Verifying Equivalence of Spark Programs

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

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

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

3 **Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA**
shachari@mit.edu

Abstract

In this thesis, we present a novel approach for verifying the equivalence of Spark programs. Spark is a popular framework for writing large scale data processing applications. Such frameworks, intended for data-intensive operations, share many similarities with traditional database systems, but do not enjoy a similar support of optimization tools. Our goal is to enable such optimizations by first providing the necessary theoretical setting for verifying the equivalence of Spark programs. This is challenging because such programs combine relational algebraic operations with *User Defined Functions* (*UDF*s) and aggregate operations.

We define a model of Spark as a programming language which imitates Relational Algebra queries in the bag semantics and allows for user defined functions expressible in Presburger Arithmetics as well as aggregations. We present a technique for verifying the equivalence of an interesting class of Spark programs, and show that it is complete under certain restrictions.

1 Introduction

Unlike traditional relational databases, which are accessed using a standard query language, NoSQL databases are often accessed via an entire program. As NoSQL databases are typically huge, optimizing such programs is an important problem. Unfortunately, although query optimization has been studied extensively over the last few decades, the techniques presented in the past no longer carry over when access to the data is via a rich programming language.

Standard query optimization techniques are often based on the notion of query equivalence, i.e., the ability to determine if different queries are guaranteed to return the same results over all database inputs. If one can decide equivalence between queries, it may be possible to devise a procedure that simplifies a given query to derive a cheaper, yet equivalent, form. In addition, the ability to decide query equivalence is a necessary component to allow rewriting of queries with views or previously computed queries. This, again, is an important optimization technique.

This paper studies the equivalence problem for fragments of the Spark programming language. Spark is a popular framework for writing large scale data processing applications. It 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 user defined function (UDF). Intuitively, map, filter and fold can be seen as extensions to project, select


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and aggregation, respectively, with arbitrary UDFs applied. Besides accessing RDDs, Spark programs can also perform arithmetic and Boolean computations.

Unsurprisingly, in the general case, the problem of determining equivalence between Spark programs is undecidable. Thus, this paper focuses on fragments of Spark programs, and gives a sound condition for equivalence. For special cases, we also provide conditions that are complete for determining equivalence. In addition, we show that, even within our limited fragments, there are still undecidable cases.

The techniques used in this paper for determining equivalence of Spark programs are rather different than traditional query equivalence characterizations. Query equivalence is usually determined by either *(1)* showing that equivalence boils down to the existence of some special mapping (e.g., isomorphism, homomorphism) between the queries *or (2)* proving that equivalence over arbitrary databases can be determined by checking for equivalence over some finite set of canonical databases. (Actually, often, both such characterizations are shown.) In this paper we use a different approach. Intuitively, we present a method to translate Spark programs into an augmented version of Presburger arithmetic. Since equivalence of Presburger arithmetic is decidable, we derive decidability of Spark programs. We note that some of the intricacies arise from the fact RDDs are bags (and not individual items), and Spark programs can contain aggregation (using the fold operation).

2 Overview

We begin with a few example programs representative of the primary results of this paper. The programs are written in a simple language we define, called SparkLite. SparkLite is a general purpose programming language with a special data type called *RDD* and operations on its. Programs are written using sequential composition of atomic commands, RDD operations and conditions, but no loops. Below we show two equivalent programs working on a collection of integers. Both programs compute a new collection containing all the elements of the input collection which equal to or bigger than 50, multiplied by two. The programs operate differently: $P1$ first multiplies, then filters, while $P2$ goes the other way around.

	P1(R: $RDD_{\texttt{Int}}$)**:**	**P2**(R: $RDD_{\texttt{Int}}$)**:**
1	$R'_1 = \texttt{map}(\lambda x.2 * x)(R)$	$R'_2 = \texttt{filter}(\lambda x.x \geq 50)(R)$
2	$R''_1 = \texttt{filter}(\lambda x.x \geq 100)(R')$	$R''_2 = \texttt{map}(\lambda x.2 * x)(R')$
2	`return` R''_1	`return` R''_2

`map` and `filter` are operations that apply a function on each element in the collection, and yield a new collection. For example, let RDD R be the bag $R = \{\!\{2, 2, 103, 64\}\!\}$ (note that repetitions are allowed). We give R as an input to 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. The latter contains the elements of R'_1 which are greater or equal to 100, i.e., $R''_1 = \{\!\{206, 128\}\!\}$. The second program begins by applying the filter operator. The latter produces an RDD R'_2 by throwing away all the elements in R which are smaller than 50, resulting in the RDD $R'_2 = \{\!\{103, 64\}\!\}$. After $P2$ applies the map operator, it ends up with an RDD R''_2 which contains the same elements as R''_1. Hence, both program 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: $ite(2 * x < 100, 2 * x, \bot)$, and $P2$ as: $(\lambda x.2 * x)(ite(x < 50, x, \bot))$, where ite denotes the if-then-else operator. 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 $\perp$ records the deletion of an element due to failing to satisfy the condition checked by the `filter` operation. The formula whose validity attests for the equivalence of $P1$ and $P2$ is thus:

$$\forall x.ite(2 * x < 100, 2 * x, \perp) = (\lambda x.2 * x)(ite(x < 50, x, \perp)) .$$

In this thesis, the functions given to the operations are expressed in (extended) Presburger Arithmetic (the theory of integers with addition, without multiplication). As a result, the validity of the produced formulae can be decided.

Next, we consider programs with aggregate operations. Aggregate operations encapsulate a loop over all elements in an RDD, which returns a single result computed from all elements. For example, the following equivalent programs check if the sum of all elements in the RDD R is divisible by 5:

	P3(R: $RDD_{\texttt{Int}}$)**:**	**P4**(R: $RDD_{\texttt{Int}}$)**:**
1	s = `fold`$(0, \lambda A, x.A + x)(R)$	R' = `map`$(\lambda x.3 * x)(R)$
2	`return` $s\%5 = 0$	s' = `fold`$(0, \lambda A, x.A + x)(R')$
3		`return` $s'\%5 = 0$

We illustrate the way these programs work using RDD $R = \{\!\{1, 2, 3, 4, 5\}\!\}$ as an example. In $P3$, the `fold` operator in the first line yields the value $s = 0 + 1 + 2 + 3 + 4 + 5 = 15$. The `fold` operator iterates over the elements of R in some arbitrary order, and sums their value starting from 0, its first argument, as the initial value. We note that had R been empty, s would have been set to 0. The program returns true because $15\%5 = 0$. In $P4$, we apply a map on R, yielding $R' = \{\!\{3, 6, 9, 12, 15\}\!\}$. The `fold` in $P4$ computes that $s' = 3 + 6 + 9 + 12 + 15 = 45$, and $P4$ returns true too as $45\%5 = 0$.

To verify the equivalence of the programs, we again encode them symbolically. We encode the summation of the elements in R as $[x]_{0,\lambda A,x.A+x}$. The $[\cdot]$ operator indicates we apply an iterative operator, so in conventional mathematical notation we would write the above summation as: $\Sigma_{x \in R} x$. In our example, we have the following encoding for $P3$: $[x]_{0,\lambda A,x.A+x}\%5 = 0$ and similarly for $P4$: $[3 * x]_{0,\lambda A,x.A+x}\%5 = 0$. For equivalence, we require:

$$(*)[x]_{0,\lambda A,x.A+x}\%5 = 0 \iff [3x]_{0,\lambda A,x.A+x}\%5 = 0$$

To handle the equivalence condition we use an inductive claim—the programs should be equal in the initial state (assuming the RDDs are empty), and in every step of the implicit loop in `fold` (which goes over all elements and performs the aggregation). We show such a check is guaranteed to be sound, but not necessarily complete, as it is possible for aggregations to diverge during the fold loop's run, and converge in the end. However, there are cases where naive induction is complete. One such case is one where the fold functions and the term for the elements on which we apply the fold operator behave as distributive functions. For these cases, it is enough to show the equality on the initial step, and the first step of the aggregation. $P3$ and $P4$ above satisfy the condition for completeness. For the initial state in $(*)$, both programs return true, as both return the boolean value of $0\%5 = 0$. We note that the completeness condition is satisfied, as every pair of applications of the sum function can be written as a single summation: $\exists z.(A + x) + y = A + z \wedge (A + 3x) + 3y = A + 3z$, and $z = x + y$. We can thus shrink the sequence of applications of the fold functions, and prove the equivalence for the first step of the induction only, instead for the nth. Therefore, it is enough to show $\forall x.x\%5 = 0 \iff 3x\%5 = 0$ to prove $(*)$.

We can find a wide range of example programs for which equivalence can be proven using direct induction. For example, suppose we maintain a database table of products and their prices, and we write a program that verifies all products' prices are not below some threshold, say 100$. Then, we wish to make a discount, while maintaining the price threshold. So we give a 20% discount for all products whose prices are above 125$, and give no discount otherwise: $discount((prod, p)) = \textbf{if } p \geq 125 \textbf{ then } (prod, 0.8p) \textbf{ else } (prod, p)$. To verify that our discount program preserves the threshold, we verify if the below programs are equivalent. The input RDD for $P5$,$P6$ is a of 2-tuple of integers, the first element in the tuple representing a product ID, and the second element in the tuple representing the price of that product. $P5$ calculates the minimum over the prices (in database terminology, it can be understood as aggregation on the second column), and assigns the aggregated result to the variable $minP$. $P5$ returns a boolean value representing whether $minP$ is at least 100 or not. $P6$ first applies the *discount* mapping specified earlier, and then applies fold to calculate the minimal price after applying the discount, assigning the result to $minDiscountP$. Like $P5$, it checks if the minimum is at least 100.

