

# Verifying Equivalence of Spark Programs

Shelly Grossman<sup>1</sup>, Sara Cohen<sup>2</sup>, Shachar Itzhaky<sup>3</sup>, Noam Rinetzky<sup>1</sup>, and Mooly Sagiv<sup>1</sup>

- 1 School of Computer Science, Tel Aviv University, Tel Aviv, Israel  
`{shellygr,maon,msagiv}@tau.ac.il`
- 2 School of Engineering and Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel  
`sara@cs.huji.ac.il`
- 3 Computer Science, Massachusetts Institute of Technology, USA  
`shachari@mit.edu`

---

## Abstract

In this paper, we present a novel approach for verifying 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 database systems, but do not enjoy a similar support of optimization tools used by traditional databases. 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* (UDFs).

We model Spark as a programming language which imitates Relational Algebra queries in the bag semantics and allows for user defined functions expressible in Presburger Arithmetics. We present a sound verification technique for verifying equivalence of Spark programs which is complete for programs without aggregate operations as well as for programs with aggregate operations which fall under a criterion we provide.

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

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

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

## 1 Preliminaries

In this section, we define a simple extension of Presburger arithmetic  $\mathbb{N}$ , which is the first-order theory of the natural numbers with addition, to tuples, and state its decidability.

**Notations.** We denote the set of natural numbers (including zero), positive numbers, and integers by  $\mathbb{N}$ ,  $\mathbb{N}^+$ , and  $\mathbb{Z}$ , respectively. We denote the *size* (number of elements) of a set  $X$  by  $|X|$ . We write  $ite(p, e, e')$  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  $X$ .

First order logic formulae

Explain extension with tuples

State decidability w/o tuples ; explain why decidable with.



© Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv;  
licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Arithmetic Exp.**      $ae ::= c \mid ae + ae \mid -ae \mid c * ae \mid ae / c \mid ae \% c$   
**Boolean Exp.**         $be ::= \text{true} \mid \text{false} \mid e = e \mid ae < ae \mid \neg be \mid be \wedge be \mid be \vee be$   
**General Basic Exp.**  $e ::= ae \mid be \mid v \mid (e, e) \mid p_i(e) \mid \text{ite}(be, e, e)$

■ **Figure 1** Terms

update the below text

The following proposition follows directly from the decidability of Presburger arithmetic [1].

► **Proposition 1.** *Given two SparkLite programs  $P$  and  $Q$  which use only integer or boolean basic types and no RDDs, PE is decidable.*

**Proof.** *sketch.* Let the signatures of  $P$  and  $Q$  be  $P(\bar{T}):\tau$ ,  $Q(\bar{T}):\tau$ . Expression and functions in SparkLite belong to an extension of the Presburger arithmetic to integer numbers, which is decidable [1]. If the return types  $\tau$  are tuples, then on per-element basis, equivalency is decidable: For each element of the returned tuple we get an equation of functions applied to the variables in the programs, definable in the Presburger arithmetic, so the problem of program equivalence reduces to solving the Presburger formula. It is decidable by using a decision procedure such as *Cooper's algorithm* [2].<sup>1</sup> ◀

## 2 The SparkLite language

In this section, we define the syntax of SparkLite, a simple imperative programming language which allows to use Spark's *resilient distributed datasets* (RDDs) [5].

### 2.1 Data Model

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

**RDDs.** In addition, SparkLite allows the user to define RDDs. RDDs are bags of *records*, all of the same type. Hence,  $RDD_\tau$  denotes bags containing records of type  $\tau$ .

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

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

<sup>1</sup> A remark on complexity: Cooper's algorithm has an upper bound of  $2^{2^{2^p n}}$  for some  $p > 0$  and where  $n$  is the number of symbols in the formula [4]. In practice, our experiments show that Cooper's algorithm on non-trivial formulas returns almost instantly, even on commodity hardware.

