $\bot$ values instead of concrete valuations. Let there be two equal fold expressions under lemma's 5 premises, and a a valuation $\vec{v}$ in the valuation sequence returning a non-$\perp$ value in $R_0$, and $\perp$ valuation to $R_1$. We get:

$$\begin{array}{rcl} g(A_{\varphi_0,i}) = g'(A_{\varphi_1,i}) \quad \wedge \quad g(f_0(A_{\varphi_0,i}, \varphi_0[FV(\varphi_0) \mapsto \vec{v}])) & = & g'(f_1(A_{\varphi_1,i}, \perp)) = g'(A_{\varphi_1,i}) \\ \Longrightarrow \quad g(f_0(A_{\varphi_0,i}, \varphi_0[FV(\varphi_0) \mapsto \vec{v}])) & = & g(A_{\varphi_0,i}) \\ \Longrightarrow_{\varphi_0[FV(\varphi_0)\mapsto\vec{v}]\neq\perp} \qquad \forall A, c. f_0(A,c) & = & A \end{array}$$

In that case, we see in the last transition that the lemma's conditions are fulfilled only if the fold functions are constant, namely the intermediate value returned is never changed by it. If we assume though, that fold UDFs are not constant in the programs we consider, then fold operations on RDDs of different cardinalities never produce an equivalent result on all input RDDs. More specifically, lemma 5 is true when the inductive claim satisfy the locality property - ranging over the entire domain is the same as ranging over the support of the RDDs' bags. This way, we don't have to consider $\perp$ valuations in order to be able to handle RDDs of different sizes.

fishy due to removing info on locality property from earlier on, think how to present it clearly

SG: Two examples to add: (1) max(v) and foldByKey(,max),max(v) (2) fold over union with an empty rdd

Lemma 5 shows that an inductive proof of the equality of folded values is *sound*. The meaning is that given any two folded expressions which are not equivalent, the lemma always reports them as non-equivalent. After presenting several examples of the application of the lemma, we show a constraint on 5, for which the method is also complete.

**Examples** We proceed with several examples, showing different combinations of *fold* , various UDFs, and relational operators.

▸ Example 6.2 (Basic (double) counting). Below is a basic example of an application of the theorem on the *fold* operation. Here the fold expressions has a different type compared to the RDDs given to the fold function.

Let: $f = \lambda A, (a,b).A + b$

| | P1($R$: $RDD_\tau$): | P2($R$: $RDD_\tau$): |
|---|---|---|
| 1 | $R' = \texttt{map}(\lambda x.(x,1))(R)$ | $R' = \texttt{map}(\lambda x.(x,2))(R)$ |
| 2 | $\texttt{return}\ 2 * \texttt{fold}(0,f)(R')$ | $\texttt{return}\ \texttt{fold}(0,f)(R')$ |

After calculating representative elements for $R'$ in both programs (which is straightforward), for equivalence we need to prove $2 * [(\mathbf{x}_r, 1)]_{0,f} = [(\mathbf{x}_r, 2)]_{0,f}$. We apply Lemma 5. We set $g(x) = id, g'(x) = (\lambda x.2 * x)$, both *init* values to 0, and check the two conditions:

$$0 = g(0) = 2 * 0 \tag{1}$$

$$\begin{aligned} \forall x, A, A'.g(A) = A' \Rightarrow g(f(A,(x,1))) &= g(A+1) \\ &= 2 * A + 2 \\ &= g(A) + 2 \\ &= A' + 2 \\ &= f(A',(x,2)) \end{aligned} \tag{2}$$

Which is what had to be proven. ■

▸ Example 6.3 (Sum with/without zeroes). The next example deals with the ability to handle neutral elements of summation filtered out from RDDs, resulting in an equal sum in both programs.

Let: $sum = \lambda A, x.A + x$

| | P1($R$: $RDD_{\mathtt{Int}}$): | P2($R$: $RDD_{\mathtt{Int}}$): |
|---|---|---|
| 1 | $R' = \mathtt{filter}(\lambda x.x > 0)(R)$ | $R' = \mathtt{filter}(\lambda x.x > -1)(R)$ |
| 2 | $\mathtt{return\ fold}(0, sum)(R')$ | $\mathtt{return\ fold}(0, sum)(R')$ |

The program term of $P1$:

$$\phi_{P1} = \left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R > 0 \\ \bot & otherwise \end{cases}\right]_{0,sum}$$

The program term of $P2$:

$$\phi_{P2} = \left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R > -1 \\ \bot & otherwise \end{cases}\right]_{0,sum}$$

For *init* case, equivalency is obvious $(0 = 0)$. For the induction step we know that $\bot$ values encountered require to leave the intermediate fold value unchanged. In our case, knowing that the operation is summation, and for brevity of writing, we replace all $\bot$ instances in the representative element with the neutral element of the given *fold* UDF, which is 0. We assume $A = A'$ and need to prove:

$$A + \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 0 \\ 0 & otherwise \end{cases} = A' + \begin{cases} \mathbf{x}_R & \mathbf{x}_R > -1 \\ 0 & otherwise \end{cases}$$

Which boils down to the following comparison according to entire the $2 \times 2$ matrix of possible cases:

$$(\mathbf{x}_R > 0 \wedge \mathbf{x}_R > -1 \wedge \mathbf{x}_R = \mathbf{x}_R) \vee (\mathbf{x}_R <= 0 \wedge \mathbf{x}_R > -1 \wedge \mathbf{x}_R = 0) \vee$$
$$(\mathbf{x}_R <= 0 \wedge \mathbf{x}_R <= -1 \wedge 0 = 0) \vee (\mathbf{x}_R > 0 \wedge \mathbf{x}_R <= -1 \wedge \mathbf{x}_R = 0)$$

A proof using an open-source implementation of Cooper's algorithm in OCaml [] is given in Figure 16. ■

```
integer_ qelim
<<forall x.
((x > 0 /\ x > -1) \/ ((x <= 0 /\ x > -1) /\ x = 0) \/ (x <= 0 /\ x <= -1) \/ ((x > 0
/\ x <= -1) /\ x = 0))>>;;
- : fol formula = <<true>>
```

**Figure 16** Output of Cooper ML implementation proving this example

▸ Example 6.4 (Divisibility by 9). Here we show two programs that get an RDD of integers, which is interpreted as a number. Every element in the RDD is a digit. By assuming that the iteration over the elements is in fixed order, the RDD indeed represents a unique number.

SG: Remark: This example is interesting because the final transformation on the aggregated expressions does not satisfy the requirements for completeness ($x\%9$ is not injective), but it is still provable by lemma 5. It is also nice because it shows the power of UDFs in Spark.

Let: $makeNumber = \lambda N, x.N * 10 + x, sum = \lambda S, x.S + x$

| | P1($R$: $RDD_{\mathtt{Int}}$): | P2($R$: $RDD_{\mathtt{Int}}$): |
|---|---|---|
| 1 | $v = \mathtt{fold}(0, makeNumber)(R)$ | $v = \mathtt{fold}(0, sum)(R)$ |
| 2 | $\mathtt{return}\ v\%9$ | $\mathtt{return}\ v\%9$ |

We get an expression for the folded value $v$ in both programs:

$$\phi_{P1} = [\mathbf{x}_R]_{0,makeNumber}$$
$$\phi_{P2} = [\mathbf{x}_R]_{0,sum}$$

Need to prove: $\phi_{P1}\%9 = \phi_{P2}\%9$, or, after substitution: $[\mathbf{x}_R]_{0,makeNumber}\%9 = [\mathbf{x}_R]_{0,sum}\%9$ By applying lemma 5, taking $g = g' = \lambda x.x\%9$, we need to prove:

$$\forall x.A, A'.A\%9 = A'\%9 \implies (A * 10 + x)\%9 = (A' + x)\%9$$

Which is provable using Cooper's algorithm (see Figure 17). ■

```
integer_ qelim
<<forall x,a,b.
exists y,z.
((a-9*y = b-9*z)/\((a-9*y)<9)/\((a-9*y)>=0)) ==>
exists u,w.
(((10 * a + x)-9*u = (b+x)-9*w)/\(((b+x)-9*w) < 9)/\(((b+x)-9*w) >= 0))>>;;
- :  fol formula = <<true>>
```

**Figure 17** Output of Cooper ML implementation proving this example

### 6.1.2 Completeness

On its surface, the inductive claim does not permit a *sound* ***and complete*** method of verifying the equivalence for the restricted class of programs we defined. However, if at least one of the transformations applied on the aggregated expression is an injection, the equivalence of the programs implies the inductive claim, making it a *complete* proof method. The underlying principle allowing it, is that if we presumed the inductive claim to be false while the equivalence of the programs is true, then we could trim the RDDs to a prefix of the same size that violates the inductive claim, which would mean a violation of the assumption on equivalence. We formulate this intuition in the next paragraphs.

We begin with showing that when we apply on folded expressions a non-injective function in both programs, the lemma fails:

▸ Example 6.5 (Non-injective modification of folded expressions). This example shows how non-injective return statements can weaken the inductive claim, resulting in failure to prove it. As a result, it cannot be used to prove the equivalence of the following two programs.

| | P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | $R' = \texttt{map}(\lambda x.x\%3)(R)$ | $R' = \texttt{fold}(0, \lambda A, x.A + x)(R)$ |
| 2 | $\texttt{return fold}(0, \lambda A, x.(A + x)\%3)(R') = 0$ | $\texttt{return } R'\%3 = 0$ |

To prove the equivalence, we should check by induction the equality of both boolean results. Taking $g(x) = \lambda x.x = 0,\ g'(x) = \lambda x.(x \ mod\ 3) = 0$ and attempt to prove by induction the following claim:

$$[x\ mod\ 3]_{0,+\ mod\ 3} = 0 \Leftrightarrow [x]_{0,+\ mod\ 3}\ mod\ 3 = 0$$

fails. To illustrate, we attempt to prove:

$$\forall x, A, A'.A = 0 \iff A'\ mod\ 3 = 0 \implies (A + x\ mod\ 3)\ mod\ 3 = 0 \iff (A' + x)\ mod\ 3 = 0$$

Suppose that in the induction hypothesis we have $A = 1, A' = 2$. Then the hypothesis that says $A = 0 \iff A'\ mod\ 3 = 0$ is satisfied, but it cannot be said that $(A + x\ mod\ 3)\ mod\ 3 = 0 \iff (A' + x)\ mod\ 3 = 0$ (take $x = 1$: $((1 + (1\%3))\%3 = 2, (2 + 1)\%3 = 0)$).

And indeed, Cooper's algorithm outputs '`false`' for it:

```
integer_qelim
<<forall x,a,b.
(a = 0 <=> (exists w.  (b-3*w) = 0))
==>
( (exists y.  ((a+x-3*y) = 0)) <=> (exists z.  ((b+x-3*z) = 0) ) ) >>;;
- :  fol formula = <<false>>
```

**Figure 18** Output of Cooper ML implementation proving the failure of the inductive claim in this example

From the above example we derive the following lemma:

**Lemma 6.** *Under the premises of lemma 5, if we have:*

$$(*) \quad g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1}))$$

*and in addition, $g$ is injective or $g'$ is injective, then we can prove $(*)$ by induction.*

**Proof.** Suppose w.l.o.g $g$ is injective. Then we attempt to prove the following by induction, which is equivalent to $(*)$:

$$(**) \quad [\varphi_0]_{init_0,f_0} = g^{-1}(g'([\varphi_1]_{init_1,f_1}))$$

**Case 1: induction base not satisfied:** Assume $init_0 \neq g^{-1}(g(init_1))$. We take $R_0, R_1$ to be empty bags. Then $[\varphi_j]_{init_j,f_j} = init_j$ for $j \in \{0,1\}$, and from $(**)$, we get: $init_0 = g^{-1}(g(init_1))$, contradiction.

**Case 2: induction step not satisfied:** Assume $init_0 = g^{-1}(g(init_1))$, and:

$$\exists \vec{v}, A_{\varphi_0}, A_{\varphi_1}.A_{\varphi_0} = g^{-1}(g(A_{\varphi_1})) \wedge f_0(A_{\varphi_0}, \varphi_0[FV(\varphi_0) \mapsto \vec{v}]) \neq g^{-1}(g'(f_1(A_{\varphi_1}, \varphi_1[FV(\varphi_1) \mapsto \vec{v}]))$$

Let the sequence of valuations $\langle \vec{a_1}, \ldots, \vec{a_n} \rangle$ be the generators of the intermediate values $A_{\varphi_0}, A_{\varphi_1}$. We take RDDs $R_0, R_1$ defined as follows for $j \in \{0,1\}$:

$$R_j = \bigcup_{i=1,\ldots,n} \{\!\{\varphi_j[FV(\varphi_j) \mapsto \vec{a_i}]\}\!\} \cup \{\!\{\varphi_j[FV(\varphi_j) \mapsto \vec{v}]\}\!\}$$

For this choice of RDDs we have:

$$[\varphi_j]_{init_j,f_j} = f_j(A_{\varphi_j}, \varphi_j[FV(\varphi_j) \mapsto \vec{v}])$$

But, from the assumption:

$$f_0(A_{\varphi_0}, \varphi_0[FV(\varphi_0) \mapsto \vec{v}]) \neq g^{-1}(g'(f_1(A_{\varphi_1}, \varphi_1[FV(\varphi_1) \mapsto \vec{v}]))$$

we get :

$$[\varphi_0]_{init_0,f_0} \neq g^{-1}(g'([\varphi_1]_{init_1,f_1}))$$

contradiction.

◀

Lemma 6 shows that the inductive process can be applied on injective mappings of folded expressions in order to prove their equivalence.