Let: $min2 = \lambda A, (x, y).min(A, y)$

	P5(R: $RDD_{\texttt{Prod}\times\texttt{Int}}$)**:**	**P6**(R: $RDD_{\texttt{Prod}\times\texttt{Int}}$)**:**
1	$minP = \texttt{fold}(+\infty, min2)(R)$	$R' = \texttt{map}(\lambda(prod, p).discount((prod, p)))(R)$
2	`return` $minP \geq 100$	$minDiscountP = \texttt{fold}(+\infty, min2)(R)$
3		`return` $minDiscountP \geq 100$

The porgrams equivalence can be written in our encoding of programs as formulas as follows:

$$[(prod, p)]_{+\infty, \lambda A,(x,y).min(A,y)} \geq 100 \iff [discount((prod, p))]_{+\infty, \lambda A,(x,y).min(A,y)} \geq 100$$

To prove it, we apply induction. In the induction base, we check for empty RDDs, for which both programs return true. Otherwise, we assume that after n prices checked, the minimum M in the first program, and the minimum M' in the second program, are simulatenously at least 100 or not. The programs will be equivalent, if this invariant is kept after checking the next price. The formula for this invariant is:

$$\begin{aligned}&\forall (prod, p), M, M'.(M \geq 100 \iff M' \geq 100) \implies \\ &\quad (min(M, p) \geq 100 \iff min(M', p_2(discount((prod, p)))) \geq 100)\end{aligned}$$

3 Main Results

The main contributions of this paper are as follows:

- We present a simplified model of Spark programs over RDDs by defining a programming language called *SparkLite*, in which UDFs are expressed as simply typed λ-calculus restricted to Presburger arithmetics (see Section 6).
- We show that the equivalence of arbitrary SparkLite programs is undecidable (Section 8.2.1).
- We define a sound method for checking equivalence of a broad class of SparkLite programs (Sections 8.2 and 8.2.4).
- We introduce an interesting and nontrivial subclass of SparkLite in which checking program equivalence is decidable, called $AggPair^1_{sync}$. Interestingly, programs in $AggPair^1_{sync}$ allow both UDFs and aggregations. The decidability of the equivalence is proven by observing that common SparkLite aggregates are *closed* in the sense that operations composed one on another can be simulated by a single operation (Section 8.2.3).

4 Outline

In Section 5 we present the preliminaries for this paper. Section 6 introduces the syntax and operational semantics of the SparkLite language, which is modeling conjunctive queries with aggregations and UDFs. In Section 7, we present alternative syntax with equivalent semantics for SparkLite, and a compilation procedure from the natural syntax of SparkLite to a symbolic one. This novel presentation of SparkLite allows to verify various properties based on the programs' representation as logical terms, and program equivalence in particular, as is discussed in Section 8. There, we first show that the general program equivalence problem is undecidable for SparkLite. Following it is a discussion of various classes of SparkLite, focusing on the decidability of the program equivalence problem. We present a decision procedure for conjunctive queries without aggregations (which is unlike known existing algorithms for the problem), and more interestingly, for a subclass of SparkLite programs with aggregations. For other classes, we present sound algorithms for verifying the equivalence. **??** shows related work on the subject of query and program equivalence. We conclude and present future work in Section 9.

5 Preliminaries

In this section, we describe a simple extension of Presburger arithmetic [10], which is the first-order theory of the natural numbers with addition, to tuples of integers, and state its decidability.

Notations. We denote the sets of natural numbers, positive natural numbers, and integers by $\mathbb{N}$, $\mathbb{N}^+$, and $\mathbb{Z}$, respectively. We denote a (possibly empty) sequence of elements coming from a set X by $\overline{X}$. We write $ite(p, e, e')$[1] to denote an expression which evaluates to e if p holds and to e' otherwise. We use $\perp$ to denote the *undefined* value. 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 $\{\!\{\}\!\}$.

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 1.[2] 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 [5, 10], even when combined with Boolean logic [2, 7],[3], and infinities [8].[4] For

[1] ite is shorthand for if-then-else
[2] We assume the reader is familiar with FOL, and omit a more formal description for brevity.
[3] 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].)
[4] We denote inifinities as $+\infty, -\infty \in \mathbb{Z}$.

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.

Figure 1 Terms of the Augmented Presburger Arithmetic

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	re	::=	$\texttt{cartesian}(\boldsymbol{r}, \boldsymbol{r}) \mid \texttt{map}(f)(\boldsymbol{r}) \mid \texttt{filter}(f)(\boldsymbol{r})$
Aggregation Exp.	ge	::=	$\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$
General Expressions	η	::=	$e \mid re \mid ge$
Let expressions	E	::=	$Let\ \boldsymbol{x} = \eta\ in\ E \mid \eta$
Programs	$Prog$	::=	$\boldsymbol{P}(\overline{\boldsymbol{r} : RDD_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v} : \boldsymbol{\tau}}) = \overline{Fdef}\quad \overline{PFdef}\quad E$

The syntax of basic expressions e is defined in Figure 1

Figure 2 Syntax for SparkLite

example, *Cooper's Algorithm* [4] is a standard decision procedure for Presburger Arithmetic[5].

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 pointwise 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.*

It will be useful in the next sections 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 we later show how formulas in APA$^+$ can be converted to APA formulas.

6 The SparkLite language

In this section, we define the syntax of SparkLite, a simple functional programming language which allows to use Spark's *resilient distributed datasets* (*RDDs*) [12].

[5] 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 [9]. However, in practice, our experiments show that Cooper's algorithm on most non-trivial formulas in this paper returns almost instantly, even on commodity hardware.

6.1 Syntax

The syntax of SparkLite is defined in Figure 2. 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: $RDD_{\boldsymbol{\tau}}$ 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 denote the set of (syntactic) variable names by `Vars`, and range over it using $\boldsymbol{v}$, $\boldsymbol{b}$, and $\boldsymbol{r}$, for variables of type integer and records, boolean, and *RDD*, respectively.

A program $\boldsymbol{P}(\boldsymbol{r} : \overline{RDD_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v} : \boldsymbol{\tau}}) = \overline{Fdef}\ \overline{PFdef}\ E$ 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 name and types of its input parameters, which may be *RDDs* ($\overline{\boldsymbol{r}}$) or integers ($\overline{\boldsymbol{v}}$). The body of the program is comprised of two sequences of function declarations ($\overline{Fdef}$ and $\overline{PFdef}$) and the program's *main expression* (E). $\overline{Fdef}$ 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 return a value of a basic type. For example,

$$\texttt{def isOdd} = \lambda y{:}\,\texttt{Int}.\ \neg(y \,\%\, 2 = 0){:}\,\texttt{Boolean}$$

defines `isOdd` to be a function which determines whether its integer argument `y` is odd or not. $\overline{PFdef}$ associates function names $\boldsymbol{F}$ with a restricted form of second-order lambda expressions, which we refer to as *parametric functions*.[6] 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,

$$\texttt{def addC} = \lambda x{:}\,\texttt{Int}.\ \lambda y{:}\,\texttt{Int}.\ x + y{:}\,\texttt{Int}$$

allows to create any first order function which adds a constant to its argument, e.g., `addC(1)` is the function $\lambda x{:}\,\texttt{Int}.\ 1 + x{:}\,\texttt{Int}$ which returns the value of its input argument incremented by one.

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* (re) or an *aggregate expression* (ge). A basic expression is a term in the Augmented Presburger Arithmetics (see Section 5). The *RDD* expression $\texttt{cartesian}(\boldsymbol{r}, \boldsymbol{r'})$ returns the cartesian product, under bag semantics, of *RDDs* $\boldsymbol{r}$ and $\boldsymbol{r'}$. The *RDD* expressions `map` and `filter` generalize the *project* and *select* operators in *Relational Algebra* (RA) [1, 3], with *user-defined functions* (*UDFs*): $\texttt{map}(f)(\boldsymbol{r})$ evaluates to an *RDD* obtained by applying the UDF f to every element x of *RDD* $\boldsymbol{r}$, with the same multiplicity x had in $\boldsymbol{r}$. $\texttt{filter}(f)(\boldsymbol{r})$ evaluates to a copy of $\boldsymbol{r}$, except that all elements in $\boldsymbol{r}$ which do not satisfy f are removed. The *aggregation expression* `fold` is a generalization of aggregate operations in SQL, e.g., `SUM` or `AVERAGE`, with *UDFs*: $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ accumulates the results obtained by iteratively applying f to every element x in $\boldsymbol{r}$, starting from the *initial element* e and applying f a total of $\boldsymbol{r}(x)$ times for every $x \in \boldsymbol{r}$. If $\boldsymbol{r}$ is empty, then $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r}) = \boldsymbol{e}$

▸ Remark. To ensure that the meaning of $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ is well defined, we require that f be a commutative function, in the following sense: $\forall x, y_1, y_2.f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$. Also, as is common in functional languages, we assume that variables are never reassigned.

[6] Parametric functions were inspired by the *Kappa Calculus* [6], which contains only first-order functions, but allows lifting them to larger product types, which is exactly the purpose of parametric functions in SparkLite.

$$isOdd = \lambda x\colon \texttt{Int}.\ \neg(x \,\%\, 2 = 0)$$

Let: $doubleAndAdd = \lambda c\colon \texttt{Int}.\lambda x\colon \texttt{Int}.\ 2 * x + c$

$$sumFlatPair = \lambda A\colon \texttt{Int}, (x, y)\colon \texttt{Int} \times \texttt{Int}.\ A + x + y$$

$\mathbf{P1}(R_0\colon RDD_{\texttt{Int}}, R_1\colon RDD_{\texttt{Int}})$**:**

1 $A = \texttt{filter}(isOdd)(R_0)$
2 $B = \texttt{map}(doubleAndAdd(1))(A)$
3 $C = \texttt{cartesian}(B, R_1)$
4 $v = \texttt{fold}(0, sumFlatPair)(C)$
5 `return` v

Figure 3 Example SparkLite Program

▸ Remark. We assume that the signature of UDFs given to either *map*, *filter*, or *fold* should match to the type of the RDD on which they are applied.

▸ **Example 1.** Consider the example SparkLite program in Figure 3. From the example program we can see the general structure of SparkLite programs: First, the functions that are used as UDFs in the program are declared and defined: $isOdd, sumFlatPair$ defined as *Fdef*, and *doubleAndAdd* defined as a *PFdef*. The name of the program ($\boldsymbol{P} = P1$) is declared with a list of input RDDs (R_0, R_1). Instead of writing *let* l_1 *in let* l_2 *in* ..., we use syntactic sugar, where each line of code contains a single l_i, and the last line denotes the return value using the `return` keyword. Here, three variables of RDD type (A, B, C) and one integer variable (v) are bound by *let*s. We can see in the definition of A an application of the *filter* operation, accepting the RDD R_0 and the function *isOdd*. For B's definition we apply the *map* operation with a parametric function *doubleAndAdd* with the parameter 1, which is interpreted as $\lambda x.\ 2 * x + 1$. C is the cartesian product of B and input RDD R_1. We apply an aggregation using *fold* on the RDD C, with an initial value 0 and the function $sumFlatPair$, which 'flattens' elements of tuples in C, taking their sum. The sum of all these elements is stored in the variable v. The returned value is the integer variable v. The program's signature is $P1(RDD_{\texttt{Int}}, RDD_{\texttt{Int}})\colon \texttt{Int}$.

6.2 Operational Semantics

In this section, we define the operational semantics of SparkLite,which allows to evaluate programs.

