Verifying Equivalence of Spark Programs*

S. Grossman[1], S. Cohen[2], S. Itzhaky[3], N. Rinetzky[1], and M. 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. *Apache Spark* is a popular framework for writing large scale data processing applications. Our long term goal is to develop automatic tools for reasoning about Spark programs. This is challenging because Spark programs combine database-like relational algebraic operations and aggregate operations, corresponding to (nested) loops, with *User Defined Functions* (*UDF*s). In this paper, we present a novel SMT-based 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 (multisets) and allows for UDFs expressible in Presburger Arithmetics. We prove that the problem of checking equivalence is undecidable even for programs which use a single aggregation operator. Thus, we present sound techniques for verifying the equivalence of interesting classes of Spark programs, and show that it is complete under certain restrictions. We implemented our technique, and applied it to a few small, but intricate, test cases.

1 Introduction

Spark [17, 29, 30] is a popular framework for writing large scale data processing applications. It is an evolution of the Map-Reduce paradigm, which provides an abstraction of the distributed data as *bags* (multisets) of items. A bag r can be accessed using higher-order operations such as *map*, which applies a *user defined function (UDF)* to all items in r; *filter*, which filters items in r using a given boolean UDF; 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. Bags also support by-key, *join* and *cartesian product* operators. A language such as *Scala* or *Python* is used as Spark's interface, allowing to embed calls to the underlying framework, as well as defining UDFs that Spark executes.

* We would like to thank the reviewers for their helpful comments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° [321174], by Len Blavatnik and the Blavatnik Family foundation, and by the Broadcom Foundation and Tel Aviv University Authentication Initiative.

This paper shows how to harness SMT solvers to automatically reason about small subsets of Spark programs. Specifically, we are interested in developing tools that can check whether two Spark programs are equivalent and produce a witness input for the different behavior of inequivalent ones. Reasoning about the equivalence of Spark programs is challenging—not only is the problem undecidable even for programs containing a single aggregate operation, some specific intricacies arise from the fact that the input datasets are bags (rather than simple sets or individual items), and that the output might expose only a *partial* view of the results of UDF-based aggregations.

Our main tool for showing equivalence of Spark programs is reducing the equivalence question to the validity of a formula in Presburger arithmetic, which is a decidable theory [12, 22]. More specifically, we present a simplified model of Spark by defining SparkLite, a functional programming language in which UDFs are expressed over a decidable theory. We show that SMT solvers can effectively verify equivalence of and detect potential differences between Spark programs. We present different verification techniques which leverage certain semantic restrictions which, in certain cases, make the problem decidable. These restrictions can also be validated through SMT. Arguably, the most interesting aspect of our technique is that it can reason about higher order operations such as *fold* and *foldByKey*, corresponding to limited usage of loops and nested loops, respectively. The key reason for the success of our techniques is that our restrictions make it possible to automatically infer inductive hypotheses simple enough to be mechanically checked by SMT solvers, e.g., [10].

Main Results Our main technical contributions can be summarized as follows:

- We prove that verifying the equivalence of SparkLite programs is undecidable even in our limited setting.
- We identify several interesting restrictions of SparkLite programs, and develop sound, and in certain cases complete, methods for proving program equivalence. (See Table 1, which we gradually explain in Section 2.)
- We implemented our approach on top of Z3 [10], and applied it to several interesting programs inspired by real-life Spark applications. When the implementation employs a complete method and determines that a pair of programs is not equivalent, it produces a (real) counterexample of bag elements which are witnesses for the difference between the programs. This counterexample is guaranteed to be valid for programs which have a complete verification method, and can help understand the differences between these programs.

2 Overview

For space considerations, we concentrate on presenting an informal overview through a series of simple examples, and formalize the results in [13].

Figure 1 shows two equivalent Spark programs and the formula that we use for checking their equivalence. The programs accept a bag of integer elements. They return another bag where each element is twice the value of the original element, for elements which are at least 50. The programs operate differently:

Method	**Syntactic restriction**	**Semantic restriction**	**Complete?**
$NoAgg$	No folds	-	✓
$AggOne^p$	Single fold, primitive output	-	-
$AggOne^b$	Single fold, bag output	-	-
$AggMult^p$	Non-nested folds, primitive output	-	-
$AggOne^p_{sync}$	Single fold, primitive output	Synchronous collapsible aggregations	✓
$AggOneK^b$	Single fold by key, bag output	Isomorphic keys	-

Table 1. Sound methods for verifying equivalence of Spark programs, their syntactic and semantic prerequisites, and completeness. By abuse of notation, we refer to SparkLite programs adhering to the syntactic restriction of one of the first four verification methods as belonging to the *class of* SparkLite *programs* of the same name.

```
P1(R: Bag_Int):                        P2(R: Bag_Int):
$R'_1 = map(\lambda x.2 * x)(R)$               $R'_2 = filter(\lambda x.x \geq 50)(R)$
$R''_1 = filter(\lambda x.x \geq 100)(R'_1)$       $R''_2 = map(\lambda x.2 * x)(R'_2)$
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.