SG: Not having injective functions leads to a discussion of finding inductive invariants, removing it for now

### 6.1.3 Induction on self cartesian products

As mentioned previously, self cartesian products, or more precisely, programs where a cartesian products has a non-empty intersection of the sets input RDD variables appearing in each of the tuple elements, are excluded. The following example shows why this exclusion was made:

▸ Example 6.6 (Aggregate on self join/cartesian - failure of lemma 5). We want to calculate $2n^2$ where $n$ is the number of ones (1's) in an RDD.

| | P1($R : RDD_{\texttt{Int}}$): | P2($R : RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | $O = \texttt{filter}(\lambda x.x = 1)(R)$ | $O = \texttt{filter}(\lambda x.x = 1)(R)$ |
| 2 | $C = \texttt{cartesian}(O, O)$ | $O2 = \texttt{map}(\lambda x.2 * x)(O)$ |
| 3 | $C' = \texttt{map}(\lambda(x, y).x + y)(C)$ | $C2 = \texttt{cartesian}(O2, O2)$ |
| 4 | $\texttt{return fold}(0, \lambda A, x.A + x)(C')$ | $\texttt{return fold}(0, \lambda A, (x, y).A + y)(C2)$ |

For $P1, P2$:

$$\phi(O) = \begin{cases} \mathbf{x}_R & \mathbf{x}_R = 1 \\ \perp & otherwise \end{cases}$$

For $P1$:

$$\phi_{P1}(C) = (\phi_{P1}(O), \phi_{P1}(O))$$

$$\phi_{P1}(C') = \phi_{P1}(O) + \phi_{P1}(O) = \begin{cases} 1 & \mathbf{x}_R^{(1)} = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & \mathbf{x}_R^{(2)} = 1 \\ \perp & otherwise \end{cases}$$

$$\begin{aligned} \phi_{P1} &= [\phi_{P1}(C')]_{0, \lambda A, x.A+x} \\ &= \left[ \begin{cases} 1 & \mathbf{x}_R^{(1)} = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & \mathbf{x}_R^{(2)} = 1 \\ \perp & otherwise \end{cases} \right]_{0, \lambda A, x.A+x} \end{aligned}$$

For $P2$:

$$\phi_{P2}(O2) = \begin{cases} 2 * \mathbf{x}_R^{(1)} & \mathbf{x}_R^{(1)} = 1 \\ \perp & otherwise \end{cases}$$

$$\phi_{P2}(C2) = (\phi_{P2}(O2), \phi_{P2}(O2))$$

$$\begin{aligned} \phi_{P2} &= [\phi_{P2}(C2)]_{0, \lambda A, (x,y).A+y} \\ &= \left[ \left( \begin{cases} 2 & \mathbf{x}_R^{(1)} = 1 \\ \perp & otherwise \end{cases}, \begin{cases} 2 & \mathbf{x}_R^{(2)} = 1 \\ \perp & otherwise \end{cases} \right) \right]_{0, \lambda A, (x,y).A+y} \end{aligned}$$

Proceeding with the application of lemma 5: Let there be $x, y. \mathbf{x}_R^{(1)} = x, \mathbf{x}_R^{(2)} = y$ and intermediate values $A, A'$, such that $A = A'$. Need to prove:

$$A + \begin{cases} 1 & x = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & y = 1 \\ \perp & otherwise \end{cases} = A' + \begin{cases} 2 & y = 1 \\ \perp & otherwise \end{cases}$$

If $x = 1, y = 1$, we get: $A + 1 + 1 = A' + 2$ which is true as induction assumption gives $A = A'$. If $x \neq 1$, then the fold function receives a $\perp$ and regards it as a neutral element, resulting in $A + 1 = A' + 2$, contradiction.

**Handling self products** The solution to the self-product problem is *multiple step inductions*. That is, we apply the induction step not on one element from the sequence, but on two or more elements. Furthermore, the elements are not chosen arbitrarily. We take advantage of the well-definition of *fold* operator and choose at once a sequence of elements which all belong to the same *symmetry class* in the complex RDD cartesian product with at least one variable appearing in both elements of the product. We start with illustrating the symmetry class for various RDDs:

1. $R \times R$ - symmetry classes are the singletons $\{(x,x)\}$ $(x \in R)$ (the diagonal of the product matrix), and $\{(x,y),(y,x)\}$ for $x,y \in R, x \neq y$.
2. $R \times R \times R$ - symmetry classes are the singletons $\{(x,x,x)\}$ $(x \in R)$, as well as the sets $\{(x,x,y),(x,y,x),(y,x,x),(x,y,y),(y,x,y),(y,y,x)\}$ for $x,y \in R, x \neq y$, and $\{(x,y,z),(x,z,y),(y,x,z),(y,z,x),(z,x,y),(z,y,x)\}$ for $x,y,z \in R, x \neq y \neq z$.
3. $R \times R \times R'$ - symmetry classes are similar to the first case: $\{(x,x,z)\}$ for $x \in R, z \in R'$ and $\{(x,y,z),(y,x,z)\}$ for $x,y \in R, x \neq y; z \in R'$
4. $R \times R' \times R \times R'$ - symmetry classes are similar to the previous case: $\{(x,y,x,y)\}$ for $x \in R, y \in R'$ and $\{(x,y,x',y'),(x,y',x',y),(x',y,x,y'),(x','y',x,y)\}$ for $x,x' \in R, x \neq x'; y,y' \in R', y \neq y'$.
5. $S = [\![\texttt{map}]\!](R \times R)(\lambda x,y.x+y)$ - symmetry classes are the same as in the first case. Even though we have a single RDD $S$, there are 2 symmetric valuations to it if the two elements of $R$ are different, and 1 if they are equal.

In general, let there be a set of input RDDs $R_1,\dots,R_n$ and an RDD $A = A_1 \times \cdot \times A_k$. For each $i \in \{1,\dots,k\}$, let $\phi(A_i) = \varphi(R_{i_{j_1}},\dots,R_{i_{j_i}})$, where $i_l \in \{1,\dots,n\}$ for $l \in \{j_1,\dots,j_i\}$. We denote the total number of appearances of an RDD $R_i$ in $A$ as $c_i$. The *symmetry family of $R_i$ in $A$ of size $m$* is the set of all symmetric partial valuations to $A$ such that in the $c_i$ appearances of $R_i$ in $A$ there are $m$ unique variables. Each such symmetric partial valuation is a *symmetry class of $R_i$ in $A$*. In general, the *symmetry family of $R_i$ in $A$* is the union of the above for all $m \in \{1,\dots,c_i\}$. For each symmetry family of size $m$ we want to know what is the cardinality of each symmetry class, because this number will define the number of induction steps required for it. Illustratively, we have $c_i$ cells that need to be filled with $m$ unique values. We choose arbitrary $m$ such values from the RDD $R_i$ and build the symmetry class for it, denote the values $\{x_1,\dots,x_m\}$. The number of times each time $x_j$ value appears is denoted $\beta_j$, and the requirement is that $\sum_{j=1}^{m}\beta_j = c_i$. This is the same as choosing a partitioning of a set of size $c_i$ to $m$ non-empty sets with importance to the order of the sets in the partitioning. This is equal to $m! \cdot S(c_i,m)$ where $S(c_i,m)$ is the *Stirling set number* [] [1]. The closed formula for the size of a symmetry class for a symmetry family of size $m$ of $R$ in $A$ is:

$$SCS(c_i,m) = \sum_{j=0}^{m}(-1)^j\binom{m}{j}(m-j)^{c_i}$$

Given some concrete valuation to all input RDDs, we can calculate for all $i \in \{1,\dots,n\}$ the number $m_i$, which is the number of unique values chosen for the $c_i$ instances of an RDD $R_i$. The number of induction steps required is the product of the symmetry class sizes for all RDDs:

$$\Pi_{i=1}^{n} SCS(c_i,m_i)$$

We now calculate it formally for the above examples:

[1] The *Stirling set number* $S(n,m)$ is the number of ways to partition a set of $n$ elements to $m$ non empty sets, and $m!$ is the number of possible orderings of the chosen sets.

1. For a valuation $(x,x)$ we get $SCS(2,1) = \binom{1}{0}1^2 - \binom{1}{1}0^2 = 1$ as expected[2].
   For a valuation $(x,y), x \neq y$ we get $SCS(2,2) = \binom{2}{0}2^2 - \binom{2}{1}1^2 + \binom{2}{2}0^2 = 4-2 = 2$ as expected.
2. For a valuation $(x,x,x)$ we get $SCS(3,1) = 1$ as expected.
   For a valuation $(x,x,y), x \neq y$ we get $SCS(3,2) = \binom{2}{0}2^3 - \binom{2}{1}1^3 + 0 = 8-2 = 6$ as expected.
   For a valuation $(x,y,z), x \neq y \neq z$ we get $SCS(3,3) = \binom{3}{0}3^3 - \binom{3}{1}2^3 + \binom{3}{2}1^3 - 0 = 27-24+3 = 6$, as expected.
3. For a valuation $(x_r, x_r, x_{r'})$ we get $SCS(2,1)SCS(1,1) = 1$ as expected.
   For a valuation $(x_r, y_r, x_{r'}), x_r \neq y_r$ we get $SCS(2,2)SCS(1,1) = 2$ as expected.
4. For a valuation $(x_r, x_{r'}, x_r, x_{r'})$ we get $SCS(2,1)SCS(2,1) = 1$ as expected.
   For a valuation $(x_r, x_{r'}, y_r, y_{r'}), x_r \neq y_r, x_{r'} \neq y_{r'}$ we get $SCS(2,2)SCS(2,2) = 4$ as expected.
5. Same as item 1.

**Completing the proof of example 6.6** Reminder: we had $\mathbf{x}_R^{(1)} = x, \mathbf{x}_R^{(2)} = y$ and intermediate values $A, A'$, such that $A = A'$, and needed to prove:

$$A + \begin{cases} 1 & x = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & y = 1 \\ \perp & otherwise \end{cases} = A' + \begin{cases} 2 & y = 1 \\ \perp & otherwise \end{cases}$$

We handle cases according to the number of choices to pick unique elements in the cartesian product (there are 2 such cases):

1. For $x = y$, we get that we need to prove:
   $$A + \begin{cases} 1 & x = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & x = 1 \\ \perp & otherwise \end{cases} = A' + \begin{cases} 2 & x = 1 \\ \perp & otherwise \end{cases}$$
   which is immediate.
2. For $x \neq y$, we prove for 2 possible valuations: $\mathbf{x}_R^{(1)} = x, \mathbf{x}_R^{(2)} = y$ and $\mathbf{x}_R^{(1)} = y, \mathbf{x}_R^{(2)} = x$. We get that we need to prove:
   $$\left(A + \begin{cases} 1 & x = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & y = 1 \\ \perp & otherwise \end{cases}\right) + \begin{cases} 1 & y = 1 \\ \perp & otherwise \end{cases} + \begin{cases} 1 & x = 1 \\ \perp & otherwise \end{cases} = \left(A' + \begin{cases} 2 & y = 1 \\ \perp & otherwise \end{cases}\right) + \begin{cases} 2 & x = 1 \\ \perp & otherwise \end{cases}$$
   which is indeed true, in all 3 possible cases where $x \neq y$: (1) $x = 1, y \neq 1$, or (2) $y = 1, x \neq 1$ or (3) $x \neq 1, y \neq 1, x \neq y$.

▸ **Lemma 7.** *Under the premises of lemma 5, where we allow self cartesian products too. For brevity, we denote the assignment of $\vec{v}$ to the free variables of $\varphi_j$ as $\psi_j(\vec{v}) = \varphi_j[FV(\varphi_j) \mapsto \vec{v}]$. We denote $SF(n)$ as the symmetry family of both $R_0, R_1$ (true because $\mathcal{M}(R_0) = \mathcal{M}(R_1)$) of size $n$. In addition, $\sigma_i(\vec{v})$ will return the i'th permutation of the vector $\vec{v} \in SF(n)$ in its symmetry class, which is known to be of size $SCS(\mathcal{M}(R_0), n)$ - so $i \in \{1, \dots, SCS(\mathcal{M}(R_0), n)\}$. We set $\sigma_1(\vec{v}) = \vec{v}$ for all $\vec{v}$. We have $g([\varphi_0]_{init_0, f_0}) = g'([\varphi_1]_{init_1, f_1})$ if:*

$$g(init_0) = g'(init_1) \tag{1}$$

[2] $\forall n. SCS(n,1) = 1$

$$\forall \vec{v}, A_{\varphi_0} : \xi_0, A_{\varphi_1} : \xi_1. \qquad g(A_{\varphi_0}) = g'(A_{\varphi_1}) \implies$$
$$\begin{cases} g(f_0(A_{\varphi_0}, \psi_0(\vec{v}))) = g'(f_1(A_{\varphi_1}, \psi_1(\vec{v}))) & \vec{v} \in SF(1) \\ g(f_0(f_0(A_{\varphi_0}, \psi_0(\vec{v}))), \psi_0(\sigma_2(\vec{v}))) = g'(f_1(f_1(A_{\varphi_1}, \psi_1(\vec{v}))), \psi_1(\sigma_2(\vec{v}))) & \vec{v} \in SF(2) \\ \qquad \ldots & \\ \begin{array}{l} g(f_0(\ldots(f_0(A_{\varphi_0}), \psi_0(\sigma_1(\vec{v})))\ldots), \sigma_{SCS(\mathcal{M}(R_0),n)}(\vec{v}))) \quad = \\ g'(f_1(\ldots(f_1(A_{\varphi_1}), \psi_1(\sigma_1(\vec{v})))\ldots), \sigma_{SCS(\mathcal{M}(R_1),n)}(\vec{v}))) \end{array} & \vec{v} \in SF(n) \end{cases} \tag{2}$$

The lemma above shows that the process described in this section allows sound and complete procedure for verifying $PE$, in the presence of self-cartesian-products.

## 6.2 Multiple aggregates

Another relatively simple case of $SPARK$ programs is when the program contains multiple aggregate operations, but they are independent of each other. Namely, the result of one aggregate operations is not used in the other one, and a final result can be proven by a set of formulas, each of which is dependent only on one aggregated result from each of the tested programs.