Program Environment. We define a unified semantic domain $\mathcal{D}$ for all types in SparkLite. The *program environment* type: $\mathcal{E} = \texttt{Vars} \to \mathcal{D}$ is a mapping from each variable in `Vars` to its value, according to type. A variable's type does not change during the program's run, nor does its value. For simplicity, the environment also stores the text of function. We denote the meaning of a function f by $[\![f]\!]$, e.g. $[\![\lambda x.2 * x]\!] = \lambda x.2 * x$

Data flow. We start with an initial environment function ρ_0 that maps all input variables and function definitions. We define the *semantic interpretation* of expressions based on an environment $\rho \in \mathcal{E}$, and specifically for $x \in \overline{\boldsymbol{r}} \cup \overline{\boldsymbol{v}}$, $[\![\boldsymbol{x}]\!](\rho) = \rho(\boldsymbol{x})$. The semantics of composite expressions are straight-forward using the semantics of their components. The semantics of *let* is to create a new environment by binding the variable name. In Figure 4 we specify the behavior of $[\![\cdot]\!](\cdot)$ for all expressions and statements.

$$
\begin{array}{lcl}
[\![c]\!](\rho) & = & c \\
[\![\boldsymbol{v}]\!](\rho) & = & \rho(v) \\
[\![\texttt{unOp}\, e]\!](\rho) & = & unOp\,[\![e]\!](\rho) \\
[\![e_1\, \texttt{binOp}\, e_2]\!](\rho) & = & [\![e_1]\!](\rho)\, binOp\, [\![e_2]\!](\rho) \\
[\![(e_1, \cdots, e_n)]\!](\rho) & = & \big([\![e_1]\!](\rho), \cdots, [\![e_n]\!](\rho)\big) \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!](\rho) & = & ite\big([\![e_1]\!](\rho), [\![e_2]\!](\rho), [\![e_3]\!](\rho)\big) \\
[\![\texttt{map}(f)(r)]\!](\rho) & = & \{\!\{\rho(f)(x) \mid x \in \rho(r)\}\!\} \\
[\![\texttt{filter}(b)(r)]\!](\rho) & = & \{\!\{x \mid x \in \rho(r) \wedge \rho(b)(x)\}\!\} \\
[\![\texttt{cartesian}(r_1, r_2)]\!](\rho) & = & \{\!\{(x_1, x_2) \mid x_1 \in \rho(r_1) \wedge x_2 \in \rho(r_2)\}\!\} \\
[\![\texttt{fold}(a_0, f)(r)]\!](\rho) & = & q_f\big([\![a_0]\!](\rho), \rho(r)\big), \quad where \\
 & & q_f(v_0, s) = \begin{cases} a_0 & s = \{\!\{\}\!\} \\ \rho(f)\big(x, q_f(v_0, s')\big) & s = \{\!\{x; 1\}\!\} \cup s' \end{cases} \\
[\![Let\ \ \boldsymbol{x} = \eta\ in\ E]\!](\rho) & = & [\![E]\!](\rho[\boldsymbol{x} \mapsto [\![\eta]\!](\rho)]) \\
[\![\boldsymbol{P}(\cdots)\ =\ \cdots\ E]\!](\rho_0) & = & [\![E]\!](\rho_0)
\end{array}
$$

$Prog = \boldsymbol{P}(\overline{\boldsymbol{r} : RDD_\tau}, \overline{\boldsymbol{v} : \tau}) = \overline{Fdef}\ \ \overline{PFdef}\ \ E$. `unOp` and `binOp` are taken from Figure 1: $\texttt{unOp} \in \{-, \neg, \boldsymbol{p}_i\}$, $\texttt{binOp} \in \{+, *, /, \%, =, <, \wedge, \vee, (,)\}$

Figure 4 Semantics of SparkLite

Example. Consider the program $P1$ shown in Figure 3. Suppose we were given the following input: $R_0 = \{\!\{(1;7), (2;1)\}\!\}, R_1 = \{\!\{(3;4), (5;2)\}\!\}$. Then: $\rho(A) = \{\!\{(1;7)\}\!\}$, $\rho(B) = \{\!\{(3;7)\}\!\}$, $\rho(C) = \{\!\{((3,3);28), ((3,5);14)\}\!\}$, $\rho(v) = 28 * (3+3) + 14 * (3+5)$ and the program returns $\rho(v) = 280$.

7 Term Semantics for SparkLite

In this section, we define an alternative, equivalent semantics for SparkLite where the program is interpreted as a term in APA$^+$.[7] This term is called the *program term* and denoted $\Phi(P)$ for program P, specified in Figure 5. These terms are assigned their own denotational semantics such that the meaning of $\Phi(P)$ is identical to the operational semantics of P. Special variables are used to refer to elements of input RDDs,[8] and a new language construct is added to denote the fold operation.

Representative elements of RDDs. The variables assigned by ϕ for input RDDs are called *representative elements*. In a program that receives an input RDD r, we denote the representative element of r as $\mathbf{x}_r$. The set of possible valuations to that variable is equal to the bag defined by r, and an additional 'undefined' value ($\perp$), for the empty RDD. Therefore $\mathbf{x}_r$ ranges over $\mathrm{dom}(r) \cup \{\perp\}$. By abuse of notations, the term for a non-input RDD, computed in a SparkLite program, is also called a representative element.

In Figure 5, we specify how programs written in SparkLite are translated to logical terms. Our programs are written as a series of 'let' expressions of the form $Let\ x = \eta\ in\ E$. The translation replaces all instances of x in E with the term for η, which is computed by a recursive call to ϕ_P. Input RDDs are simply translated to their representative element

[7] Reminder: APA$^+$ is equal to APA with additional *ite* and $\perp$ expressions,
[8] To avoid overhead of notations, we assume the programs do not contain self-products (for every cartesian product in the program, the sets of variables appearing in each component must be disjoint).

$$
\begin{array}{lcl}
\phi_P(\mathit{Let}\ x = \eta\ \mathit{in}\ E) & = & \phi_P(E)[\phi_P(\eta)/x] \\
\phi_P(e) & = & e \\
\phi_P(\texttt{map}(f)(r)) & = & f(\phi_P(r)) \\
\phi_P(\texttt{filter}(f)(r)) & = & \mathit{ite}(f(\phi_P(r)) = tt, \phi_P(r), \bot) \\
\phi_P(\texttt{cartesian}(r_1, r_2)) & = & (\phi_P(r_1), \phi_P(r_2)) \\
\phi_P(\texttt{fold}(f, e)(r)) & = & [\phi_P(r)]_{e,f} \\
\phi_P(r) & = & \begin{cases} \mathbf{x}_r & r \in \overline{r} \\ r & \mathit{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{c}
\mathit{Let}\ P : \boldsymbol{P}(\overline{\boldsymbol{r}}, \overline{\boldsymbol{v}}) = \overline{\boldsymbol{F}}\ \overline{\boldsymbol{f}}\ E \\
\Phi(P) = \phi_P(E)
\end{array}
$$

Figure 5 Compiling SparkLite to Logical Terms

$$
\begin{array}{lcl}
[\![\mathbf{x}_{r_i}]\!](\rho_0) & = & \rho_0(r_i) \\
[\![v]\!](\rho_0) & = & \rho_0(v) \\
[\![f]\!](\rho_0) & = & \rho_0(f) \\
[\![f(t_1, \ldots, t_n)]\!](\rho_0) & = & \{\!\{[\![f]\!](y_1, \ldots, y_n) \mid \mathit{if}\ t_i \in \mathit{RDD}\ \mathit{then}\ y_i \in [\![t_i]\!](\rho_0)\ \mathit{else}\ y_i = [\![t_i]\!](\rho_0)\}\!\} \\
[\![\mathit{ite}(f(t), t, \bot)]\!](\rho_0) & = & \{\!\{z \mid z \in [\![t]\!](\rho_0) \wedge [\![f]\!](\rho_0)(z)\}\!\} \\
[\![(t, t')]\!](\rho_0) & = & \{\!\{(z, z') \mid z \in [\![t]\!](\rho_0) \wedge z' \in [\![t]\!](\rho_0)\}\!\} \\
[\![[t]_{e,f}]\!](\rho_0) & = & q_f\big([\![e]\!](\rho), [\![t]\!](\rho_0)\big)
\end{array}
$$

For the definition of q_f, refer to the definition of `fold` in Figure 4.

Figure 6 Semantics of Terms

variable. If ϕ_P is applied on an RDD which is not an input RDD, it is not modified. By construction, it is guaranteed that this can only happen in the context of a 'let' expression, so after full evaluation of the 'let' expression, the resulting term has only variables which are representative elements of input RDDs. Non-RDD expressions are not modified by the translation. Map is translated to an application of the map UDF f on the term of the RDD on which we apply the map. Filter is translated to an *ite* expression, which returns the term of the RDD on which the `filter` operator is applied, if that term satisfies the given UDF f, and no element otherwise (represented by the value $\bot$). The cartesian product is translated to a tuple of terms of the argument RDDs. `fold` is a challenging operator, as it cannot be expressed as a first-order term. We solve this by introducing a special notation for the folded value of a term of an RDD r with a given initial value e and fold UDF f: $[\phi_P(r)]_{e,f}$.