$P1$ 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 bag, and yield a new bag. For example, let bag 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 bag, R'_1, by doubling every element of R, i.e., $R'_1 = \{\!\{4, 4, 206, 128\}\!\}$. The `filter` operator in the second line generates bag 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 a bag R'_2 of all the elements in R which are not smaller than 50, resulting in the bag $R'_2 = \{\!\{103, 64\}\!\}$. $P2$ applies the map operator to produce bag 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*, i.e., given the same inputs produce the same outputs, 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 a *program term*: $\phi(P1) = ite(2 * x \geq 100, 2 * x, \bot)$, and $P2$ as: $\phi(P2) = 2 * ite(x \geq 50, x, \bot)$, where *ite* denotes the if-then-else operator and $\bot$ is used to denote that the element has been removed. The variable symbol x can be thought of as an arbitrary element in the bag R, and the terms $\phi(P1)$ and $\phi(P2)$ 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 $\forall x. \phi(P1) = \phi(P2)$. It is expressible in a decidable extension of Presburger Arithmetics, which supports the special $\bot$ symbol (see [13, Section 8]). 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 to its multiplicity (the number of duplicates it has in the bag) or the presence of other elements. Thus, we can symbolically represent the effect of the program on any bag, by encoding its effect on a single arbitrary element from that bag. Interestingly, the locality property transcends to the *cartesian product* operator which conjoins items across bags.

Decidability. The validity of the aforementioned formula suffices to prove the equivalence of $P1$ and $P2$ due to a tacit fact: both programs operate on the same bag. Consider, however, programs $P1'$ and $P2'$ which receive bags R_1 and R_2 as inputs. $P1'$ maps all the elements of R_1 to 1 and $P2'$ does the same for R_2. Their symbolic encoding is $\phi(P1') = (\lambda x.1)x_1$ and $\phi(P1') = (\lambda x.1)x_2$, where x_1 and x_2 represent, respectively, arbitrary elements from R_1 and R_2. The formula $\forall x_1, x_2.\phi(P1') = \phi(P2')$ is valid. Alas, the programs produce different results if R_1 and R_2 have different sizes. Interestingly, we show that unless both programs always return the empty bag, they are equivalent *iff* their program terms are equivalent *and* use the same variable symbols.[4] Furthermore, it is possible to decide whether a program always returns the empty bag by determining if its program term is equivalent to $\bot$. Theorem 1 (Section 4.1) shows that the equivalence of *NoAgg* programs, i.e., ones not using aggregations, can be decided.

Usage of inductive reasoning We use inductive reasoning to determine the equivalence of programs that use aggregations. Theorem 2 (presented later on) shows that equivalence in $AggOne^p$, that is, of programs that use a single `fold` operation and return a primitive value, is undecidable. Thus, we consider different classes of programs that use aggregations in limited ways.

Figure 2 contains an example of two simple equivalent $AggOne^p$ programs. The programs operate over a bag of pairs (product IDs, price). The programs check if the minimal price in the bag is at least 100. The second program does this by subtracting 20 from each price in the bag 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 bag 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 a bag 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*

[4] Recall that intuitively, these variables pertain to arbitrary elements in the input bags. In our example, $\phi(P1')$ uses variable x_1 and $\phi(P2')$ uses x_2.

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

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

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

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

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

bag 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 [17].[5]) The base of the induction hypothesis checks that the programs are equivalent when the input bags 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 bag.[6] In our example, when the bags 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 $((\mathit{prod}, p), (\mathit{prod}', p'))$ giving updated intermediate values M_2 and M_2'.

Completeness of the inductive reasoning. In the example in Figure 2, we use a simple form of induction by proving that two higher-order operations are equivalent iff they are equivalent on every input element and arbitrary temporarily accumulated values (M_1 and M_1' in Figure 2). Such an approach is incomplete. We now show an example for incompleteness, and a modified verification formula that is complete for a subset of AggOne^p, called AggOne^p_{sync}. In Figure 3, $P3$ and $P4$ were rewritten into $P5$ and $P6$, respectively, by using $=$ instead of $\geq$. The rewritten programs are equivalent. We show both the "naïve" formula, similar to the formula from Figure 2, and a revised version of it. (We 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 and M_1'. For example,

[5] We note that our results do not require UDFs to be associative, however, Spark does.
[6] Note that AggOne^p programs can fold bags produced by a sequence of filter, map, and cartesian product operations. Our approach is applicable to such programs because if the program terms of two folded bags use the same variable symbols, then any selection of elements from the input bags produces an element in the bag being folded in one program iff it produces an element in the bag that the other program folds. (See Lemma 1)

for $M_1 = 60$, $M_1' = 120$, and $p = 100$, we get a spurious counterexample to equality, leading to the wrong conclusion that the programs may not be equivalent. In fact, if $P5$ and $P6$ iterate over the input bag in the same order, it is not possible that their (temporarily) accumulated values are 60 and 120 at the same time.

Luckily, we observe that, often, the `fold` UDFs are somewhat restricted. One such natural property, is the ability to "collapse" any sequence of applications of the aggregation function f using a single application. We can leverage this property for more complete treatment of equivalence verification, if the programs collapse in *synchrony*; given their respective fold functions f_1, f_2, initial values i_1, i_2, and the symbolic representation of the program term pertaining to the folded bags φ_1, φ_2, the programs collapse in synchrony if the following holds:

$$\begin{aligned} \forall x, y. \exists a.\ & f_1(f_1(i_1, \varphi_1(x)), \varphi_1(y)) = f_1(i_1, \varphi_1(a)) \\ \wedge\ & f_2(f_2(i_2, \varphi_2(x)), \varphi_2(y)) = f_2(i_2, \varphi_2(a)) \end{aligned} \tag{1}$$

Note that the same input a is used to collapse both programs. 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 decide this property.

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 and 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.

Handling ByKey Operations Spark is often used to aggregate values of groups of records identified by a shared key. For example, in Figure 4 we present two equivalent programs that given a bag of pairs of student IDs and grades, return a histogram graph of all passing grades (≥ 60), in deciles. $P7$ first maps each student's grade to its decile, while making the decile the key. (The key is the first component in the pair.) Then, it computes the count of all students in a certain decile using the `foldByKey` operation, and filters out all non-passing deciles (< 6) from the resulting histogram. $P8$ first filters out all failing grades, and then continues similarly with the histogram computation.