▸ Example 6.7 (Independent fold). Below are 2 programs which return a tuple containing the sum of positive elements in its first element, and the sum of negative elements in the second element. We show that by applying lemma 7 on each element of the resulting tuple, we are able to show the equivalency.

We define a fold function:

$$\text{Let:} \quad h : (\lambda(P, N), x. \begin{cases} (P + x, N) & x \geq 0 \\ (P, N - x) & otherwise \end{cases}$$

| P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|
| **return** $\texttt{fold}((0,0), h)(R)$ | $R_P = \texttt{filter}(\lambda x.x \geq 0)(R)$ |
| | $R_N = \texttt{map}(\lambda x. - x)(\texttt{filter}(\lambda x.x < 0)(R))$ |
| | $p = \texttt{fold}(0, \lambda A, x.A + x)(R_P)$ |
| | $n = -\texttt{fold}(0, \lambda A, x.A + x)(R_N)$ |
| | **return** $(p, n)$ |

w.l.o.g. we show the application of lemma 7 to second element of the tuple. First,

$$\phi_{P2}(R_N) = \begin{cases} -\mathbf{x}_R & \mathbf{x}_R < 0 \\ \bot & otherwise \end{cases}$$

We let $g = \lambda x.x$ and $g' = \lambda x. - x$. Induction base case is obvious. Induction step:

$$\forall x, (P, N), N'.N' = N \implies p_2(h((P, N), x)) = N' + \begin{cases} -x & x < 0 \\ \bot & otherwise \end{cases}$$

Substituting for $p_2 \circ h$, we get that we need to prove:

$$\begin{aligned}
\begin{cases} N & x \geq 0 \\ N - x & otherwise(x < 0) \end{cases} &=^{?} N' + \begin{cases} -x & x < 0 \\ \bot & otherwise \end{cases} \\
&=_{\bot \texttt{ is neutral}} \begin{cases} N' - x & x < 0 \\ N' & otherwise \end{cases} \\
&=_{N=N'} \begin{cases} N - x & x < 0 \\ N & otherwise \end{cases} \\
&= \begin{cases} N - x & x < 0 \\ N & x \geq 0 \end{cases}
\end{aligned}$$

And the equivalence follows. ∎

▸ **Lemma 8.** *Let $R_1 \in RDD_{\sigma_1}, \ldots, R_k \in RDD_{\sigma_k}$, and denote the representative elements $\varphi_i$ for $i \in \{1, \ldots, k\}$. We assume the $\varphi_i$ are based only on* map *and* filter *operations (*the extension to self cartesian products is implemented naturally on each RDD $R_i$ on which a fold is applied separately*). Let there be $k$* fold *UDFs $f_i : \xi_i \times \sigma_i \to \xi_i$, and $k$ initial values $init_i$:$\xi_i$. Let a similar set of RDDs, representative elements,* fold *UDFs and initial values, with all denotions having a $'$. Let there be 2 functions $g$:$\xi_1 \times \cdots \times \xi_k \to \xi, g'$:$\xi'_i \times \cdots \times \xi'_{k'} \to \xi$. We denote a vector $\vec{v} = FV(\varphi_1, \ldots, \varphi_k, \varphi'_1, \ldots, \varphi'_{k'})$ of the free variables in all representative elements. We have: if*

$$g(init_1, \ldots, init_k) = g'(init'_1, \ldots, init'_{k'}) \tag{1}$$

$$\begin{gathered}
\forall \vec{v}, A_{\varphi_1}:\xi_1, \ldots, A_{\varphi_k}:\xi_k, A_{\varphi'_1}:\xi'_1, \ldots, A_{\varphi'_{k'}}:\xi'_{k'}.g(A_{\varphi_1}, \ldots, A_{\varphi_k}) = g'(A_{\varphi'_1}, \ldots, A_{\varphi'_{k'}}) \implies \\
g(f_1(A_{\varphi_1}, \varphi_1[FV(\varphi_1) \mapsto \vec{v}]), \ldots, f_k(A_{\varphi_k}, \varphi_k[FV(\varphi_k) \mapsto \vec{v}])) = \\
g'(f'_1(A_{\varphi'_1}, \varphi'_1[FV(\varphi'_1) \mapsto \vec{v}]), \ldots, f'_k(A_{\varphi'_{k'}}, \varphi'_{k'}[FV(\varphi'_{k'}) \mapsto \vec{v}]))
\end{gathered} \tag{2}$$

*then* $g([\varphi_1]_{init_1, f_1}, \ldots, [\varphi_k]_{init_k, f_k}) = g'([\varphi'_1]_{init'_1, f'_1}, \ldots, [\varphi'_{k'}]_{init'_{k'}, f'_{k'}})$

**Proof.** SG: natural extension of the previous lemma. The difference is that we choose $k + k'$ intermediate values in each step.

◂

## 6.3 A class for which *PE* is undecidabile

We show a reduction of Hilbert's 10'th problem to *PE*. We assume towards a contradiction that *PE* is decidable under the premises of lemma 8 with represntative elements based also on the *cartesian* operation. Let there be a polynom $p$ over $k$ variables $x_1, \ldots, x_k$, and coefficients $a_1, \ldots, a_k$. For each variable $x_i$ we assume the existence of some RDD $R_i$ with $x_i$ elements. We use *SPARK* operations and the input RDDs $R_i$ to represent the value of the polynom $P$ for some valuation of the $x_i$. For each summand in the polynom $p$, we define a translation $\varphi$:

- $x_i \longrightarrow^{\varphi} [\texttt{map}(\lambda x.1)(R_i)]_{0,+}$
- $x_i x_j \longrightarrow^{\varphi} [\texttt{cartesian}(\texttt{map}(\lambda x.1)(R_i), \texttt{map}(\lambda x.1)(R_j))]_{0,+}$
- By induction, $x_i^2 \longrightarrow^{\varphi} [\texttt{cartesian}(\texttt{map}(\lambda x.1)(R_i), \texttt{map}(\lambda x.1)(R_i))]_{0,+}$.
  This rule as well as the rest of the powers follow according to the previous rule. For a degree $k$ monom, we apply the *cartesian* operation $k$ times.

- ${x_i}^0 \longrightarrow^{\varphi} 1$, trivially.
- $am(x) \longrightarrow^{\varphi} a\varphi(m(x))$ where $m(x)$ is a monomial with coeffeicient 1 of the variable $x$, thus we have already defined $\varphi$ for it.
- $aq(x_{i_1}, \ldots, x_{i_j}) \longrightarrow^{\varphi} a\varphi(q(x_{i_1}, \ldots, x_{i_j}))$ where $q(x)$ is a monomial with coefficient 1 and multiple variables for which $\varphi$ was defined in the previous rules.
- $\varphi(p(a_1, \ldots, a_k; x_1, \ldots, x_k)) = \sum_{i=1}^{k} \varphi(a_i q(x_{i_1}, \ldots, x_{i_k})))$ follows by structural induction on the previous rules.

We generate the following instance of the *PE* problem:

| | P1($R_1, \ldots, R_k$: $RDD_{\texttt{Int}}$): | P2($R_1, \ldots, R_k$: $RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | `return` $\varphi(p) \neq 0$ | `return` $tt$ |

By choosing input RDDs of the cardianlity of $R_i$ equal to the matching variable $x_i$ we can simulate any valuation to the polynom $p$. If $P1$ returns true, then the valuation is not a root of the polynom $p$. Thus, if it is equivalent to the 'true program' $P2$, then the polynom $p$ has no roots. Therefore, if the algorithm solving *PE* outputs 'equivalent' then the polynom $p$ has no root, and if it outputs 'not equivalent' then the polynom $p$ has some root, where $x_i = |[\![R_i]\!]|$. Thus we have polynomial reduction to Hilbert's 10'th problem.

## 6.4 Aggregated values on Union

SG: Skipping right now

▸ **Lemma 9.** *Let $R_1 \in RDD_\sigma, R_2 \in RDD_\sigma, R' \in RDD_{\sigma'}$, and denote the representative elements $\varphi_1, \varphi_2, \varphi'$. Let there be* fold *UDFs $f : \xi \times \sigma \to \xi, f' : \xi' \times \sigma' \to \xi'$, and initial values $init{:}\xi, init'{:}\xi'$. Let there be 2 functions $g{:}\xi \to \Xi, g'{:}\xi' \to \Xi$. We denote a vector $\vec{v} = FV(\varphi_1, \varphi')$ of the free variables in all representative elements. Suppose we have a mapping between the RDDs $R_1, R_2$: $s : \sigma \to \sigma$, such that if w.l.o.g. $|R_1| > |R_2|$, then $\forall x \in R_1, s(x) \in R_2 \cup \{\perp\}$, and $s$ is onto $R_2$ and injective on $R_2$'s preimage. We have: if*

$$g(init) = g'(init') \tag{1}$$

$$\begin{gathered}\forall \vec{v}, A{:}\xi, A'{:}\xi'.g(A) = g'(A') \implies \\ g(f(f(A, \varphi_1[\varphi_1 \mapsto \vec{v}]), s(\varphi_2[\varphi_2 \mapsto \vec{v}]))) = g'(f'(A', \varphi'[FV(\varphi') \mapsto \vec{v}]))\end{gathered} \tag{2}$$

*then $g([\varphi_1 \cup \varphi_2]_{init,f}) = g'([\varphi']_{init',f'})$*

▸ **Lemma 10.** *Let $R_1 \in RDD_\sigma, R_2 \in RDD_\sigma, R_1' \in RDD_{\sigma'}, R_2' \in RDD_{\sigma'}$, and denote the representative elements $\varphi_1, \varphi_2, \varphi_1', \varphi_2'$. Let there be* fold *UDFs $f : \xi \times \sigma \to \xi, f' : \xi' \times \sigma' \to \xi'$, and initial values $init{:}\xi, init'{:}\xi'$. Let there be 2 functions $g{:}\xi \to \Xi, g'{:}\xi' \to \Xi$. We denote a vector $\vec{v} = FV(\varphi_1, \varphi_1')$ of the free variables in all representative elements. We have: if*

$$g(init) = g'(init') \tag{1}$$

$$\begin{gathered}\forall \vec{v}, A{:}\xi, A'{:}\xi'.g(A) = g'(A') \implies \\ g(f(f(A, \varphi_1[\varphi_1 \mapsto \vec{v}]), \varphi_2[\varphi_2 \mapsto \vec{v}])) = g'(f'(f'(A', \varphi_1'[FV(\varphi_1') \mapsto \vec{v}]), \varphi_2'[FV(\varphi_2') \mapsto \vec{v}]))\end{gathered} \tag{2}$$

*then $g([\varphi_1 \cup \varphi_2]_{init,f}) = g'([\varphi_1' \cup \varphi_2']_{init',f'})$*

▸ Example 6.8 (Aggregate with union). Self union and summation, is like doubling elements and summation:

Let: $sum = \lambda A, y.A + y$

| | P1($R : RDD_{\texttt{Int}}$): | P2($R : RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | **return** $\texttt{fold}(0, sum)(\texttt{union}(R, R))$ | **return** $\texttt{fold}(0, sum)(\texttt{map}(\lambda x.2 * x)(R))$ |

We need to prove:

$$[(\mathbf{x}_R^{(1)} \cup \mathbf{x}_R^{(2)})]_{0,sum} = [2 * \mathbf{x}_R]_{0.sum}$$

To apply lemma 5 on the union, we need to apply the fold function for it twice, one for each RDD participating in the union. We take $s : \texttt{Int} \to \texttt{Int}$ to be the identity function. Then, $\mathbf{x}_{R'} \mapsto^s \mathbf{x}_R$. We get that for ranging $x$ over $\mathbf{x}_R$ we need to prove by lemma 9:

$$\forall x, A, A'.A = A' \implies (A + x) + x = A' + 2 * x$$

which is immediate.

