Verifying Equivalence of Spark Programs

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

[1] Tel Aviv University, Israel. {shellygr,maon,msagiv}@tau.ac.il
[2] The Hebrew University of Jerusalem, Israel. sara@cs.huji.ac.il
[3] Massachusetts Institute of Technology, USA. shachari@mit.edu

Abstract. *Spark* is a popular framework for writing large scale data processing applications. Such frameworks, intended for data-intensive operations, share many similarities with database systems, but do not enjoy a similar support of optimization tools. Our goal is to develop tools for reasoning about Spark programs. This is challenging because Spark programs combine relational algebraic operations and aggregate operations with *User Defined Functions* (*UDF*s).
We present the first technique for verifying the equivalence of Spark programs. We model Spark as a programming language whose semantics imitates Relational Algebra queries (with aggregations) over bags (multi-sets) and allows for UDFs expressible in Presburger Arithmetics. While the problem of checking equivalence is undecidable in general, we present a sound technique for verifying the equivalence of an interesting class of Spark programs, and show that it is complete under certain restrictions. We implemented our technique in a prototype tool, and used it to verify the equivalence of a few small, but intricate, open-source Spark programs.

1 Introduction

Spark [18, 28, 29] is a popular framework for writing large scale data processing applications. Such frameworks, intended for data-intensive operations, share many similarities with NoSQL database systems (see, e.g., [3]). Unlike traditional relational databases, which are accessed using a standard query language, NoSQL databases are often accessed via an entire program. A key property of Spark is the ability to employ User Defined Functions (UDFs), which are used in higher-order operations such as *map*, *filter*, and *fold*.

Spark was developed in reaction to the data flow limitations of the Map-Reduce paradigm. Importantly, Spark programs are centered on the resilient distributed dataset (RDD) structure, which contains a bag of (distributed) items. An RDD r can be accessed using operations such as *map*, which applies a function to all items in r, *filter*, which filters items in r using a given boolean function, and *fold* which aggregates items together, again using a UDF. Intuitively, map, filter and fold can be seen as extensions to the standard database operations *project*, *select* and *aggregation*, respectively, with arbitrary UDFs applied. A language such as *Scala* or *Python* is used as Spark's interface, which allows to embed calls

Class	Description	Decidable?	Method coverage
$NoAgg$	No aggregations	Yes	Sound and complete
Agg^1	Single aggregation, primitive output	No	Sound
$AggPair^1_{sync}$	Collapsible aggregations	Yes	Sound and complete
Agg^1_R	Single aggregation, RDD output	No	Sound
Agg^n	Multiple non-nested aggregations	No	Sound

Table 1. Classes of Spark programs, their decidability result, and the soundness/completeness of their verification techniques. ($AggPair^1_{sync}$ is a class of pairs of programs.)

to the underlying framework in standard programs, as well as to define the UDFs that Spark executes.

This paper addresses the problem of reasoning about Spark programs. In particular, we are interested in questions of checking whether two Spark programs are equivalent and identifying the differences between two programs. This problem is important for query optimization [5], program correctness, program regression, and software understanding. The problem of checking equivalence in Spark is undecidable in general, even for programs containing a single aggregate operation. Therefore, we are interested in (1) identifying sub-cases in which it is decidable to check equivalence, and (2) identifying sound and useful methods for dealing with large classes of Spark programs. We note that some of the intricacies arise from the fact RDDs are bags (and not sets or individual items), and Spark programs may contain aggregations (fold operations).

1.1 Main Results

Our technical contributions can be summarized as follows:

- We present a simplified model of Spark by defining SparkLite, a functional programming language in which UDFs are expressed over a decidable theory.
- We identify several interesting classes of SparkLite programs and develop sound, and in certain cases complete, methods for proving program equivalence. (See Table 1.)
- We implemented our approach over Z3 [12], and applied it to several interesting programs taken out of real-life Spark applications. In case the tool fails to verify equivalence, it produces a counterexample of RDD elements which are witnesses for the difference between the programs. This counterexample is guaranteed to be real for programs which have a complete verification method, and can help understand the differences between these programs.

1.2 Overview

We now provide an informal overview on our verification technique, and its scope. Our main tool for showing equivalence of Spark programs is reducing the equivalence of Spark programs to the validity of a formula in Presburger

P1(R: $RDD_{\texttt{Int}}$)**:**	**P2**(R: $RDD_{\texttt{Int}}$)**:**
$R'_1 = \texttt{map}(\lambda x.2 * x)(R)$	$R'_2 = \texttt{filter}(\lambda x.x \geq 50)(R)$
$R''_1 = \texttt{filter}(\lambda x.x \geq 100)(R'_1)$	$R''_2 = \texttt{map}(\lambda x.2 * x)(R'_2)$
`return` R''_1	`return` R''_2

$$\forall x.ite(2 * x \geq 100, 2 * x, \bot) = 2 * ite(x \geq 50, x, \bot)\,.$$

Fig. 1. Equivalent Spark programs and a formula attesting for their equivalence.

arithmetic, which is a decidable theory [14,23]. Our practical tool uses Z3. This is surprising since Spark programs contain implicit loops and higher-order functions. Our insight is that in many cases these loops are simple enough to allow reasoning by using universal formulae for programs without aggregations and extract simple inductive invariants for programs with aggregation. We now demonstrate that through a series of simple examples.

A simple example. Figure 1 shows two equivalent Spark programs and the formula that we use for checking their equivalence. The programs accept an RDD of integer elements. They return another RDD where each element is twice the value of the original element, for elements which are at least 50. The programs operate differently: $P1$ first multiplies, then filters, while $P2$ goes the other way around. `map` and `filter` are operations that apply a function on each element in the RDD, and yield a new RDD. For example, let RDD R be the bag $R = \{\!\{2, 2, 103, 64\}\!\}$ (note that repetitions are allowed). R is an input of both $P1$ and $P2$. The `map` operator in the first line of $P1$ produces a new RDD, R'_1, by doubling every element of R, i.e., $R'_1 = \{\!\{4, 4, 206, 128\}\!\}$. The `filter` operator in the second line generates RDD R''_1, containing the elements of R'_1 which are at least 100, i.e., $R''_1 = \{\!\{206, 128\}\!\}$. The second program first applies the filter operator, producing an RDD R'_2 of all the elements in R which are smaller than 50, resulting in the RDD $R'_2 = \{\!\{103, 64\}\!\}$. $P2$ applies the map operator to produce RDD R''_2 which contains the same elements as R''_1. Hence, both programs return the same value.

To verify that the programs are indeed equivalent, we encode them symbolically using formulae in first-order logic, such that the question of equivalence boils down to proving the validity of a formula. In this example, we encode $P1$ as: $ite(2 * x \geq 100, 2 * x, \bot)$, and $P2$ as: $2 * ite(x \geq 50, x, \bot)$, where ite denotes the if-then-else operator and $\bot$ is used to denote that the element as been removed. The variable symbol x can be thought of as an arbitrary element in the dataset R, and the terms on the left and right side of the equality sign record the effect of $P1$ and $P2$, respectively, on x. The constant symbol $\bot$ records the deletion of an element due to not satisfying the condition checked by the `filter` operation. The formula whose validity attests for the equivalence of $P1$ and $P2$ is thus: $\forall x.ite(2 * x \geq 100, 2 * x, \bot) = 2 * ite(x \geq 50, x, \bot)$. Here, the formula is expressed in a decidable extension of Presburger Arithmetic, which allows to handle $\bot$ (see Appendix A), thus its validity can be decided.

This example points out an important property of the *map* and *filter* operations, namely, their *locality*: they handle every element separately, with no regard

$$discount = \lambda(prod, p).(prod, p - 20)$$
$$min2 = \lambda A, (x, y).if\ A < y\ then\ A\ else\ y$$

P3(R: $RDD_{\texttt{Prod}\times\texttt{Int}}$)**:**
$minP = \texttt{fold}(+\infty, min2)(R)$
`return` $minP \geq 100$

P4(R: $RDD_{\texttt{Prod}\times\texttt{Int}}$)**:**
$R' = \texttt{map}(\lambda(prod, p).discount((prod, p)))(R)$
$minDiscountP = \texttt{fold}(+\infty, min2)(R')$
`return` $minDiscountP \geq 80$

$$\begin{pmatrix} prod' = prod \wedge p' = p - 20 & assumptions \\ \wedge M_2 = ite(M_1 < p, M_1, p) \wedge M_2' = ite(M_1' < p', M_1', p') & assumptions \end{pmatrix}$$
$$\implies (+\infty \geq 100 \iff +\infty \geq 80) \quad base\ case$$
$$\wedge((M_1 \geq 100 \iff M_1' \geq 80) \implies (M_2 \geq 100 \iff M_2' \geq 80))\ induction\ step$$

Fig. 2. Equivalent Spark programs with aggregations and an inductive equivalence formula. Variables $prod, p, prod', p', M_1, M_1', M_2, M_2'$ are universally quantified.

to its multiplicity (the number of duplicates it has in the *RDD*) or the presence of other elements. Thus, we can symbolically represent the effect of the program on any *RDD* by encoding its effect on a single arbitrary element.

Usage of inductive reasoning. We also handle programs that use aggregations in a limited way. For example, we give a sound verification technique for Agg^1, which is a class of programs that use a single `fold` operation and returns a primitive value. Interestingly, we show that even in this limited class, deciding equivalence is undecidable in general. Figure 2 contains a slightly more interesting example of two equivalent Spark programs. The programs operate over an RDD of pairs (product IDs, price). The programs check if the minimal price of the RDD is at least 100. The second program does it by subtracting 20 from each price in the RDD and comparing the minimum to 80. $P3$ computes the minimal price in R using `fold`, and then returns *true* if it is at least 100 and *false* otherwise. $P4$ first applies *discount* to every element, resulting in a temporary *RDD* R', and then computes the minimum of R'. It returns *true* if the minimum is at least 80, and *false* otherwise. The `fold` operation combines the elements of an RDD by repeatedly applying a UDF. `fold` cannot be expressed in first order terms. Thus, we use induction to verify that two `fold` results are equal. Roughly speaking, the induction leverages the somewhat *local* nature of the `fold` operation, specifically, that it does not track *how* the temporarily accumulated value is obtained: Note that the elements of R' can be expressed by applying the *discount* function on the elements of R. Thus, intuitively, we can assume that in both programs, `fold` iterates on the *input* RDD R in the same order. (It is permitted to assume a particular order because the applied UDFs must be commutative for the `fold` to be well-defined [18].) The base of the induction hypothesis checks that the programs are equivalent when the input RDDs are empty, and the induction step verifies the equivalence is retained when we apply the `fold`'s UDF on some arbitrary accumulated value and an element coming from each input RDD. In our example, when the RDDs are empty both programs return *true*. (The `fold` operation returns $+\infty$.) Otherwise, we assume that after n prices checked, the

minimum M_1 in $P3$ is at least 100 iff the minimum M'_1 in $P4$ is at least 80. The programs are equivalent if this invariant is kept after checking the next product and price $((prod, p), (prod', p'))$ giving updated intermediate values M_2, M'_2.

Completeness of the inductive reasoning. In the example in Figure 2 we use a simple form of induction by essentially proving that two higher-order operations are equivalent iff they are equivalent on every element. Such an approach is incomplete. We now show an example for incompleteness, and a modified verification formula which is complete for a designated set of Spark programs called $AggPair^1_{sync}$, which is a subclass of Agg^1. In Figure 3, the program was rewritten by using $=$ instead of $\geq$. In this example the programs are equivalent. We show both the "naïve" formula, similar to the formula from Figure 2, and a revised version of it. We will explain shortly how the revised formula is obtained. The naïve formula is not valid, since it requires that the returned values be equivalent ignoring the history of applied `fold` operations generating the intermediate values M_1, M'_1. In particular, for $M_1 = 60, M'_1 = 120, p = 100$ we get a counterexample leading to the wrong conclusion that the programs are potentially inequivalent. However, we note it is actually impossible to reach $M_1 = 60, M'_1 = 120$ in these programs.