**Interpretation operator** We use  $\llbracket \cdot \rrbracket$  (semantic brackets) to denote the mathematical interpretation of an expression. We shall see in the next subsections that it may be a function of an *environment* when the expressions contain variables. The exact meaning of environments and semantic interpretation of syntactic strings under environments is fully explained in Section 2.4.

## 2.2 Functional Model

**Operations** *RDDs* are analogous to database tables and as such the methods to query the *RDDs* are inspired by both *Relational Algebra* (*RA*) [1] and Spark. *RA* has 5 basic operators, which are *Select*, *Project*, *Cartesian Product*, *Union* and *Subtract*. This paper focuses on the first three operators. The *Select* operator is analogous to *filter* in SparkLite, and *Project* is analogous to *map*. The expressive power of SparkLite’s *map* and *filter* is greater than their analogous *RA* operations thanks to the UDFs (see next), which allow *extended projection* as well as greater flexibility in executing complex operations on elements of different types.

**UDFs.** A special property of Spark is in allowing some of its standard operations to be *higher order functions* - to receive a function and to apply it on an *RDD* in a method defined by the operation. For example, we can *fold* an *RDD* containing integer numbers by providing a sum function of two integers to the *fold* transform. Such a function is called a “*User-Defined Function*”, or simply *UDF*, for short. The *signature* of a UDF contains information on the return type and the arguments types, and when applied in the context of an *RDD* operation, the signature should match both the *RDD* type and the operation on it (see typing rules in appendix ??). The syntax of the *body* of a UDF is the same as that of first-order simply typed lambda expressions [2]. For example:  $sumMod10: \text{Int} \times \text{Int} \rightarrow \text{Int}$  is the signature of the following function:  $sumMod10 = \lambda x, y. x \% 10 + y \% 10$ . The types are omitted from the body of the function for brevity, but when the types are not clear from the context, we may also write it as:  $sumMod10 = \lambda x: \text{Int}, y: \text{Int}. x \% 10 + y \% 10: \text{Int}$ . Note that  $sumMod10$  has two arguments, but we could also write it as a function with single argument having the following signature:  $sumPairMod10: (\text{Int} \times \text{Int}) \rightarrow \text{Int}$ , with body:  $sumPairMod10 = \lambda(x, y). x \% 10 + y \% 10$ .

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

## 2.3 Syntax

The syntax of SparkLite language is defined in Figure 3.

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

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

	$P1(R_0: RDD_{\text{Int}}, R_1: RDD_{\text{Int}}):$
1	$A = \text{filter}(isOdd)(R_0)$
2	$B = \text{map}(doubleAndAdd(1))(A)$
3	$C = \text{cartesian}(B, R_1)$
4	$v = \text{fold}(0, sumFlatPair)(C)$
5	<b>return</b> $v$

■ **Figure 2** Example SparkLite program

**Program structure** The header of a program contains function definitions. Loops are not allowed in the body of a program. Variable declarations are in *SSA (Static Single Assignment)* form [3]. Variables are immutable by this construction. Programs have no side effects, do not change the inputs, and always return a value. The *program signature* will consist of its name, its input types and return type:  $P(\overline{\mathcal{T}}_i, \overline{RDD}_i): \mathcal{T}_o$