As an example, we take the program $P1$ from Figure 3, and show how to construct $\Phi(P1)$ by applying ϕ_P. The representative element of the RDD A is a conditional term, containing elements received from valuations to R_0 (the variable $\mathbf{x}_{R_0}$) which are odd. The even valuations return $\bot$, so these elements do not belong to A. For the RDD B, the term describes that we apply the $\mathit{doubleAndAdd}(1)$ function ($\lambda x.2 * x + 1$) to the representative element of A, resulting in a composition of representative elements. The RDD C is a pair containing the representative element for B in the first tuple element and the representative element for R_1 in the second tuple element. The program term for $P1$ is the application of the special fold notation on the term for the RDD C.

$$
\begin{aligned}
\phi_{P1}(A) &= \phi_{P1}(\texttt{filter}(isOdd)(R_0)) = ite(isOdd(\phi_{P1}(R_0)), \phi_{P1}(R_0), \perp) \\
&= ite(isOdd(\mathbf{x}_{R_0}), \mathbf{x}_{R_0}, \perp) \\
\phi_{P1}(B) &= \phi_{P1}(doubleAndAdd(1)(A)) = doubleAndAdd(1)(\phi_{P1}(A)) \\
&= doubleAndAdd(1)(ite(isOdd(\mathbf{x}_{R_0}), \mathbf{x}_{R_0}, \perp)) \\
\phi_{P1}(C) &= \phi_{P1}(\texttt{cartesian}(B, R_1)) = (\phi_{P1}(B), \phi_{P1}(R_1)) \\
&= (doubleAndAdd(1)(ite(isOdd(\mathbf{x}_{R_0}), \mathbf{x}_{R_0}, \perp)), \mathbf{x}_{R_1}) \\
\Phi(P1) &= \phi_{P1}(\texttt{fold}(0, sumFlatPair)(C)) = [\phi_{P1}(C)]_{0,sumFlatPair} \\
&\quad [(\phi_{P1}(B), \phi_{P1}(R_1))]_{0,sumFlatPair} \\
&= [(doubleAndAdd(1)(ite(isOdd(\mathbf{x}_{R_0}), \mathbf{x}_{R_0}, \perp)), \mathbf{x}_{R_1})]_{0,sumFlatPair}
\end{aligned}
$$

In addition, we define in Figure 6 the semantics of those logical terms, so we could evaluate programs given in FOL form on concrete input variables and RDDs. We refer to ρ_0 from the previous section to denote the meaning of input variables and function definitions. The program term may have the form of either an input term, map term, filter term, cartesian term or fold term. Every sub-term also belongs to one of these classes, and can thus be calculated using the semantics. Map, filter and cartesian return bags, with multiplicities which are consistent with the original RDDs. For example, in map, if the map UDF f is constant, then every element in the RDD on which the map is applied donates its multiplicity to the final constant element in the resulting RDD. For example, if $R = \{\!\{(1;3),(2;2)\}\!\}$ and $f = \lambda x.0$, then the multiplicity of 0 in $[\![(\lambda x.0)(\mathbf{x}_R)]\!]$ is $3 + 2 = 5$. The semantics for filter are such that only elements that satisfy the filter UDF are left in the resulting RDD. There are no $\perp$ elements — $\perp$ is merely a syntactic denotation of a non-existent element. In cartesian, any combination of possible values of the terms yields an element in the cartesian product. Therefore the multiplicity is equal to the product of the multiplicities of each constituent in the resulting tuple. The semantics of the special fold notation are defined similarly to the semantics of fold operator in the operational semantics, using the q_f function.

We show how the term semantics are evaluated on the terms of the example program in *Figure* 3. The terms will be evaluated on the same input which we used before: $R_0 = \{\!\{(1;7),(2;1)\}\!\}, R_1 = \{\!\{(3;4),(5;2)\}\!\}$. By abuse of notations, we refer to the bags defined by the semantics of the terms by their respective variable names. For A, we take the bag that is received from substituting $\mathbf{x}_{R_0}$ with an element of R_0. The elements of R_0 are 1 with multiplicity 7, and 2 with multiplicity 1. For $\mathbf{x}_{R_0} = 2$, $isOdd(2) = ff$ so $2 \notin A$. For $\mathbf{x}_{R_0} = 1$, $isOdd(1) = tt$, so $1 \in A$ with multiplicity 7 (the operation is applied on each instance of the value, and thus returns the same multiplicity). For B, we apply the map $doubleAndAdd(1)$ on the elements of $A = \{\!\{(1;7)\}\!\}$. Therefore, $B = \{\!\{(3;7)\}\!\}$. For C, we take all possible pairings of elements with the first element coming from B, and the second element coming from R_1. Here, we will have the elements $(3,3),(3,5)$. To calculate the multiplicity, we note 3 appears in B a total of 7 times, and in R_1, 5 appears 2 times, so there are 14 combinations of the different appearances of $(3,5)$ in the cartesian product C. By the same logic, the multiplicity of $(3,3)$ is 28. Finally, the application of the fold operator on C, yielding the variable v which is returned as the program's output, is done in the same manner as in the operational semantics, by applying $q_{sumFlatPair}$. The element $(3,3)$ is aggregated by the $sumFlatPair$ function a total of 28 times, and $(3,5)$ is aggregated 14 times, so $v = 0 + 28 * (3 + 3) + 14 * (3 + 5) = 280$.

Formalization of the Term Semantics for SparkLite**.** Let P be a SparkLite program. We use the standard notation ρ_0 for the environment of input variables and function definitions.

The term $\Phi(P)$ is called the *program term of* P as before. The *Term Semantics* (TS) of a program that returns an RDD-type output is the bag that is obtained from all possible valuations to the free variables:

$$TS(P)(\rho_0) = [\![\Phi(P)]\!](\rho_0)$$

Where the meaning of $[\![\Phi(P)]\!]$ is determined according to Figure 6. Assigning a concrete valuation to the free variables of $\Phi(P)$ returns an element in the output RDD r^{out}. By taking all possible valuations to the term with elements from $\bar{r}$, we get the bag equal to r^{out}. For fold operations, a non-RDD value is returned.

▸ **Proposition 2.** *Let* $P : \boldsymbol{P}(\bar{\boldsymbol{r}}) = \bar{\boldsymbol{F}}\ \bar{\boldsymbol{f}}\ E$ *be a* SparkLite *program. Let* $[\![P]\!]$ *be the interpretation of its output according to the operational semantics, and* $TS(P)$ *according to the term semantics of* P*. Then, for any environment* ρ_0*, we have:*

$$TS(P)(\rho_0) = [\![P]\!](\rho_0)$$

8 Verifying Equivalence of SparkLite Programs

In this section we present techniques for verifying the equivalence of SparkLite programs, based on the representation of programs as logical terms, described in the last section. We begin by formally defining the program equivalence problem.

The Program Equivalence (*PE*) problem. Let P_1 and P_2 be SparkLite programs, with signature $P_i(\overline{T}, \overline{RDD_T}) : \tau$ for $i \in \{1, 2\}$. We denote by ρ_0 the environment of input variables $\overline{v}$, input RDDs $\overline{r}$ and function definitions. We use $[\![P_i]\!](\rho_0)$ to denote the result of P_i. We say that P_1 and P_2 are *equivalent*, if for all environments ρ_0, it holds that $[\![P_1]\!](\rho_0) = [\![P_2]\!](\rho_0)$.

8.1 Verifying Equivalence of SparkLite Programs without Aggregations

We are given two programs P_1, P_2 receiving as input a series of RDDs $\overline{r} = (r_1, \ldots, r_n)$. We assume without loss of generality, the programs receive only RDD arguments $\overline{r}$.[9] We let $\overline{x} = (x_1, \ldots, x_n)$ be a concrete valuation for all input RDDs representative elements: $\mathbf{x}_{r_i}$ will map to x_i. We denote the substitution of the concrete valuation in a term t over $\overline{\mathbf{x}_r}$ as: $t(\overline{x}) = t[x_1/\mathbf{x}_{r_1}, \ldots, x_n/\mathbf{x}_{r_n}]$.

We define a class of programs which do not contain any aggregate expressions in the program term.

▸ **Definition 1** (The *NoAgg* class)**.** *A program* P *satisfies* $P \in NoAgg$ *if* $\Phi(P)$ *does not contain any aggregate terms (i.e. terms of the form:* $[t]_{i,f}$*).*

Any program without *fold* operations belongs to *NoAgg*. For example, programs $P1, P2$ in the overview section are *NoAgg* programs.

In Figure 7 we present an algorithm for deciding the equivalence of *NoAgg* programs. The algorithm considers two possibilites: one, that programs return an empty RDD for all inputs. In that case, if both programs return empty RDDs, they are trivially equivalent. In the second case, we note that it is not enough to compare the evaluations of program terms, but also the multiplicities of those values in the output RDD.

[9] The extension to equivalence of terms which is based also on non-RDD inputs is immediate by quantification on the non-RDD variables in the term.

```
if 𝒩(Φ(P1) = ⊥ ∧ Φ(P2) = ⊥) = tt then
 | return equivalent
else
 | if FV(Φ(P1)) ≠ FV(Φ(P2)) then
 |  | return not equivalent
 | else
 |  | if 𝒩(∃x̄.Φ(P1)[x̄/FV(Φ(P1))] ≠ Φ(P2)[x̄/FV(Φ(P2))]) = tt then
 |  |  | return not equivalent
 |  | else
 |  |  | return equivalent
 |  | end
 | end
end
```

Figure 7 Algorithm for deciding $NoAgg$

▸ Example 1 (The importance of multiplicities for equivalence). Let $P1(R_0, R_1) = \texttt{map}(\lambda x.1)(R_0)$ and $P2(R_0, R_1) = \texttt{map}(\lambda x.1)(R_1)$. $P1$ and $P2$ have the same program term (the constant 1),[10] but the multiplicity of that element in the output bag is different and depends on the source input RDD. In $P1$, its multiplicity is the same as the size of R_0, and in $P2$ it is the same as the size of R_1. $P1$ and $P2$ are therefore not equivalent, for inputs R_0, R_1 of different sizes. Therefore, for each program term $\Phi(P)$ we consider the *set of free variables*, denoted $FV(\Phi(P))$. Each free variable has some source input RDD. In the example, $FV(\Phi(P1)) = \{\mathbf{x}_{R_0}\}$, and $FV(\Phi(P2)) = \{\mathbf{x}_{R_1}\}$.

This example provides the motivation for the definition of the algorithm to first compare the sets of free variables. After the sets of free variables are found to be equal, the algorithm calls the solver on the equivalence formula of the program terms.

The below proposition claims that terms for non-empty RDDs can be equal only if the sets of free variables of the terms are equal. Otherwise, even if the terms themselves evaluate to the same element for some valuation, the multiplicity of these elements is different in the two resulting RDDs, because the underlying variables have different multiplicites in their input RDDs. We show that in two programs without aggregations, different sets of free variables of the program terms imply the existence of input RDDs for which the two program terms evaluate to different bags, thus they are semantically inequivalent.

▸ **Proposition 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 $FV(t_1) \neq FV(t_2)$, and $t_1 \neq \bot \vee t_2 \neq \bot$. Then, $\exists \overline{r}.[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$.*

The program terms are written in APA$^+$, and may contain *ite* and $\bot$ expressions, therefore we need to encode the formulas in APA by eliminating *ite* and $\bot$. We present a translation procedure for converting APA$^+$ formulas to APA. Let φ be a formula in APA$^+$. Following the standard notation of *sub-terms*, *positions*, and *substitutions* in [11],[11] 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 the translation from APA$^+$ to APA. If

[10] To be more precise, the term is an application of the function $\lambda x.1$ on a free variable, $\mathbf{x}_{R_0}$ or $\mathbf{x}_{R_1}$. ϕ does not apply beta-reduction on the UDFs.