In many of the examples, the operations used in the `fold` are somewhat restricted. This can be leveraged for more complete treatment of equivalence verification. A natural property which we observed, is the following closure property, for functions $f_1, f_2, \varphi_1, \varphi_2$ and an initial value A:

$$\begin{aligned} \forall x, y. \exists a.\ & f_1(f_1(A, \varphi_1(x)), \varphi_1(y)) = f_1(A, \varphi_1(a)) \\ \wedge\ & f_2(f_2(A, \varphi_2(x)), \varphi_2(y)) = f_2(A, \varphi_2(a)) \end{aligned} \tag{1}$$

In our example, $\min(\min(+\infty, x), y) = \min(+\infty, a)$, and $\min(\min(+\infty, x - 20), y - 20) = \min(+\infty, a - 20)$, for $a = \min(x, y)$. The reader may be concerned how this closure property can be checked. Interestingly, for formulas in Presburger arithmetic, an SMT solver can check this property in a sound and complete way.

We utilized the above closure property by observing that any pair of intermediate results can be expressed as single applications of the UDF. Surely any M_1 must have been obtained by repeating applications of the form $f_1(f_1(\cdots))$, and similarly for M'_1 with $f_2(f_2(\cdots))$. Therefore, in the revised formula, instead of quantifying on any M_1, M'_1, we quantify over the argument a to that single application, and introduce the assumption incurred by Equation (1). We can then write an induction hypothesis that holds iff the two fold operations return an equal result.

Decidability. Table 1 characterizes the programs for which our method is applicable together with the strength of the method. The example program in Figure 1 is representative of programs that belong to the $NoAgg$ class of programs, defined as programs without `fold` operations, for which we have a decision procedure showing it is decidable. Programs with a `fold` operation belong to one of the four classes pertaining to them: $Agg^1, AggPair^1_{sync}, Agg^1_R, Agg^n$. Equivalence in Agg^1 is undecidable, and the result is extended naturally to Agg^1_R and Agg^n. On the other hand, $AggPair^1_{sync}$ is decidable and we provide a decision procedure.

$$min2 = \lambda A, (x, y).\ ite(A < y, A, y)$$

P5(R: $RDD_{\texttt{Prod}\times\texttt{Int}}$)**:**	**P6**(R: $RDD_{\texttt{Prod}\times\texttt{Int}}$)**:**
$minP = \texttt{fold}(+\infty, min2)(R)$	$R' = \texttt{map}(\lambda(prod, p).discount((prod, p)))(R)$
`return` $minP = 100$	$minDiscountP = \texttt{fold}(+\infty, min2)(R')$
	`return` $minDiscountP = 80$

Naive formula:

$$\begin{pmatrix} prod' = prod \wedge p' = p - 20 & assumptions \\ \wedge\ M_2 = ite(M_1 < p, M_1, p) \wedge M_2' = ite(M_1' < p', M_1', p') & assumptions \end{pmatrix}$$
$$\implies (+\infty = 100 \iff +\infty = 80) \quad base\ case$$
$$\wedge((M_1 = 100 \iff M_1' = 80) \implies (M_2 = 100 \iff M_2' = 80))\ induction\ step$$

Revised formula:

$$\begin{pmatrix} prod' = prod \wedge p' = p - 20 & assumptions \\ \left.\begin{array}{r} \wedge\ a = (a_0, a_1) \wedge M_1 = ite(+\infty < a_1, +\infty, a_1) \\ \wedge\ M_1' = ite(+\infty < a_1 - 20, +\infty, a_1 - 20) \end{array}\right\} & \begin{array}{r} closure \\ property \end{array} \\ \wedge\ M_2 = ite(M_1 < p, M_1, p) \wedge M_2' = ite(M_1' < p', M_1', p') & assumptions \end{pmatrix}$$
$$\implies (+\infty = 100 \iff +\infty = 80) \quad base\ case$$
$$\wedge((M_1 = 100 \iff M_1' = 80) \implies (M_2 = 100 \iff M_2' = 80))\ induction\ step$$

Fig. 3. Equivalent Spark programs for which a more elaborate induction is required. All variables are universally quantified.

The programs in Figures 2 and 3 belong to $AggPair^1_{sync}$. Note that, formally, $AggPair^1_{sync}$ contains pairs of programs, and thus in fact is a *relation* between programs. Deciding whether a pair of programs belong to $AggPair^1_{sync}$ is done by checking the validity of Equation (1), which is expressed in Presburger arithmetic.

1.3 Limitations

Several limitations were imposed on our model of SparkLite. Some of the limitations can be lifted easily, such as self-products support, by-key support, and reduce operator addition. We also ignore certain aspects of Spark, such as distribution and partitioning of the data, and ordering of the elements in an RDD. We also omit other relational algebra operators such as *union* and *subtract*. We are encouraged by the fact that this class of programs covers several interesting programs that we found in the Internet and in textbooks, see Section 4. Presburger arithmetic can be implemented with solvers like Cooper's algorithm [11]. For simplicity we use Z3 which does not support full Presburger arithmetic, but support the fragment of Presburger arithmetic used in this paper. Z3 also supports uninterpreted functions which are useful to prove equivalence of other Spark programs, which are beyond the scope of this paper.

First-Order Functions	*Fdef*	$::= \texttt{def}\ \boldsymbol{f}\ =\ \lambda\overline{\boldsymbol{y}:\boldsymbol{\tau}}.\, e:\boldsymbol{\tau}$
Second-Order Functions	*PFdef*	$::= \texttt{def}\ \boldsymbol{F}\ =\ \lambda\overline{\boldsymbol{x}:\boldsymbol{\tau}}.\, \lambda\overline{\boldsymbol{y}:\boldsymbol{\tau}}.\, e:\boldsymbol{\tau}$
Function Expressions	f	$::= \boldsymbol{f} \mid \boldsymbol{F}(\overline{e})$
RDD Expressions	μ	$::= \texttt{cartesian}(\mu, \mu') \mid \texttt{map}(f)(\mu) \mid \texttt{filter}(f)(\mu) \mid \boldsymbol{r}$
General Expressions	η	$::= e \mid \mu \mid \texttt{fold}(\boldsymbol{e}, f)(\mu)$
Let expressions	E	$::= \mathit{let}\ \boldsymbol{x} = \eta\ \mathit{in}\ E \mid \eta$
Programs	*Prog*	$::= \boldsymbol{P}(\overline{\boldsymbol{r}:\mathit{RDD}_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v}:\boldsymbol{\tau}})\ =\ \overline{\mathit{Fdef}}\quad \overline{\mathit{PFdef}}\quad E$

Fig. 4. Syntax for SparkLite

2 The SparkLite language

In this section, we define SparkLite, a simple functional programming language which allows to use Spark's *resilient distributed datasets* (*RDDs*) [28].

Preliminaries. We denote a (possibly empty) sequence of elements coming from a set X by $\overline{X}$. An *if-then-else* expression $ite(p, e, e')$ denotes an expression which evaluates to e if p holds and to e' otherwise. A *bag* m over a domain X is a multiset, i.e., a set which allows for repetitions, with elements taken from X. We denote the *multiplicity* of an element x in bag m by $m(x)$, where for any x, either $0 < m(x)$ or $m(x)$ is undefined. We write $x \in m$ as a shorthand for $0 < m(x)$. We write $\{\!\{x; n(x) \mid x \in X \wedge \phi(x)\}\!\}$ to denote a bag with elements from X satisfying some property ϕ with multiplicity $n(x)$, and omit the conjunct $x \in X$ if X is clear from context. We denote the *size* (number of elements) of a set X by $|X|$ and that of a bag m of elements from X by $|m|$, i.e., $|m| = \Sigma_{x \in X}\, ite(x \in m, m(x), 0)$. We denote the empty bag by $\{\!\{\}\!\}$.

Syntax of SparkLite The syntax of SparkLite is defined in Figure 4. SparkLite supports two primitive types: *integers* (`Int`) and *booleans* (`Boolean`). On top of this, the user can define *record types* $\boldsymbol{\tau}$, which are Cartesian products of primitive types, and *RDD*s: $\mathit{RDD}_{\boldsymbol{\tau}}$ is (the type of) bags containing elements of type $\boldsymbol{\tau}$. We refer to primitive types and tuples of primitive types as *basic types*, and, by abuse of notation, range over them using $\boldsymbol{\tau}$. We use e to denote a *basic expression* containing only basic types, written in Presburger arithmetics extended to include tuples in a straightforward way. (See Appendix A.) We range over variables using $\boldsymbol{v}$ and $\boldsymbol{r}$ for variables of basic types and *RDD*, respectively.

A program $\boldsymbol{P}(\overline{\boldsymbol{r}:\mathit{RDD}_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v}:\boldsymbol{\tau}})\ =\ \overline{\mathit{Fdef}}\ \overline{\mathit{PFdef}}\ E$ is comprised of a *header* and a *body*, which are separated by the = sign. The header contains the name of the program ($\boldsymbol{P}$) and the names and types of its input parameters, which may be *RDDs* ($\overline{\boldsymbol{r}}$) or records or primitive types ($\overline{\boldsymbol{v}}$). The body of the program is comprised of two sequences of function declarations ($\overline{\mathit{Fdef}}$ and $\overline{\mathit{PFdef}}$) and the program's *main expression* (E). $\overline{\mathit{Fdef}}$ binds function names $\boldsymbol{f}$ with first-order lambda expressions, i.e., to a function which takes as input a sequence of arguments of basic types and return a value of a basic type. $\overline{\mathit{PFdef}}$ associates function names $\boldsymbol{F}$ with a restricted form of second-order lambda expressions, which we refer to as *parametric functions*. As in the *Kappa Calculus* [16], a parametric

function $\boldsymbol{F}$ receives a sequence of basic expressions and returns a first order function. Parametric functions can be instantiated to form an unbounded number of functions from a single pattern. For example, $\texttt{def addC} = \lambda x\colon \texttt{Int}.\, \lambda y\colon \texttt{Int}.\, x + y\colon \texttt{Int}$ can create any first order function which adds a constant to its argument, e.g., $\texttt{addC}(1) = \lambda x\colon \texttt{Int}.\, 1 + x\colon \texttt{Int}$ and $\texttt{addC}(2) = \lambda x\colon \texttt{Int}.\, 2 + x\colon \texttt{Int}$.

The program's main expression is comprised of a sequence of *let* expression which bind general expressions to variables. A general expression is either a *basic expression* (e), an RDD *expression* (μ) or an *aggregate expression* ($\texttt{fold}(\boldsymbol{e}, f)(\eta)$). The expression $\texttt{cartesian}(\eta, \eta')$ returns the cartesian product of η and η'. The expressions `map` and `filter` generalize the *project* and *select* operators in *Relational Algebra* (*RA*) [1, 8], with *user-defined functions* (*UDFs*): $\texttt{map}(f)(\eta)$ produces an *RDD* by applying the unary UDF f to every element x of η. $\texttt{filter}(f)(\eta)$ evaluates to a copy of η, except that all elements in η which do not satisfy f are removed. The aggregate expression is a generalization of the aggregate operations in SQL, e.g., `SUM` or `AVERAGE`, with *UDFs*: $\texttt{fold}(\boldsymbol{e}, f)(\eta)$ accumulates the results obtained by iteratively applying the binary UDF f to every element x in an RDD η in some arbitrary order together with the accumulated result obtained so far, which is initialized to the *initial element* $\boldsymbol{e}$. If η is empty, then $\texttt{fold}(\boldsymbol{e}, f)(\eta) = \boldsymbol{e}$.

Remarks. Firstly, as is common in functional languages, variables are never reassigned. We assume that the signature of UDFs given to either *map*, *filter*, or *fold* match the type of the RDD on which they are applied. Also, to ensure that the meaning of $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ is well defined, we require, as Spark does [18], that f be a commutative on its second argument, i.e., $\forall x, y_1, y_2. f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$. For clarity, we omit the *let* expressions from examples, and write programs as a sequential composition of variable assignments.

Semantics of SparkLite We define a denotational semantics for SparkLite. From this point on, we assume, without loss of generality, that programs do not contain *let* expressions; if a program does, we first substitute every variable by its definition, resulting in a program containing a single generalized expression defined using input parameters, basic expressions, and RDD operations.

Let P be a SparkLite program and η its main expression. We calculate the meaning of P for a given input *input environment* ρ_0 which maps P's input variables to their values in two steps. Firstly, we extend ρ_0 to include a mapping from function names to their definitions in P. Secondly, we use the augmented ρ_0 to evaluate η according to the rules in Figure 5, which are rather straightforward.