**Example program** Consider the example SparkLite program in Figure 2.

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 ( $P = P1$ ) is announced with a list of input RDDs ( $R_0, R_1$ ) (*Prog* rule). Instead of writing *Let  $l_1$  in Let  $l_2$  in ...*, we use syntactic sugar, where each line of code contains a single ‘*Let*’ definition, and the target expression (which may be a ‘*Let*’ expression itself) follows. Then, 3 variables of RDD type are defined ( $A, B, C$ ) and one integer variable ( $v$ ). We can see in the definition of  $A$  an application of the *filter* operation, accepting the RDD  $R_0$  and the function *isOdd*. For  $B$ ’s definition we apply the *map* operation with a parametric function *doubleAndAdd* with the parameter 1, or simply the function  $\lambda x. 2 * x + 1$ .  $C$  is the cartesian product of  $B$  and input RDD  $R_1$ . We apply an aggregation using *fold* on the RDD  $C$ , with an initial value 0 and the function *sumFlatPair*, which ‘flattens’ elements of tuples in  $C$  by taking their sum, and summing all those vectorial sums to a single value stored in the variable  $v$ . As we omit ‘*Let*’ expressions from the example, we use the **return** keyword to return a value. The returned value is the integer variable  $v$ . The program’s signature is  $P1(RDD_{\text{Int}}, RDD_{\text{Int}}): \text{Int}$ .

## 2.4 Semantics

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

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

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

**Data flow** The *environment function*  $\rho \in \mathcal{E}$  denotes the environment of the program. The initial environment function  $\rho_0$  maps all input variables and function definitions: For  $x \in \overline{\mathcal{T}} \cup \overline{\text{Vars}} \cup \overline{Fdef} \cup \overline{PFdef}$ ,  $\rho_0(x) = \llbracket x \rrbracket$ , where  $\llbracket \cdot \rrbracket$  is used to express the interpretation of the

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

■ **Figure 3** Syntax for SparkLite

input in the semantic domain. We denote  $\llbracket v \rrbracket(\rho) = \rho(v)$  as the *interpreted value* of the variable  $v$  of type  $\mathcal{D}$ . The semantics of expressions are straight-forward and we provide the semantics with the current environment  $\rho$  (using  $\llbracket \cdot \rrbracket(\cdot)$ ), see Figure 4 for details.

In Figure 4 we specify the behavior of  $\llbracket \cdot \rrbracket(\cdot)$  on the body of the program.

**Function and UDF semantics** For UDFs, which are based on a restricted fragment of the *simply typed lambda calculus* [], we assume the syntax and semantics are the same as in the  $\lambda$ -calculus. Note however that the syntax does not allow passing higher order functions as UDFs, and forces any higher order function to be reduced to a first-order function beforehand. In addition, all parameters passed to UDF which are based on higher-order functions are read-only. The semantics of function definitions do not depend on the environment, and appear in Figure 4.

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

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

- In *map*, if several elements  $y$  map to  $x$  by  $f$ , then the multiplicity of  $x$  is the sum of multiplicities of all  $y$  elements. In other words, if an element  $x$  appears  $n$  times, we apply the map UDF on it  $n$  times.
- *fold is well defined*: It should be noted that the Spark specification requires UDFs passed to aggregate operations to be *commutative* and *associative* for the value to be uniquely defined. In SparkLite, only commutativity is necessary. By the assumed commutativity of  $f$ , the order in which elements from the bag are chosen in the *fold* operation does not affect the result.
- *foldByKey* returns an RDD where each *key*  $k$  (defined to be the first element of the RDD type tuple) appears once, and the value is replaced with the fold result of all values  $v_i$