Verifying the equivalence of $P7$ and $P8$ is challenging because, intuitively, the by-key operation corresponds to a nested loop: It partitions the bag into *"buckets"* according to the key element of the bag and folds every bucket separately. Furthermore, note that the two programs fold bags which contain different keys.

Our approach to verify programs using by-key operations is based on a reduction to the problem of verifying programs using *fold*: We rewrite the programs, so instead of applying the fold operation on one bucket at a time (as `foldByKey` does), we apply it on the entire bag to get the global aggregated result. We then

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

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

Naïve formula:

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

Revised formula:

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

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

map each key to the global aggregated result, instead of the aggregated result for the bucket. It is then possible to write an inductive hypothesis based on the rewritten program. The reduction is sound if the two compared programs partition the bag's elements to buckets consistently: If program $Q1$ sends two elements to the same bucket, then $Q2$ must also send those two elements to the same bucket (although it does not have to be the same bucket as $Q1$), and vice versa. As with the property of collapsibility seen earlier, this property can also be expressed in Presburger arithmetic, and be verified using an SMT solver: for functions k_1 and k_2 that describe expressions for keys, we require:

$$\forall x, x'.\big((k_1(x) = k_1(x') \wedge k_1(x) \neq \bot) \implies (k_2(x) = k_2(x'))\big) \tag{2}$$

$$\forall x, x'.\big((k_2(x) = k_2(x') \wedge k_2(x) \neq \bot) \implies (k_1(x) = k_1(x'))\big) \tag{3}$$

Figure 4 shows the inductive hypothesis whose validity ensures the equivalence of $P7$ and $P8$, as well as the resulting instantiation of Equations (2) and (3). $AggOneK^b$ is a sound method for verifying equivalence of pairs of programs that use single `foldByKey` and satisfy Equations (2) and (3). (See [13, Lemma 7].)[7]

[7] Our approach is not sound if Equations (2) and (3) do not hold. To illustrate such a case, consider a hypothetical a case in which $P7'$ computes the histogram by deciles, $P8'$ by percentiles, and then both programs map all the elements to a constant, ignoring the aggregated value. $P7'$ produces at most 10 elements (one per decile), while $P8'$ produces at most 100, so they are clearly inequivalent.

$$getDecile = \lambda(sId, g).\ (g/10, sId)\,; \quad count = \lambda A, v.\ A + 1$$
$$isPassingDecile = \lambda(d, sId).d \geq 6\,; \quad isPassingGrade = \lambda(sId, g).g \geq 60$$

```
P7(R: Bag_{StudentID×Int}):                    P8(R: Bag_{StudentID×Int}):
R' = map(getDecile)(R)                          R' = filter(isPassingGrade)(R)
H = foldByKey(0, count)(R')                     R'' = map(getDecile)(R')
return filter(isPassingDecile)(H)               return foldByKey(0, count)(R'')
```

$$\begin{aligned}
&\big(d = g/10 \quad \textit{assumptions}\big)\\
&\quad\implies \Big(ite(d \geq 6, (d, 0), \bot) = (ite(g \geq 60, d, \bot), 0) && \textit{base case}\\
&\quad\wedge\big(ite(d \geq 6, (d, C), \bot) = (ite(g \geq 60, d, \bot), C') \implies && \textit{induction step}\\
&\qquad ite(d \geq 6, (d, C + 1), \bot) = (ite(g \geq 60, d, \bot), C' + 1)\big)\Big)
\end{aligned}$$

$$\forall g, g'.(g/10 = g'/10 \wedge g \neq \bot) \implies ite(g \geq 60, g/10, \bot) = ite(g' \geq 60, g'/10, \bot) \tag{2}$$
$$\forall g, g'.ite(g \geq 60, g/10, \bot) = ite(g' \geq 60, g'/10, \bot) \wedge ite(g \geq 60, g/10, \bot) \neq \bot \implies g/10 = g'/10 \tag{3}$$

Fig. 4. Equivalent Spark programs with aggregation by-key. All variables are universally quantified. If any component of the tuple is $\bot$, then the entire tuple is considered as $\bot$.

Decidability. Table 1 characterizes the programs for which our method is applicable, together with the strength of the method.[8] The example programs in Figure 1 are representative of programs that belong to the *NoAgg* class of programs, for which we have a decision procedure for verifying equivalence. We consider five classes of programs containing `fold` operations. Equivalence in $AggOne^p$ is undecidable, and the result is extended naturally to the special cases of $AggOneK^b$, $AggOne^b$ and $AggMult^p$. On the other hand, $AggOne^p_{sync}$ is a complete verification method. The equivalence of the programs in Figures 2 and 3 can be verified using $AggOne^p_{sync}$. Note that applying $AggOne^p_{sync}$ and $AggOneK^b$ require also checking the validity of Equation (1), respectively Equations (2) and (3). Fortunately, these requirements are expressed in Presburger arithmetic and thus can be decided.

Limitations We restrict ourselves to programs using *map*, *filter*, *cartesian product*, *fold*, and *foldByKey* where UDFs are defined in Presburger Arithmetic. We forbid self products—it is possible, but technically cumbersome, to extend our work to support self-products. However, supporting operators such as *union* and *subtract* can be tricky because of the bag semantics. Presburger arithmetic can be implemented with solvers such as Cooper's algorithm [9]. For simplicity we use Z3 which does not support full Presburger arithmetic, but supports the fragment of Presburger arithmetic used in this paper. Z3 also supports uninterpreted functions, which are useful to prove equivalence of other classes of Spark programs, but this is beyond the scope of this paper.