3 Verifying Equivalence of SparkLite Programs

Programs P_1 and P_2 are *equivalent* if (i) they receive the same sequence of formal input parameters, and (ii) for any input environment ρ_0, it holds that $[\![P_1]\!](\rho_0) = [\![P_2]\!](\rho_0)$. Unfortunately, the equivalence of general SparkLite programs is undecidable; Theorem 2 indicates that the problem is undecidable even if we only consider programs containing a single aggregate operation. Thus, in

$$[\![\boldsymbol{r}]\!](\rho_0) = \rho_0(r) \quad [\![\boldsymbol{f}]\!](\rho_0) = \rho_0(\boldsymbol{f}) \quad [\![\boldsymbol{F}(e_1,\ldots,e_n)]\!](\rho_0) = \rho_0(\boldsymbol{F})([\![e_1]\!](\rho_0),\ldots,[\![e_n]\!](\rho_0))$$
$$[\![\texttt{map}(f)(\mu)]\!](\rho_0) = \{\!\{[\![f]\!](\rho_0)(x) \mid x \in [\![\mu]\!](\rho_0)\}\!\}$$
$$[\![\texttt{filter}(f)(\mu)]\!](\rho_0) = \{\!\{x \in [\![\mu]\!](\rho_0) \mid [\![f]\!](\rho_0)(x)\}\!\}$$
$$[\![\texttt{cartesian}(\mu,\mu')]\!](\rho_0) = \{\!\{(x,x') \mid x \in [\![\mu]\!](\rho_0) \wedge x' \in [\![\mu']\!](\rho_0)\}\!\}$$
$$[\![\texttt{fold}(\boldsymbol{e},f)(\mu)]\!](\rho_0) = q_f\big([\![e]\!](\rho_0),[\![\mu]\!](\rho_0)\big) \quad \text{where} \quad q_f(v_0,s) = \begin{cases} v_0 & s = \{\!\{\}\!\} \\ \rho_0(f)\big(q_f(v_0,s'),x\big) & s = \{\!\{x;1\}\!\} \cup s' \end{cases}$$

Fig. 5. Semantics of SparkLite. The (standard) meaning of basic expressions is omitted.

this section, we explore various techniques for verifying program equivalence for restricted classes of SparkLite programs.

3.1 Verifying Equivalence of Programs without Aggregations

We first present a sound and complete technique for verifying the equivalence of *NoAgg* programs, the class of SparkLite programs that do not use aggregations.

Program terms. The first step of our technique is the construction of *program terms*: Given a program P with main expression η, we generate an (intermediate) *uninterpreted program term* $\phi(P)$ which, roughly speaking, reflects the effect of the program on arbitrary elements taken from its input RDDs. $\phi(P)$ refers to functions by name. We produce the *program term* of P, denoted by $\Phi(P)$, by applying beta reduction according to the definition of every function in P.

We obtain the uninterpreted program term of P by applying the translation function ϕ, shown in Figure 6, on P's main expression. ϕ recursively traverses the expression and generates a first-order logical term over the vocabulary of built-in operations and UDFs defined in P. The base cases of the recursion are plain arithmetic expressions e and RDD variables r. ϕ acts as the identity function for expressions and replaces any input RDD variable r with a fresh variable $\mathbf{x}_r$ which we refer to as r's *representative variable*. Strictly speaking, the latter transformation is not necessary technically. It is done merely to emphasize that $\mathbf{x}_r$ is used to represent an arbitrary element of r and not a whole RDD. Translation of an RDD operation produces a term corresponding to the application of its UDF on a single RDD element: A $\texttt{map}(f)(\mu)$ operation is translated into a function application. A $\texttt{filter}(f)(\mu)$ operation is translated to an *ite* expression which returns the program term of μ on the *then* branch and $\perp$ on the *else* branch. We use $\perp$ to denote the *non-existing* value. $\perp$ cannot be used in a program, and appears only in formulae to record the effect of filtering an element out of an RDD. The cartesian product is translated to a pair of uninterpreted program terms pertaining to its arguments. (For now, ignore the translation of `fold` operations.)

Example 1. Consider the main expression $\eta = \texttt{filter}(geq(100))(\texttt{map}(double)(R))$ of the program $P1'$ obtained by inlining the *let* expressions in program $P1$ (see Section 1), defining the doubling function as $double = \lambda x.2 * x$, and instantiating the parametric function $geq = \lambda y.\, \lambda x.x \geq y$ to act as the condition of the filter. The uninterpreted program term of $P1'$ is $\phi(\eta) = ite(geq(100)(double(\mathbf{x}_R)), double(\mathbf{x}_R), \perp)$

$$
\begin{array}{llll}
\phi(e) & = e & \phi(\texttt{map}(f)(\mu)) & = f(\phi(\mu)) \\
\phi(r) & = \mathbf{x}_r & \phi(\texttt{filter}(f)(\mu)) & = ite(f(\phi(\mu)) = tt, \phi(\mu), \bot) \\
\phi(\texttt{fold}(f,e)(\mu)) & = [\phi(\mu)]_{e,f} & \phi(\texttt{cartesian}(\mu_1,\mu_2)) & = (\phi(\mu_1), \phi(\mu_2))
\end{array}
$$

Fig. 6. A translation of a general expression to an uninterpreted program term.

bool EqNoAgg(P_1, P_2):
if $\mathit{RepVarSet}(\phi(P_1)) = \mathit{RepVarSet}(\phi(P_2))$ **then**
  return Valid$(\forall \overline{x}.\Phi(P_1)[\overline{x}/FV(\phi(P_1))] = \Phi(P_2)[\overline{x}/FV(\phi(P_2))])$
else
  return Valid$(\forall \overline{x}.(\Phi(P_1)[\overline{x}/FV(\phi(P_1))] = \bot \wedge \Phi(P_2)[\overline{x}/FV(\phi(P_2))] = \bot))$

Fig. 7. Algorithm for deciding equivalence for $NoAgg$ programs. $FV(\phi(P))$ denotes the set of free variables in the program term of P. $\mathit{RepVarSet}(\phi(P))$ denotes the subset of $FV(\phi(P))$ comprised of representative variables.

and its program term is $\Phi(P1') = ite(2 * \mathbf{x}_R \geq 100, 2 * \mathbf{x}_R, \bot)$. Intuitively, we can learn how $P1'$ affects every element of, e.g., input RDD $\{\!\{2, 2, 103, 64\}\!\}$, by treating $\Phi(P1')$ as a "function" of $\mathbf{x}_R$ and "applying" it to 2, 2, 103, and 64.

Verifying equivalence. In Figure 7 we present an algorithm for deciding the equivalence of $NoAgg$ programs. The algorithm first checks if both uninterpreted program terms are defined using the same representative variables. If they are the same, the algorithm returns that the two programs are equivalent *iff* the program terms of both programs evaluate to the same value, including $\bot$, for any possible assignment of their representative variables. Otherwise, the algorithm returns that two programs are equivalent *iff* they always return an empty RDD, in which case they are trivially equivalent. This check is done by asking the solver to determine whether both program terms are equivalent to $\bot$, i.e., no matter which elements we pick from the RDD, the program would filter them out.

Theorem 1 (Decidability of the $NoAgg$ class). *Let P_1 and P_2 be $NoAgg* SparkLite *program. $P1$ is equivalent to $P2$ iff* **EqNoAgg**$(P1, P2)$ *returns true.*

The proof of the theorem (see Appendix C) is based on three key observations: (i) Let R be the RDD returned by a $NoAgg$ program P. For any element x, $x \in R$ if and only if x can be generated by considering input *singleton* RDDs, containing the single elements of the input RDDs that contributed to the generation of x. This allows to determine that the RDDs produced by two programs contain the same elements by verifying that their program terms, as first-order terms, are equivalent. (ii) Because we forbid self-joins, the multiplicity of x in R can be calculated by summing the product of the multiplicities of every possible combination of single elements taken fromm the input RDDs that can generate x. This allows to determine that programs having equivalent program terms also agree on the multiplicities of elements in the output they produce by syntactically checking that the sets of representative variables used by the uninterpreted program term. (iii) A program always returns an empty RDD iff its program term evaluates to $\bot$ for any assignment for its free variables.

Example 2. Let $\texttt{cartesian}(\texttt{filter}(geq(100))(R_0), \texttt{map}(double)(R_1))$ be the main expression of a program P. Thus, the program term of P is $(ite(\mathbf{x}_{R_0} \geq 100, \mathbf{x}_{R_0}, \bot), 2*\mathbf{x}_{R_1})$. An arbitrary element (a, b) can appear in the RDD R that P returns if $a \geq 100$ and b is even. Moreover, (i) we can find (a, b) in the output only if $a \in R_0$ and $b/2 \in R_1$, and (ii), we can construct singleton input RDDs that can generate any such pair, provided $a \geq 100$ and b is even. Furthermore the multiplicity of an element (a, b) in the output that P produces is $R_0(a) * R_1(b/2)$, if $a \geq 100$ and b is even. If (a, b) is not of that form, then $\Phi(P)(a, b)$ evaluates to $\bot$.

The reason we use the uninterpreted program terms to check that the sets of representative variables agree, and not, e.g., the program terms, is that programs may ignore the values found in the input RDDs. As a result, even if the programs have identical program terms, they may produce the same elements but with different multiplicities, as the following example illustrates.

Example 3. Let $P5(R_0, R_1) = one = \lambda x.1\ \texttt{map}(one)(R_0)$ and $P6(R_0, R_1) = one = \lambda x.1\ \texttt{map}(one)(R_1)$ be SparkLite programs. $P5$ and $P6$ have the same program term (the constant 1). Thus, the RDDs they produce would only contain 1s. However, $P5$ generates a 1 for every element in R_0 whereas $P6$ generate a 1 for every element in R_1. Hence, if the size of R_0 is different from that of R_1, the two programs would produce different outputs. Note that the uninterpreted program terms of $P5$ and $P6$ are $one(\mathbf{x}_{R_0})$ and $one(\mathbf{x}_{R_1})$, respectively, and that they have the same program term, namely 1. Our algorithm would find out that $RepVarSet(\phi(P5)) = \{\mathbf{x}_{R_0}\} \neq \{\mathbf{x}_{R_1}\} = RepVarSet(\phi(P6))$, and would determine that the programs are not equivalent.

Indeed, non-empty RDDs can be equal only if their uninterpreted terms have equal sets of representative variables.

Lemma 1. *Let P and P' be $NoAgg$ programs such that at least one of them does not always produce an empty RDD. If $RepVarSet(\phi(P)) \neq RepVarSet(\phi(P'))$ then the programs are not equivalent.*

Note that Lemma 1, proven in Appendix B, does not apply when the two programs always produce the empty RDD. Indeed, if the two programs always filters out the elements they inspect, it does not matter which RDDs their uninterpreted program terms use. This is the reason we have to take special care about programs which always return an empty RDD.

The underlying theory. **EqNoAgg** is sound regardless of the kind of basic expressions that programs can use, provided we have a sound technique for verifying validity of the generated formulae. It is complete whenever these formulae are in a decidable theory. Appendix A includes a description of a simple decidable extension to Presburger arithmetic which serves as the underlying theory of the formulae generated for SparkLite programs.

3.2 Verifying Equivalence of SparkLite Programs with Aggregation

In this section, we adapt our verification technique to handle programs containing aggregations. We focus on programs in classes Agg^1 and $AggPair^1_{sync}$ which have

a single aggregation operation. For space reasons, we relegate to Appendix G the discussion of handling programs in classes Agg^1_R and Agg^n, for which we provide sound techniques for verifying program equivalence by generalizing the technique we use for Agg^1 programs.

Program terms for aggregations. We first extend our notion of (uninterpreted) program terms to reflect the presence of `fold` operations. The resulting terms are no longer legal terms in first order logic. Thus, we cannot use them directly in formulae. Instead, we extract out of them a set of formulae whose validity, intuitively, amounts to the establishment of an inductive invariant regarding the effect of `fold` operations.

The construction of the uninterpreted program terms for `fold` operations is shown in Figure 6. We are using a special operator $[\phi(\mu)]_{i,f}$, where $\phi(\mu)$ is the uninterpreted term pertaining to the RDD being folded, i is the initial value, and f is the fold function. We refer to $[\phi(\mu)]_{i,f}$ as an aggregate term. Intuitively, an aggregate operator indicates that calculating the effect of the `fold` requires iterating over all the elements of μ. Clearly, the translation of `fold` cannot be masquerade as a first-order term.