$\llbracket c \rrbracket(\rho)$	$= c$
$\llbracket v \rrbracket(\rho)$	$= \rho(v)$
$\llbracket \text{uOp } e \rrbracket(\rho)$	$= \text{uOp } \llbracket e \rrbracket(\rho)$
$\llbracket e \text{ binOp } e \rrbracket(\rho)$	$= \llbracket e \rrbracket(\rho) \text{ binOp } \llbracket e \rrbracket(\rho)$
$\llbracket \text{if } e \text{ then } e \text{ else } e \rrbracket(\rho)$	$= \text{ite}(\llbracket e \rrbracket(\rho), \llbracket e \rrbracket(\rho), \llbracket e \rrbracket(\rho))$
$\llbracket \text{map}(f)(r) \rrbracket(\rho)$	$= \{\{\rho(f)(x) \mid x \in \rho(r)\}\}$
$\llbracket \text{filter}(b)(r) \rrbracket(\rho)$	$= \sigma_{\{x \mid b(x)\}}(r)$
$\llbracket \text{cartesian}(r_1, r_2) \rrbracket(\rho)$	$= \{(x_1, x_2) \mid r_1(x_1) \cdot r_2(x_2) \mid x_1 \in r_1 \wedge x_2 \in r_2\}$
$\llbracket \text{fold}(a_0, f)(r) \rrbracket(\rho)$	$= \begin{cases} f(\llbracket \text{fold} \rrbracket(a_0, f)(r'), a) & r = r' \cup \{a; 1\}, \text{ where } a \in r \\ v & r = \perp \end{cases}$
$\llbracket \text{foldByKey}(a_0, f)(r) \rrbracket(\rho)$	$= \{(k, v); 1 \mid (k, \_) \in r \wedge v = \llbracket \text{fold} \rrbracket(a_0, f)(\pi_2(r \upharpoonright_{(k, \_)}))\}$
$\llbracket \text{Let } x = \eta \text{ in } E \rrbracket(\rho)$	$= \llbracket E \rrbracket(\rho[x \mapsto \llbracket \eta \rrbracket(\rho)])$
$\llbracket \text{Prog} \rrbracket(\rho_0)$	$= \llbracket E \rrbracket(\rho_0)$

■ **Figure 4** Semantics of SparkLite.  $\text{Prog} = P(\bar{r} : \overline{RDD\tau}, \bar{v} : \bar{\tau}) = \overline{Fdef} \ \overline{PFdef} \ E$ .  $\text{uOp}$ ,  $\text{binOp}$ , and  $\text{terOp}$  are taken from Figure 3:  $\text{uOp} \in \{-, \neg, \pi_i\}$ ,  $\text{binOp} \in \{+, *, /, \%, =, <, \wedge, \vee, (, )\}$

<b>Let expr.:</b>	$\phi_P(\text{Let } x = \eta \text{ in } E, k)$	$= (t_{E'}[t_\eta/x], m), \text{ where } (t_{E'}, n) = \phi_P(E', k), (t_\eta, m) = \phi_P(\eta, n)$
<b>Basic exps.:</b>	$\phi_P(e, k)$	$= (e, k)$
<b>Map:</b>	$\phi_P(\text{map}(f)(r), k)$	$= (f(t), m), \text{ where } (t, m) = \phi_P(r, k)$
<b>Filter:</b>	$\phi_P(\text{filter}(f)(r), k)$	$= \text{ite}(f(t) = tt, (t, m), \perp), \text{ where } (t, m) = \phi_P(r, k)$
<b>Cartesian:</b>	$\phi_P(\text{cartesian}(r_1, r_2), k)$	$= ((t_{r_1}, t_{r_2}), m), \text{ where } (t_{r_1}, n) = \phi_P(r_1, k), (t_{r_2}, m) = \phi_P(r_2, n)$
<b>Input RDDs:</b>	$\phi_P(r, k)$	$= \begin{cases} (\mathbf{x}_r^{(k)}, k+1) & r \in \bar{r} \\ (r, k) & \text{otherwise} \end{cases}$

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

■ **Figure 5** Compiling SparkLite to logical terms ( $\phi$ ).

such that that were mapped to the same key in the original RDD:  $(k, v_i) \in r$ . Thus, we have a fold operator acting on the restriction of  $r$  to some key  $k$ , projected to the values.