▸ Example 6.9 (Aggregate on empty union). Self union and summation, is like doubling elements and summation:

Let: $count = \lambda A, y.A + 1$

| | P1($R : RDD_{\texttt{Int}}$): | P2($R : RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | **return** $\texttt{fold}(0, count)(R)$ | $R' = \texttt{filter}(\lambda x.0 = 1)(R)$ |
| 2 | | **return** $\texttt{fold}(0, count)(\texttt{union}(R, R'))$ |

We have $\phi_{P1}(r^o) = [\mathbf{x}_R]_{0,count}$. For $P2$, $\phi_{P2}(R') = \bot$ for any valuation to $\mathbf{x}_R$. Therefore: $\phi_{P2}(r^o) = [(\mathbf{x}_R \cup \mathbf{x}_{R'})]_{0,count}$

And by applying lemma 9, any mapping $s$ just maps to $\bot$, so we get that we need to prove:

$$\forall x, A, A'.A = A' \implies count(A, x) = count(count(A, x), \bot)$$

As $count$ is naturally extended to be constant on $\bot$, the implication is equivalent to:

$$count(A, x) = count(A, x)$$

and the equivalence follows. ■

▸ **Lemma 11.** *We can extend lemma 8 with the* union *operation, using the method described in lemmas 9,10.*

▸ Example 6.10 (Sum over union). Summing the elements of two RDDs separately in $P2$ and of the union in $P1$:

Let: $sum = \lambda A, y.A + y$

| | P1($R_0 : RDD_{\texttt{Int}}, R_1 : RDD_{\texttt{Int}}$): | P2($R_0 : RDD_{\texttt{Int}}, R_1 : RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | **return** $\texttt{fold}(0, sum)(\texttt{union}(R_0, R_1))$ | **return** $\texttt{fold}(0, sum)(R_0) + \texttt{fold}(0, sum)(R_1)$ |

To prove equivalence, we need $[(\mathbf{x}_{R_0} \cup \mathbf{x}_{R_1})]_{0,sum} = [\mathbf{x}_{R_0}]_{0,sum} + [\mathbf{x}_{R_1}]_{0,sum}$. We apply Lemma 11. Base case is trivial. Step: Assume $A = B + C$. We need to prove:

$$\forall x, y, A, B, C.(A + x) + y = B + x + C + y$$

▸ Example 6.11 (Different number of unions). This example shows aggreagte on unions with different cardinalities can be proven equal.

| P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|
| $R' = \lambda x.2 * x$ | $R' = \lambda x.3 * x$ |
| return $\texttt{fold}(0,+)(\texttt{union}(\texttt{union}(R',R'),R'))$ | return $\texttt{fold}(0,+)(\texttt{union}(R',R'))$ |

Need to prove:

$$[((2 * \mathbf{x}_R^{(1)} \cup 2 * \mathbf{x}_R^{(2)}) \cup 2 * \mathbf{x}_R^{(3)})]_{0,+} = [(3 * \mathbf{x}_R^{(1)} \cup 3 * \mathbf{x}_R^{(2)})]_{0,+}$$

We apply lemma 10 , we choose $\mathbf{x}_R^{(2)} = \mathbf{x}_R^{(1)}, \mathbf{x}_R^{(3)} = \mathbf{x}_R^{(1)}$.
We get:

$$\forall x, A, A'.A = A' \implies ((A + 2x) + 2x) + 2x = (A' + 3x) + 3x$$

need another generalization! probably normal form where all unions are visible

▸ **Theorem 12** (Basic decidability for *PEMT* ). *For two SPARK programs $Q_1, Q_2$ returning a single or a tuple of RDDs which do not depend on the result of a* subtract *operation, and for two SPARK programs $T_1, T_2$ which act on the output of $Q_1, Q_2$ respectively by applying fold functions on them,* PEMT *problem is decidable if either $T_1$ or $T_2$ apply an injective transform on the folded expressions.*

and union - remove todo after formalizing for union

another problem is that in *PEMT* we assume the output is single and not a tuple, and we do not have RDD tuples formally

**Proof.** A conclusion from lemmas 5, 6, 7, 8 ◂

▸ **Corollary 13.** *For two SPARK programs $P_1, P_2$ which can be expressed by SPARK programs $Q_1, Q_2, T_1, T_2$ satisfying $P_i = T_i \circ Q_i$ and the premises of theorem 12,* PE *is decidable.*

## 6.5 Nested aggregations

We saw in 12 a proof of decidability for a fragment of *SPARK* programs. In the following subsection we present more complex *SPARK* programs on which the theorem applies, the method for proving the equivalence, and the cases on which it fails. Those programs have a value of an aggregate operation used in later aggregations (i.e. 'nested' aggregations). We see that the inductive method is sound in handling those cases, and that under certain conditions, it is complete too.

▸ Example 6.12 (Conditional summation). The following example takes the *sum* of all elements which are greater than the *count* of elements in an RDD.

$$\text{Let:} \quad \begin{array}{l} f : (\lambda A, (a,b).A + b) \\ +: \lambda A, x.A + x \end{array}$$

| P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|
| $R' = \texttt{map}(\lambda x.(x,2))(R)$ | $R' = \texttt{map}(\lambda x.(x,1))(R)$ |
| $sz = \texttt{fold}(0,f)(R')$ | $sz = \texttt{fold}(0,f)(R')$ |
| $B = \texttt{filter}(\lambda x.x > sz)(R)$ | $B = \texttt{filter}(\lambda x.x > 2 * sz)(R)$ |
| return $\texttt{fold}(0,+)(B)$ | return $\texttt{fold}(0,+)(B)$ |

The equivalence condition is:

$$\left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R > \phi_{P1}(sz) \\ \perp & \text{otherwise} \end{cases}\right]_{0,+} = \left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R > 2 * \phi_{P2}(sz) \\ \perp & \text{otherwise} \end{cases}\right]_{0,+}$$

Replacing $sz, sz'$ we get:

$$\left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R > [(\mathbf{x}_R^{(1)}, 2)]_{0,f} \\ \perp & \text{otherwise} \end{cases}\right]_{0,+} = \left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R > 2 * [(\mathbf{x}_R^{(1)}, 1)]_{0,f} \\ \perp & \text{otherwise} \end{cases}\right]_{0,+}$$

After formally applying lemma 5 we get that the above is equivalent if and only if $\phi_{P1}(sz) = 2 * \phi_{P2}(sz)$, that is:

$$[(\mathbf{x}_R^{(1)}, 2)]_{0,f} = 2 * [(\mathbf{x}_R^{(1)}, 1)]_{0,f}$$

In this case, we set $g = \lambda x.x, g' = \lambda x.2 * x$, and get:

$$0 = g(0) = 2 * 0 \tag{1}$$

$$\forall x, A, A'.A = 2 * A' \implies f(A,(x,2)) \begin{aligned}[t] &= A + 2 \\ &= 2 * A' + 2 \\ &= 2 * (A' + 1) \\ &= f(A',(x,1)) \end{aligned} \tag{2}$$

Proving the equivalence. ∎

This property is reflected in the following proposition:

▸ Proposition 6.1. Let there be two $SPARK$ programs $P_1, P_2$ and returning aggregated expressions $[a_i(\vec{r})]_{init_i,f_i}$. Let there be two other $SPARK$ programs, $T_1, T_2$, also returning aggregated expressions, $[b_i(\vec{r'}, a)]_{init_T,g}$ (the aggregation function is equal in both $T_1, T_2$). $PEMT$ is decidable for $P, T$ and $P', T'$ if and only if $PE$ is decidable for $P, P'$.

**Proof.** $\phi_{P_1} = [a_1(\vec{r})]_{init_1,f_1}, \phi_{P_2} = [a_2(\vec{r})]_{init_2,f_2}$ are the program terms of $P_1, P_2$, respectively. $a, a'$ are terms without aggregations. Let $[b_i(\vec{r'}, \phi_{P_i})]_{init_T,g}$ for $i \in \{1,2\}$ be the program terms for $T_1 \circ P_1, T_2 \circ P_2$, respectively. The programs are equivalent if and only if the program terms are equivalent. As we have the same aggregation function on both representations, we can check the equivalence of $b_1(\vec{r'}, [a_1(\vec{r})]_{init_1,f_1}))$ and $b_2(\vec{r'}, [a_2(\vec{r})]_{init_2,f_2})$ instead. The decidability or undecidability is determined by corollary 13. ◂

From proposition 6.1, it can be concluded that representative elements which are an injection as a function of the folded values have a decidable equivalence checking procedure. In the above example, the expression:

$$f(sz) = \lambda x. \begin{cases} x & x > sz \\ \perp & \text{otherwise} \end{cases}$$

is an injective function of $sz$ - each such expression, when choosing a certain $sz$, is a different function of $x$.

This allows us to define an algorithm for verifying complex equivalences with aggregated queries. The idea is to apply nested inductive proofs (on the nested expressions) during the proof of the induction step of the outer aggregations.

▸ **Theorem 14.** *Let there be two $SPARK$ programs $P_1, P_2$, returning an expression of the form: $f_i([\phi_i(\vec{x}, a_i(\vec{y}))]_{init_i,g_i})$, The* PE *problem is decidable if exists $i \in \{1,2\}$ such that the following expressions are all injective:*

1. $f_i$
2. $\mathcal{F} = \lambda A, \vec{x}. f_i(g_i(A_i, \phi_i(\vec{x}, a_i(\vec{y}))))$

**Proof.** We apply lemma 5 on the expressions returned by $P_i$:

$$f_1(init_1) = f_2(init_2) \tag{1}$$

$$\forall\vec{x}, A_1, A_2.f_1(A_1) = f_2(A_2) \implies f_1(g_1(A_1, \phi_1(\vec{x}, a_1(\vec{y})))) = f_2(g_2(A_2, \phi_2(\vec{x}, a_2(\vec{y})))) \tag{2}$$

$a_1, a_2$ are aggregated expressions: $a_i = [\psi_i(\vec{y})]_{j_i,h_i}$. So we apply lemma 5 again. For brevity, we denote: $\mathcal{F}' = f_i(g_i(A_i, \phi_i(\vec{x}, a_i(\vec{y})))) = \mathcal{F}(A_i, \vec{x})$

$$\mathcal{F}'(j_1) = \mathcal{F}'(j_2) \tag{1}$$

$$\forall\vec{y}, B_1, B_2.\mathcal{F}'(B_1) = \mathcal{F}'(B_2) \implies \mathcal{F}'(h_1(B_1, \psi_1(\vec{y}))) = \mathcal{F}'(h_2(B_2, \psi_2(\vec{y}))) \tag{2}$$

As we know that for at least one of the sides we have injective functions for both applications of lemma 5, then by corollary 13 we have a decision procedure for the equivalence.

SG: is 'only if' also true here? can't prove right now

◂

Therefore, by using the conditions determined by theorem 14, there is no need to assume the outer aggregation function is equal, as stated in this lemma:

▸ **Lemma 15.** *Let there be two SPARK programs $P_1, P_2$ and returning aggregated expressions $[a_i(\vec{r})]_{init_i,f_i}$. Let there be two other SPARK programs, $T_1, T_2$, also returning aggregated expressions, $[b_i(\vec{r'}, a)]_{init_{T_i},g_i}$.* PEMT *is decidable for $P, T$ and $P', T'$ if and only if* PE *is decidable for $P, P'$.*

**Proof.** $\phi_{P_1} = [a_1(\vec{r})]_{init_1,f_1}, \phi_{P_2} = [a_2(\vec{r})]_{init_2,f_2}$ are the program terms of $P_1, P_2$, respectively. $a, a'$ are terms without aggregations. Let $[b_i(\vec{r'}, \phi_{P_i})]_{init_{T_i},g_i}$ for $i \in \{1, 2\}$ be the program terms for $T_1 \circ P_1, T_2 \circ P_2$, respectively. The programs are equivalent if and only if the program terms are equivalent. By applying lemma 5 on the outer aggregation, we need to check:

$$g_1(init_{T_1}) = g_2(init_{T_2}) \tag{1}$$

$$\begin{aligned}&\forall\vec{x'}, A_1, A_2.g_1(A) = g_2(A_2) \implies \\ &g_1(f_1(A_1, b_1(\vec{r'}, \phi_P)[\vec{r'} \mapsto \vec{x'}])) = g_2(f_2(A_2, b_2(\vec{r'}, \phi_{P'})[\vec{r'} \mapsto \vec{x'}]))\end{aligned} \tag{2}$$

Verifying the second equation is decidable under the assumptions of theorem 14. ◂

Below are several examples:

▸ Example 6.13 (Increase all elements by count of elements and count). In this example we add the *count* of the RDD's elements to all elements, and count the elements of the result. Of course, this is the same as simply counting the elements.

Let: $count = \lambda A, x.A + 1$

| | P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|---|
| 1 | $c = \texttt{fold}(0, count)(R)$ | $c = \texttt{fold}(0, count)(R)$ |
| 2 | $R' = \texttt{map}(\lambda c.\lambda x.x + c)(R)$ | $\texttt{return}\ c$ |
| 3 | $\texttt{return fold}(0, count)(R')$ | |

Program term of $P1$: $\phi_{P1} = [\mathbf{x}_R + [\mathbf{x}_R]_{0,count}]_{0,count}$. Program term of $P2$: $\phi_{P2} = [\mathbf{x}_R]_{0,count}$. We apply lemma 15. Induction: base case is trivial. Step: Let $A = A'$. Need to prove $A + 1 = A' + 1$ which is immediate. *count* maps each element which is not $\perp$ to the intermediate value, so the value of $[\mathbf{x}_R]_{0,count}$ does not affect the result. ■

▸ Example 6.14 (Increase all elements by count of elements and sum). If the number of elements in the input RDD is $n$ and their sum is $s$, then the following two programs calculate $s + n * n$.

$$\text{Let:}\quad \begin{aligned} count &= \lambda A, x.A + 1\\ sum &= \lambda A, x.A + x\\ count &= \lambda A, x.A + 1\\ addConst &= \lambda c.\lambda x.x + c\\ replaceWith &= \lambda c.\lambda x.c \end{aligned}$$

| P1(R): | P2(R): |
|---|---|
| $c = \texttt{fold}(0, count)(R)$ | $c = \texttt{fold}(0, count)(R)$ |
| $R' = \texttt{map}(addConst(c))(R)$ | $s = \texttt{fold}(0, sum)(R)$ |
| $\texttt{return fold}(0, sum)(R')$ | $R' = \texttt{map}(replaceWith(c))(R)$ |
| | $\texttt{return fold}(0, sum)(R') + s$ |

Program term of $P1$:

$$\phi_{P1} = [\mathbf{x}_R + [\mathbf{x}_R]_{0,count}]_{0,sum}$$

Program term of $P2$:

$$\phi_{P2} = [[\mathbf{x}_R]_{0,count}]_{0,sum} + [\mathbf{x}_R]_{0,sum}$$

We have both nested aggregations and multiple aggregate terms in $P2$. We apply 2 lemmas: lemma 8 and lemma 15. Base case: 0=0+0. Step: Let $A = B + C$. Need to prove: $A + (\mathbf{x}_R + [\mathbf{x}_R^{(1)}]_{0,count}) = B + [\mathbf{x}_R^{(1)}]_{0,count} + C + \mathbf{x}_R$. We get immediately that $[\mathbf{x}_R^{(1)}]_{0,count}$ cancel out of both sides of the equation, and it is enough to prove $A + \mathbf{x}_R = B + C + \mathbf{x}_R$, which is true by the induction hypothesis. ■

▸ Example 6.15 (Count number of elements bigger than half the maximal element). This example once again shows how sum and count can be equivalent, but the aggregations are applied on an RDD modified according to the result of a previous aggregation.