Verifying equivalence of Agg^1 programs Arguably, the simplest class of programs with aggregations is the class of programs which return a primitive expression which depends on the result of the aggregation operation. Technically, a SparkLite program P is in class Agg^1 if its uninterpreted program term is of the form $g[[\phi(\mu)]_{i,f}/v]$, where g is an expression in Presburger Arithmetic which has a single free variable v and μ is an RDD expression which does not include `fold` operations, i.e., $\phi(P)$ can be obtained by substituting v with the aggregate term pertaining to the application of a `fold` operation on μ. In the following, we use the functional notation $g(x)$ as a shorthand for $g[x/v]$.

Frustratingly, the following theorem, proven in Appendix D via a reduction from the halting problem for 2-counter machines, indicates that verifying program equivalence remains undecidable even if we only consider this kind of programs.

Theorem 2. *The problem of deciding whether two arbitrary* Agg^1 SparkLite *programs are equivalent is undecidable.*

Recall that in Section 1.2 we demonstrated the use of a sound method for verifying equivalence of Agg^1 programs by proving that $P3$ and $P4$ (see Figure 2) are equivalent using inductive reasoning. The following lemma, proven in Appendix E, formalizes the method.

Lemma 2 (Sound method for verifying equivalence of Agg^1 programs). *Let P_1 and P_2 be Agg^1 programs, whose uninterpreted program terms are $\phi(P_1) = g_1([\phi(\mu_1)]_{i_1,f_1})$ and $\phi(P_2) = g_2([\phi(\mu_2)]_{i_2,f_2})$, respectively, where $f_1 = \lambda x, y.\, e_1$*

and $f_2 = \lambda x, y.\, e_2$. *$P_1$ and P_2 are equivalent if the following three conditions hold:*

$$RepVarSet(\varphi_1) = RepVarSet(\varphi_2) \tag{2}$$

$$\textbf{valid}(g_1(i_1)[\overline{v}/FV(\phi(P_1))] = g_2(i_2)[\overline{v}/FV(\phi(P_2))]) \tag{3}$$

$$\textbf{valid}(\forall \overline{v}, M_1, M_2.\, g_1(M_1)[\overline{v}/FV(\phi(P_1))] = g_2(M_2)[\overline{v}/FV(\phi(P_2))] \implies \tag{4}$$
$$g_1(e_1[M_1/x, \Phi(\mu_1)/y])[\overline{v}/FV(\phi(P_1))] = g_2(e_2[M_2/x, \Phi(\mu_2)/y])[\overline{v}/FV(\phi(P_2))])$$

Intuitively, Equations (3) and (4) formalize the concept of inductive reasoning described in Section 1.2 for the base of the induction and the induction step, respectively. Equation (2) requires that the free variables of the folded RDD expressions use the same representative variables. Recall that in **NoAggEq**, this check ensured that the produced RDDs agree on the multiplicities of elements. Here, we use this check for a similar purpose: It ensures that the two `fold` operations iterate over RDDs of the same size. Note that we do not require that the RDD folded by the two programs be equivalent. However, in Equation (4) we still use the fact that every element in the folded RDD can be produced by instantiating its program terms with elements from the input RDDs.

Complete verification techniques for subclasses of Agg^1 There are several cases in which one or more of the requirements of Lemma 2 are not satisfied, yet the aggregate terms are equivalent. However, some of these cases can be identified and subsequently have equivalence verified using other methods. As an appetizer, we consider a simple "corner case", where the `fold` UDF function f always returns the initial value. (For example, a program might have a trivial fold if f always returns its first argument or if the folded RDD is always empty.) More formally, let P_j, for $j = 1, 2$, be Agg^1 programs such that $\phi(P_j) = g_j([\phi(\mu_j)]_{i_j, f_j})$ and $f_j = \lambda x, y.\, e_j$. We say that P_j uses a *trivial fold* if the formula $\forall \overline{v}.\, (i_j = e_j)[i_j/x, \Phi(\mu_j)/y])[\overline{v}/FV(\phi(P_j))]$ is valid. (Note that it is possible to decide whether the above formula is valid or not.) Establishing the equivalence of such programs merely requires that Equation (3) holds, i.e., that $\forall \overline{v}.\, (g_1(i_1) = g_2(i_2))[\overline{v}/FV(\phi(P_j))]$ is valid.

In Section 1.2 we showed that although programs $P5$ and $P6$ do not satisfy the requirements of Lemma 2, we can verify their equivalence using a more specialized verification technique targeting the $AggPair^1_{sync}$ subclass of programs, for which we have a sound and complete technique for verifying program equivalence. The $AggPair^1_{sync}$ class is characterized by a decidable semantic property of the `fold` UDF, the aggregate terms, and the initial values, which we refer to as *collapsing*. Intuitively, the collapsing property states that the value obtained by any two consecutive application of the `fold` UDF can be obtained using a single application. For example, if the UDF is $sum = \lambda x, y.\, x + y$ and the initial value is 0, then the result obtained by applying sum consecutively on any two elements a and b can also be obtained by applying sum on $a + b$. Note that if a program enjoys the collapsing property it is possible to collapse any sequence of iterated applications of the `fold` UDF starting from the initial value, using a single application of the UDF.

To apply our complete verification technique we require that the program have collapsible aggregation in *synchrony*; it should be possible to construct the

value used to collapse two consecutive application of the `fold` UDF on elements in the folded RDD using the same input in both programs.

We say that two programs belong to the class $AggPair^1_{sync}$ if for both programs, it is possible to collapse a sequence of iterated applications of the fold UDF starting from the initial value, using the same element in both programs.

Definition 1 (The $AggPair^1_{sync}$ class). *Let there be two Agg^1 programs P_1, P_2 with equal signatures, whose program terms are $g_j([\phi(\mu_j)]_{i_j,f_j})$ for $j = 1, 2$, and where $f_1 = \lambda x, y.\, e_1$ and $f_2 = \lambda x, y.\, e_2$. We say that $\langle P_1, P_2 \rangle \in AggPair^1_{sync}$ if:*

$$RepVarSet(\varphi_1) = RepVarSet(\varphi_2) \tag{5}$$

$$\begin{aligned}\forall \bar{v_1}, \bar{v_2}.\exists \bar{v}'.e_1[(e_1[i_1/x, \Phi(\mu_1)[\bar{v_1}/FV(\phi(\mu_1))]/y)/x, \Phi(\mu_1)[\bar{v_2}/FV(\phi(\mu_1))]/y] \\ = e_1[i_1/x, \Phi(\mu_1)[\bar{v}'/FV(\phi(\mu_1))]/y] \\ \wedge\, e_2[(e_2[i_2/x, \Phi(\mu_2)[\bar{v_1}/FV(\phi(\mu_2))]/y)/x, \Phi(\mu_2)[\bar{v_2}/FV(\phi(\mu_2))]/y] \\ = e_2[i_2/x, \Phi(\mu_2)[\bar{v}'/FV(\phi(\mu_2))]/y]\end{aligned} \tag{6}$$

Interestingly, for this class, equivalence is decidable.

Theorem 3 (Equivalence in $AggPair^1_{sync}$ is *decidable*). *Let P_1 and P_2 be Agg^1 programs as in Lemma 2. Assume that P_1 and P_2 are a pair of $AggPair^1_{sync}$ programs. P_1 and P_2 are equivalent if and only if the following holds:*

$$\mathbf{valid}(g_1(i_1)[\bar{v}/FV(\phi(P_1))] = g_2(i_2))[\bar{v}/FV(\phi(P_2))] \tag{7}$$

$$\mathbf{valid}\Big(\begin{array}{l}\forall \bar{v}, \bar{w}, M_1, M_2.\, \big(M_1 = e_1[i_1/x, (\Phi(\mu_1)[\bar{w}/FV(\phi(\mu_1))])/y] \wedge \\ \qquad\qquad M_2 = e_2[i_2/x, (\Phi(\mu_2)[\bar{w}/FV(\phi(\mu_2))])/y]\big) \implies Ind\end{array}\Big)$$

$$\begin{aligned}where\ Ind = \big(g_1(M_1) = g_2(M_2) \implies \\ g_1(e_1[M_1/x, \Phi(\mu_1)/y]) = g_2(e_2[M_2/x, \Phi(\mu_2)/y])\big)[\bar{v}/FV(\phi(P_1))]\end{aligned} \tag{8}$$

Note that checking if two programs P_1, P_2 belong to $AggPair^1_{sync}$ involves solving an additional formula (Equation (6)).[4]

4 Prototype Implementation

We developed a prototype implementation verifying the equivalence of Spark programs. The tool is written in Python 2.7 and uses the Z3 Python interface to prove formulas. We ran our experiments on a 64-bit Ubuntu host with a quad core 2.30 GHz Intel Core i5-6200U processor, with 8GB memory.

The tool accepts pairs of Spark program written using the Python interface, determines the class of SparkLite program they belong to, and verifies their equivalence using the appropriate method. The main technical challenge is the symbolic analysis of UDFs, given as pure Python code. For simplicity, we limit the tool to handle programs with up to two aggregations.

[4] For brevity of notations, it is not mentioned that $\bar{v}$ and $\bar{w}$ should assign the same values to non-RDD parameters of $P1$ and $P2$. The same applies to the definition of the $AggPair^1_{sync}$ class.

A total of 19 test-cases of both equivalent and non-equivalent instances were tested, including all the examples from this paper. The examples, listed in Figure 9 in Appendix H, were inspired by real Spark uses taken from [18, 27] and online resources (e.g., open-source Spark clients). They include join optimizations, different aggregations, and various UDFs. For each instance, the tool either verifies that the given programs are equivalent, or produces a counterexample, that is, an input for which the programs produce different outputs. All examples were checked in less than 0.2 seconds.

5 Related Work

This paper bridges the areas of databases and programming languages. The problem considered (i.e., determining equivalence of expressions accessing a dataset) is a classic topic in database theory. The solution approach (i.e., translation into a symbolic representation in a decidable theory) is one that is often employed in the programming language community. In this section we discuss related work from both of these areas.

Query containment and equivalence were first studied in the seminal work [4]. This work was extended in numerous papers, e.g., [19] for queries with inequalities and [7] for acyclic queries. Of most relevance to this paper are the extensions to queries evaluated under bag and bag-set semantics [6], and to aggregate queries, e.g., [9, 10, 15]. The latter papers consider specific aggregate functions, such as min, count, sum and average, or aggregate functions defined by operations over abelian monoids. Equivalence is characterized in terms of special types of homomorphisms or by considering a finite set of canonical databases.

In the field of verification and programming languages there were several works addressing relational algebra operators. For example, El Ghazi et al. [13] took the SMT solver approach to verify relational constraints in Alloy [17]. Smith and Albarghouthi [26] presented an algorithm for synthesizing Spark programs by analyzing user-provided examples fitted into higher-order sketches (map followed by reduce, etc.) and a set of UDFs.

6 Conclusion and Future Work

To conclude, we saw that the problem of checking query equivalence (where queries were written as programs in the SparkLite language), can be modeled with formulas in a decidable fragment of first-order logic. We showed that in the presentation of a query equivalence instance as a formula, the solver for the formula is capable of proving equivalence of programs without aggregation (Theorem 1). Furthermore, we provided a classification of programs with aggregations, and presented a sound method for equivalence testing for a basic class, with generalizations to more intricate classes (Lemmas 2 and Appendix G), and a sound and complete method for one particular non-trivial class of programs (Theorem 3).

We hope the foundations laid in this paper will lead to development of tools that handle formal verification and optimization of clients written in Spark and similar frameworks, by building upon the concepts presented here and extending them to more elaborate structures, such as queries with nested aggregation, unions, and multiple step-inductions for self joins.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. 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.
3. Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.
4. Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM.
5. Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.
6. Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '93, pages 59–70, New York, NY, USA, 1993. ACM.
7. Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211 – 229, 2000.
8. E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
9. Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2), 2007.
10. Sara Cohen, Yehoshua Sagiv, and Werner Nutt. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Logic*, 6(2):328–360, April 2005.
11. David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.
12. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.
13. Aboubakr Achraf El Ghazi and Mana Taghdiri. *Relational Reasoning via SMT Solving*, pages 133–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
14. Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of presburger arithmetic. Technical report, Massachusetts Institue of Technology, Cambridge, MA, USA, 1974.
15. Stéphane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. On the equivalence and rewriting of aggregate queries. *Acta Inf.*, 40(8):529–584, 2004.
16. Masahito Hasegawa. *Decomposing typed lambda calculus into a couple of categorical programming languages*, pages 200–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.
17. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
18. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Inc., 1st edition, 2015.
19. Anthony Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, January 1988.

20. Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Autom. Reasoning*, 36(3):213–239, 2006.
21. Aless Lasaruk and Thomas Sturm. *Effective Quantifier Elimination for Presburger Arithmetic with Infinity*, pages 195–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
22. Derek C. Oppen. A 222pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323 – 332, 1978.
23. M. Presburger. ÃIJber die vollstÃd'ndigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervor. *Comptes Rendus du I congrÃĺs de MathÃľmaticiens des Pays Slaves*, pages 92–101, 1929.
24. Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.
25. Asankhaya Sharma, Shengyi Wang, Andreea Costea, Aquinas Hobor, and Wei-Ngan Chin. *Certified Reasoning with Infinity*. Springer International Publishing, 2015.
26. Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 326–340, New York, NY, USA, 2016. ACM.
27. Josh Wills, Sean Owen, Uri Laserson, and Sandy Ryza. *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, Inc., 1st edition, 2015.
28. 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.
29. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

A A decidable extension of Presburger Arithmetic suitable for SparkLite

Presburger Arithmetic. We consider a fragment of first-order logic (FOL) with equality over the integers, where expressions are written in the rather standard syntax specified in Figure 8.[5] Disregarding the tuple expressions ($(pe\,,\overline{pe})$ and $\boldsymbol{p}_i(e)$) and *ite*, the resulting first-order theory with the usual $\forall$ and $\exists$ quantifiers is called the *Presburger Arithmetic*. The problem of checking whether a sentence in Presburger arithmetic is valid has long been known to be decidable [14, 23], even when combined with Boolean logic [2, 20],[6] and infinities [21, 25].[7] For example, *Cooper's Algorithm* [11] is a standard decision procedure for Presburger Arithmetic.[8]

In this paper, we consider a simple extension to this language by adding a *tuple constructor* $(pe\,,\overline{pe})$, which allows us to create k-tuples, for some $k \geq 1$, of primitive expressions, and a projection operator $\boldsymbol{p}_i(e)$, which returns the i-th component of a given tuple expression e. We extend the equality predicate to tuples in a point-wise manner, and call the extended logical language *Augmented Presburger Arithmetic* (APA). The decidability of Presburger Arithmetic, as well as Cooper's Algorithm, can be naturally extended to APA. Intuitively, verifying the equivalence of tuple expressions can be done by verifying the equivalence of their corresponding constituents.

Proposition 1. *The theory of formulas over $\mathbb{Z}^n$ with terms in the Augmented Presburger Arithmetic is decidable.*

Proof. Let φ be a quantified formula over $\bigcup_n \mathbb{Z}^n$ with terms in Augmented Presburger Arithmetic. We shall translate φ to a formula in Presburger Arithmetic. For any atom $A\colon\ = a = b$, where $a, b \in \mathbb{Z}^k$ for some $k > 0$, we build the following formula: $\bigwedge_{i=1}^{k} p_i(a) = p_i(b)$ and replace it in place of A. In the resulting formula, we assign new variable names, replacing the projected tuple variables: For $a \in \mathbb{Z}^k$ we define $x_{a,i} = p_i(a)$ for $i \in \{1, \ldots, k\}$. Variable quantification extends naturally, i.e. $\forall a$ becomes $\forall x_{a,1}, \ldots, x_{a,k}$, and similarly for $\exists$.

To be compatible with SparkLite's requirements, it will be useful to discuss an extension of APA in which terms are allowed to contain two additional constructs: *ite* expressions, and $\perp$ values. We denote this extension APA$^+$, and show how formulas in APA$^+$ can be converted to APA formulas.

The program terms may contain *ite* and $\perp$ expressions, therefore we need to encode the formulas in APA. We present a translation procedure $\mathcal{N}$ for converting

[5] We assume the reader is familiar with FOL, and omit a more formal description.

[6] Originally, Presburger Arithmetic was defined as a theory over natural numbers. However, its extension to integers and booleans is also decidable. (See, e.g., [2].)

[7] We denote infinities as $+\infty, -\infty$ and extend the underlying domain $\mathbb{Z}$ to hold elements which represent .

[8] The complexity of Cooper's algorithm is $O(2^{2^{2^{pn}}})$ for some $p > 0$ and where n is the number of symbols in the formula [22].

Arithmetic Expression	ae	$::= c \mid v \mid ae + ae \mid -ae \mid c * ae \mid ae \,/\, c \mid ae \,\%\, c$
Boolean Expression	be	$::= \texttt{true} \mid \texttt{false} \mid b \mid e = e \mid ae < ae \mid \neg be \mid be \wedge be \mid be \vee be$
Primitive Expression	pe	$::= ae \mid be$
Basic Expression	e	$::= pe \mid \boldsymbol{v} \mid (pe\,, \overline{pe}) \mid \boldsymbol{p}_i(e) \mid \texttt{ite}(be, e, e)$

c, v, and b denote integer numerals, integer variables, and boolean variables, respectively. % denotes the modulo operator.

Fig. 8. Terms of the Augmented Presburger Arithmetic

APA$^+$ formulas to APA. Let φ be a formula. Following the standard notation of *sub-terms*, *positions*, and *substitutions* in [24],[9] we use $\varphi|_p$ to denote the *sub-term* of φ in a specific *position* p and by $\varphi[r]_p$ the substitution of the sub-term in position p with r. We use this notation to define $\mathcal{N}$. If $\varphi|_p = ite(\varphi_1, \varphi_2, \varphi_3)$, then φ is converted to: $(\varphi_1 \implies \varphi[\varphi_2]|_p) \wedge (\neg\varphi_1 \implies \varphi[\varphi_3]|_p)$. In addition, for every sub-term of the form $\varphi|_p = f(t_1, \ldots, t_n)$, if some t_i is equal (syntactically) to $\bot$, then $\varphi|_p = \bot$, as there is no meaning to evaluating functions on $\bot$ symbols, which represent non-existing RDD elements. Finally, we replace $\bot = \bot, x \neq \bot$ with tt, and $\bot \neq \bot, x < \bot, x \leq \bot, x > \bot, x \geq \bot, x = \bot$ with ff. We define a translation function $\mathcal{N}(\varphi)$, which goes over all positions in φ and performs substitutions as above. For example, $\varphi = (ite(x{>}0, x, \bot){=}\bot)$ is translated to: $((x{>}0 \implies ff) \wedge (x{\leq}0 \implies tt))$. Indeed, both $\varphi, \mathcal{N}(\varphi)$ are true only for $x \leq 0$.

Proposition 2 (φ and $\mathcal{N}(\varphi)$ are equivalent). *For every APA$^+$ formula φ, the APA formula $\varphi' = \mathcal{N}(\varphi)$, received by replacing all ite sub-terms with two implication conjuncts, all function calls with $\bot$ arguments to $\bot$, and all equalities and inequalities containing a $\bot$ symbol with either tt or ff, is equivalent to φ:* $\varphi \iff \mathcal{N}(\varphi)$

B Proof of Lemma 1

Lemma 1 follows directly from the following lemma. Lemma 3 show that in two programs without aggregations, different sets of representative variables of the terms imply the existence of input RDDs for which the two program terms evaluate to different bags, thus they are semantically inequivalent.

Lemma 3. *Let there be two programs $P_1, P_2 \in NoAgg$, over input RDDs $\overline{r}$ and program terms $\phi(P_i) = t_i$. such that $RepVarSet(t_1) \neq RepVarSet(t_2)$, and $t_1 \neq \bot \vee t_2 \neq \bot$. Then, $\exists \overline{r}.[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$.*

Proof. By symmetry, we assume without loss of generality $t_1 \neq \bot$. Therefore, there is an element in the RDD defined by t_1: $\exists \overline{x}, y.y = t_1(\overline{x}) \wedge y \neq \bot$. Denoting $\overline{x} = (x_1, \ldots, x_l)$, we choose input RDDs $\overline{r}$ such that each input RDD has a single element x_i of multiplicity n_i: $r_i = \{\!\{x_i; n_i\}\!\}$, for $i = 1, \ldots, l$. If $t_2(\overline{x}) \neq y$ then $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required. Otherwise, the multiplicity of y in $[\![t_1]\!]$ is $\Pi_{\mathbf{x}_{r_i} \in RepVarSet(t_1)} n_i$, and in $[\![t_2]\!]$ it is $\Pi_{\mathbf{x}_{r_i} \in RepVarSet(t_2)} n_i$. As $RepVarSet(t_1) \neq RepVarSet(t_2)$, there are $n_i > 1$ such that $\Pi_{\mathbf{x}_{r_i} \in RepVarSet(t_1)} n_i \neq \Pi_{\mathbf{x}_{r_i} \in RepVarSet(t_2)} n_i$, thus $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required.

[9] For brevity, we omit the technical details of these standard definitions.

C Proof of Theorem 1

Proof. For non-RDD return types, the proof follows from Proposition 1, as the returned expression is expressible in APA. Therefore, let us assume that the return type is an RDD. For RDD return type, we use the algorithm **EqNoAgg** in Figure 7, which is a decision procedure, as we shall prove.

We begin with the following lemma:

Lemma 4. *Let P be a NoAgg program with inputs $R_1, \ldots, R_k, y_1, \ldots, y_m$, returning an RDD R, and $RepVarSet(\phi(P)) = \mathbf{x}_{R_{j_1}}, \ldots, \mathbf{x}_{R_{j_n}}$. We denote $\hat{R} = \Pi_{\{j|\mathbf{x}_{R_j} \in RepVarSet(\phi(P))\}} R_j$. By abuse of notation we mark the term of the RDD using Φ, and identify the program P with its returned RDD R. Then:*

$$\forall x \in R.R(x) = \sum_{\substack{(v_1,\ldots,v_k)\in\hat{R}.\\ \Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x}} \hat{R}(v_1,\ldots,v_k) \tag{9}$$

$$(\exists v_1,\ldots,v_k.\Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m) = x \wedge x \neq \bot) \iff x \in R \tag{10}$$

Proof. We prove by structural induction on the *NoAgg* program P after inlining. P is actually a shorthand for the RDD returned. We apply the induction on all operations except for `fold`, which is irrelevant to the *NoAgg* class.

- No operation—return an input RDD ($P = \texttt{return}\ R_i$): We have $\phi(P) = \mathbf{x}_{R_i}$, and $\hat{R}_i = R_i$.
  Equation (9) follows immediately, as $\hat{R}_i = R_i$:

  $$\sum_{v_i \in \hat{R}_i.\Phi(P)(v_i)=x} \hat{R}_i(v_i) = R_i(x)$$

  and so does Equation (10).
- Map ($P = \texttt{map}(R)(f)$): We have $\phi(P) = f(\phi(R))$, and thus $\hat{P} = \hat{R}$ We know by induction that

  $$\forall x \in R.R(x) = \sum_{(v_1,\ldots,v_k)\in\hat{R}.\Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x} \hat{R}(v_1,\ldots,v_k)$$

  Let $x' \in \texttt{map}(R)(f)$. From the semantics of SparkLite, we know that there are elements $z_1, \ldots, z_l \in R$ such that $f(z_i) = x'$. Therefore, the multiplicity of x' in P is the sum of multiplicities of all z_i in R. Also, $\Phi(\texttt{map}(R)(f))(v_1,\ldots,v_k,y_1,\ldots,y_m) = f(\Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m) = x'$. Then, Equation (9) follows:

  $$\begin{aligned}\texttt{map}(R)(f)(x') &= \textstyle\sum_{\{z_i|f(z_i)=x\}} R(z_i) = \\ &= \textstyle\sum_{\{z_i|f(z_i)=x\}} \sum_{(v_1,\ldots,v_k)\in\hat{R}.\Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=z_i} \hat{R}(v_1,\ldots,v_k) \\ &= \textstyle\sum_{(v_1,\ldots,v_k)\in\hat{R}.\Phi(\texttt{map}(R)(f))(v_1,\ldots,v_k,y_1,\ldots,y_m)=x'} \hat{R}(v_1,\ldots,v_k)\end{aligned}$$

  Equation (10) follows immediately because `map` can not transform elements to $\bot$: if $x \in P$, there is a $z \in R$ for which $f(z) = x$, and then by induction we can

find suitable $v_1, \ldots, v_k$ such that $\Phi(R)(v_1, \ldots, v_k, y_1, \ldots, y_m) = z$. For the same $v_1, \ldots, v_k$, $\Phi(P)(v_1, \ldots, v_k, y_1, \ldots, y_m) = f(z) = x$. Conversely, if there are such $v_1, \ldots, v_k$, then $\Phi(R)(v_1, \ldots, v_k, y_1, \ldots, y_m) = z$, and $z \in R$, and `map` can not transform elements to $\perp$, we get that $\Phi(P)(v_1, \ldots, v_k, y_1, \ldots, y_m) = f(z) \in P$.