[8] Due to space considerations, we do not discuss equivalence of programs from mixed syntactic classes with comparable output types. In essence, there is a reduction from these instances such that one of the methods presented here will be applicable.

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	$::= \boldsymbol{f} \mid \boldsymbol{F}(\overline{e})$
Bag Expressions	μ	$::= \texttt{cartesian}(\mu, \mu') \mid \texttt{map}(f)(\mu) \mid \texttt{filter}(f)(\mu) \mid \boldsymbol{r}$
General Expressions	η	$::= e \mid \mu \mid \texttt{fold}(\boldsymbol{e}, f)(\mu) \mid \texttt{foldByKey}(\boldsymbol{e}, f)(\mu)$
Let expressions	E	$::= let\ \boldsymbol{x} = \eta\ in\ E \mid \epsilon$
Programs	$Prog$	$::= \boldsymbol{P}(\overline{\boldsymbol{r}: Bag_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v}:\boldsymbol{\tau}}) = \overline{Fdef}\quad \overline{PFdef}\quad E\ \eta$

Fig. 5. Syntax for SparkLite

3 The SparkLite language

In this section, we describe SparkLite, a simple functional programming language based on the operations provided by Spark [29].

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 that 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 bag m by $|m|$ and the empty bag by $\{\!\{\}\!\}$. We denote the i-th component of a tuple x by $p_i(x)$, and extend $p_i(\cdot)$ to bags containing tuples in the natural way.

SparkLite The syntax of SparkLite is defined in Figure 5. SparkLite supports two primitive types: *integers* (`Int`) and *booleans* (`Boolean`). On top of this, the user can define *record types* $\boldsymbol{\tau}$, which are tuples of primitive types, and *Bag*s:[9] $Bag_{\boldsymbol{\tau}}$ is (the type of) bags containing elements of type $\boldsymbol{\tau}$. We refer to primitive types and records 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 [13, Section 8].) We range over variables using $\boldsymbol{v}$ and $\boldsymbol{r}$ for variables of basic types and *Bag*, respectively.

A program $\boldsymbol{P}(\overline{\boldsymbol{r}: Bag_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v}:\boldsymbol{\tau}}) = \overline{Fdef}\,\overline{PFdef}\,E\ \eta$ 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 a sequence of the names and types of its input formal parameters, which may be *Bags* ($\overline{\boldsymbol{r}}$) or records or primitive types ($\overline{\boldsymbol{v}}$). The body of the program is comprised of two sequences of function declarations ($\overline{Fdef}$ and $\overline{PFdef}$), variable declarations (E), and the program's *main expression* (η). $\overline{Fdef}$ 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 returns a value of a basic type. $\overline{PFdef}$ 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*

[9] *Bag*s is an abstraction of the main data-structure used in Spark, called *RDD* [17,29,30].

Calculus [15], 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, `def addC` $= \lambda x\colon$ `Int.` $\lambda y\colon$ `Int.` $x + y\colon$ `Int` can create any first order function which adds a constant to its argument, e.g., `addC(1)` $= \lambda x\colon$ `Int.` $1 + x\colon$ `Int` and `addC(2)` $= \lambda x\colon$ `Int.` $2 + x\colon$ `Int`.

The program declares variables with a sequence of *let* expressions which bind general expressions to variables. A general expression is either a *basic expression* (e), a *bag expression* (μ), or an *aggregate expression* ($\texttt{fold}(\boldsymbol{e}, f)(\mu)$ or $\texttt{foldByKey}(\boldsymbol{e}, f)(\mu)$). The expression $\texttt{cartesian}(\mu, \mu')$ returns the cartesian product of μ and μ'. $\texttt{map}(f)(\mu)$ produces a *Bag* by applying the unary UDF f to every element x of μ. $\texttt{filter}(f)(\mu)$ evaluates to a copy of μ, except that all elements in μ which do not satisfy f are removed. The aggregate expression $\texttt{fold}(\boldsymbol{e}, f)(\mu)$ accumulates the results obtained by iteratively applying the binary UDF f to every element x in a *Bag* μ in some arbitrary order together with the accumulated result obtained so far, which is initialized to the *initial element* $\boldsymbol{e}$. If μ is empty, then $\texttt{fold}(\boldsymbol{e}, f)(\mu) = \boldsymbol{e}$. The $\texttt{foldByKey}(\boldsymbol{e}, f)$ operation applied on a *Bag* μ of record type $K \times V$ produces a *Bag* of pairs, where every key $k \in K$ which appears in μ is associated with the result obtained by applying $\texttt{fold}(\boldsymbol{e}, f)$ to the *Bag* containing all the values associated with k in μ.

We denote the meaning of a SparkLite program P by $[\![P]\!]$, which receives *input environments* ρ_0, assigning values to P's formal variables, to either bags or basic types. (See [13, Section 7].)

Remarks. We assume that the signature of UDFs given to either *map*, *filter*, *fold* or *foldByKey* match the type of the *Bag* on which they are applied. Also, to ensure that the meaning of $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ and $\texttt{foldByKey}(\boldsymbol{e}, f)(\boldsymbol{r})$ is well defined, i.e., we require, as Spark does [17], that f be commutative on its second argument: $\forall x, y_1, y_2. f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$.