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

$\varphi|_p = ite(\varphi_1, \varphi_2, \varphi_3)$, 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 $\perp$, then:

$$\varphi|_p = \perp$$

as there is no meaning to evaluating functions on $\perp$ symbols, which represent non-existing RDD elements. Finally, we replace $\perp = \perp, x \neq \perp$ with tt, and $\perp \neq \perp, x < \perp, x \leq \perp, x > \perp, x \geq \perp, x = \perp$ with ff. We define a translation function $\mathcal{N}(\varphi)$, which goes over all positions in φ and performs substitutions as above. For example, $\varphi = (ite(x > 0, x, \perp) = \perp)$ is translated to: $x > 0 \implies x = \perp \wedge x \leq 0 \implies \perp = \perp$, which then is translated to: $((x > 0 \implies ff) \wedge (x \leq 0 \implies tt)) = \mathcal{N}(\varphi)$. Both $\varphi, \mathcal{N}(\varphi)$ are true only for $x \leq 0$.

▸ **Proposition 4** (Translation of APA$^+$ to APA yields equivalent formulas)**.** *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 $\perp$ arguments to $\perp$, and all equalities and inequalities containing a $\perp$ symbol with either tt or ff, is equivalent to φ:*

$$\varphi \iff \mathcal{N}(\varphi)$$

Before presenting the decidability result for $NoAgg$, we begin with an example.

▸ Remark. All examples use syntactic sugar for '*let*' expressions. For brevity, instead of applying ϕ on the underlying '*let*' expressions, we apply it line-by-line from the top-down. In addition, we assume that in programs returning an RDD-type, the RDD is named r^{out}, and the programs always end with `return` r^{out}, even if not explicitly written so in the example. Thus, $\Phi(P) = \phi_P(r^{out})$.

▸ Example 2 (Basic optimization — operator pushback). This example shows a common optimization of pushing the filter/selection operator backward, to decrease the size of the dataset.

	P1(R: $RDD_{\texttt{Int}}$)**:**	**P2**(R: $RDD_{\texttt{Int}}$)**:**
1	$R' = \texttt{map}(\lambda x.4 * x/5)(R)$	$R' = \texttt{filter}(\lambda x.x \geq 125)(R)$
2	`return` $\texttt{filter}(\lambda x.x \geq 100)(R')$	`return` $\texttt{map}(\lambda x.4 * x/5)(R')$

Both programs may return an non-empty RDD of integers, and the sets of free variables are equal: $FV(\Phi(P1)) = \{\mathbf{x}_R\} = FV(\Phi(P2))$. We analyze the representative elements:

$$\begin{aligned}
&\phi_{P1}(R') = 4 * \mathbf{x}_R/5 \\
&\phi_{P1}(r^{out}) = ite(\phi_{P1}(R') \geq 100, \phi_{P1}(R'), \perp) = ite(4 * \mathbf{x}_R/5 \geq 100, 4 * \mathbf{x}_R/5, \perp) \\
&\phi_{P2}(R') = ite(\mathbf{x}_R \geq 125, \mathbf{x}_R, \perp) \\
&\phi_{P2}(r^{out}) = (\lambda x.4 * x/5)(\phi_{P1}(R')) = (\lambda x.4 * x/5)(ite(\mathbf{x}_R \geq 125, \mathbf{x}_R, \perp))
\end{aligned}$$

We need to verify that:

$$\forall \mathbf{x}_R.ite(4 * \mathbf{x}_R/5 \geq 100, 4 * \mathbf{x}_R/5, \perp) = (\lambda x.4 * x/5)(ite(\mathbf{x}_R \geq 125, \mathbf{x}_R, \perp))$$

Using Proposition 4:

$$\begin{aligned}
&\mathcal{N}(ite(4 * \mathbf{x}_R/5 \geq 100, 4 * \mathbf{x}_R/5, \perp) = (\lambda x.4 * x/5)(ite(\mathbf{x}_R \geq 125, \mathbf{x}_R, \perp))) = \\
&(4 * \mathbf{x}_R/5 \geq 100 \implies (\mathbf{x}_R \geq 125 \implies 4 * \mathbf{x}_R/5 = (\lambda x.4 * x/5)(\mathbf{x}_R) \wedge \neg(\mathbf{x}_R \geq 125) \implies ff)) \wedge \\
&(\neg(4 * \mathbf{x}_R/5 \geq 100) \implies (\mathbf{x}_R \geq 125 \implies ff \wedge \neg(\mathbf{x}_R \geq 125) \implies tt))
\end{aligned}$$

The algorithm in Figure 7 is used to verify the equivalence of $NoAgg$ programs. First, we check if the programs return an empty RDD, in which case they are equivalent. We do so by computing the program term $\Phi(P_i)$ and solving for $\Phi(P_i) = \bot$, after converting to APA. Then, we check if the sets of free variables of the program terms are equal. If not, the algorithm returns 'not equivalent'. Otherwise, we solve the equivalence formula by attempting to find a counter-example to equivalence.

▸ **Theorem 2** (Decidability of the $NoAgg$ class). *Given two* SparkLite *programs* $P_1, P_2 \in NoAgg$, PE *is decidable.*

Proof. For non-RDD return types, the absence of aggregate operators implies we can use Proposition 1, as the returned expression is expressible in APA. For RDD return type, we use the algorithm in Figure 7, which is a decision procedure:

- If both program terms evaluate to the empty bag for any choice of input RDDs, the algorithm detects it and outputs the programs are equivalent.
- Otherwise, the algorithm checks syntactically that $FV(\Phi(P_1)) = FV(\Phi(P_2))$. If that is not the case, then by Proposition 3 we can conclude the programs are not equivalent.
- The correctness of the algorithm in the next step follows from the equivalence of the denotational TS semantics and the operational semantics defined in 6.2 (Proposition 2). If such an $\bar{x}$ is found, we take input RDDs $\bar{R}$, where $R_i = \{\!\{x_i; 1\}\!\}$, for which $[\![\Phi(P_i)]\!](\bar{R}) = \Phi(P_i)[\bar{x}/FV(\Phi(P_i))]$, thus $[\![\Phi(P_1)]\!](\bar{R}) \neq [\![\Phi(P_2)]\!](\bar{R})$. Otherwise, the formula is unsatisfiable. As $FV(\Phi(P_1)) = FV(\Phi(P_2))$ from the previous step, we know that the additional multiplicity donated by any valuation $\bar{x} \in \bar{R}$ is equal to $\Pi_i R_i(x_i)$ in both programs. We conclude that for all possible choices of input RDDs, the resulting bags have the same elements, and those elements have the same multiplicities in each bag, as required.

◂

▸ Remark. The equivalence formula generated does not contain aggregate terms, and applications of *ite* are normalized using the transformation $\mathcal{N}$, resulting in a formula definable in APA. The algorithm generates formulas in APA twice: once to verify whether the programs do not return the empty bag for all inputs, and second to test for equivalence.[12] From Proposition 1 and Proposition 4, all the formulas can be decided.

8.2 Verifying Equivalence of a Class of SparkLite Programs with Aggregation

In this section, we discuss how the existing framework can be extended to prove equivalence of SparkLite programs containing aggregate expressions. The terms for aggregate operations are given using a special operator $[t]_{i,f}$, where t is the term being folded, i is the initial value, and f is the fold function. The operator *binds* all free variables in the term t, thus the free variables of t are not contained in the free variables set of the term that includes the aggregate term.

We begin by showing there is no algorithm to decide PE for general SparkLite programs with aggregation.

[12] In the next section, program terms may contain aggregate expressions. In that case, there may be more formulas generated, and subsequently more calls to Cooper's Algorithm.

8.2.1 Proof of Undecidability of *PE*

▸ **Theorem 3.** PE *for general* SparkLite *programs is undecidable.*

Proof. We show a reduction of Hilbert's 10^{th} problem to *PE*. Hilbert's 10^{th} problem is the problem of finding a general algorithm that given a polynomial with integer coefficients, decides whether it has integer roots. We assume towards a contradiction that *PE* is decidable. Let there be a polynomial p over k variables $x_1, \ldots, x_k$, and coefficients $a_1, \ldots, a_n$. We use SparkLite operations and input RDDs R_i to represent the value of the polynomial p for some valuation of the x_i. We define a translation φ from monomials to SparkLite expressions [13]. Note, that we allow to nest RDD operations inside other RDD operations, which while not being explicitly allowed according to the SparkLite syntax, can be readily formulated properly as a series of 'let' expressions.

- $\varphi(x_i) = R_i$
- $\varphi(x_{i_1}^{n_1} \cdots x_{i_l}^{n_l}) = \texttt{cartesian}(R_{i_1}, \varphi(x_{i_1}^{n_1-1} \cdots x_{i_l}^{n_l}))$ $(n_j > 0 \text{ for } j = 1, \ldots, l)$

In addition, given a monomial m, we define $\hat{\varphi}(m) = [\varphi(m)]_{0,\lambda A,\bar{x}.A+1}$. Given a polynomial $p(a_1 \ldots, a_n; x_1, \ldots, x_k) = \sum_{l=1}^{n} a_l m_l$ where m_l are monomials over $x_1, \ldots, x_k$, we generate the following instance of the *PE* problem:

	$\mathbf{P1}(R_1, \ldots, R_k : RDD_{\texttt{Int}})$**:**	$\mathbf{P2}(R_1, \ldots, R_k : RDD_{\texttt{Int}})$**:**
1	`return` $\sum_{l=1}^{n} a_l \hat{\varphi}(m_l) \neq 0$	`return` tt

By choosing input RDDs such that the size of R_i is equal to the matching variable x_i, we can simulate any valuation to the polynomial p. If $P1$ returns true, then the valuation is not a root of the polynomial p. Thus, if it is equivalent to the 'true program' $P2$, then the polynomial p has no roots. Therefore, if the algorithm solving *PE* outputs 'equivalent' then the polynomial p has no roots, and if it outputs 'not equivalent' then the polynomial p has some root, where $x_i = |[\![R_i]\!]|$ for the R_i's which serve as the witness for nonequivalence. Thus we have a reduction of Hilbert's 10^{th} problem, proving *PE* is undecidable. ◂

8.2.2 Single Aggregate

The simplest class of programs in which an aggregation operator appears, is the class of programs whose program terms are a function of an aggregate term, that is have the form $g([t]_{i,f})$.

▸ **Definition 2** (The Agg^1 class)**.** *Let there be a program P with $\Phi(P) = g([t]_{i,f})$. $P \in Agg^1_R$ if t does not contain aggregate terms.*

The most simple case in which two Agg^1 programs are equivalent, is when the fold function f does not change the initial value *init*:

▸ **Definition 3** (Trivial fold)**.** $[\varphi]_{init,f}$ *is a* trivial fold *if:*

$$\forall \bar{v}. f(init, \varphi(\bar{v})) = init$$

[13] Note we allow self cartesian products, i.e. expressions like $\texttt{cartesian}(R, R)$. It is possible to have an equivalent reduction which is not using self cartesian products, by representing each variable x_i, whose highest power in the polynomial is p_i, using p_i RDDs. The output program will first verify that for any variable x_i, all p_i RDDs representing x_i have the same size (this can be done using the *fold* operation and comparison of the results). If not, the program returns true. Otherwise, any power of x_i up to p_i will be represented using a cartesian product of a subset of the different RDDs of x_i. The rest of the reduction's details are the same as in this reduction.