**Example** For the example program Figure 2, 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  $\rho(v) = 280$ .

### 3 Term Semantics for SparkLite

In this section, we present an alternative, equivalent semantics for SparkLite where the program is interpreted as a term in first order logic over the theory of integer arithmetic. This term is called the *program term* and denoted  $\Phi(P)$  for program  $P$ , and its exact specification appears in Figure 5. The variables of the term are taken from the input RDDs. We take for example the program in Figure 2. The  $\Phi$  function is defined using the function  $\phi$  whose purpose is to maintain consistent variable renaming.  $\phi$  is applied recursively on the expression returned by the program. We simplify by running  $\phi_{P_1}$  on each line of the

program, top-down:

$$\begin{aligned}
\phi_P(A, 0) &= \phi_P(\text{filter}(\text{isOdd})(R_0), 0) = (\text{ite}(\text{isOdd}(t), t, \perp), n \text{ where } (t, n) = \phi_P(R_0, 0)) \\
&=_{\phi_P(R_0, 0) = \mathbf{x}_{R_0}^{(0)}, 1} \text{ite}(\text{isOdd}(\mathbf{x}_{R_0}^{(0)}), \mathbf{x}_{R_0}^{(0)}, \perp), 1 \\
\phi_P(B, 1) &= \phi_P(\text{doubleAndAdd}(1)(A), 1) = \text{doubleAndAdd}(1)(A), 1 \\
&= \text{doubleAndAdd}(1)(\text{ite}(\text{isOdd}(\mathbf{x}_{R_0}^{(0)}), \mathbf{x}_{R_0}^{(0)}, \perp)), 1 \\
\phi_P(C, 1) &= \phi_P(\text{cartesian}(B, R_1), 1) =_{\phi_P(B, 1) = B, 1} (B, p_1(\phi_P(R_1, 1)), p_2(\phi_P(R_1, 1))) \\
&=_{\phi_P(R_1, 1) = \mathbf{x}_{R_1}^{(1)}, 2} (B, \mathbf{x}_{R_1}^{(1)}), 2 \\
\phi_P(v, 2) &= \phi_P(\text{fold}(0, \text{sumFlatPair})(C), 2) =_{\phi_P(C, 2) = C, 2} [C]_{0, \text{sumFlatPair}}, 2 \\
\Phi(P) &= p_1(\phi_P(v, 2)) = [C]_{0, \text{sumFlatPair}} = [(B, \mathbf{x}_{R_1}^{(1)})]_{0, \text{sumFlatPair}} \\
&= [(\text{doubleAndAdd}(1)(\text{ite}(\text{isOdd}(\mathbf{x}_{R_0}^{(0)}), \mathbf{x}_{R_0}^{(0)}, \perp)), \mathbf{x}_{R_1}^{(1)})]_{0, \text{sumFlatPair}}
\end{aligned}$$

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

**Matching representative elements** For two program terms to be comparable, they must depend on the same input RDDs. For example:

	$P1(R_0: \text{RDD}_{\text{Int}}, R_1: \text{RDD}_{\text{Int}}):$	$P2(R_0: \text{RDD}_{\text{Int}}, R_1: \text{RDD}_{\text{Int}}):$
1	$\text{return map}(\lambda x.1)(R_0)$	$\text{return map}(\lambda x.1)(R_1)$

We see that  $P1$  and  $P2$  have the same program term (1), but the multiplicity of that element in the output bag is different and depends on the source input RDD. In  $P1$ , its multiplicity is the same as the size of  $R_0$ , and in  $P2$  it’s the same as the size of  $R_1$ .  $P1$  and  $P2$  are therefore not equivalent, because we can provide inputs  $R_0, R_1$  of different sizes.

For each program term  $\Phi(P)$  we therefore consider  $FV(\Phi(P))$ , the *set of free variables*. Each free variable has some source input RDD, and an input RDD may have more than one free variable representing it in the program term. In the example, the set of free variables of  $P1$  is  $FV(\Phi(P1)) = \{\mathbf{x}_{R_0}\}$ , and of  $P2$  it is  $FV(\Phi(P2)) = \{\mathbf{x}_{R_1}\}$ . For programs to be equivalent, there must be an isomorphism between the sets, mapping each free variable to a single variable with the same source input RDD.