$$\text{Let:}\quad \begin{aligned} max &= \lambda A, x.ite(A < x, x, A)\\ sum_2 &= \lambda A, (x, y).A + y\\ count &= \lambda A, x.A + 1\\ biggerThanHalfOf &= \lambda m.\lambda x.2 * x > m \end{aligned}$$

| P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|
| $m = \texttt{fold}(\perp, max)(R)$ | $m = \texttt{fold}(\perp, max)(R)$ |
| $R' = \texttt{filter}(biggerThanHalfOf(m))(R)$ | $R' = \texttt{filter}(biggerThanHalfOf(m))(R)$ |
| $R'' = \texttt{map}(\lambda x.(x, 1))(R')$ | $\texttt{return fold}(0, count)(R')$ |
| $\texttt{return fold}(0, sum_2)(R')$ | |

Program term of $P1$:

$$\phi_{P1} = \left[ \begin{cases} (\mathbf{x}_R, 1) & 2 * \mathbf{x}_R > [\mathbf{x}_R^{(1)}]_{\perp,max} \\ \perp & otherwise \end{cases} \right]_{0,sum_2}$$

Program term of $P2$:

$$\phi_{P2} = \left[ \begin{cases} \mathbf{x}_R & 2 * \mathbf{x}_R > [\mathbf{x}_R^{(1)}]_{\perp,max} \\ \perp & otherwise \end{cases} \right]_{0,count}$$

According to theorem 14, we can check the induction base case for the outer aggregation which is trivial: $0 = 0$, for the step we assume $A = A'$, and we need to prove:

$$\begin{cases} A+1 & 2 * \mathbf{x}_R > [\mathbf{x}_R^{(1)}]_{\perp,max} \\ A & otherwise \end{cases} = \begin{cases} A'+1 & 2 * \mathbf{x}_R^{(1)} > [\mathbf{x}_R^{(1)}]_{\perp,max} \\ A' & otherwise \end{cases}$$

Which is straightforward. ■

▸ Example 6.16 (Sum and count polynomial from two RDDs). In this example we use two RDDs.

$$\text{Let:}\quad \begin{aligned} sum &= \lambda A, x.A + x \\ count &= \lambda A, x.A + 1 \\ addConst &= \lambda c.\lambda x.x + c \\ replaceWith &= \lambda c.\lambda x.c \end{aligned}$$

| P1($R_0$: $RDD_{\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}}$): | P2($R_0$: $RDD_{\texttt{Int}}$, $R_1$: $RDD_{\texttt{Int}}$): |
|---|---|
| $c = \texttt{fold}(0, count)(R_0)$ | $c = \texttt{fold}(0, count)(R_0)$ |
| $R' = \texttt{map}(addConst(c))(R_1)$ | $s = \texttt{fold}(0, sum)(R_1)$ |
| $\texttt{return fold}(0, sum)(R')$ | $R' = \texttt{map}(replaceWith(c))(R)$ |
| | $\texttt{return fold}(0, sum)(R') + s$ |

Program term of $P1$:

$$\phi_{P1} = [\mathbf{x}_{R_1} + [\mathbf{x}_{R_0}]_{0,count}]_{0,sum}$$

Program term of $P2$:

$$\phi_{P2} = [[\mathbf{x}_{R_0}]_{0,count}]_{0,sum} + [\mathbf{x}_{R_1}]_{0,sum}$$

We apply theorem 14. Base case: 0=0+0. Step: Let $x_0$ a valuation for $\mathbf{x}_{R_0}$, and $x_1$ a valuation for $\mathbf{x}_{R_1}$. Assume $A = B + C$. Need to prove: $A + (x_1 + [\mathbf{x}_{R_0}]_{0,count}) = B + [\mathbf{x}_{R_0}]_{0,count} + C + x_1$. We get that it is enough to prove $A + x_1 = B + C + x_1$, $x_0$, which is true by the induction hypothesis. $x_0$ is not used. ■

SG: TODO - continue from here

▸ Example 6.17 (Nested and dependent aggregates). The following example is significantly more elaborate than the previous examples, and will fail due to non injective functions.

SG: This example fails because it does not satisfy the conditions of theorem 14. TODO cleanup

SG: This example is important. Go back to it, replace fold operators with macros

An example for calculating the minimal 2 elements of the RDD in two methods, that result in dependent aggregates, for which the induction method fails due to missing inductive invariants on the UDFs.

$$\text{Let:}\quad \begin{aligned} min &= \lambda M, x.\texttt{if}(x < M)\,\texttt{then}\,\{x\}\,\texttt{else}\,\{M\} \\ f &= \lambda m_1, m_2.\texttt{if}(m_2 \le x)\,\texttt{then}\,\{ \\ &(m_1, m_2)\}\,\texttt{else}\,\{\texttt{if}(x < m_1)\,\texttt{then}\,\{(x, m_1)\}\,\texttt{else}\,\{(m_1, x)\}\} \end{aligned}$$

| P1($R$: $RDD_{\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}}$): |
|---|---|
| $m1 = \texttt{fold}(-\infty, min)(R)$ | $\texttt{return fold}((-\infty, -\infty), f)(R)$ |
| $R' = \texttt{filter}((\lambda m.\lambda x.x \ne m)(m1))(R)$ | |
| $m2 = \texttt{fold}(-\infty, min)(R')$ | |
| $\texttt{return}\ (m1, m2)$ | |

Program term of $P1$:

$$\left([\mathbf{x}_R]_{-\infty,min}, \left[\begin{cases} \mathbf{x}_R & \mathbf{x}_R \ne [\mathbf{x}_R]_{-\infty,min} \\ \perp & otherwise \end{cases}\right]_{-\infty,min}\right)$$

Program term of $P2$:

$$[\mathbf{x}_R]_{(-\infty,-\infty),f}$$

We attempt to prove by induction. Induction: base case is trivial. Step: Let $(A,B) = (C,D)$. We attempt to prove for the first element in the tuple as a first step. For that, need to prove $min(A,x) = \pi_1(f((C,D),x))$, which stands for:

$$ite(\mathbf{x} < A, \mathbf{x}, A) = ite(D \leq \mathbf{x}, C, ite(\mathbf{x} < C, \mathbf{x}, C))$$

Using induction hypothesis, by replacing all $C$ with $A$ and $D$ with $B$ we get:

$$ite(\mathbf{x} < A, \mathbf{x}, A) = ite(B \leq \mathbf{x}, A, ite(\mathbf{x} < A, \mathbf{x}, A))$$

Assume $\mathbf{x} < A$, the value of the LHS is $\mathbf{x}$. But, if $B \leq \mathbf{x}$, then RHS is equal to $A$, yielding $\mathbf{x} = A$, contrary to the assumption. Therefore the proof fails. The reason the proof failed is that we did not keep the information on either $B > A$ or $D > C$, which is an invariant of the program. Had it been part of the induction hypothesis, the proof would have succeeded. A possible fix is defining $f$ as follows: $(\lambda m_1, m_2.ite(m_2 < m_1, \bot, ite(m_2 \leq x, (m_1, m_2), ite(x < m_1, (x, m_1), (m_1, x))))$ Another reason the proof failed is due to examining the equivalence under $pi_1$ which is not injective. 2nd attempt: we continue the induction step but we want to prove:

$$(min(A,\mathbf{x}), min\left(B, \begin{cases} \mathbf{x} & \mathbf{x} \neq [\mathbf{x}_R]_{-\infty,min} \\ \bot & otherwise \end{cases}\right)) = f((C,D),x)$$

Which is equivalent to:

$$(min(A,\mathbf{x}_R), min\left(B, \begin{cases} \mathbf{x} & \mathbf{x} \neq [\mathbf{x}_R]_{-\infty,min} \\ \bot & otherwise \end{cases}\right)) = f((A,B),x) = ite(B \leq \mathbf{x}, (A,B), ite(\mathbf{x} < A, (\mathbf{x},A), (A,\mathbf{x})))$$

We take assumptions on RHS.

- $B \leq \mathbf{x} \implies$ RHS $= (A,B)$. We assume to the contrary that $A > \mathbf{x}$. But then, $min(A,\mathbf{x}) = \mathbf{x}$, contradiction to LHS=RHS. Therefore $A \leq \mathbf{x}$. As $B \leq \mathbf{x}$ and $min(B,\bot) = B$, the second tuple element of LHS is also B. So LHS=RHS, as required.
- $\mathbf{x} < B, \mathbf{x} < A \implies$ RHS=$(\mathbf{x},A)$. For the first tuple element of RHS, it is clear that $min(A,\mathbf{x}) = \mathbf{x}$ so there is equality. For the second element we need to prove $\begin{cases} \mathbf{x} & \mathbf{x} \neq [\mathbf{x}_R]_{-\infty,min} \\ \bot & otherwise \end{cases} = A$
  Which is obviously unprovable.
  It is interesting to understand why it fails even after getting an injective function. It relates first to the fact that the two tuple elements of LHS are unrelated. We know that $\mathbf{x}$ is the candidate for the real minimum, but we can't show in induction on LHS that A is "demoted" to the second tuple element. It never shows up there as part of the term.
  The deep reason for the failure is that if we iterate on $R, R'$ according to the ordering of their index set $I$, and take $A$ as the current value as of $i$ for the first element in LHS, where $\mathbf{x}$ at $i+1$ is smaller than $A$, then $A$ is also the current value of the second tuple element in LHS. It means that until we encountered the real minimum, the induction hypothesis for LHS is $(A,A)$, whereas the RHS always keeps two distinct elements $C, D$ where $C < D$ at every step of the induction.
- $\mathbf{x} < B, \mathbf{x} \geq A \implies$ RHS=$(A,\mathbf{x})$. (Can analyze, but the proof already failed)

▸ Example 6.18 (Filter and Max aggregate). SG: TODO continue from here

SG: replace fold with macros

The example programs below always return the same maximal element. The first program first checks if there is an element bigger than 6 and takes the max from a filtered RDD of elements bigger than 3, and the second program checks if there is an element bigger than 3 and takes the maximum from a filtered RDD of elements bigger than 2. Otherwise, both programs take the maximum from an unaltered RDD.

$$\text{Let:}\quad \begin{aligned} existElementBiggerThan &= \lambda a.\lambda b,x.\mathtt{if}\,(b \vee (x > a))\,\mathtt{then}\,\{tt\}\,\mathtt{else}\,\{\mathit{ff}\} \\ biggerThan &= \lambda a.\lambda x.\mathtt{if}\,(x > a)\,\mathtt{then}\,\{tt\}\,\mathtt{else}\,\{\mathit{ff}\} \\ max &= \lambda M,x.\mathtt{if}\,(x > M)\,\mathtt{then}\,\{x\}\,\mathtt{else}\,\{M\} \end{aligned}$$

| | P1($R$: $RDD_{\mathtt{Int}}$): | P2($R$: $RDD_{\mathtt{Int}}$): |
|---|---|---|
| 1 | $b = \mathtt{fold}(\mathtt{false}, existElementBiggerThan(6))(R)$ | $b = \mathtt{fold}(\mathtt{false}, existElementBiggerThan(3))(R)$ |
| 2 | $R' = \mathtt{if}(b)\,\mathtt{then}\,\{\mathtt{filter}(biggerThan(3))(R)\}\,\mathtt{else}\,\{R\}$ | $R' = \mathtt{if}(b)\,\mathtt{then}\,\{\mathtt{filter}(biggerThan(2))(R)\}\,\mathtt{else}\,\{R\}$ |
| 3 | $\mathtt{fold}(\bot, max)(R')$ | $\mathtt{fold}(\bot, max)(R')$ |

$$\phi_{P1}(b) = [\mathbf{x}_R]_{\mathit{ff}, existElementBiggerThan(6)}$$

$$\phi_{P2}(b) = [\mathbf{x}_R]_{\mathit{ff}, existElementBiggerThan(3)}$$

$$\phi_{P1}(R') = \begin{cases} \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 3 \\ \bot & otherwise \end{cases} & \phi_{P1}(b) = tt \\ \mathbf{x}_R & \end{cases}$$

$$\phi_{P2}(R') = \begin{cases} \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 2 \\ \bot & otherwise \end{cases} & \phi_{P2}(b) = tt \\ \mathbf{x}_R & \end{cases}$$

$$\phi_{P1}(r^o) = \left[ \begin{cases} \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 3 \\ \bot & otherwise \end{cases} & \phi_{P1}(b) = tt \\ \mathbf{x}_R & \end{cases} \right]_{\bot.max}$$

$$\phi_{P2}(r^o) = \left[ \begin{cases} \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 2 \\ \bot & otherwise \end{cases} & \phi_{P1}(b) = tt \\ \mathbf{x}_R & \end{cases} \right]_{\bot.max}$$

Let $x$ be a variable for $\mathbf{x}_R$ and $A = A'$ intermediate values of the external $max$ aggregation. Need to prove:

$$max\left(A, \begin{cases} \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 3 \\ \bot & otherwise \end{cases} & \phi_{P1}(b) = tt \\ \mathbf{x}_R & \end{cases}\right) = max\left(A', \begin{cases} \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 2 \\ \bot & otherwise \end{cases} & \phi_{P2}(b) = tt \\ \mathbf{x}_R & \end{cases}\right)$$

Which is equivalent to:

$$max\left(A, \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 3 \wedge \phi_{P1}(b) = tt \\ \bot & \mathbf{x}_R \leq 3 \wedge \phi_{P1}(b) = tt \\ \mathbf{x}_R & \phi_{P1}(b) = \mathit{ff} \end{cases}\right) = max\left(A, \begin{cases} \mathbf{x}_R & \mathbf{x}_R > 2 \wedge \phi_{P2}(b) = tt \\ \bot & \mathbf{x}_R \leq 2 \wedge \phi_{P2}(b) = tt \\ \mathbf{x}_R & \phi_{P2}(b) = \mathit{ff} \end{cases}\right)$$

Bringing it to a standard form:

$$max\left(A, \begin{cases} \mathbf{x}_R & (\mathbf{x}_R > 3 \wedge \phi_{P1}(b) = tt) \vee \phi_{P1}(b) = ff \\ \bot & otherwise \end{cases}\right) = max\left(A, \begin{cases} \mathbf{x}_R & (\mathbf{x}_R > 2 \wedge \phi_{P2}(b) = tt) \vee \phi_{P2}(b) = ff \\ \bot & otherwise \end{cases}\right)$$

We simplify the 2nd arguments in both $max$ calls as $\psi_1, \psi_2$ and want to know when:

$$max(A, \psi_1) = max(A, \psi_2)$$

which can be written as:

$$\begin{cases} A & (A \geq \psi_1 \wedge \psi_1 \neq \bot) \vee \psi_1 = \bot \\ \psi_1 & otherwise \end{cases} = \begin{cases} A & (A \geq \psi_2 \wedge \psi_2 \neq \bot) \vee \psi_2 = \bot \\ \psi_2 & otherwise \end{cases}$$

So what we need to check is

$$((A \geq \psi_1 \wedge \psi_1 \neq \bot) \vee \psi_1 = \bot \iff (A \geq \psi_2 \wedge \psi_2 \neq \bot) \vee \psi_2 = \bot) \wedge ((A < \psi_1 \wedge \psi_1 \neq \bot) \implies \psi_1 = \psi_2)$$

We start with substituting back $\psi_1, \psi_2$ in the first conjunct, and prove w.l.o.g. (symmetry) the $\implies$ direction. Assume:

$$A \geq \mathbf{x}_R \vee (\phi_{P1}(b) = tt \wedge \mathbf{x}_R \leq 3)$$

Need to prove:

$$A \geq \mathbf{x}_R \vee (\phi_{P2}(b) = tt \wedge \mathbf{x}_R \leq 2)$$

SG: ...