4 Verifying Equivalence of SparkLite Programs

Programs P_1 and P_2 are *comparable* if they receive the same sequence of formal input parameters, and produce the same output type. They are *equivalent* if, in addition, for any input environment ρ_0, it holds that $[\![P_1]\!](\rho_0) = [\![P_2]\!](\rho_0)$. We assume that we only check the equivalence of comparable programs. Also, without loss of generality, we define programs without *let* expressions; as variables are never reassigned, these can always be eliminated by substituting every variable by its definition.We can now state our result regarding decidability of *NoAgg* programs, defined as programs without aggregate terms. (cf. [13, Section 9].)

Theorem 1. *The equivalence of programs in the NoAgg class is decidable.*

Unsurprisingly, however, equivalence in the general case is undecidable. The reduction in [13, Theorem 2] from the halting problem for 2-counter machines shows that verifying equivalence of $AggOne^p$ programs, is an undecidable problem.

Theorem 2. *The problem of deciding whether two arbitrary* $AggOne^p$ SparkLite *programs are equivalent is undecidable.*

4.1 Program Terms

The first step of our technique is the construction of *program terms*: Given a program P with main expression η, we generate a *program term* $\phi(P)$ which, roughly speaking, reflects the effect of the program on arbitrary elements taken from its input bags. It is obtained by applying the translation function ϕ, shown in Figure 6, on P's main expression. ϕ recursively traverses the expression and generates a logical term over the vocabulary of built-in operations and UDFs defined in P. The base case of the recursion is input bag variables r, which ϕ replaces with fresh variables $\mathbf{x}_r$. We refer to these variables as *representative variables*. Translation of a SparkLite operation on *Bags* produces a term corresponding to the application of its UDF on a single *Bag* element, which is a new bag expression: A $\texttt{map}(f)(\mu)$ operation is translated into the expression received by applying the lambda expression that corresponds to f, on the program term of μ. 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. The $\texttt{cartesian}(\mu, \mu')$ operation is translated to a pair of program terms pertaining to its arguments. Note that in the absence of aggregate operations, $\phi(\cdot)$ is a first-order term and thus can be used directly in formulas.

Aggregate operations require iterating over all the elements of μ. Therefore, it is clear that the translation of `fold` cannot be masqueraded as a first-order term. For $\texttt{fold}(e, f)(\mu)$ we are using a special operator $[\phi(\mu)]_{i,f}$, where $\phi(\mu)$ is the term pertaining to the bag being folded, i is the initial value, and f is the fold function. We refer to $[\phi(\mu)]_{i,f}$ as an aggregate term.

RepVarSet. For an expression μ consisting only of input bags and input parameters of basic types, $\mathit{RepVarSet}(\mu)$ denotes the set of all representative variables corresponding to the input bags appearing in μ. We can thus similarly define $\mathit{RepVarSet}(P)$ for the main expression of P. $\mathsf{FV}(P)$ denotes the entire set of free variables (both representative and non-bag inputs) in the program term of P.

Example 1. Consider the main expression $\eta = \texttt{filter}(\mathit{geq}(100))(\texttt{map}(\mathit{double})(R))$ of the program $P1''$ obtained by inlining the *let* expressions in program $P1$ (see Section 2), defining the doubling function as $\mathit{double} = \lambda x.2 * x$, and instantiating the parametric function $\mathit{geq} = \lambda y.\, \lambda x.x \geq y$ to act as the condition of the filter. The program term of $P1''$ is $\phi(P1'') = \mathit{ite}(2 * \mathbf{x}_R \geq 100, 2 * \mathbf{x}_R, \perp)$. Intuitively, we can learn how $P1''$ affects every element of, e.g., input *Bag* $\{\!\{2, 2, 103, 64\}\!\}$, by treating $\phi(P1'')$ as a "function" of $\mathbf{x}_R$ and "applying" it to 2, 2, 103, and 64. It is easy to see that $\mathsf{FV}(P1'') = \mathit{RepVarSet}(P1'') = \{\mathbf{x}_R\}$. Consider now instead $P5'$ also obtained by inlining of the *let* expressions in $P5$. In this case, $\phi(P5'') = [\mathbf{x}_R]_{+\infty,\lambda A,(x,y).\mathit{ite}(A<y,A,y)} = 100$.

4.2 Verifying Equivalence of SparkLite Programs with Aggregation

In this section, we discuss the generation of inductive hypotheses for programs with aggregations. We focus on the AggOne^p and AggOne^p_{sync} methods (recall Table 1), applicable on programs with a single fold operation. For space reasons, we

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

$\phi(e)$ is defined recursively based on the structure of e, e.g. $\phi(e_1 + e_2) = \phi(e_1) + \phi(e_2)$.

Fig. 6. A translation of a general expression to program terms.

relegate to [13, Section 13 and 14] the discussion of the other methods: $AggOne^b$, $AggMult^p$ and $AggOneK^b$, which are all sound techniques generalizing $AggOne^p$.

We note that in the presence of `fold` operations, The resulting terms are no longer legal terms in first order logic, and 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.

Verifying equivalence of $AggOne^p$ programs Arguably, the simplest class of programs with aggregations is the class of programs that return a primitive expression that depends on the result of the aggregation operation. Technically, a pair of SparkLite programs is in class $AggOne^p$ if each program P in the pair belongs to $AggOne^p$, i.e., there is a an expression g in Presburger Arithmetic with a single free variable x such that the program term of P is of the form $g[[\phi(\mu)]_{i,f}/x]$, where μ is a bag expression that does not include `fold` or `foldByKey` operations; that is, if $\phi(P)$ can be obtained by substituting x in g with the aggregate term pertaining to the application of a `fold` operation on μ. In the following, we refer to g as P's *top expression.* By abuse of notation, we use the functional notation $g(t)$ as a shorthand for $g[t/x]$, the expression obtained by substituting the term t with g's free variable. Similarly, given an expression e with two free variables x and y, we write $e(t_1, t_2)$ as a shorthand for $e[t_1/x, t_2/y]$.