– Filter ($P = \texttt{filter}(R)(f)$): We have $\phi(P) = ite(f(\phi(R)), \phi(R), \perp)$, and thus $\hat{P} = \hat{R}$. By induction,

$$\forall x \in R.R(x) = \sum_{\substack{(v_1,\ldots,v_k)\in\hat{R}.\\ \Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x}} \hat{R}(v_1, \ldots, v_k)$$

Let $x' \in \texttt{filter}(R)(f)$. From the semantics of SparkLite, if $x' \in \texttt{filter}(R)(f)$, then $f(x') = tt$. Therefore,
$\Phi(\texttt{filter}(R)(f))(v_1, \ldots, v_k, y_1, \ldots, y_m) = \Phi(R)(v_1, \ldots, v_k, y_1, \ldots, y_m) = x'$.
The multiplicity of x' in $\texttt{filter}(R)(f)$ is the same as that of x' in R. Thus, we receive:

$$\begin{aligned}
\texttt{filter}(R)(f)(x') &= R(x) \\
&= \textstyle\sum_{(v_1,\ldots,v_k)\in\hat{R}.\Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x} \hat{R}(v_1, \ldots, v_k) \\
&= \textstyle\sum_{(v_1,\ldots,v_k)\in\hat{R}.\Phi(\texttt{filter}(R)(f))(v_1,\ldots,v_k,y_1,\ldots,y_m)=x'} \hat{R}(v_1, \ldots, v_k) \\
&= \textstyle\sum_{(v_1,\ldots,v_k)\in\hat{R}.\Phi(P)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x'} \hat{P}(v_1, \ldots, v_k)
\end{aligned}$$

To prove Equation (10), we note that if $x \in \texttt{filter}(R)(f)$, then $x \in R$.
Thus, $\exists v_1, \ldots, v_k$ such that:
$x = \Phi(R)(v_1, \ldots, v_k, y_1, \ldots, y_m) = \Phi(\texttt{filter}(R)(f))(v_1, \ldots, v_k, y_1, \ldots, y_m)$,
as required. Conversely, If $\exists v_1, \ldots, v_k.\Phi(\texttt{filter}(R)(f))(v_1, \ldots, v_k, y_1, \ldots, y_m) = x \neq \perp$, then necessarily $f(x) = tt$ and $x \in R$, so $x \in \texttt{filter}(R)(f)$ by the semantics of SparkLite.

– Cartesian product ($P = \texttt{cartesian}(R, R')$): We have $\phi(P) = (\phi(R), \phi(R'))$. As we assumed there are no self products, we can say $RepVarSet(\phi(R)) \cap RepVarSet(\phi(R')) = \emptyset$, and set $R'' = \texttt{cartesian}(R, R')$, and $RepVarSet(\phi(R'')) = RepVarSet(\phi(R)) \cup RepVarSet(\phi(R'))$and $\hat{R''} = \hat{R} \times \hat{R'}$. By induction,

$$\begin{aligned}
\forall x \in R.R(x) &= \textstyle\sum_{\substack{(v_1,\ldots,v_k)\in\hat{R}.\\ \Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x}} \hat{R}(v_1, \ldots, v_k) \\
\forall x \in R'.R'(x) &= \textstyle\sum_{\substack{(v_1,\ldots,v_{k'})\in\hat{R'}.\\ \Phi(R')(v_1,\ldots,v_{k'},y_1,\ldots,y_m)=x}} \hat{R'}(v_1, \ldots, v_{k'})
\end{aligned}$$

Let $x' \in \texttt{cartesian}(R, R')$. From the semantics of SparkLite, the multiplicity of $x' = (x_1, x_2)$ is equal to product of the multiplicity of $x_1 \in R$ and the multiplicity of $x_2 \in R'$. Therefore,

$$\begin{aligned}
&\texttt{cartesian}(R, R')(x') = R(x_1)R'(x_2) \\
&= \textstyle\sum_{\substack{(v_1,\ldots,v_k)\in\hat{R}.\\ \Phi(R)(v_1,\ldots,v_k,y_1,\ldots,y_m)=x_1}} \hat{R}(v_1, \ldots, v_k) \sum_{\substack{(v_1,\ldots,v_{k'})\in\hat{R'}.\\ \Phi(R')(v_1,\ldots,v_{k'},y_1,\ldots,y_m)=x_2}} \hat{R'}(v_1, \ldots, v_{k'}) \\
&= \textstyle\sum_{\substack{(v_1,\ldots,v_k,v'_1,\ldots,v'_{k'})\in\hat{R''}.\\ \Phi(R'')(v_1,\ldots,v_k,v'_1,\ldots,v'_{k'})=(x_1,x_2)}} \hat{R''}(v_1, \ldots, v_k, v'_1, \ldots, v'_{k'})
\end{aligned}$$

as required. The proof of Equation (10) follows directly from applying it on each of the constituents in the structural induction and noting that the cartesian product can not make elements equal to $\perp$ unless one of the constituents is equal to $\perp$, and also that if one if the constituents is equal $\perp$, then the entire resulting pair is also equal to $\perp$.

We continue with the proof of the correctness of the algorithm.

Sound: Need to prove: *If* $\mathbf{EqNoAgg}(P1, P2)$ *returns true, then* $P1$ *is equivalent to* $P2$. We assume towards a contradiction that $P1$ is not equivalent to $P2$. Therefore, there are input RDDs and parameters $\bar{R}, \bar{y}$ such that either:

- Without loss of generality (by symmetry), $\exists x.x \in P1(\bar{R}, \bar{y}) \wedge x \notin P2(\bar{R}, \bar{y})$: As $x \in P1(\bar{R}, \bar{y})$, we can deduce from Equation (10) in Lemma 4 that there are $\bar{v} \in \bar{R}$ such that $\Phi(P1)(\bar{v}, \bar{y}) = x$. We also know that **EqNoAgg** returned true, thus either $RepVarSet(\phi(P1)) \neq RepVarSet(\phi(P2))$ and both program terms are equal to $\perp$, which is impossible by Equation (10) (because $\Phi(P1)(\bar{v}, \bar{y}) = x$ and $x \neq \perp$ by Equation (10)), or $RepVarSet(\phi(P1)) = RepVarSet(\phi(P2))$ and then $\forall z.\Phi(P1)(z) = \Phi(P2)(z)$. But then, $\Phi(P2)(\bar{v}, \bar{y}) = x$, so from Equation (10) in Lemma 4, we get a contradiction: $x \in P2(\bar{R}, \bar{y})$.
- $\exists x.P1(\bar{R}, \bar{y})(x) \neq P2(\bar{R}, \bar{y})(x)$: We can assume that $x \in P1(\bar{R}, \bar{y}) \wedge x \in P2(\bar{R}, \bar{y})$ (otherwise, we return to the case of the previous bullet), thus we can apply Equation (9) in Lemma 4. By the same deduction done in the previous bullet, we must have $RepVarSet(\phi(P1)) = RepVarSet(\phi(P2))$, and that $\forall \bar{v}.\Phi(P1)(\bar{v}) = \Phi(P2)(\bar{v})$. Thus, we can set $\hat{R} = \Pi_{R_j \in RepVarSet(\phi(P1))} R_j$, and have that:

$$\sum_{\bar{v} \in \hat{R}.\Phi(P1)(\bar{v}, \bar{y}) = x} \hat{R}(\bar{v}) \neq \sum_{\bar{v} \in \hat{R}.\Phi(P2)(\bar{v}, \bar{y}) = x} \hat{R}(\bar{v})$$

  Therefore, $\{\bar{v} \in \hat{R} | \Phi(P1)(\bar{v}, \bar{y}) = x\} \neq \{\bar{v} \in \hat{R} | \Phi(P2)(\bar{v}, \bar{y}) = x\}$. But this is a contradiction, because this would mean $\forall \bar{v}.\Phi(P1)(\bar{v}, \bar{y}) = \Phi(P2)(\bar{v}, \bar{y})$ is invalid, contradicting the assumption that $\mathbf{EqNoAgg}(P1, P2)$ returns true.

Complete: Need to prove: *If* $P1 = P2$*, then* $\mathbf{EqNoAgg}(P1, P2)$ *returns true.* We assume towards a contradiction that $\mathbf{EqNoAgg}(P1, P2)$ returns false. Then, either (1) $RepVarSet(\phi(P1)) = RepVarSet(\phi(P2))$ and $\exists \bar{v}.\Phi(P1)(\bar{v}, \bar{y}) \neq \Phi(P2)(\bar{v}, \bar{y})$, or (2) $RepVarSet(\phi(P1)) \neq RepVarSet(\phi(P2))$ and $\exists \bar{v}\Phi(P1)(\bar{v}, \bar{y}) \neq \perp \vee \Phi(P2)(\bar{v}, \bar{y}) \neq \perp$.

If (1), then we take such a witness $\bar{v}$, and write it explicitly as $(v_1, \ldots, v_n)$. For each input RDD that its representative variable belongs to $RepVarSet(\phi(P1))$, we generate R_i such that $R_i = \{\!\{v_i; 1\}\!\}$, Otherwise, we pick it arbitrarily. We denote the sequence of RDDs generated as $\bar{R}$. We assume without loss of generality that $\Phi(P1)(\bar{v}, \bar{y}) \neq \perp$. (Otherwise, $\bar{v}$ is not a witness.) Then we denote $z = \Phi(P1)(\bar{v}, \bar{y})$ and note that by Equation (10) in Lemma 4, $z \in P1(\bar{R}, \bar{y})$. Furthermore, as we chose all multiplicities of $v_i \in R_i$ to be equal to 1, we can deduce that $P1(\bar{R}, \bar{y}) = \{\!\{z; 1\}\!\}$, and that $P2(\bar{R}, \bar{y}) = \{\!\{z'; 1\}\!\}$ for $z' = \Phi(P2)(\bar{v}, \bar{y})$. As $z \neq \Phi(P2)(\bar{v}, \bar{y})$, we can deduce that $z \neq z'$, thus $P1(\bar{R}, \bar{y}) \neq P2(\bar{R}, \bar{y})$, contradicting the assumption that $P1$ and $P2$ are equivalent.

If (2), then we apply Lemma 1, and it immediately follows that $P1$ and $P2$ are not equivalent, in contradiction to our assumption.

D Proof of Theorem 2

Theorem 2 is a direct corollary of the following theorem.

Theorem 4. *The halting problem for 2-counter machines (2CM) reduces to the problem of program equivalence in* SparkLite *(*PE *for short).*

Proof. We show a reduction of the halting problem for 2CM to *PE*. Given a 2CM machine $(\{c_1, c_2\}, L, T)$ where c_1, c_2 are counters in $\mathbb{N}$, $L \subset \mathbb{N}$ is a finite set of instruction locations, and T is the transition function of the states, which are 3-tuples of the location, and counter values: (l, n_1, n_2). We denote by $s_i = (l_i, n_{i_1}, n_{i_2})$ the initial state of the machine, and $s_h = (l_h, n_{h_1}, n_{h_2})$ as the halting state of the machine. We generate the following instance of the *PE* problem:

$$f = \lambda S, x.T(S)$$

P7(R: $RDD_{\texttt{Int}}$)**:**	**P8**(R: $RDD_{\texttt{Int}}$)**:**
`return` `fold`$(s_i, f)(R) = s_f$	`return` ff

If the two programs are equivalent, then the 2CM never reaches s_f, and therefore does not halt. Otherwise, if there is some RDD R such that $P1$ returns tt, then the programs are not equivalent, and the 2CM halts after $|R|$ steps. The size of the input RDD R determines how many steps the 2CM will make when simulated by $P1$. In addition, as the number of locations in L is finite, and as the allowed instructions in a 2CM are definable in our extended Presburger arithmetics, T, and consequently the fold UDF f, are also definable in the extended Presburger arithmetics. Thus, $P7$ and $P8$ are both valid SparkLite programs. Furthermore, $P7$ belongs to the Agg^1 and so can $P8$ (by taking $g = \lambda x.True$ to act on any aggregated term). Due to this, this is a reduction of the 2CM halting problem to the program equivalence problem in Agg^1.