Putting in the second conjunct, we assume: $A < \psi_1 \wedge \psi_1 \neq \bot$, which is essentially

$$A < \mathbf{x}_R \wedge (\mathbf{x}_R > 3 \wedge \phi_{P1}(b) = tt) \vee \phi_{P1}(b) = ff$$

and prove:

$$\begin{cases} \mathbf{x}_R & (\mathbf{x}_R > 3 \wedge \phi_{P1}(b) = tt) \vee \phi_{P1}(b) = ff \\ \bot & otherwise \end{cases} = \begin{cases} \mathbf{x}_R & (\mathbf{x}_R > 2 \wedge \phi_{P2}(b) = tt) \vee \phi_{P2}(b) = ff \\ \bot & otherwise \end{cases}$$

using the assumption, this boils down to:

$$((\mathbf{x}_R > 3 \wedge \phi_{P1}(b) = tt) \vee \phi_{P1}(b) = ff) \iff (\mathbf{x}_R > 2 \wedge \phi_{P2}(b) = tt) \vee \phi_{P2}(b) = ff$$

The ($\leftarrow$) direction is clear from the assumption, so we prove ($\implies$).

So we assume first $\mathbf{x}_R > 3 \wedge \phi_{P1}(b) = tt$. If $\phi_{P2}(b) = ff$ the proof is completed, but if not we need to prove in addition that $\mathbf{x}_R > 2$, which we already know. Now we assume $\phi_{P1}(b) = ff$. We only need to prove that if $\phi_{P2}(b) = tt$ then $\mathbf{x}_R > 2$. Substituting for $\phi_{P2}(b)$ we get:

$$[\mathbf{x}_R]_{ff, existElementBiggerThan(3)} = tt \implies \mathbf{x}_R > 2$$

We proceed with induction again. Let some $y$, and $B$ an intermediate value. We need to prove that if $B = tt \implies \mathbf{x}_R > 2$ then $existElementBiggerThan(3)(B, y) = tt \implies \mathbf{x}_R > 2$. If $B$ is true then by definition $existElementBiggerThan(3)(B, y)$ is also true, and $\mathbf{x}_R > 2$ by the induction assumption. Otherwise,

SG: ... what a mess. I suspect we miss a few inductive invariants here to be able to give a proper proof

SG: On closer examination, the conditions are not fulfilled. both sides of the internal aggregation are not injective. We take a constant A,A', and x. We get a function of $\phi_{Pi}(b)$ which for certain values of $x$ (i.e. $x > 3$) is constant for choice of $\phi_{Pi}(b)$.

SG: We can add an inductive invariant and then prove it. But how to find it?

SG: TODO: Move to nested fold section and complete

## 6.6 By-key operations

SG: use new macros in the entire section

Previously we defined the fold by key operator $\langle x_r \rangle_{i,f}$ and the semantics of the *foldByKey* operation. According to the semantics, the folded value for each key is affected only by the set of values that match to the key in the bag. When the set of variables appearing in the term for the key and the set of variables appearing in the term for the value are disjoint, we have: $\langle x_r \rangle_{i,f} = (p_1(x_r), [p_2(x_r)])_{i,f}$. Otherwise, the term is more complex, and we shall illustrate it using the following example:

▸ Example 6.19 (Mixed key-value variables in value). In this example there is an RDD with dependencies between the key and the value:

$$\text{Let:} \quad \begin{aligned} f &= \lambda A, x.A + x \\ f' &= \lambda A, (x, y).A + y \end{aligned}$$

| | P1($R$: $RDD_{K\times V}$): | P2($R$: $RDD_{K\times V}$): |
|---|---|---|
| 1 | `return` $\mathtt{foldByKey}(0, f)(R)$ | $R' = \mathtt{map}(\lambda(k, v).(k, (k, v)))(R)$ |
| 2 | | `return` $\mathtt{foldByKey}(0, f')(R')$ |

We denote $\mathbf{x}_R = (\mathbf{x}_R^k, \mathbf{x}_R^v)$. For $P1$: $\phi_{P1}(r^o) = \langle(\mathbf{x}_R^k, \mathbf{x}_R^v)\rangle_{0,f} = (\mathbf{x}_R^k, [\mathbf{x}_R^v]_{0,f})$. For $P2$: $\phi_{P2}(r^o) = \langle(\mathbf{x}_R^k, (\mathbf{x}_R^k, \mathbf{x}_R^v)\rangle_{0,f'}$. Suppose we take some constant $\mathbf{x}_R^k$, and choose some $\mathbf{x}_R^v$ such that $(\mathbf{x}_R^k, \mathbf{x}_R^v) \in [\![R]\!]$. If so, we need to prove $[(\mathbf{x}_R^k, \mathbf{x}_R^v)]_{0,f'} = [\mathbf{x}_R^v]_{0,f}$ By induction: base case is trivial, for $A, A'$ such that $A = A'$, and some arbitrary choice of $\mathbf{x}_R^v$, we have:

$$f'(A', (\mathbf{x}_R^k, \mathbf{x}_R^v)) = A' + \mathbf{x}_R^v = f(A', \mathbf{x}_R^v) = f(A, \mathbf{x}_R^v)$$

as required.

**Specification of the *foldByKey* term:** Let us denote $FV_K(\phi)$ the set of free variables of the key section of a representative element $\phi$: $FV(p_1(\phi))$, and $FV_V(\phi)$ the set of free variables of the value section of a representative element $\phi$: $FV(p_2(\phi))$. We denote a new function based on the original *foldByKey* UDF: $f_k = \lambda\vec{k}{:}T^{|FV_K(\phi)|}.\lambda\vec{v}{:}T^{|FV_V(\phi)\setminus FV_K(\phi)|}.f[FV_K(\phi) \mapsto \vec{k}]$

Then: $\langle\phi\rangle_{i,f} = (p_1(\phi), [p_2(\phi)]_{i,f_k(\vec{a})})$ where $\vec{a}{:}T^{|FV_K(\phi)|}$

The *fold* operator binds all free variables in the representative element on which the *fold* operator is applied. The *foldByKey* operator binds only the variables that appear in the value section of the representative element but not in the key-section. Thus, it is a representative element whose free variables are those that belong to the key-section of the representative element on which the *foldByKey* operator was applied.

For two *foldByKey* expressions, we consider first the methodology for testing two RDDs with independent key and value terms for equivalence.

▸ **Lemma 16.** *Let there be two programs $P, P'$ with program terms $\phi_P(r^o) = (p_1(t), [p_2(t)]_{i,f})$ and $\phi_{P'}(r^o) = (p_1(t'), [p_2(t')]_{i',f'})$. We have $\phi_P(r^o) = \phi_{P'}(r^{o\prime})$ if and only if:*

- $p_1(t) = p_1(t')$ *and*
- $[p_2(t)]_{i,f} = [p_2(t')]_{i',f'}$

For non-independent key-value terms, the extension to lemma 16 is natural:

▸ **Lemma 17.** *Let there be two programs $P, P'$ with program terms $\phi_P(r^o) = (p_1(t), [p_2(t)]_{i,f_k(\vec{a})})$ and $\phi_{P'}(r^o) = (p_1(t'), [p_2(t')]_{i',f'_k(\vec{a}')})$. We have $\phi_P(r^o) = \phi_{P'}(r^{o\prime})$ if and only if:*

- $p_1(t) = p_1(t')$ *and*

- $[p_2(t)]_{i,f_k(\vec{a})} = [p_2(t')]_{i',f'_k(\vec{a}')}$

**Proof.** Correctness is by definition. ◂

We can therefore extend previous results to aggregated results which are by-key.

▸ **Corollary 18.** *Theorem ?? can be applied to the aggregated value of a by-key expression once the equivalence of the keys was proven.*

▸ Example 6.20. Basic example:

$$\text{Let: } \begin{array}{c} sum = \lambda A, x.A + 2 * x \\ double_2 = \lambda(x, y).(x, 2 * y) \end{array}$$

| | P1($R$: $RDD_{\texttt{Int}\times\texttt{Int}}$): | P2($R$: $RDD_{\texttt{Int}\times\texttt{Int}}$): |
|---|---|---|
| 1 | $R' = \texttt{foldByKey}(0, sum)(R)$ | $R' = \texttt{map}(double_2)(R)$ |
| 2 | $\texttt{return map}(double_2)(R')$ | $\texttt{return foldByKey}(0, sum)(R')$ |

The representative elements and program term of $P1$ are:

$$\phi_{P1}(R') = (p_1(\mathbf{x}_R), [p_2(\mathbf{x}_R)]_{0,sum})$$

$$\phi_{P1}(r^o)(p_1(\mathbf{x}_R), 2 * [p_2(\mathbf{x}_R)]_{0,sum})$$

The representative elements and program term of $P2$ are:

$$\phi_{P2}(R') = (p_1(\mathbf{x}_R, 2 * p_2(\mathbf{x}_R))$$

$$\phi_{P2}(r^o)(p_1(\mathbf{x}_R), [2 * p_2(\mathbf{x}_R)]_{0,sum})$$

Need to prove, by lemma 16: $p_1(\mathbf{x}_R = p_1(\mathbf{x}_R$ (immediate), and

$$2 * [p_2(\mathbf{x}_R)]_{0,sum} = [2 * p_2(\mathbf{x}_R)]_{0,sum}$$

We proceed with an application of lemma 5:

$$\forall x = (x_1, x_2), A, A'.2 * A = A' \implies 2 * (A + x_2) = A' + (2 * x_2)$$

We have $2 * (A + x_2) = 2 * A + 2 * x_2 = A' + 2 * x_2$ as required. ■

▸ Example 6.21. This example shows how we can prove equivalence of by-key transformations even when we attempt to 'trick' the theory by making the definition of the keys 'fluid' - meaning keys with tuples of variant length.

$$\text{Let: } \begin{array}{c} f = \lambda A, (x_1, x_2).(max(p_1(A), x_1), min(p_2(A), x_2)) \\ g = \lambda A, x.max(A, x) \end{array}$$

| | P1($R$: $RDD_{\texttt{Int}\times(\texttt{Int}\times\texttt{Int})}$): | P2($R$: $RDD_{\texttt{Int}\times(\texttt{Int}\times\texttt{Int})}$): |
|---|---|---|
| 1 | $R_1 = \texttt{filter}(\lambda(x, (y, z)).x = y \wedge y = z)(R)$ | $R_1 = \texttt{filter}(\lambda(x, (y, z)).x = y \wedge y = z)(R)$ |
| 2 | $R_2 = \texttt{map}(\lambda(x, (y, z)).((x, y), z))(R_1)$ | $\texttt{return foldByKey}((\bot, \bot), f)(R_1)$ |
| 3 | $R_3 = \texttt{foldByKey}(\bot, g)(R_2)$ | |
| 4 | $\texttt{return map}(\lambda((x, y), z).(x, (y, z)))(R_3)$ | |

SG: [originally, in the PODS98 paper, there was sum(y) instead of max(y). But this is not really equivalent when relations are allowed to contain bags, for example two instances of (1,1,1) in R]

The representative elements and program term of $P1$ are:

$$\phi_{P1}(R_1) = \begin{cases} \mathbf{x}_R & p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R)) \\ \bot & otherwise \end{cases}$$

$$\phi_{P1}(R_2) = \begin{cases} ((p_1(p_1(\mathbf{x}_R)), p_1(p_2(\mathbf{x}_R))), p_2(p_2(\mathbf{x}_R))) & p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R)) \\ \perp & otherwise \end{cases}$$

$$\phi_{P1}(R_3) = \begin{cases} ((p_1(p_1(\mathbf{x}_R)), p_1(p_2(\mathbf{x}_R))), [p_2(p_2(\mathbf{x}_R))]_{\perp,g}) & p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R)) \\ \perp & otherwise \end{cases}$$

$$\phi_{P1}(r^o) = \begin{cases} (p_1(p_1(\mathbf{x}_R)), (p_1(p_2(\mathbf{x}_R)), [p_2(p_2(\mathbf{x}_R))]_{\perp,g})) & p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R)) \\ \perp & otherwise \end{cases}$$