**Compiling SparkLite to logical terms using representative elements** Let  $P$  be a SparkLite program. We use standard notations  $r^{in}$  for the inputs of  $P$ , and  $r^{out}$  for the output of  $P$ . The term  $\Phi(P)$  is called the *program term of  $P$*  and it is defined by structural induction according to the notations in Figure 5. We write the set of free variables of the program term  $FV(\Phi(P))$  as a vector:  $(\mathbf{x}_{r_{j_k}^{in}})_{k=1}^{n_P}$ , where  $n_P$  is the number of free variables. A vector of valuations to the free variables is denoted  $\bar{x} = (x_1, \dots, x_n)$  and satisfies  $x_k \in r_{j_k}^{in}$  for  $k \in \{1, \dots, n\}$ . The *SR semantics* (*Symbolic Representation Semantics*) of a program that returns an RDD-type output is the bag that is obtained from all possible valuations to the free variables:

$$SR(P)(\overline{r^{in}}) = \{ \{ \Phi(P)[\bar{x}/FV(\Phi(P))] \mid \Phi(P)[\bar{x}/FV(\Phi(P))] \neq \perp \wedge \forall k \in \{1, \dots, n_P\}. x_k \in r_{j_k}^{in} \} \}$$



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  $\overline{r^{in}}$ , we get the bag equal  $r^{out}$ . We elaborate later on programs that return a basic type by using aggregate operations on RDDs.

► **Proposition 2.** *Let  $P : P(\bar{r}) = \bar{F} \bar{f} E$  be a SparkLite program,  $\llbracket P \rrbracket$  be the interpretation of its output according to the defined semantics, and  $SR(P)$  by the symbolic representation semantics of  $P$  as described earlier. Without loss of generality, we do not consider programs with non-RDD inputs. Then, for any input  $\overline{r^{in}}$ , we have:*

$$SR(P)(\overline{r^{in}}) = \llbracket P \rrbracket(\llbracket \overline{r^{in}} \rrbracket)$$

**Proof.** See appendix ??.

**Program equivalence problem formalization using representative elements** Given two programs  $P, Q$  receiving as input a series of RDDs  $\overline{r^{in}} = (r_1^{in}, \dots, r_n^{in})$ . We assume w.l.o.g. the programs are non-trivial, meaning they do not return the empty RDD for any choice of inputs. We define isomorphism of sets of free variables for  $P, Q$  as an injective and onto mapping:  $\mathcal{S} : FV(\Phi(P)) \xrightarrow{\sim} FV(\Phi(Q))$  such that  $\forall k \in \{1, \dots, n\}. S(\mathbf{x}_{r_k^{in}}^{(k)}) = \mathbf{x}_{r_{j_k}^{in}}^{(i)} \wedge j_k = j_i$

The *PE* problem becomes the problem of proving the following:

$$\begin{aligned} (*) \quad & FV(\Phi(P)) \simeq^{\mathcal{S}} FV(\Phi(Q)) \\ (**) \quad & \forall \bar{x} \Phi(P)[\bar{x}/FV(\Phi(P))] = \Phi(Q)[\bar{x}/\mathcal{S}(FV(\Phi(P)))] \end{aligned}$$

where the choice of  $\overline{r^{in}}$  is arbitrary, and  $\mathcal{S}$  is non-deterministically chosen from all legal isomorphisms.

## 4 Verifying Equivalence of SparkLite Programs

**The Program Equivalence (PE) problem** Let  $P_1$  and  $P_2$  be SparkLite programs, with signature  $P_i(\bar{T}, RDD_{\bar{T}}) : \tau$  for  $i \in \{1, 2\}$ . We use  $\llbracket P_i \rrbracket(\llbracket \bar{v} \rrbracket, \llbracket \bar{r} \rrbracket)$  to denote the result of  $P_i$ . We say that  $P_1$  and  $P_2$  are *equivalent*, if for all input values  $\bar{v}$  and RDDs  $\bar{r}$ , it holds that  $\llbracket P_1 \rrbracket(\llbracket \bar{v} \rrbracket, \llbracket \bar{r} \rrbracket) = \llbracket P_2 \rrbracket(\llbracket \bar{v} \rrbracket, \llbracket \bar{r} \rrbracket)$ .