Lemma 1 formalizes the sound method that we used in Section 2 to show that $P3$ and $P4$ (see Figure 2) are equivalent.

Lemma 1 (Sound method for verifying equivalence of Agg^1 programs). *Let P_1 and P_2 be $AggOne^p$ programs such that $\mathsf{FV}(P_1) = \mathsf{FV}(P_2)$. Assume that $\phi(P_1) = g_1([\phi(\mu_1)]_{i_1,f_1})$ and $\phi(P_2) = g_2([\phi(\mu_2)]_{i_2,f_2})$, 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 conditions hold:*

$$RepVarSet(\mu_1) = RepVarSet(\mu_2) \tag{4}$$

$$\mathbf{valid}\big(\forall \mathsf{FV}(P_1).\, g_1(i_1) = g_2(i_2)\big) \tag{5}$$

$$\mathbf{valid}\big(\forall \mathsf{FV}(P_1), M_1, M_2.\, g_1(M_1) = g_2(M_2) \implies g_1(e_1(M_1, \phi(\mu_1))) = g_2(e_2(M_2, \phi(\mu_2)))\big) \tag{6}$$

Intuitively, Equations (5) and (6) formalize the concept of inductive reasoning described in Section 2 for the base of the induction and the induction step, respectively. Equation (4) requires that the free variables of the folded bag expressions use the same representative variables. It ensures that the two `fold` operations iterate over bags of the same size. Note that we do not require that the bag folded by the two programs be equivalent. However, in Equation (6) we still use the fact that corresponding elements in the two folded bags can be produced by instantiating the program terms $e_{1,2}$ with corresponding elements from the input bags.

Complete verification techniques for subclasses of $AggOne^p$. Lemma 1 provides a sound, but incomplete, verification technique. This means that there are cases in which a pair of equivalent programs does not satisfy one or more of the requirements of Lemma 1. Luckily, some of these cases can be identified and subsequently have their equivalence verified using other methods. As a simple example, in [13] we show that the equivalence of SparkLite programs whose `fold` operations return a constant value can be reduce to the (decidable) problem of verifying equivalence of $NoAgg$ programs. We now describe the $AggOne^p_{sync}$ verification method.

In Section 2 we showed that although programs $P5$ and $P6$ do not satisfy the requirements of Lemma 1, we can verify their equivalence using a more specialized verification technique, $AggOne^p_{sync}$. We now present a more detailed discussion of $AggOne^p_{sync}$. We recall that the three main properties of pairs of programs that $AggOne^p_{sync}$ applies to are (1) both belong to $AggOne^p$; (2) the folds in both programs can be collapsed; and (3) the process of collapsing the folds can be done in synchrony.

The collapsing property states that any value produced by consecutive applications of the `fold` UDF can be obtained by 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 once on $a + b$. Also, recall that the bag being folded contains elements which are obtained via a sequence of `map`, `filter` and `cartesian` operations applied to elements taken out of the input bags. Synchronized collapsing occurs when given the same input elements to two consecutive applications of the `fold` UDF, it is possible to collapse them both using the same input element.

Thus, *synchronized collapsing* is a semantic property of `fold` UDFs, aggregated terms, and initial values of a pair of programs that belong to $AggOne^p_{sync}$. In the following, we denote by $\mathsf{FV_r}(P)$ and $\mathsf{FV_b}(P)$ the subsets of $\mathsf{FV}(P)$ comprised of bag, respectively, non-bag, input formal parameters.

Definition 1 (The $AggOne^p_{sync}$ class). *Let P_1 and P_2 be $AggOne^p$ programs such that $\mathsf{FV}(P_1) = \mathsf{FV}(P_2)$. Assume that $\phi(P_1) = g_1([\phi(\mu_1)]_{i_1,f_1})$ and $\phi(P_2) = g_2([\phi(\mu_2)]_{i_2,f_2})$, where $f_1 = \lambda x, y.\, e_1$ and $f_2 = \lambda x, y.\, e_2$. We say that P_1 and P_2 belong together to $AggOne^p_{sync}$, denoted by $\langle P_1, P_2\rangle \in AggOne^p_{sync}$, if the following conditions hold:*

$$RepVarSet(\mu_1) = RepVarSet(\mu_2) \tag{7}$$

$$\begin{aligned}\forall \bar{b}, \bar{u}, \bar{v}. \exists \bar{w}.\, & e_1(i_1, \phi(\mu_1))[\bar{b}/\mathsf{FV_b}, \bar{w}/\mathsf{FV_r}] = \\ & \qquad e_1((e_1(i_1, \phi(\mu_1))[\bar{b}/\mathsf{FV_b}, \bar{u}/\mathsf{FV_r}], \phi(\mu_1)[\bar{b}/\mathsf{FV_b}, \bar{v}/\mathsf{FV_r}]) \\ & \wedge e_2(i_2, \phi(\mu_2))[\bar{b}/\mathsf{FV_b}, \bar{w}/\mathsf{FV_r}] = \\ & \qquad e_2((e_2(i_2, \phi(\mu_2))[\bar{b}/\mathsf{FV_b}, \bar{u}/\mathsf{FV_r}], \phi(\mu_2)[\bar{b}/\mathsf{FV_b}, \bar{v}/\mathsf{FV_r}])\end{aligned} \tag{8}$$

Note that in Equation (8), all applications of the `fold` UDF functions agree on the values of the non-bag input formal parameters used to "generate" the accumulated elements. Also note that checking if $\langle P_1, P_2\rangle \in \mathit{AggOne}^p_{sync}$ involves determining the validity of an additional decidable formula, namely Equation (8). Theorem 3 shows that verifying the equivalence of a pair of programs in AggOne^p_{sync} effectively reduces to checking a single application of the `fold` UDFs.