If two instances of Agg^1: $\Phi(P1) = g_1([\varphi_1]_{init_1,f_1}), \Phi(P1) = g_2([\varphi_2]_{init_2,f_2})$ have trivial folds, then equality of the initial values under g ($g_1(init_1) = g_2(init_2)$) is enoguh to show equivalence:

$$g_1([\varphi_1]_{init_1,f_1}) = g_1(init_1) \wedge g_2([\varphi_2]_{init_2,f_2}) = g_2(init_2) \wedge g_1(init_1) = g_2(init_2) \implies g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$$

When the *fold* is not trivial, we require the sets of free variables to be equal. We saw that equal sets of free variables imply equally sized RDDs. This allows us to formulate the equivalence of the *fold* operation as an inductive process: By first checking the *fold* results are equal for empty RDDs, and then are equal for every element produced by an assignment of the free variables. The sequence of assignments must be of equal size for both RDDs, as the sets of free variables are equal, so the inductive computation terminates at the same time in both *fold* operations. While *fold* operations may be equal even on RDDs of different size for non-trivial folds, we wish to avoid such peculiarities, and require equal sets of free variables in aggregate terms to be able to apply the inductive technique for equivalence verification.

▸ Remark. The transformation $\mathcal{N}$ we defined earlier for converting APA$^+$ formulas to APA formulas changes a bit when handling functions used as fold functions. $\perp$ elements indicate non-existent elements, either from an empty RDD, or from an RDD whose some of its elements were filtered out by a `filter` operation. Therefore, if f is used as a fold function, then $\forall A.f(A, \perp) = A$. The motivation is to avoid update of the aggregated value when f is applied on elements that were filtered out from the RDD previously.

We proceed with a technique for proving equivalence of Agg^1 programs based on proving inductive claims:

▸ **Lemma 4** (Sound method for verifying equivalence of Agg^1 programs)**.** *Let P_1, P_2 be Agg^1 programs, with $\Phi(P_i) = g_i([\varphi_i]_{init_i,f_i})$. The terms φ_1, φ_2 are representative elements of two RDDs R_1, R_2 of types σ_1, σ_2, respectively. Let $f_1 : \xi_1 \times \sigma_1 \to \xi_1, f_2 : \xi_2 \times \sigma_2 \to \xi_2$ be two* fold *functions, $init_1$:ξ_1, $init_2$:ξ_2 be* initial *values, and $g_1 : \xi_1 \to \xi, g_2 : \xi_2 \to \xi$ be functions. We have $g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$ if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{1}$$

$$g_1(init_1) = g_2(init_2) \tag{2}$$

$$\forall \bar{v}, A_{\varphi_1} : \xi_1, A_{\varphi_2} : \xi_2 . g_1(A_{\varphi_1}) = g_2(A_{\varphi_2}) \implies g_1(f_1(A_{\varphi_1}, \varphi_1(\bar{v}))) = g_2(f_2(A_{\varphi_2}, \varphi_2(\bar{v}))) \tag{3}$$

▸ Example 3 (Maximum and minimum). Below are two equivalent programs belonging to Agg^1. The programs compute the maximum element of a numeric RDD in two different methods: in the first program by getting the maximum directly, and in the second by getting the additive inverse of the minimum of the additive inverses of the elements.

Let: $max = \lambda M, x.ite(x > M, x, M)$
$min = \lambda M, x.ite(x < M, x, M)$

	P1($R : RDD_{\texttt{Int}}$)**:**	**P2**($R : RDD_{\texttt{Int}}$)**:**
1	`return fold`$(-\infty, max)(R)$	$R' =$ `map`$(\lambda x. -x)(R)$
2		`return` $-$ `fold`$(+\infty, min)(R')$

The equivalence formula is:

$$[\mathbf{x}_R]_{-\infty,max} = -[-\mathbf{x}_R]_{+\infty,min}$$

We apply Lemma 4: The two program apply a fold operation on a term of the same RDD R. $init_0 = -\infty, init_1 = +\infty$ and $g_1 = \lambda x.x., g_2 = \lambda x. -x$, therefore $g_1(-\infty) = g_2(+\infty)$ as required. We check the inductive claim:

$$\forall x, A, A'.A = -A' \implies max(A, \mathbf{x}_R(x)) = -min(A', -\mathbf{x}_R(x))$$

We assume $A = -A'$ and attempt to prove $max(A, x) = -min(A', -x)$, which requires verifying:

$$ite(x > A, x, A) \quad = -ite(-x < A', -x, A') = ite(-x < A', x, -A')$$

We thus need to verify the following APA formula:

$$\forall x, A, A'.A = -A' \implies ((x > A \wedge -x < A' \implies x = x) \wedge (x > A \wedge -x \geq A' \implies x = -A') \\ \wedge (x \leq A \wedge -x < A' \implies A = x) \wedge (x \leq A \wedge -x \geq A' \implies A = -A'))$$

which is true. ■

The application of Lemma 4 to the algorithm presented in Theorem 2 involves one syntactic check (Equation (1)), and two calls to Cooper's algorithm (Equations (2) and (3)). Lemma 4 shows that an inductive proof of the equality of folded values is *sound*. Therefore, given two folded expressions which are not equivalent, the lemma is guaranteed to report so.

8.2.3 A Complete Subclass

There are several cases in which one or more of the requirements of Lemma 4 are not satisfied, yet the aggregate expressions are equal. Moreover, some of these cases can be identified and subsequently have equivalence verified without applying the inductive verification technique. The first requirement, $FV(\varphi_0) = FV(\varphi_1)$, is not necessary when the fold functions applied on the terms in both programs are both trivial (Definition 3).

We proceed with an example showing a case which Lemma 4 does not cover due to Equation (3).

▸ Example 4 (Non-injective modification of folded expressions). Non-injective transformations of the aggregate expression can weaken the inductive claim, resulting in failure to prove it. As a result, Lemma 4 fails to prove the equivalence of the following two Agg_1 programs. The two programs return a boolean value, indicating if the sum of the elements in the input RDD R is divisible by 3. The difference is that $P1$ takes the sum modulo 3 of each element modulo 3, and $P2$ applies modulo 3 on the original sum of the elements.

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

To prove the equivalence, we should check by induction the equality of both boolean results:

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

Taking $g_1(x) = \lambda x.x = 0$, $g_2(x) = \lambda x.(x\%3) = 0$, the attempt to use Lemma 4 fails with a counterexample to Equation (3):

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

The counter-example is: $A = 1, A' = 2, x = 1$. The hypothesis $A = 0 \iff A'\%3 = 0$ is satisfied, but $(A + x\%3)\%3 = 2 \neq 0$, and $(A' + x)\%3 = 0$.

Despite the fact that Lemma 4 did not cover Example 4, this example belongs to a subclass of Agg^1 for which a sound and complete equivalence test method exists. This class is characterized by a verifiable semantic property of the fold functions and the initial values. This semantic property states that an application of *fold* on a sequence of two elements can be expressed as an application of *fold* on a single element. For example, if the fold function is *sum*, we say that applying *sum* on an aggregated value A and two elements x, y can be written as a sum of A and $x + y$. We name this process 'shrinking' for short. We say that two programs belong to a class called $AggPair^1_{sync}$ if for both programs, it is possible to 'shrink' a sequence of iterated applications of the fold function starting from the initial value, using the same element in both programs. In the overview, we saw that shrinking was performed for $\lambda A, x.A + x$ and $\lambda A, x.A + 3x$.

▸ **Definition 4** (The $AggPair^1_{sync}$ class). *Let there be two Agg^1 programs P_1, P_2 with equal signature, whose program terms are $g_i([\varphi_i]_{init_i,f_i})$ for $i = 1, 2$. We say that $\langle P_1, P_2\rangle \in AggPair^1_{sync}$ if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{4}$$

$$\forall \bar{v_1}, \bar{v_2}.\exists \bar{v}'.f_1(f_1(init_1, \varphi_1(\bar{v_1})), \varphi_1(\bar{v_2})) = f_1(init_1, \varphi_1(\bar{v}')) \tag{5}$$
$$\wedge\, f_2(f_2(init_2, \varphi_2(\bar{v_1})), \varphi_2(\bar{v_2})) = f_2(init_2, \varphi_2(\bar{v}'))$$

The $AggPair^1_{sync}$ class contains pairs of programs in which repeated application of Equation (5) can shrink multiple applications of the *fold* function starting from the same initial value and on the same sequence of valuations, to a single application of the *fold* function, and it can be done using the same valuation for both programs.

▸ **Theorem 5** (Equivalence in $AggPair^1_{sync}$ is *decidable*). *Let P_1, P_2 such that $\langle P_1, P_2\rangle \in AggPair^1_{sync}$, with input RDDs $\bar{r}$. Let $\Phi(P_i) = g_i([\varphi_i]_{init_i,f_i})$. Then, $\Phi(P_1) = \Phi(P_2)$ if and only if:*

$$g_1(init_1) = g_2(init_2) \tag{6}$$

$$\forall \bar{v}, \bar{y}, A_{\varphi_1}, A_{\varphi_2}.(A_{\varphi_1} = f_1(init_1, \varphi_1(\bar{y})) \wedge A_{\varphi_2} = f_2(init_2, \varphi_2(\bar{y})) \wedge g_1(A_{\varphi_1}) = g_2(A_{\varphi_2})) \tag{7}$$
$$\implies g_1(f_1(A_{\varphi_1}, \varphi_1(\bar{v}))) = g_2(f_2(A_{\varphi_2}, \varphi_2(\bar{v})))$$

▸ Example 5 (Completing Example 4). We have:

$$f_0(f_0(0, x\%3), y\%3) = (x)\%3)\%3 + (y\%3)\%3 = f_0(0, (x + y)\%3) = ((x + y)\%3)\%3$$
$$f_1(f_1(0, x), y) = x + y = f_1(0, x + y)$$

So Equation (5) is true (for arbitrary x, y, $x + y$ can reduce the two fold applications), and the programs belong to $AggPair^1_{sync}$. We are left with proving:

$$\forall x, y.((0 + y\%3)\%3 = 0 \iff (0 + y)\%3 = 0) \implies ((y + x\%3)\%3 = 0 \iff (y + x)\%3 = 0)$$

which is correct — so we were able to prove the equivalence with the stronger lemma.