### 4.1 Verifying Equivalence of SparkLite Programs without Aggregations

**Program equivalence problem formalization using representative elements** Given two programs  $P, Q$  receiving as input a series of RDDs  $\overline{r^{in}} = (r_1^{in}, \dots, r_n^{in})$ . We assume w.l.o.g. the programs are non-trivial, meaning they do not return the empty RDD for any choice of inputs. We define isomorphism of sets of free variables for  $P, Q$  as an injective and onto mapping:  $\mathcal{S} : FV(\Phi(P)) \xrightarrow{\sim} FV(\Phi(Q))$  such that  $\forall k \in \{1, \dots, n\}. S(\mathbf{x}_{r_k^{in}}^{(k)}) = \mathbf{x}_{r_{j_k}^{in}}^{(i)} \wedge j_k = j_i$

The *PE* problem becomes the problem of proving the following:

$$\begin{aligned} (*) \quad & FV(\Phi(P)) \simeq^{\mathcal{S}} FV(\Phi(Q)) \\ (**) \quad & \forall \bar{x} \Phi(P)[\bar{x}/FV(\Phi(P))] = \Phi(Q)[\bar{x}/\mathcal{S}(FV(\Phi(P)))] \end{aligned}$$

where the choice of  $\overline{r^{in}}$  is arbitrary, and  $\mathcal{S}$  is non-deterministically chosen from all legal isomorphisms.

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



1. Verify that:

$$FV(\Phi(P)) = FV(\Phi(Q))$$

If not, output **not equivalent**. The definition of  $\Phi$  appears in Figure 5.

2.
  - a. Choose an isomorphism  $\mathcal{S}$  of the representative elements of the input RDDs in both  $P, Q$ .
  - b. We check the following formula is satisfiable:

$$\exists \bar{v}. \Phi(P)[\bar{v}/FV(\Phi(P))] \neq \Phi(Q)[\bar{v}/\mathcal{S}(FV(\Phi(Q)))]$$

- c. If it is satisfiable, go back to (a) and repeat until finding an unsatisfiable formula, or all possible isomorphisms were exhausted.
  - d. If the formula is unsatisfiable, return **equivalent**.
  - e. If all isomorphisms were exhausted without finding an unsatisfiable formula, then return **not equivalent**.

■ **Figure 6** An algorithm for solving  $PE$  for two programs  $P, Q$  with the same signature

**Proof.** For non-RDD return types we already proved in Proposition 1 - without aggregate operations, basic types can not be influenced by operations performed on RDDs. Thus, we can remove all RDD operations from the program and get an equivalent program in the setting of Proposition 1. For RDDs we provide an algorithm in Figure 6, which is a decision procedure. The correctness of the algorithm follows from the equivalency of the  $SR()$  semantics and the semantics defined in 2.4. It checks that two  $SR()$  expressions are equal, and the process terminates:

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

◀

**Note:** All examples use syntactic sugar for ‘*Let*’ expressions. For brevity, instead of applying  $\phi$  on the underlying ‘*Let*’ expressions, we apply it line-by-line from the top-down. Finding the isomorphism  $\mathcal{S}$  between variable names in both programs is done automatically in all programs, but formally it is part of the decision procedure. In addition, we assume that in programs returning an RDD-type, the RDD is named  $r^{out}$ , and the programs always end with **return**  $r^{out}$ . Thus,  $\Phi(P) = \phi_P(r^{out})$ .

► **Example 1 (Basic optimization - operator pushback).** This example shows a common optimization of pushing the filter/selection operator backward, to decrease the size of the dataset.