Theorem 3 (Equivalence in AggOne^p_{sync} is *decidable*). *Let P_1 and P_2 be AggOne^p programs as in Lemma 1, such that $\langle P_1, P_2\rangle \in \mathit{AggOne}^p_{sync}$. P_1 and P_2 are equivalent if and only if the following holds:*

$$\mathbf{valid}(\forall \mathsf{FV}(P_1).\, g_1(i_1) = g_2(i_2)) \tag{9}$$

$$\mathbf{valid}\begin{pmatrix} \forall \bar{v}, \bar{w}, M_1, M_2.\, \big(\, M_1 = e_1(i_1, \phi(\mu_1)[\bar{v}/\mathsf{FV}(P_1)])\wedge \\ \qquad\qquad M_2 = e_2(i_2, \phi(\mu_2)[\bar{v}/\mathsf{FV}(P_1)])\big) \implies \mathit{Ind} \end{pmatrix}$$

$$\textit{where } \mathit{Ind} = \big(g_1(M_1) = g_2(M_2) \implies \tag{10}$$
$$g_1(e_1(M_1, \phi(\mu_1))) = g_2(e_2(M_2, \phi(\mu_2)))\big)[\bar{w}/\mathsf{FV}(P_1))]$$

5 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 Windows host with a quad core 3.40 GHz Intel Core i7-6700U processor, with 32GB 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.

A total of 23 test-cases of both equivalent and non-equivalent instances were tested, including all the examples from this paper. In Figure 7, we highlight test cases inspired by real Spark uses taken from [17, 28] and online resources (e.g., open-source Spark clients), and belong to one of the defined SparkLite classes. The full list of tested programs appears in [13, Section 15]. 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. Each example was analyzed in less than 0.5 seconds. It is also interesting to note that most examples with a primitive aggregation output are verified using AggOne^p_{sync} and not AggOne^p, indicating that the AggOne^p_{sync} class is not esoteric, but wide enough to cover useful programs. Our tool was able to prove the equivalence of all equivalent programs, and find counterexamples for inequivalent ones, with the exception of . $P15''$ and $P16''$ which belong to AggOne^p. While it is immediate that these programs are equivalent (we note the intermediate fold results in both programs are the same, and apply the same transformation on the fold result), our tool was not able to show the equivalence. This is because the AggOne^p_{sync} technique is not applicable to this particular example, as *count* is not a collapsible fold function, and the AggOne^p technique is effective only when the equivalence claim is inductive, which is not the case here.

Test	**Description**	**Eq.**	**Ver.**	**Method**
$P1$,$P2$	From Section 2. Showing map and filter commutativity.	Y	Y	$NoAgg$
$P1$,$P2'$	$P2$ changed to filter elements smaller than 100.	N	Y	$NoAgg$
$P3$, $P4$	From Section 2. Also proved using $AggOne^p_{sync}$.	Y	Y	$AggOne^p$
$P5$, $P6$	From Section 2.	Y	Y	$AggOne^p_{sync}$
$P7$, $P8$	From Section 2. Describe distribution of passing students' grades.	Y	Y	$AggOneK^b$
$P9$, $P10$	Distributivity of map UDFs with respect to join.	Y	Y	$NoAgg$
$P9'$, $P10$	Map UDFs which are not distributive with respect to join.	N	Y	$NoAgg$
$P11$, $P12$	Distributivity of filter UDFs with respect to join.	Y	Y	$NoAgg$
$P13$, $P14$	Count on a filtered bag / sum on a bag mapped to a constant (0/1).	Y	Y	$AggOne^p$
$P15$, $P16$	Modular arithmetic: Divisibility by 5 of the sum of the elements, vs. divisibility by 5 of the sum of the elements, each multiplied by 3.	Y	Y	$AggOne^p_{sync}$
$P15'$, $P16'$	Modular arithmetic: Divisibility by 6 instead of 5 is not retained.	N	Y	$AggOne^p_{sync}$
$P15''$, $P16''$	Modular arithmetic: Divisibility by 5 of the elements' count, vs. divisibility by 5 of the count after multiplying the elements by 3.	Y	N	$AggOne^p$
$P17$, $P18$	Maximum is expressed as inverted minimum of inverted elememts.	Y	Y	$AggOne^p_{sync}$
$P17'$, $P18$	As above, but there is a bug in the initial value of the maximum.	N	Y	$AggOne^p_{sync}$
$P19$, $P20$	Summation (by key) of positive vs. non-negative integers.	Y	Y	$AggOneK^b$
$P21$, $P22$	Summation of both keys and values in different ways.	Y	Y	$AggOneK^b$

Fig. 7. Highlighted test cases. Note that the join operator was implemented as a combination of `cartesian`, `filter` and `map` operations, with designated UDFs.

6 Related Work and Conclusion

The problem considered (i.e., determining equivalence of expressions accessing a dataset) is a classic topic in database theory. Query containment and equivalence were first studied in seminal work by Chandra et al. [2]. This work was extended in numerous papers, e.g., [18] for queries with inequalities and [4] for acyclic queries. Of most relevance to this paper are the extensions to queries evaluated under bag and bag-set semantics [3], and to aggregate queries, e.g., [7, 8, 14]. The latter papers consider specific aggregate functions, such as min, count, sum and average, or aggregate functions defined by operations over abelian monoids. In comparison, we do not restrict UDFs to monoids, and provide a different characterization for decidability.