E Proof of Lemma 2

Proof. (Lemma 2) 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`: $\langle a_1, \ldots, a_n \rangle$, where n is size of the bag R. We also know that a requirement of `fold` UDFs is that they are *commutative*, so the order of elements chosen does not change the final result. We also remark that we extended the definition of f_i in the underlying theory to $\perp$ arguments by setting $f_i(M, \perp) = M$ ($\perp$ is defined to behave as the neutral element for f_i, and cannot appear as the accumulated value argument to f_i). The motivation is to avoid updating the

intermediate value when f_i is applied on elements that were filtered out from the RDD previously. We denote $\phi(R_1) = \mu_1, \phi(R_2) = \mu_2$. To prove $P1$ and $P2$ are equivalent, we need to show that for every $\bar{v}$ of assignments to $FV(\phi(P_i))$, To prove $g_1([\phi(\mu_1)]_{init_1,f_1})[\bar{v}/FV(\phi(P_1))] = g_2([\phi(\mu_2)]_{init_2,f_2})[\bar{v}/FV(\phi(P_2))]$, it is necessary to prove that:

$$g_1(\texttt{fold}(init_1, f_1)(R_1))[\bar{v}/FV(\phi(P_1))] = g_2(\texttt{fold}(init_2, f_2)(R_2))[\bar{v}/FV(\phi(P_2))]$$

We set $M_{j,0} = init_j$ for $j = 1, 2$. Each element of R_1 and R_2 is expressible by providing a concrete valuation to the free variables of μ_1 and μ_2, namely, the vector $\bar{v}$.

We prove the equality by induction on the *size* of the RDDs R_1, R_2, denoted n.[10] We choose an arbitrary sequence of n valuations $\langle \bar{a}_1, \ldots, \bar{a}_n \rangle$, and plug them into the `fold` operation for both R_1, R_2. The result is two sequences of *intermediate values* $\langle M_{1,1}, \ldots, M_{1,n} \rangle$ and $\langle M_{2,1}, \ldots, M_{2,n} \rangle$. From the semantics of `fold`, we have that $M_{j,i} = e_j[M_{j,i-1}/x, \Phi(\mu_j)/y][\bar{a}_i/FV(\phi(\mu_j))]$ for $j = 1, 2$. Our goal is to show $g_1(M_{1,n})\bar{v}/FV(\phi(P_1))[\bar{v}/FV(\phi(P_1))] = g_2(M_{2,n})[\bar{v}/FV(\phi(P_2))]$ for all n.

Case $n = 0$: When $R_1 = R_2 = \{\!\{\}\!\}$, we have $\texttt{fold}(init_1, f_1)(R_1) = init_1$ and $\texttt{fold}(init_2, f_2)(R_2) = init_2$. From Equation (3), $g_1(init_1)[\bar{v}/FV(\phi(P_1))] = g_2(init_2)[\bar{v}/FV(\phi(P_2))]$, as required.

Case $n = i$, assuming correct for $n \leq i-1$: By assumption, we know that the sequence of intermediate values up to $i-1$ satisfies: $g_1(M_{1,i-1})[\bar{v}/FV(\phi(P_1))] = g_2(M_{2,i-1})[\bar{v}/FV(\phi(P_2))]$. We are given the i'th valuation, denoted $\bar{a}_i$. We need to show $M_{1,i} = M_{2,i}$, so we use the formula for calculating the next intermediate value:

$$M_{1,i} = e_1[M_{1,i-1}/x, \Phi(\mu_1)/y][\bar{a}_i/FV(\phi(\mu_1))]$$
$$M_{2,i} = e_2[M_{2,i-1}/x, \Phi(\mu_2)/y][\bar{a}_i/FV(\phi(\mu_2))]$$

We use Equation (4), plugging in $\bar{v} = \bar{a}_i$, $M_1 = M_{1,i-1}$, and $M_2 = M_{2,i-1}$. By the induction assumption, $g_1(M_{1,i-1})[\bar{v}/FV(\phi(P_1))] = g_2(M_{2,i-1})[\bar{v}/FV(\phi(P_2))]$, therefore $g_1(M_1)[\bar{v}/FV(\phi(P_1))] = g_2(M_2)[\bar{v}/FV(\phi(P_2))]$, so Equation (4) yields:

$$g_1(e_1[M_1/x, \Phi(\mu_1)/y][\bar{a}_i/FV(\phi(\mu_1))])[\bar{v}/FV(\phi(P_1))] =$$
$$g_2(e_2[M_2/x, \Phi(\mu_2)/y][\bar{a}_i/FV(\phi(\mu_2))])[\bar{v}/FV(\phi(P_2))]$$

By substituting back M_j and the formula for the next intermediate value, we get: $g_1(M_{1,i})[\bar{v}/FV(\phi(P_1))] = g_2(M_{2,i})[\bar{v}/FV(\phi(P_2))]$ as required.

[10] It is important to note that not every n can be a legal size of the RDDs. For example, if $R_1 = \texttt{cartesian}(R, R)$, then its size must be quadratic ($|R|^2$). The induction we apply here, is actually stronger than what is required for equivalence, because we prove the equivalence even for subsets of the RDDs which may not be expressible using SparkLite operations. In any case, the soundness argument is valid.

F Proof Theorem 3

Proof. **Sound (if):** We prove the equality $g_1([\phi(\mu_1)]_{i_1,f_1})(\bar{R}) = g_2([\phi(\mu_2)]_{i_2,f_2})(\bar{R})$. [11] by induction on the size of the RDDs μ_1, μ_2, denoted n.[12] For $n = 0$, $\mu_1(\bar{r}, \bar{y}) = \mu_2(\bar{r}, \bar{y}) = \{\!\{\}\!\}$, thus $[\phi(\mu_j)]_{i_j,f_j} = i_j\,(j = 1, 2)$, and the equality follows from Equation (7). Assuming for n and proving for $n + 1$: We let a sequence of intermediate values $M_{j,k}$, $(j = 1, 2; k = 1, \ldots, n + 1)$, for which we know in particular that $g_1(M_{1,n}) = g_2(M_{2,n})$, and we need to prove $g_1(M_{1,n+1}) = g_2(M_{2,n+1})$. We denote $M_{j,0} = i_j$, and then we have $M_{j,k} = e_j[M_{j,k-1}/x, \Phi(\mu_j)[\bar{a_k}/FV(\phi(\mu_j))]/y]\,(k = 1, \ldots, n+1)$ for some $\bar{a_k}$. According to Equation (6): $M_{j,2} = e_j[M_{j,1}/x, \Phi(\mu_j)[\bar{a_2}/FV(\phi(\mu_j))]/y] = e_j[e_j[i_j/x, \Phi(\mu_j)[\bar{a_1}/FV(\phi(\mu_j))]/y]/x, \Phi(\mu_j)[\bar{a_2}$ yields:
$\exists \bar{a_2}'. \bigwedge_{j=1,2} M_{j,2} = e_j[i_j/x, \Phi(\mu_j)[\bar{a_2}'/FV(\phi(\mu_j))]/y]$. We can thus use Equation (6) to prove by induction that $\exists \bar{a_k}'. \bigwedge_{j=1,2} M_{j,k} = e_j[i_j/x, \Phi(\mu_j)[\bar{a_k}'/FV(\phi(\mu_j))]/y]$, and in particular $\exists \bar{a_n}'. \bigwedge_{j=1,2} M_{j,n} = e_j[i_j/x, \Phi(\mu_j)[\bar{a_n}'/FV(\phi(\mu_j))]$. By applying Equation (8) for $\bar{v} = \bar{a_{n+1}}, \bar{y} = \bar{a_n}'$, we get: $g_1(M_{1,n+1}) = g_2(M_{2,n+1})$, as required.
Complete (only if): Assume towards a contradiction that either Equation (7) or Equation (8) are false. If the requirement of Equation (7) is not satisfied, yet the aggregates are equivalent, i.e.

$$g_1([\phi(\mu_1)]_{i_1,f_1}) = g_2([\phi(\mu_2)]_{i_2,f_2}) \wedge g_1(i_1) \neq g_2(i_2)$$

then we can get a contradiction by choosing all input RDDs to be empty. Thus, for $\bar{R} = \{\!\{\}\!\}$, $[\phi(\mu_1)]_{init_1,f_1} = i_1 \wedge [\phi(\mu_2)]_{i_2,f_2} = i_2 \implies g_1(i_1) = g_2(i_2)$, which is a contradiction. The conclusion is that Equation (7) is a necessary condition for equivalence. Therefore, we assume just Equation (8) is false. Let there be counterexamples $\bar{v}, \bar{y}$ to Equation (8), [13] and let:

$$E_j = e_j[e_j[i_j/x, \Phi(\mu_j)[\bar{y}/FV(\phi(\mu_j))]/y]/x, \Phi(\mu_j)[\bar{v}/FV(\phi(\mu_j))]/y]$$

Then $g_1(E_1)(\bar{R}) \neq g_2(E_2)(\bar{R})$. By Equation (6) we can write E_j as: $E_j = e_j[i_j/x, \Phi(\mu_j)[\bar{w}/FV(\phi(\mu_j))]/y]$ for some $\bar{w}$. We take an RDD $R = \{\!\{\bar{w}; 1\}\!\}$. Then $\mu_j(R) = \{\!\{\Phi(\mu_j)[\bar{w}/FV(\phi(\mu_j))]; 1\}\!\}$, for which: $[\Phi(\mu_j)]_{i_j,f_j}(R) = E_j$. By the assumption, $g_1([\phi(\mu_1)]_{i_1,f_1})](R) = g_2([\phi(\mu_2)]_{i_2,f_2})](R)$, but then $g_1(E_1)(\bar{R}) = g_2(E_2)(\bar{R})$. Contradiction.

G Additional Classes with Sound Equivalence Verification Methods

A natural extension of the Agg^1 class is to programs that use an aggregated expression in another RDD operation. For example, filtering elements strictly larger than

[11] Following Footnote 4, we will simply ignore the assignments to non-RDD variables for readability.
[12] The comment in footnote 10 regarding the validity of the soundness argument, even if μ_i can not have size n, is still valid here.
[13] Note that the M_j are determined immediately by choosing $\bar{y}$: $M_j = e_j[i_j/x, \Phi(\mu_j)[\bar{y}/FV(\phi(\mu_j))]/y]$.

any element in another RDD: $\texttt{filter}((\lambda x.\lambda y.y > x)(\texttt{fold}(-\infty, max)(R_1)))(R_0)$, for which the program term is $ite(\mathbf{x}_{R_0} > [\mathbf{x}_{R_1}]_{-\infty,max}, \mathbf{x}_{R_0}, \bot)$.

Definition 2 (The Agg^1_R class). *Let there be a program P with $\phi(P) = \psi$. We say that $P \in Agg^1_R$ if ψ contains a single aggregate term $[\varphi]_{i,f}$, which is denoted γ, and in addition, φ has no aggregate terms. We write $\phi(P) = \psi[M/\gamma]$, where M is the value of the aggregate sub-term.*

Lemma 5 (Lifting Lemma 2 to Agg^1_R). *Let two* SparkLite *programs P_1, P_2 in Agg^1_R with terms ψ_j and aggregate expressions $\gamma_j = [\phi(\mu_j)]_{i_j,f_j}$, and $f_j = \lambda x, y.\, e_j$ for $j \in \{1, 2\}$. P_1 is equivalent to P_2 if:*

$$RepVarSet(\phi(\mu_1)) = RepVarSet(\phi(\mu_2)) \tag{11}$$

$$RepVarSet(\phi(\psi_1)) = RepVarSet(\phi(\psi_2)) \tag{12}$$

$$\forall \bar{x}.\psi_1[i_1/\gamma][\bar{x}/FV(\phi(P_1))] = \psi_2[i_2/\gamma][\bar{x}/FV(\phi(P_2))] \tag{13}$$

$$\begin{aligned}\forall \bar{x}, \bar{v}, M_1, M_2.(\psi_1[M_1/\gamma][\bar{x}/FV(\phi(P_1))] = \psi_2[M_2/\gamma][\bar{x}/FV(\phi(P_2))]) \implies \\ \big(\psi_1[e_1[M_1/x, \Phi(\mu_1)[\bar{v}/FV(\phi(\mu_1))]/y]/\gamma][\bar{x}/FV(\phi(P_1))] \\ = \psi_2[e_2[M_2/x, \Phi(\mu_2)[\bar{v}/FV(\phi(\mu_2))/y]/\gamma][\bar{x}/FV(\phi(P_2))]\big)\end{aligned} \tag{14}$$

Proof. The proof follows along the lines of the proof of Lemma 2. We need to prove $\Phi(P_1) = \Phi(P_2)$, or $\forall \bar{x}, A_1, A_2.\psi_1[A_1/\gamma_1][\bar{x}/FV(\phi(P_1))] = \psi_2[A_2/\gamma_2][\bar{x}/FV(\phi(P_2))]$, where $\gamma_j = [\phi(\mu_j)]_{i_j,f_j}$ and $\bar{x}$ is a vector of valuations to $RepVarSet(\psi_1)$, $RepVarSet(\psi_2)$ which are equal sets (Equation (12)). We shall prove it by induction on the size of the RDDs μ_1 and μ_2, generating the underlying terms of γ_1 and γ_2.