Note that checking if two programs P_1, P_2 belong to $AggPair^1_{sync}$ involves a syntactic check of the free variables, and verification of an additional APA formula (Equation (5)).

8.2.4 Sound Methods for Additional Classes

A natural extension of the Agg^1 class is to programs that use the aggregated expression to perform an operation on RDDs. For example, a program that returns an RDD where all elements are strictly larger than all elements in another RDD:

$$\begin{array}{rl} & \mathrm{P1}(R_0 : RDD_{\texttt{Int}}, R_1 : RDD_{\texttt{Int}}): \\ 1 & \texttt{filter}((\lambda x.\lambda y.y > x)(\texttt{fold}(-\infty, max)(R_1)))(R_0) \end{array}$$

The program term is $ite(\mathbf{x}_{R_0} > [\mathbf{x}_{R_1}]_{-\infty,max}, \mathbf{x}_{R_0}, \perp)$.

▸ **Definition 5** (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 instance of an aggregate term in a position p: $\psi|_p = [\varphi]_{init,f}$, which is denoted γ, and in addition, φ has no aggregate terms. We write $\Phi(P) = \psi[\gamma]|_p$, where γ the value of the aggregate sub-term.*

▸ **Lemma 6** (Lifting Lemma 4 to Agg^1_R). *Let two* SparkLite *programs P_1, P_2 in Agg^1_R with terms ψ_i and aggregate expressions $\gamma_i = [\varphi_i]_{init_i,f_i}$, $i \in \{1,2\}$ in position p_i. P_1 is equivalent to P_2 if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{8}$$
$$FV(\psi_1) = FV(\psi_2) \tag{9}$$
$$\forall \bar{x}.\psi_1[init_1]|_{p_1}(\bar{x}) = \psi_2[init_2]|_{p_2}(\bar{x}) \tag{10}$$
$$\forall \bar{x}, \bar{v}, A_1, A_2.(\psi_1[A_1]|_{p_1}(\bar{x}) = \psi_2[A_2]|_{p_2}(\bar{x})) \implies (\psi_1[f_1(A_1, \varphi_1(\bar{v}))]|_{p_1}(\bar{x}) = \psi_2[f_2(A_2, \varphi_2(\bar{v}))]|_{p_2}(\bar{x})) \tag{11}$$

Lemmas 4,6 show that Agg^1, Agg^1_R have a sound equivalence verification method, and Theorem 5 shows that $AggPair^1_{sync}$ has a sound and complete equivalence verification method.[14] The sound technique can be further generalized to programs with multiple aggregate terms, where the aggregated terms are not nested in one another. Each aggregate term does not contain an aggregate term in its definition. We denote this class Agg^n.

▸ **Definition 6** (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. $P \in Agg^n$ if $t_1, \ldots, t_n$ do not contain aggregate terms.*

▸ **Lemma 7.** *Let P_1, P_2 be two programs in Agg^n, such that $\Phi(P_i) = g_i(\overline{[\varphi_i]_{init_i,f_i}})$. We have $g_1(\overline{[\varphi_1]_{init_1,f_1}}) = g_2(\overline{[\varphi_2]_{init_2,f_2}})$ if:*

$$\forall j_1, j_2.FV(\varphi_{1,j_1}) = FV(\varphi_{2,j_2}) \tag{12}$$
$$g_1(\overline{init_1}) = g_2(\overline{init_2}) \tag{13}$$
$$\forall \overline{v}, \overline{A_{\varphi_1}}, \overline{A_{\varphi_2}}.g_1(\overline{A_{\varphi_1}}) = g_2(\overline{A_{\varphi_2}}) \implies g_1(\overline{f_1(A_{\varphi_1}, \varphi_1(\overline{v}))}) = g_2(\overline{f_2(A_{\varphi_2}, \varphi_2(\overline{v}))}) \tag{14}$$

The proof of the lemma follows along the lines of the proofs of Lemmas 4,6, where the induction is on the size of the RDDs whose terms are φ_{i,j_i}.

We note that the subset of Agg^n programs that can be handled with Lemma 7 could be extended if we relaxed Equation (12). Lemma 7 requires all aggregate terms to be on RDDs of the same size, to allow equal induction lengths. We show a motivating example for relaxing Equation (12):

▸ Example 6. Let two Agg^n programs that sum the eleemnts of RDD R_0. $P2$ will also apply a trivial fold on R_1 and return the sum of the aggregations. As the fold on R_1 is trivial, it will not affect the final result.

[14] Even when the completeness criterion for $AggPair^1_{sync}$ is not met, we may be successful in proving the equivalence using Lemma 4. For example, $[((\lambda x.1)(\mathbf{x}_{r_0}), (\lambda x.1)(\mathbf{x}_{r_1}))]_{0,\lambda A,(x,y).A+x+y} = [((\lambda x.1)(\mathbf{x}_{r_1}), (\lambda x.1)(\mathbf{x}_{r_0}))]_{0,\lambda A,(x,y).A+x+y}$ can be proved by induction, but for $f = \lambda A, (x,y).A+x+y$, $f(f(A,(1,1)),(1,1)) = A+4 \neq f(A,(1,1)) = A+2$ (the choice of valuation does not change the result). Thus, it does not satisfy the completeness criterion.

	$\mathbf{P1}(R_0{:}\,RDD_{\texttt{Int}}, R_1{:}\,RDD_{\texttt{Int}})$:	$\mathbf{P2}(R_0{:}\,RDD_{\texttt{Int}}, R_1{:}\,RDD_{\texttt{Int}})$:
1	$v = \texttt{fold}(0, \lambda A, x.A + x)(R_0)$	$v' = \texttt{fold}(0, \lambda A, x.A + x)(R_0)$
2	`return` v	$u = \texttt{fold}(0, \lambda A, x.0)(R_1)$
3		`return` $v' + u$

We see that as $P2$ has an aggregate term with a set of free variables equal to $\{\mathbf{x}_{R_1}\}$ and the other aggregate terms have $\{\mathbf{x}_{R_0}\}$, Lemma Theorem 7 returns 'not equivalent', while $P1$ and $P2$ are actually equivalent.

We note that in order to analyze such programs, we need to verify equivalence in the case some of the participating RDDs are empty, while others are not. However, Agg^n still contains non-trivial programs, as the below example shows:

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

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

	$\mathbf{P1}(R{:}\,RDD_{\texttt{Int}})$:	$\mathbf{P2}(R{:}\,RDD_{\texttt{Int}})$:
1	`return` $\texttt{fold}((0,0), h)(R)$	$R_P = \texttt{filter}(\lambda x.x \geq 0)(R)$
2		$R_N = \texttt{map}(\lambda x. -x)(\texttt{filter}(\lambda x.x < 0)(R))$
3		$p = \texttt{fold}(0, \lambda A, x.A + x)(R_P)$
4		$n = -\texttt{fold}(0, \lambda A, x.A + x)(R_N)$
5		`return` (p, n)

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

We let $g_1 = g_2 = \lambda(x, y).(x, y)$. We apply Lemma 7 to prove:

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

We note that Equation (12) is satisfied: $FV(R) = FV(R_P) = FV(R_N) = \{x_R\}$. Induction base case (Equation (13)) is trivial. Induction step for proving Equation (14):

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

Substituting for h, we get a formula in APA$^+$ and proceed as usual. ■

9 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 logical formulas. Specifically, we looked at formulas over a decidable extension of the Presburger arithmetic, and added a special operation for representing fold operations. We also showed that in the presentation of a query equivalence instance as a logical formula, the solver for the formula is capable of proving equivalence of conjunctive queries (Theorem 2). Furthermore, we provided a classification of programs with the fold operation (aggregations), and presented a sound method for equivalence testing for each presented class (Lemmas 4, 6, 7), and a sound and complete method for one particular non-trivial class of programs (Theorem 5).

We hope the foundations laid in this paper will open numerous new possibilities: First, it allows writing tools that handle formal verification and optimization of clients written

in Spark and similar frameworks, by building upon the concepts presented here to more elaborate structures involving queries with nested aggregation, unions, and multiple step-inductions for self joins. In addition, other decidable theories are applicable to programs used in practice, replacing Presburger arithmetics.

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 E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

4 David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.

5 Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of presburger arithmetic. Technical report, Massachusetts Institue of Technology, Cambridge, MA, USA, 1974.

6 Masahito Hasegawa. *Decomposing typed lambda calculus into a couple of categorical programming languages*, pages 200–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

7 Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. Deciding boolean algebra with presburger arithmetic. *J. Autom. Reasoning*, 36(3):213–239, 2006.

8 Aless Lasaruk and Thomas Sturm. *Effective Quantifier Elimination for Presburger Arithmetic with Infinity*, pages 195–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

9 Derek C. Oppen. A 222pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323 – 332, 1978.

10 M. 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.

11 Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

12 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.

A Extending Cooper's Algorithm to the Augmented Presburger Arithmetic

Proof. Let φ be a quantified formula over $\bigcup_n \mathbb{Z}^n$ with terms in the Augmented Presburger Arithmetic. We shall translate φ to a formula in the Presburger Arithmetic. For any atom $A := a = b$, 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 quantificaion extends naturally, i.e. $\forall a$ becomes $\forall x_{a,1}, \ldots, x_{a,k}$, and similarly for $\exists$. ◂

B Typing rules for SparkLite

Booleans

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

Integers

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

Integer ops

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

Boolean ops

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

Tuples

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

UDFs

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

RDD

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

$$\frac{\rho \vdash r:RDD_\tau, r':RDD_{\tau'}}{\rho \vdash \texttt{cartesian}(r,r'):RDD_{\tau \times \tau'}} \qquad \frac{\rho \vdash r:RDD_\tau, f:\tau' \times \tau \to \tau, init:\tau'}{\rho \vdash \texttt{fold}(init,f)(r):\tau'}$$

Figure 8 Typing Rules for SparkLite

C Proof of Proposition 3

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 $\Pi_{\mathbf{x}_{r_i} \in FV(t_1)} n_i$, and in $[\![t_2]\!]$ it is $\Pi_{\mathbf{x}_{r_i} \in FV(t_2)} n_i$. As $FV(t_1) \neq FV(t_2)$, there are $n_i > 1$ such that $\Pi_{\mathbf{x}_{r_i} \in FV(t_1)} n_i \neq \Pi_{\mathbf{x}_{r_i} \in FV(t_2)} n_i$, thus $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required. ◂

D Proof of Lemma 4

Proof. (Lemma 4) 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 aggregating operations' UDFs is that they are *commutative*, so the order of elements chosen does not change the final result. We also recall we extended f_i to $\xi_i \times (\sigma_i \cup \{\bot\})$ by setting $f_i(A, \bot) = A$ ($\bot$ is defined to behave as the neutral element for f_i). We denote $[\![\varphi_1]\!] = R_1, [\![\varphi_2]\!] = R_2$. To prove $g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$, it is necessary to prove that

$$g_1([\![\texttt{fold}]\!](f_1, init_1)(R_1)) = g_2([\![\texttt{fold}]\!](f_2, init_2)(R_2))$$

We set $A_{\varphi_j,0} = init_j$ for $j = 1, 2$. Each element of R_1 and R_2 is expressible by providing a concrete valuation to the free variables of φ_1, φ_2, namely the vector $\bar{v}$. We prove the equality by induction on the *size* of the RDDs R_1, R_2, denoted n.[15] 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* $\langle A_{\varphi_1,1}, \ldots, A_{\varphi_1,n} \rangle$ and $\langle A_{\varphi_2,1}, \ldots, A_{\varphi_2,n} \rangle$. From the semantics of `fold`, we have that $A_{\varphi_j,i} = f_j(A_{\varphi_j,i-1}, \varphi_j(\bar{a}_i))$ for $j = 1, 2$. Our goal is to show $g(A_{\varphi_1,n}) = g'(A_{\varphi_2,n})$ for all n.