The representative elements and program term of $P2$ are:

$$\phi_{P2}(R_1) = \begin{cases} \mathbf{x}_R & p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R)) \\ \perp & otherwise \end{cases}$$

$$\phi_{P2}(r^o) = \begin{cases} (p_1(p_1(\mathbf{x}_R)), [(p_1(p_2(\mathbf{x}_R)), p_2(p_2(\mathbf{x}_R)))]_{(\perp\perp),f} & p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R)) \\ \perp & otherwise \end{cases}$$

As the filter conditions in both programs are equal, we only need to prove, with the knowledge that:

$$(*) p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) \wedge p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R))$$

the following:

$$(p_1(p_1(\mathbf{x}_R)), (p_1(p_2(\mathbf{x}_R)), [p_2(p_2(\mathbf{x}_R))]_{\perp,g})) = (p_1(p_1(\mathbf{x}_R)), [(p_1(p_2(\mathbf{x}_R)), p_2(p_2(\mathbf{x}_R)))]_{(\perp\perp),f})$$

Which in its turn can be simplified to proving:

$$(p_1(p_2(\mathbf{x}_R)), [p_2(p_2(\mathbf{x}_R))]_{\perp,g}) = [(p_1(p_2(\mathbf{x}_R)), p_2(p_2(\mathbf{x}_R)))]_{(\perp\perp),f}$$

We write $X = p_1(p_1(\mathbf{x}_R)) = p_1(p_2(\mathbf{x}_R)) = p_2(p_2(\mathbf{x}_R))$, and simplify:

$$(X, [X]_{\perp,g}) = [(X, X)]_{(\perp,\perp),f}$$

The sloppy handling of *foldByKey* representative elements brought us to an equivalence formula which is not provable (as $X$ ranges over all of the integer domain). But, under the $(*)$ assumption, we rewrite some of the representative elements using the fold by key operator:

$$\phi_{P1}(R_3) = \langle((p_1(p_1(\mathbf{x}_R)), p_1(p_2(\mathbf{x}_R))), p_2(p_2(\mathbf{x}_R)))\rangle\perp, g$$
$$\phi_{P1}(r^o) = (p_1(p_1(\phi_{P1}(R_3))), (p_2(p_1(\phi_{P1}(R_3))), p_2(\phi_{P1}(R_3))))$$
$$\phi_{P2}(r^o) = \langle\mathbf{x}_R\rangle_{(\perp\perp),f}$$

Under the $(*)$ assumption, the following is translated to:

$$\phi_{P1}(R_3) = \langle((X, X), X)\rangle\perp, g$$
$$\phi_{P1}(r^o) = (p_1(p_1(\langle((X, X), X)\rangle\perp, g)), (p_2(p_1(\langle((X, X), X)\rangle\perp, g)), p_2(\langle((X, X), X)\rangle\perp, g)))$$
$$\phi_{P2}(r^o) = \langle(X, (X, X))\rangle_{(\perp\perp),f}$$

According to the rule defined in lemma 18, all instances of the key variables in an RDD element in the fold functions $f$ and $g$ become constant. In our case, the key variable is also the value variable, which is $X$. Thus, $f_k == (X, X)$, $g_k == X$, and we get in both programs, that for each element of the RDD $R$ satisfying $(*)$, an element of the form $(X, (X, X))$. ■

This example may be a bit contrived, but it shows several properties:

- It is possible to consider keys of variable length if mapped correctly, even though real-life code hardly contain such abnormalities.
- Under certain conditions, it is possible to compare aggregated values with an RDD's elements
- There is a great importance to keeping track of the key variables of the representative element, after any kind of operation applied on the RDD. This allows to fixate the values correctly.

### 6.6.1 *groupByKey*

SG: Skip this section for now. It may be solved once the *union* case is solved

The *groupByKey* transform is interesting as it allows to postpone the actual aggregation to a later phase. The natural definition for a representative element $R.groupByKey()$ for an RDD of type $K \times \tau$ is $(k, [t]^k)$, where $(k, t)$ is the representative element of $R$.

A *mapValues* operation is then used to apply the actual aggregation. For example, continuing the previous example, $R.groupByKey().mapValues(sum)$ yields: $(k, [t]^k_{sum})$. It's worth to note, that the function given to the *mapValues* transform acts on the entire set of values belonging to some key. This means that given a set, the UDF of the *mapValues* is also allowed to return a set. Therefore, it is required for that UDF to be either a definite aggregation function, or to behave like a regular *map* on the set of values. Namely, the allowed operations for the UDF are operations that consider the input set as a "Sub-RDD" of type $\tau$ (and does not introduce a result which is part aggregate, part regular map, or another byKey operation [SG TODO bring example!]).

| | | | |
|---|---|---|---|
| **Bag Exp.** | $bag$ | ::= | $bag \cup bag \mid bag \setminus bag \mid bag \times bag \mid bag\|_{\boldsymbol{f}} \mid bag\|_{\pi_{\boldsymbol{f}}}$ |
| **Partitioning Ops.** | $P$ | ::= | $\texttt{mapPartitions}(\boldsymbol{BagF})$ |
| **Bag functions** | $BagFdef$ | ::= | $\texttt{def } \boldsymbol{BagF} \texttt{ as } \lambda \boldsymbol{x}: Bag_\tau \;\; bag: Bag_\tau$ |
| **Program** | $Prog$ | ::= | $\boldsymbol{P}(\overline{\boldsymbol{r}: RDD_\tau}, \overline{\boldsymbol{v}:\tau}) \texttt{ is } \overline{Fdef} \;\; \overline{PFdef} \;\; \overline{BagFdef} \;\; E$ |

**Figure 19** Syntax for $SPARK$ with partitions

# 7 Extension to partitions

When programming in Spark, it is convenient for us to think of the RDD as a single bag, without considering how it is distributed in the network. However, it is a simplistic view that by ignoring how data is distributed across the network, many opportunities for optimizations are lost. Spark contains many constructs in controlling the **partitioning** of the data: changing the number of partitions, the method of partitioning. It is done either as part of a dedicated transform such as *shuffle* or *repartition*, or as a side-effect of a transform that requires shuffling, such as *reduceByKey* or *foldByKey* . Better partitioning accounts for a more balanced distribution of the data, giving a fair share of computation to each participating "worker" node, improving running times. Nevertheless, repartitioning itself may be responsible for delays and may be a source of inefficiency. Striking the right balance between fairness and laziness in shuffles is difficult, and is not handled in this work [SG - TODO refer to relevant works, probably from the system field]. It is still important, however, to detect special transforms that consider partitioning, and be able to prove equivalency, as the equivalency is the basis on which optimizations can be performed.

## 7.1 The partitioned RDD

We extend the previously defined $SPARK$ language to support a unique operation of Spark for partitioned datasets, called *mapPartitions* . An additional, special type is added, known as a *bag*, and named $Bag_\tau$. It is a data type for which the five basic operations presented in Section 2 are supported. We add to the syntax *bag expressions*, based on the operations presented in section 2, and a type of UDFs that work exclusively on bags, and return bags.

Spark's internal implementation of RDDs is based on partitions. Loosely speaking, if we have a dataset distributed among many nodes, each node is holding one or more partitions, but a partition cannot be shared among nodes. The *mapPartitions* operation is in particular useful to various optimizations made possible from Spark's architecture. Despite the definition of partitioned bags, however, capturing Spark's partitioning semantics in full is both a difficult challenge and an unrewarding one - the semantics change between versions, and some of it requires insight into data we do not have available such as hash values and sampling of RDDs. Spark has many more operations, that can be expressed using the basic operations, but sometimes differ in how they partition the data. Spark's partitioning is very dynamic in nature and cannot be calculated statically. However, we can keep some invariants: the number of partitions an RDD has, and whether the 'unified' RDD, or the RDD that is received when merging all partitions, is the same no matter how different operations change the partitioning in the course of the program.

**Partitioned RDD Semantics** The semantics of *partitioned resilient distributed datasets* is that of *partitioned multisets*:

$$r: RDD_\tau \longrightarrow [\![r]\!]() = m^p: (\tau \hookrightarrow (\{1, \ldots, p\} \to \mathbb{N}))$$

The exact partitioning of the RDD is unspecified. We see that functionality-wise, most operations are not affected by the partitioning. Operations that are *sensitive to partitioning*, such as *mapPartitions* , are expected to produce the same unified RDD independent of the underlying partitioning. In addition, each transform is either *partitioning-preserving* or not. A transform that is **not** *partitioning-preserving* is one that *shuffles* items that were not in the same partition before the operation to the same partition, or that separate items that originated in the same partition to other partitions. *foldByKey* is an example for a transform which is not *partitioning-preserving*, by re-organizing the partitions according to the key in some non-deterministic way. For a more thorough discussion of the challenges in implementing compleete partitioning semantics, refer to the examples in Appendix...

SG: TODO?

SG: TODO Put semantics after finalizing regular semantics

$$\textbf{MapPartitions:}\quad [\![\texttt{mapPartitions}(\boldsymbol{BagF})]\!]() \;=\; \lambda r. \oplus_i [\![\boldsymbol{BagF}]\!]()(P_i([\![r]\!]()))$$

▸ Example 7.1. The importance of *mapPartitions* transformation in real-life optimizations is in the ability to build and load complex objects only once **per partition**, instead of once **per element**. Considering the following real Spark code:

$$\begin{aligned}&R.map(element\\&\qquad => \{val\ data = loadData();\\&\qquad\quad return\ f(element, data);\})\end{aligned}$$

which is optimized to:

$$\begin{aligned}&R.mapPartitions(elementIter =>\\&\qquad\qquad \{val\ data = loadData();\\&\qquad\qquad elementIter.map(element =>\\&\qquad\qquad\qquad f(element, data);)\})\end{aligned}$$

An optimization which is essentialy a loop-invariant code motion.

In $SPARK^{Agg}$ syntax, in which we augment the UDFs with the ability to load data from the outside as well, as well as the Presburger arithmetic already allowed, the programs look very similar to the above. Analyzing it in the form of a $\kappa$ representative element yields a $val\ data = loadData();\ f(r.x, data)$ for the first program, and $(val\ data = loadData(); r.map(\kappa x.f(x, data))) = f(r.x, data)$. The only needed test is that it doesn't matter when $val\ data = loadData()$ is performed. **TODO improve phrasing** - this is due to the $\kappa$-calculus parametric abstraction. While all $\kappa$ expressions act on the RDDs' actual elements, it is possible, by the $\kappa$-calculus, to extend any $\kappa$ expression's domain with another type, standing for a parameter. This allows various "load" operations in Spark, a pre-processing done before calling the actual function with all arguments. These "load" operations can be added anywhere - either before the call to transform, or during it - and it is captured in the same way in the $\kappa$ expression.

**RDD equivalency in the partitioned case** We define two notions for RDD equivalency, *weak* and *strong*. Weak equivalency is equivalency of RDDs when merged to a regular bag. Strong equivalency is equivalency requiring number of partitions to be equal and also per partition bag equivalency. Throughout the rest of this paper, **RDD equivalency refers exclusively to weak equivalency**.

SG: TODO formal with $\mathcal{M}$

For brevity, the examples that are not affected by partitioning-sensitive operations are analyzed as single-partitioned bags. All previously obtained results transfer automatically to the partitioned case in the absence of *mapPartitions*.

SG: TODO - extend results to aggregate/aggregateByKey, because the partitioned case requires specification of aggregation for intermediate values (combiner). For example, a sufficient condition is equivalence of "local", partition fold case, according to our standard lemmas, + equality of the intermediate value combiner functions. A necessary condition could be that we have partitioning independence. For example, we assume all partitions to be of size 1, and then we prove as usual for the combiner function, acting on elements of the form $foldF(init, x)$

SG: TODO - Add remark on the fact that foldByKey preserves the number of partitions in an RDD, but shuffles and chagnes the partitioning according to some hash function on the first element of the RDD's tuple elements or some other strategy which is not encoded in $SPARK$.

## 8 Related Work

- [8] contains a formal discussion of a logic with aggregate operators and arithmetic functions over $\mathbb{Q}$, and an embedding of a commercial SQL fragment (and its common aggregate functions) to the aggregate logic.
- [11] describe an extension of Presburger arithmetic with boolean algebras of sets of uninterpreted elements, with capabilites to describe correlations between integer variables and set cardinalities, which is decidable. One of the applications of their theory is constraint database query evaluation.
- [10] is a non-recent work reviewing

## 9 Future work

- Defining abstractions for Spark programs.
- Check properties of Spark programs, not just equivalence/equivalence modulo a transform
- Sound verification of programs with conditions in the program body, not just in UDFs.
- Loops, both in program body and UDFs. This will allow verifying a wider range of programs:

  ▸ Example 9.1. Nice example with finding the median:

  ```
  Algo 1:
  val C = R.fold(0,count);
  val median = ⊥;
  for i = 0; i < C;
          val min = R.fold(⊥,min);
          val countMin = R.filter(x == min).fold(0,count);
          i = i + countMin;
          if (2*i >= C)
                  median = min;
                  break;
          R = R.filter(x != min);
  ```

  ```
  Algo 2:
  Replace min with max.
  ```

  These programs both compute the median, using different aggregate functions.
- Implementing the procedure using Cooper's algorithm for Presburger arithmetic.
- Adding ordering to our definition of RDDs?