For size 0, we have $A_j = i_j$, and from Equation (13) we have $\Phi(P_1) = \Phi(P_2)$ as required.

Assuming for size n and proving for $n + 1$: The RDDs μ_1, μ_2 are now generated using $a_1, \ldots, a_{n+1}$, with intermediate values $M_{j,1}, \ldots, M_{j,n+1}$ for $j = 1, 2$. By assumption, $\forall x.\psi_1[M_{1,n}/\gamma_1] = \psi_2[M_{2,n}/\gamma_2]$, and we need to prove $\forall \bar{x}.\psi_1[M_{1,n+1}/\gamma_1][\bar{x}/FV(\phi(P_1))] = \psi_2[M_{2,n+1}/\gamma_2][\bar{x}/FV(\phi(P_2))]$. In addition: $M_{j,n+1} = e_j[M_{i,n}/x, \Phi(\mu_j)[a_{n+1}/FV(\phi(\mu_j))]]$ for $j = 1, 2$. We let some $\bar{x}$ and we need to prove for it $\psi_1[M_{1,n+1}/\gamma_1][\bar{x}/FV(\phi(P_1))] = \psi_2[M_{2,n+1}/\gamma_2][\bar{x}/FV(\phi(P_2))]$. We apply Equation (14) with $\bar{x}$ as $\bar{x}$, $\bar{v} = a_{n+1}$, and $M_{1,n}$ and $M_{2,n}$ as M_1 and M_2 respectively, concluding that:

$$\begin{aligned}&\psi_1[e_1[M_{1,n}/x, \Phi(\mu_1)[a_{n+1}/FV(\phi(\mu_1))]/y]/\gamma_1][\bar{x}/FV(\phi(P_1))] \\ &= \psi_2[e_2[M_{2,n}/x, \Phi(\mu_2)[a_{n+1}/FV(\phi(\mu_2))]/y]/\gamma_2)[\bar{x}/FV(\phi(P_2))]\end{aligned}$$

Replacing for $M_{j,n+1}$, we get what had to be proven.

The sound technique can be further generalized to programs with multiple aggregate terms, which are not nested — each aggregate term does not contain an aggregate term in its definition. We denote this class Agg^n.

Definition 3 (The Agg^n class). *Let there be a program P with $\Phi(P) = g([t_1]_{i_1,f_1}, \ldots, [t_n]_{i_n,f_n})$, or $g(\overline{[t_i]_{i_i,f_i}})$ for short, and $t_i = \Phi(\mu_i)$. $P \in Agg^n$ if $t_1, \ldots, t_n$ do not contain aggregate terms.*

Lemma 6. *Let P_1, P_2 be two programs in Agg^n, such that $\phi(P_j) = g_j(\overline{[\phi(\mu_j)]_{i_j,f_j}})$ and $f_j = \lambda x, y.\, e_j$ for $j = 1, 2$. We have $g_1(\overline{[\phi(\mu_1)]_{i_1,f_1}}) = g_2(\overline{[\phi(\mu_2)]_{i_2,f_2}})$ if:*

$$\forall j_1, j_2. RepVarSet(\phi(\mu_{1,j_1})) = RepVarSet(\phi(\mu_{2,j_2})) \tag{15}$$

$$g_1(\overline{i_1})[\overline{v}/FV(\phi(P_1))] = g_2(\overline{i_2})[\overline{v}/FV(\phi(P_2))] \tag{16}$$

$$\begin{aligned}&\forall \overline{v}, \overline{M_1}, \overline{M_2}. g_1(\overline{M_1})[\overline{v}/FV(\phi(P_1))] = g_2(\overline{M_2})[\overline{v}/FV(\phi(P_2))] \implies \\ &\quad g_1(\overline{e_1[M_1/x, \Phi(\mu_1)[\overline{v}/FV(\phi(\mu_1))]/y}][\overline{v}/FV(\phi(P_1))] \\ &\quad = g_2(\overline{e_2(M_2/x, \Phi(\mu_2)[\overline{v}/FV(\phi(\mu_2))]/y}[\overline{v}/FV(\phi(P_2))]\end{aligned} \tag{17}$$

We note that the subset of Agg^n programs that can be handled with Lemma 6 could be extended if we relaxed Equation (15). We show a motivating example for relaxing Equation (15):

Example 4. Let two Agg^n programs $P1, P2$ that sum the elements of an input RDD R_0. $P2$ will also apply a trivial fold on input RDD R_1 and return the sum of the aggregations. As the fold on R_1 is trivial, it will not affect the final result.

$$sum = \lambda A, x.A + x$$
$$zero = \lambda A, x.0$$

P9(R_0: $RDD_{\texttt{Int}}, R_1$: $RDD_{\texttt{Int}}$):	**P10**(R_0: $RDD_{\texttt{Int}}, R_1$: $RDD_{\texttt{Int}}$):
$v = \texttt{fold}(0, sum)(R_0)$	$v' = \texttt{fold}(0, sum)(R_0)$
`return` v	$u = \texttt{fold}(0, zero)(R_1)$
	`return` $v' + u$

We see that as $P10$ has an aggregate term with a set of representative variables equal to $\{\mathbf{x}_{R_1}\}$ and the other aggregate terms have $\{\mathbf{x}_{R_0}\}$, and as a result Lemma 6 returns 'not equivalent', while $P9$ and $P10$ are actually equivalent.

In order to analyze such programs, we need to verify equivalence in the case one or more of the fold operations were completed. However, Agg^n contains non-trivial programs, as the below example shows:

Example 5 (Independent fold). The below programs return a tuple containing the sum of positive elements in its first element, and the sum of negative elements in the second element. With Lemma 6, we are able to show the equivalence.

$$h : (\lambda(P, N), x.ite(x \geq 0, (P + x, N), (P, N - x))$$

P11(R: $RDD_{\texttt{Int}}$):	**P12**(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)

$$\phi(P1) = [\mathbf{x}_R]_{(0,0),h}; \quad \phi(P2) = ([\phi(R_P)]_{0,+}, -[\phi(R_N)]_{0,+})$$
$$\Phi(R_P) = ite(\mathbf{x}_R \geq 0, \mathbf{x}_R, \perp)\,; \Phi(R_N) = ite(\mathbf{x}_R < 0, -\mathbf{x}_R, \perp)$$

We set $g_1 = g_2 = \lambda(x, y).(x, y)$, and apply Lemma 6 to prove:

$$[\mathbf{x}_R]_{(0,0),h} = ([ite(\mathbf{x}_R \geq 0, \mathbf{x}_R, \perp)]_{0,+}, -[ite(\mathbf{x}_R < 0, -\mathbf{x}_R, \perp)]_{0,+})$$

Equation (15) is satisfied: $RepVarSet(\phi(P_i)) = \{x_R\}$ for $i = 11, 12$. Induction base case (Equation (16)) is trivial. Induction step (Equation (17)) can be written simply as:

$$\forall x, A, B, C. p_1(A) = B \wedge p_2(A) = C \implies$$
$$p_1(h(A, x)) = B + ite(x \geq 0, x, 0) \wedge p_2(h(A, x)) = C + ite(x < 0, -x, 0)$$

H Details of test-cases

Figure 9 shows a detailed view of all 19 test cases. The first column shows the class to which the program equivalence problem instance belongs, while the second and fourth columns show the programs themselves written concisely.

Class	**Program 1**	$\stackrel{?}{=}$	**Program 2**	**Result**
$NoAgg$	P1 (Section 1.2)	=	P2 (Section 1.2)	Verified
$NoAgg$	P1 (Section 1.2)	≠	P2 (Section 1.2) changed to filter elements smaller than 100	Found CEX
$NoAgg$	map(m2)(R1 x R2)	=	map(m2)(R1) x map(m2)(R2)	Verified
$NoAgg$	map(m2)(R)	≠	map(m2p1)(R)	Found CEX
$NoAgg$	filter(ba50)(R1 x R2)	=	filter(a50)(R1) x filter(a50)(R2)	Verified
$NoAgg$	filter(ea50)(R1 x R2)	≠	filter(a50)(R1) x filter(a50)(R2)	Found CEX
Agg^1	fold(0,count)(filter(odd)(R))	=	fold(0,sum)(map(if odd then 1 else 0)(R))	Verified
$AggPair^1_{sync}$	fold(1000,min)(R)	≠	fold(1000,min)(map(dis)(R))	Found CEX
$AggPair^1_{sync}$	fold(1000,min)(R)≥100	=	fold(1000,min)(map(dis)(R))≥80	Verified
$AggPair^1_{sync}$	fold(1000,min)(R)=100	=	fold(1000,min)(map(dis)(R))=80	Verified
$AggPair^1_{sync}$	fold(0,sMod5)(R)	≠	fold(0,sMod5)(map(m3)(R))	Found CEX
$AggPair^1_{sync}$	fold(0,sum)(R)=0%5	=	fold(0,sum)(map(m3)(R))=0%5	Verified
$AggPair^1_{sync}$	fold(0,sum)(R)=0%6	≠	fold(0,sum)(map(m3)(R))=0%6	Found CEX
$AggPair^1_{sync}$	fold(-100,max)(R)	=	-fold(100,min)(map(invert)(R))	Verified
$AggPair^1_{sync}$	fold(0,max)(R)	≠	-fold(100,min)(map(invert)(R))	Found CEX
$NoAgg$	map(m2)(S1)⋈ map(m2)(S2)	=	map(dd)(S1⋈S2)	Verified
$NoAgg$	map(dV)(S1) x map(dV)(S2)	=	map(deV)(S1 x S2)	Verified
$NoAgg$	map(dV)(S1)⋈ map(dV)(S2)	≠	map(dA)(S1⋈S2)	Found CEX
$NoAgg$	filter(oddk)(S1)⋈filter(oddk)(S2)	=	filter(oddk)(S1⋈S2)	Verified

Fig. 9. Benchmarks. The $\stackrel{?}{=}$ column records whether the programs are equivalent. The result column tells whether our tool verified the equivalence or found a counter example. R1, R2 represent RDDs of integer type. S1, S2 represent RDDs of pairs of integers. 'A x B' is a shorthand for cartesian product of RDDs A and B. 'S1⋈ S2' is a shorthand for $\text{map}(\lambda((x, y), (z, w)).(x, (y, w)))(\text{filter}(\lambda((x, y), (z, w)).x == z)(\text{S1 x S2}))$.

UDF definitions for Figure 9:

$$
\begin{array}{ll}
\text{m2} & = \lambda x.2 * x \\
\text{m2p1} & = \lambda x.2 * x + 1 \\
\text{m3} & = \lambda x.3 * x \\
\text{invert} & = \lambda x. - x \\
\text{a50} & = \lambda x.x \geq 50 \\
\text{ba50} & = \lambda(x, y).x \geq 50 \wedge y \geq 50 \\
\text{ea50} & = \lambda(x, y).x \geq 50 \vee y \geq 50 \\
\text{min} & = \lambda(x, y).ite(x < y, x, y) \\
\text{max} & = \lambda(x, y).ite(x > y, x, y) \\
\text{dis} & = \lambda x.x - 20 \\
\text{sum} & = \lambda(x, y).x + y \\
\text{sMod5} & = \lambda(x, y).(x + y)\%5 \\
\text{count} & = \lambda(x, y).x + 1 \\
\text{odd} & = \lambda x.x\%2 == 1 \\
\text{oddk} & = \lambda(x, y).x\%2 == 1 \\
\text{dd} & = \lambda(x, y).(2 * x, 2 * y) \\
\text{dV} & = \lambda(x, y).(x, 2 * y) \\
\text{deV} & = \lambda((x, y), (z, w)).((x, 2 * y), (z, 2 * w)) \\
\text{dA} & = \lambda(x, (y, z)).(2 * x, (2 * y, 2 * z))
\end{array}
$$