In the field of verification and programming languages, several works address properties of relational algebra operators. Most notably, *Cosette* [6], is a fully automated prover for SQL equivalences, which provides a proof or a counterexample to equivalence by utilizing both a theorem prover and a solver. The approach supports standard SQL features as well as predetermined aggregation functions such as count, sum, and average. On the other hand, by addressing Spark programs, our approach focuses on custom UDFs for selects, projections, and aggregation. Similarly, *Spec#* [20] has a fixed set of comprehensions such as sum, count, min and max, fitted into templates with both filters and expression terms akin to map, which are encoded into the SMT solver using specialized axioms, e.g. the distribution of plus over min/max. Our techniques, on the other hand, extract automatically properties of comprehensions to define suitable verification conditions for equivalence. El Ghazi et al. [11] took the SMT solver approach to verify relational constraints in Alloy [16], in order to be able to provide proofs, and not just counterexamples. There is, however, no guarantee on completeness, or the ability of the solver to provide a proof. It differs from this work, which carefully defines criteria for decidability and soundness, even in the expense of expressivity. Loncaric et al. [21] utilize a small-model property of sets to verify synthesized data structures which is similar to the one we leverage in the $NoAgg$ method. We extend this property to bags and aggregate operations. Smith and Albarghouthi [25] presented an algorithm for synthesizing Spark programs by analyzing user examples fitted into higher-order sketches. They use SMTs to verify commutativity of the *fold* UDFs. Chen et al. [5], studied the decidability of the latter problem. We use SMT to verify program equality assuming that the *fold* UDFs are commutative. In this sense, our approaches are complementary.

There are also generic frameworks for verifying functional programs, such as *F** [27] and *Liquid Types* [23, 24]. These prove program safety via type checking, which also utilizes SMT to check validity of implications. Both approaches require additional manual effort to verify programs like the ones we explore: in *Liquid Types*, there is no notion of equivalence, so a suitable summary must be given that holds for both programs. In *F**, equivalence can be expressed via assertions, but verifying assertions in *F** is incomplete with respect to inductive data types, such as lists. Appropriate invariants must be provided manually, essentially the same ones that are constructed automatically in this paper. Another approach to verifying functional programs is applied by *Leon* [1, 26], whose engine is based on decision procedures for the quantifier-free theory of algebraic data types with different fold functions, which allow handling recursive functions with first-order constraints. However, the approach relies on finite unrolling of the recursive calls, thus it cannot verify the equivalence of two programs when the equivalence property is not inductive by itself. In contrast, our approach is successful because of the novel specialized treatment of synchronous collapsible UDFs.

Dafny [19] supports functional programming, inductive data types, higher-order functions, and also provides some automatic induction. Dafny can automatically verify our $NoAgg$ test cases. However, applying it to certain $AggOne^p$ programs required supplying auxiliary lemmas. For example, verifying the equivalence of $P15$ and $P16$ required the use of a lemma asserting that multiplying the sum of elements in a bag by three produce the same result as summing the bag obtained by multiplying every element by three. Essentially, the lemma establishes equivalence relations between subprograms, and gives rise to a possible heuristic extension of our tool by searching for relations between subprograms.

Conclusion. The main conceptual contribution of this paper is that the problem of checking program equivalence of SparkLite programs, which reflect an interesting subset of Spark programs, can be addressed via a reduction to the validity of formulas in a decidable fragment of first-order logic. We believe the foundations laid in this paper will lead to the development of tools that handle formal verification and optimization of more classes of programs written in Spark and similar frameworks, e.g., ones with nested aggregations and unions.

References

1. Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.
2. 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.
3. 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.
4. Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211 – 229, 2000.
5. Yu-Fang Chen, Chih-Duo Hong, Nishant Sinha, and Bow-Yaw Wang. *Commutativity of Reducers*, pages 131–146. Springer Berlin Heidelberg, 2015.
6. Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.
7. Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Deciding equivalences among conjunctive aggregate queries. *J. ACM*, 54(2), 2007.
8. Sara Cohen, Yehoshua Sagiv, and Werner Nutt. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Logic*, 6(2):328–360, April 2005.
9. David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.
10. 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.
11. Aboubakr Achraf El Ghazi and Mana Taghdiri. *Relational Reasoning via SMT Solving*, pages 133–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
12. Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. Technical report, Massachusetts Institue of Technology, Cambridge, MA, USA, 1974.
13. Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of Spark programs. Tech. rep., Tel Aviv University, April 2017. URL: http://www.cs.tau.ac.il/%7Eshellygr/pubs/sparkeq-tr.pdf.
14. Stéphane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. On the equivalence and rewriting of aggregate queries. *Acta Inf.*, 40(8):529–584, 2004.
15. Masahito Hasegawa. *Decomposing typed lambda calculus into a couple of categorical programming languages*, pages 200–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.
16. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
17. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics.* O'Reilly Media, Inc., 1st edition, 2015.
18. Anthony Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, January 1988.

19. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
20. K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order smt solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 615–622, New York, NY, USA, 2009. ACM.
21. Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 355–368, New York, NY, USA, 2016. ACM.
22. Mojżesz Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervor. *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.
23. Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 159–169. ACM, January 2008.
24. Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 131–144. ACM, January 2010.
25. 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.
26. Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 199–210, New York, NY, USA, 2010. ACM.
27. Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.
28. 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.
29. 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.
30. 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.