SG: TODO remove future tense from the rest of the paper

## 10 Appendix 1: Typing rules for $SPARK$

**Booleans**

$$\frac{}{\rho\vdash\texttt{true}:Boolean} \qquad \frac{}{\rho\vdash\texttt{false}:Boolean}$$

**Integers**

$$\frac{}{\rho\vdash 0,1,\ldots:Integer}$$

**Integer ops**

$$\frac{\rho\vdash i:Integer, j:Integer, op\in\{+,-,*,\%\}}{\rho\vdash i\ op\ j:Integer} \qquad \frac{\rho\vdash i:Integer, j:Integer, op\in\{<,\leq,=,\geq,>\}}{\rho\vdash i\ op\ j:Boolean}$$

**Boolean ops**

$$\frac{\rho\vdash b:Boolean}{\rho\vdash !b:Boolean} \qquad \frac{\rho\vdash b_1:Boolean, b_2:Boolean, op\in\{\wedge,\vee\}}{\rho\vdash b_1\ op\ b_2:Boolean}$$

**Tuples**

$$\frac{\rho\vdash e_1:\tau_1, e_2:\tau_2}{\rho\vdash (e_1,e_2):\tau_1\times\tau_2} \qquad \frac{\rho\vdash e:\tau_1\times\ldots\times\tau_n}{\rho\vdash p_i(e):\tau_i}$$

**UDFs**

$$\frac{\rho\vdash f:C_1\times\cdots\times C_n\rightarrow(\tau\rightarrow\tau'), \vec{e}:C_1\times\cdots\times C_n}{\rho\vdash f(\vec{e}):\tau\rightarrow\tau'} \qquad \frac{\rho\vdash f:\tau\rightarrow\tau', t:\tau}{\rho\vdash f(t):\tau'}$$

**RDD**

$$\frac{\rho\vdash r:RDD_\tau, f:\tau\rightarrow\tau'}{\rho\vdash\texttt{map}(f)(r):RDD_{\tau'}} \qquad \frac{\rho\vdash r:RDD_\tau, f:\tau\rightarrow Boolean}{\rho\vdash\texttt{filter}(f)(r):RDD_\tau}$$

$$\frac{\rho\vdash r:RDD_\tau, r':RDD_\tau, op\in\{union, subtract\}}{\rho\vdash\texttt{op}(r,r'):RDD_\tau} \qquad \frac{\rho\vdash r:RDD_\tau, r':RDD_{\tau'}}{\rho\vdash\texttt{cartesian}(r,r'):RDD_{\tau\times\tau'}}$$

$$\frac{\rho\vdash r:RDD_\tau, f:\tau'\times\tau\rightarrow\tau, init:\tau'}{\rho\vdash\texttt{fold}(init,f)(r):\tau'} \qquad \frac{\rho\vdash r:RDD_{K\times V}, f:(V'\times V)\rightarrow V', init:V'}{\texttt{foldByKey}(init,f)(r):RDD_{K\times V'}}$$

$$\frac{\rho\vdash r:RDD_\tau, f:RDD_\tau\rightarrow RDD_{\tau'}}{\rho\vdash\texttt{mapPartitions}(r)(f):RDD_{\tau'}}$$

**Figure 20** Typing rules for $SPARK$

fix mappartitions type rules after defining partitions

## 11 Appendix 2: Proof of the equivalence of the standard semantics and SR(P) semantics for $SPARK$

Let $P : \boldsymbol{P}(\overline{\boldsymbol{r}}, \overline{\boldsymbol{v}}) = \overline{\boldsymbol{F}}\,\overline{\boldsymbol{f}}\, E$ be a program with input $\vec{r^i}$, $[\![P]\!]$ be the interpretation of its output according to the defined semantics, and $SR(P)$ by the symbolic representation semantics of $P$ as described earlier. Then:

$$p_1(SR(P)(\vec{r^i})) = [\![P]\!]([\![\vec{r^i}]\!])$$

The proof follows by structural induction on the available operations in the $SPARK$ program $P$ (The syntactic term $E$). We also assume the RDD given as arguments to the operations are input RDDs. In addition, multiplicity is expressed using the choice of a representative element: $\mathcal{M}(r) = ite(\mathbf{x}_r \in r, r(\mathbf{x}_r), 0)$ and $\mathcal{M}(P)$ is calculated accordingly (see Figure 10).

- *map*: Suppose $E = \texttt{map}(f)(r)$. We have

  $$p_1(SR(P)(r)) = \{\!\{\Phi(P)(\mathbf{x}_r); r(\mathbf{x}_r) \mid \Phi(P)(\mathbf{x}_r) \neq \bot \wedge \mathbf{x}_r \in r\}\!\} = \{\!\{f(\mathbf{x}_r)); r(\mathbf{x}_r) \mid f(\mathbf{x}_r) \neq \bot \wedge \mathbf{x}_r \in r\}\!\}$$

  While in the original semantics:

  $$[\![P]\!] = [\![\texttt{map}]\!](f)(r) = \pi_2(\{\!\{(x, f(x)); r(x) \mid x \in r\}\!\})$$

  Recall that $[\![\texttt{map}]\!]$ returns a bag, so by taking some $y$ such that $y \in [\![\texttt{map}]\!](f)(r)$, we know how to calculate its multiplicity in the bag:

  $$([\![\texttt{map}]\!](f)(r))(y) = \sum_{(x,f(x)) \in \{\!\{(x,f(x));r(x) \mid x \in r\}\!\} \wedge f(x)=y} \{\!\{(x, f(x)); r(x) \mid x \in r\}\!\}(x, f(x)) = \sum_{x \in r \wedge f(x)=y} r(x)$$

  which is the canonical representation of the bag defined by $SR(P)(r)$ - the multiplicity of $f(\mathbf{x}_r)$ is equal to the sum of all possible preimages $r(\mathbf{x}_r)$ in $r$.
- *filter*: Suppose $E = \texttt{filter}(f)(r)$.

  $$\begin{aligned} p_1(SR(P)(r)) &= \{\!\{\Phi(P)(\mathbf{x}_r); r(\mathbf{x}_r) \mid \Phi(P)(\mathbf{x}_r) \neq \bot \wedge \mathbf{x}_r \in r\}\!\} \\ &= \{\!\{ite(f(\mathbf{x}_r) = tt, (\mathbf{x}_r; r(\mathbf{x}_r)), \bot) \mid f(\mathbf{x}_r) = tt \wedge \mathbf{x}_r \in r\}\!\} \\ &= \{\!\{(\mathbf{x}_r; r(\mathbf{x}_r)) \mid f(\mathbf{x}_r) = tt \wedge \mathbf{x}_r \in r\}\!\} \end{aligned}$$

  While:

  $$[\![\texttt{filter}]\!](f)(r) = r\restriction_{\{x \mid f(x)\}} = \sigma_{x \in \{x \mid f(x)\}}(r) = \{\!\{x; r(x) \mid x \in \{x \mid f(x)\}\}\!\}$$

  And the equality of the bags follows.
- *cartesian*: Suppose $E = \texttt{cartesian}(r_1, r_2)$. We have:

  $$\begin{aligned} p_1(SR(P)(r_1, r_2)) &= \{\!\{\Phi(P)(\mathbf{x}_{r_1}, \mathbf{x}_{r_2}); r_1(\mathbf{x}_{r_1}) r_2(\mathbf{x}_{r_2}) \mid \Phi(P)(\mathbf{x}_{r_1}, \mathbf{x}_{r_2}) \neq \bot \wedge \mathbf{x}_{r_1} \in r_1 \wedge \mathbf{x}_{r_2} \in r_2\}\!\} \\ &= \{\!\{(\mathbf{x}_{r_1}, \mathbf{x}_{r_2}); r_1(\mathbf{x}_{r_1}) r_2(\mathbf{x}_{r_2}) \mid (\mathbf{x}_{r_1}, \mathbf{x}_{r_2}) \neq \bot \wedge (x, y) \in (r_1, r_2)\}\!\} \\ &= \{\!\{(\mathbf{x}_{r_1}, \mathbf{x}_{r_2}); r_1(\mathbf{x}_{r_1}) r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_1} \in r_1 \wedge \mathbf{x}_{r_2} \in r_2\}\!\} \end{aligned}$$

  In the standard semantics, we have:

  $$[\![\texttt{cartesian}]\!](r_1, r_2) = r_1 \times r_2 = \{\!\{(x_1, x_2)\,;\, r_1(x_1) \cdot r_2(x_2) \mid x_1 \in r_1 \wedge x_2 \in r_2\}\!\}$$

  And the equality is straightforward. For self-cartesian-products, we will have two unique names in $SR(P)$: $\mathbf{x}_r^{(1)}, \mathbf{x}_r^{(2)}$, so again equality will follow immediately.

still cleaning it up

- *union* : Suppose $E = \texttt{union}(r_1, r_2)$. We have:

$$p_1(SR(P)(r_1,r_2)) \quad = \quad \{\!\{\Phi(P)(\mathbf{x}_{r_1},\mathbf{x}_{r_2}); \mathcal{M}(r_1,r_2)(\mathbf{x}_{r_1}, r_2(\mathbf{x}_{r_2}) \mid \Phi(P)(\mathbf{x}_{r_1},\mathbf{x}_{r_2}) \neq \bot \wedge (\mathbf{x}_{r_1},\mathbf{x}_{r_2}) \in (r_1,r_2)\}\!\}$$
$$=^{def} \quad \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\} \cup \{\!\{\mathbf{x}_{r_2}; r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_2} \in r_2\}\!\}$$

while:

$$[\![\texttt{union}]\!](r_1,r_2) = r_1 \cup r_2 = \{\!\{x\,;\, ite(r_1(x) \neq \bot, r_1(x), 0) + ite(r_2(x) \neq \bot, r_2(x), 0) \mid x \in r_1 \cup r_2\}\!\}$$

We prove containment of both sides:

- $p_1(SR(P)(r_1,r_2)) \subset [\![\texttt{union}]\!](r_1,r_2)$: Let $a \in p_1(SR(P)(r_1,r_2))$. Then $a \in \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\}$ or $a \in \{\!\{\mathbf{x}_{r_2}; r_2[\mathbf{x}_{r_2} \mid \mathbf{x}_{r_2} \in r_2\}\!\}$. Suppose w.l.o.g. (symmetry) that $a \in \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\}$. Then $a$ has multiplicity $r_1(a)$ in $r_1$. If we also have $a \in \{\!\{\mathbf{x}_{r_2}; r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_2} \in r_2\}\!\}$ then $a$ has multiplicity $r_2(a)$ in $r_2$ and we get that $a \in r_1 \cup r_2$ with multiplicity $r_1(a) + r_2(a)$. Therefore $a \in \{\!\{x\,;\, ite(r_1(x) \neq \bot, r_1(x), 0) + ite(r_2(x) \neq \bot, r_2(x), 0) \mid x \in r_1 \cup r_2\}\!\} = [\![\texttt{union}]\!](r_1,r_2)$ as required. If $a \notin \{\!\{\mathbf{x}_{r_2}; r_2(\mathbf{x}_{r_2}) \mid \mathbf{x}_{r_2} \in r_2\}\!\}$ then $r_2(a) = \bot$. Still, $a \in r_1 \cup r_2$ but with multiplicity $r_1(a)$, thus: $a \in \{\!\{x\,;\, ite(r_1(x) \neq \bot, r_1(x), 0) + ite(r_2(x) \neq \bot, r_2(x), 0) \mid x \in r_1 \cup r_2\}\!\} = [\![\texttt{union}]\!](r_1,r_2)$ with multiplicity $r_1(a) + 0$, as required.
- $[\![\texttt{union}]\!](r_1,r_2) \subset p_1(SR(P)(r_1,r_2))$: Let $a \in [\![\texttt{union}]\!](r_1,r_2)$. Then $a \in r_1 \cup r_2$ and from the condition we know that at least one of $r_1(a)$ or $r_2(a)$ are not $\bot$. Assume w.l.o.g. (symmetry) that $r_1(a) \neq \bot$. Then $a \in \{\!\{\mathbf{x}_{r_1}; r_1(\mathbf{x}_{r_1}) \mid \mathbf{x}_{r_1} \in r_1\}\!\} \subset p_1(SR(P)(r_1,r_2))$ as required.

- *fold* : Suppose $E = \texttt{fold}(e,f)(r)$. We have: $p_1(SR(P)(r)) =^{def} [\mathbf{x}_r]_{e,f} =^{def} [\![\texttt{fold}]\!](e,f)(r)$ pending aggregate completion
- *foldByKey* : Suppose $E = \texttt{foldByKey}(e,f)(r)$. We have:

pending by-key completion

$$p_1(SR(P)(r)) \quad =^{def} \{\!\{\langle \mathbf{x}_r \rangle_{e,f}; 1 \mid \mathbf{x}_r \in r\}\!\} = \{\!\{(p_1(\mathbf{x}_r), [p_2(\mathbf{x}_r)]_{e,f}; 1 \mid \mathbf{x}_r \in r\}\!\}$$

By proving: $[p_2(\mathbf{x}_r)]_{e,f} = [\![\texttt{fold}]\!](e,f)(\pi_2(r \upharpoonright_{(\mathbf{x}_r^k,\_)}))$ we get $p_1(SR(P)(r)) = [\![\texttt{foldByKey}]\!](e,f)(r)$, as required.

## References


1 Java. `http://java.net`. Accessed: 2016-07-19.
2 Python. `https://www.python.org/`. Accessed: 2016-07-19.
3 Scala. `http://www.scala-lang.org`. Accessed: 2016-07-19.
4 Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
5 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: `http://doi.acm.org/10.1145/115372.115320`, `doi:10.1145/115372.115320`.
6 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1251254.1251264`.
7 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. URL: `http://doi.acm.org/10.1145/945445.945450`, `doi:10.1145/945445.945450`.

8 Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, July 2001. URL: `http://doi.acm.org/10.1145/502090.502100`, `doi:10.1145/502090.502100`.

9 Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):pp. 299–314, 1996.

10 Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982. URL: `http://doi.acm.org/10.1145/322326.322332`, `doi:10.1145/322326.322332`.

11 Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Autom. Reasoning*, 36(3):213–239, 2006. `doi:10.1007/s10817-006-9042-1`.

12 Derek C. Oppen. A 222pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323 – 332, 1978. URL: `http://www.sciencedirect.com/science/article/pii/0022000078900211`, `doi:http://dx.doi.org/10.1016/0022-0000(78)90021-1`.

13 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.