Case $n = 0$: $R_1 = R_2 = \{\!\{\}\!\}$, so $[\![\texttt{fold}]\!](f_1, init_1)(R_1) = init_1$ and $[\![\texttt{fold}]\!](f_2, init_2)(R_2) = init_2$. From Equation (2), $g_1(init_1) = g_2(init_2)$, as required.

Case $n = i$, assuming correct for $n \le i-1$: By assumption, we know that the sequence of intermediate values up to $i-1$ satisfies: $g_1(A_{\varphi_1,i-1}) = g_2(A_{\varphi_2,i-1})$. We are given the i'th valuation, denoted $\bar{a}_i$. We need to show $A_{\varphi_1,i} = A_{\varphi_2,i}$, so we use the formula for calculating the next intermediate value:

$$\begin{array}{rcl} A_{\varphi_1,i} & = & f_1(A_{\varphi_1,i-1}, \varphi_1(\bar{a}_i)) \\ A_{\varphi_2,i} & = & f_2(A_{\varphi_2,i-1}, \varphi_2(\bar{a}_i)) \end{array}$$

We use Equation (3), plugging in $\bar{v} = \bar{a}_i$, $A_{\varphi_1} = A_{\varphi_1,i-1}$, and $A_{\varphi_2} = A_{\varphi_2,i-1}$. By the induction assumption, $g_1(A_{\varphi_1,i-1}) = g_2(A_{\varphi_2,i-1})$, therefore $g_1(A_{\varphi_1}) = g_2(A_{\varphi_2})$, so Equation (3) yields $g_1(f_1(A_{\varphi_1}, \varphi_1(\bar{a}_i))) = g_2(f_2(A_{\varphi_2}, \varphi_2(\bar{a}_i)))$. By substituting back A_{φ_j} and the formula for the next intermediate value, we get: $g_1(A_{\varphi_1,i}) = g_2(A_{\varphi_2,i})$ as required. ◂

E Proof Theorem 5

Proof. Sound (if): We prove the equality $g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$ by induction on the size of the RDDs $[\![\varphi_1]\!], [\![\varphi_2]\!]$, denoted n.[16] For $n = 0$, $[\![\varphi_1]\!](\bar{r}) = [\![\varphi_2]\!](\bar{r}) = \{\!\{\}\!\}$, thus $[\varphi_i]_{init_i,f_i} = init_i\,(i = 1, 2)$, and the equality follows from Equation (6). Assuming for n and proving for $n+1$: We let a sequence of intermediate values $A_{\varphi_i,k}$, $(i = 1, 2; k = 1, \ldots, n+1)$, for which we know in particular that $g_1(A_{\varphi_1,n}) = g_2(A_{\varphi_2,n})$, and we need to prove $g_1(A_{\varphi_1,n+1}) = g_2(A_{\varphi_2,n+1})$. We denote $A_{\varphi_i,0} = init_i$, and then we have $A_{\varphi_i,k} = f_i(A_{\varphi_i,k-1}, \varphi_i(\bar{a_k}))\,(k = 1, \ldots, n+1)$ for some $\bar{a_k}$. According to Equation (5), $A_{\varphi_i,2} = f_i(A_{\varphi_i,1}, \varphi_i(\bar{a_2})) = f_i(f_i(init_i, \varphi_i(\bar{a_1})), \varphi_i(\bar{a_2}))$ yields $\exists \bar{a_2}'. \bigwedge_{i=1,2} A_{\varphi_i,2} = f_i(init_i, \varphi_i(\bar{a_2}'))$. We can thus use Equation (5) to prove by induction that $\exists \bar{a_k}'. \bigwedge_{i=1,2} A_{\varphi_i,k} = f_i(init_i, \varphi_i(\bar{a_k}'))$, and in particular $\exists \bar{a_n}'. \bigwedge_{i=1,2} A_{\varphi_i,n} = f_i(init_i, \varphi_i(\bar{a_n}'))$. By applying Equation (7) for $\bar{v} =$

[15] It is important to note that not every n can be a legal size of the RDDs. For example, if $R_1 =$ `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.

[16] The comment in footnote 15 regarding the validity of the soundness argument, even if $[\![\varphi_i]\!]$ can not have size n, is still valid here.

$a_{n+1}, \bar{y} = \bar{a_n}'$, we get:

$$\begin{array}{lll} g_1(f_1(f_1(init_1, \varphi_1(\bar{y})), \varphi_1(\bar{v}))) & = g_2(f_2(f_2(init_2, \varphi_2(\bar{y})), \varphi_2(\bar{v}))) & \implies \\ g_1(f_1(f_1(init_1, \varphi_1(\bar{a_n}')), \varphi_1(a_{n+1}))) & = g_2(f_2(f_2(init_2, \varphi_2(\bar{a_n}')), \varphi_2(a_{n+1}))) & \implies \\ g_1(f_1(A_{\varphi_1,n}, \varphi_1(a_{n+1}))) & = g_2(f_2(A_{\varphi_2,n}, \varphi_2(a_{n+1}))) & \implies \\ g_1(A_{\varphi_1,n+1}) & = g_2(A_{\varphi_2,n+1}) & \end{array}$$

as required.

Complete (only if): Assume towards a contradiction that either Equation (6) or Equation (7) are false. If the requirement of Equation (6) is not satisfied, yet the aggregates are equivalent, i.e.

$$g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2}) \wedge g_1(init_1) \neq g_2(init_2)$$

then we can get a contradiction by choosing all input RDDs to be empty. Thus, for $R = \{\!\{\}\!\}$, $[\![[\varphi_1]_{init_1,f_1}]\!](R) = init_1 \wedge [\![[\varphi_2]_{init_2,f_2}]\!](R) = init_2 \implies g_1(init_1) = g_2(init_2)$, which is a contradiction. The conclusion is that Equation (6) is a necessary condition for equivalence. Therefore, we assume just Equation (7) is false. Let there be counter-examples $\bar{v}, \bar{y}$ to Equation (7), [17] and let:

$$F_i = f_i(f_i(init_i, \varphi_i(\bar{y})), \varphi_i(\bar{v}))$$

Then $g_1(F_1) \neq g_2(F_2)$. By Equation (5) we can write F_i as: $F_i = f_i(init_i, \varphi_i(\bar{w}))$ for some $\bar{w}$. We take an RDD $R = \{\!\{\bar{w}; 1\}\!\}$. Then $[\![\varphi_j]\!](R) = \{\!\{\varphi_j(\bar{w}); 1\}\!\}$, for which: $[\![[\varphi_j]_{init_j,f_j}]\!](R) = F_i$. By the assumption, $[\![g_1([\varphi_1]_{init_1,f_1})]\!](R) = [\![g_2([\varphi_2]_{init_2,f_2})]\!](R)$, but then $g_1(F_1) = g_2(F_2)$. Contradiction. ◂

F Proof of Lemma 6

Proof. The proof follows along the lines of the proof of Lemma 4. We need to prove $\Phi(P_1) = \Phi(P_2)$, or $\forall \bar{x}. \psi_1[\gamma_1]|_{p_1}(\bar{x}) = \psi_2[\gamma_2]|_{p_2}(\bar{x})$, where $\gamma_i = [\varphi_i]_{init_i,f_i}$ and $\bar{x}$ is a vector of valuations to $FV(\psi_1), FV(\psi_2)$ which are equal sets (Equation (9)). We shall prove it by induction on the size of the RDDs R_1, R_2, generating the underlying terms of γ_1, γ_2.

For size 0, we have $\gamma_i = init_i$, and from Equation (10) we have $\Phi(P_1) = \Phi(P_2)$ as required.

Assuming for size n and proving for $n+1$: The RDDs R_1, R_2 are now generated using $a_1, \ldots, a_{n+1}$, with intermediate values $A_{\varphi_i,1}, \ldots, A_{\varphi_i,n+1}$ for $i = 1, 2$. By assumption, $\forall x. \psi_1[A_{\varphi_1,n}]|_{p_1} = \psi_2[A_{\varphi_2,n}]|_{p_2}$, and we need to prove $\forall \bar{x}. \psi_1[A_{\varphi_1,n+1}]|_{p_1}(\bar{x}) = \psi_2[A_{\varphi_2,n+1}]|_{p_2}(\bar{x})$. In addition, $A_{\varphi_i,n+1} = f_i(A_{\varphi_i,n}, a_{n+1})$ for $i = 1, 2$. We let some $\bar{x}$ and we need to prove for it $\psi_1[A_{\varphi_1,n+1}]|_{p_1}(\bar{x}) = \psi_2[A_{\varphi_2,n+1}]|_{p_2}(\bar{x})$. We apply Equation (11) with $\bar{x}$ as $\bar{x}$, $\bar{v} = a_{n+1}$, and $A_{\varphi_1,n}, A_{\varphi_2,n}$ as A_1, A_2, concluding that: $\psi_1[f_1(A_{\varphi_1,n}, a_{n+1})]|_{p_1}(\bar{x}) = \psi_2[f_2(A_{\varphi_2,n}, a_{n+1})]|_{p_2}(\bar{x})$. Replacing for $A_{\varphi_i,n+1}$, we get what had to be proven. ◂

[17] Note that the A_{φ_i} are determined immediately by choosing $\bar{y}$: $A_{\varphi_i} = f_i(init_i, \varphi_i(\bar{y}))$.