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

**RDD return type:** Both programs return an RDD of integers.

**Free variables:**  $FV(\Phi(P1)) = \{x_R\} = FV(\Phi(P2))$ . Thus, the sets of free variables are equal.

## XX:10 Verifying Equivalence of Spark Programs

**Analysis of representative elements:**

$$\begin{aligned} \phi_{P1}(R') = 2 * \mathbf{x}_R, \text{ and } \phi_{P1}(r^{out}) &= \begin{cases} \varphi & \varphi < 14 \wedge \varphi = \phi_{P1}(R') \\ \perp & \text{otherwise} \end{cases} = \begin{cases} 2 * \mathbf{x}_R & 2 * \mathbf{x}_R < 14 \\ \perp & \text{otherwise} \end{cases} \\ \phi_{P1}(R') &= \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \perp & \text{otherwise} \end{cases}, \text{ and } \phi_{P2}(r^{out}) = (\lambda x.x + x)(\phi_{P1}(R')) = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \perp & \text{otherwise} \end{cases} + \\ &\begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \perp & \text{otherwise} \end{cases}. \end{aligned}$$

We need to verify that:

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

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

► **Proposition 3 (Schemes for converting conditionals to a normal form).** *We write a series of universally true schemes for translating the filter cased expression to Presburger arithmetic when appearing in an equivalence formula:*

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

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

2. *Equivalence of functions of conditionals:*

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

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

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

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

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

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

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

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

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

■ **Figure 7** Aggregate operators for SparkLite programs with aggregations

**Fold:**  $\phi_P(\llbracket \text{fold} \rrbracket(f, e)(r), k) = ([t]_{e,f}, m), \text{ where } (t, m) = \phi_P(r, k)$   
**FoldByKey:**  $\phi_P(\llbracket \text{foldByKey} \rrbracket(f, e)(r), k) = (\langle t \rangle_{e,f}, m), \text{ where } (t, m) = \phi_P(r, k)$

■ **Figure 8** Compiling SparkLite to logical terms for aggregate operations.

## 7. Unnsetting of nested conditionals

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

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

For additional examples, refer to appendix ??.

► **Theorem 2.** *The algorithm described in Figure 6 is a decision procedure for the PE problem in SparkLite programs without aggregations (fold, foldByKey).*

## 4.2 Verifying Equivalence of a Class of SparkLite Programs with Aggregation

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

**Extending SR(P) with aggregate expressions** The terms for aggregate operations are given using special operators (Figure 7). We extend the definition of  $\phi$  accordingly in Figure 8. The  $[\varphi]_{\cdot}$  operator *binds* all variables in  $\varphi$ . The  $\langle \varphi \rangle_{\cdot}$  operator is syntactic sugar for  $(p_1(\varphi), [p_2(\varphi)]_{\cdot})$  which *binds* all variables in  $p_2(\varphi)$  which are not in  $p_1(\varphi)$ . That is, the variables of  $p_1(\varphi)$  are still free in the fold expression. The motivation is explained in appendix ??.

### 4.2.1 Single aggregate as the final expression

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

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

Let: $\begin{aligned} \max &= \lambda M, x. \text{if}(x > M) \text{ then } \{x\} \text{ else } \{M\} \\ \min &= \lambda M, x. \text{if}(x < M) \text{ then } \{x\} \text{ else } \{M\} \end{aligned}$		
	$P1(R : RDD_{Int}):$	$P2(R : RDD_{Int}):$
1	<b>return</b> fold( $\perp, \max$ )( $R$ )	$R' = \text{map}(\lambda x. 0 - x)(R)$
2		<b>return</b> 0- fold( $\perp, \min$ )( $R'$ )

## XX:12 Verifying Equivalence of Spark Programs

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

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

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

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

$$\begin{aligned} \max(A, x) &\stackrel{?}{=} -\min(A', -x) \\ \begin{cases} A & A > x \\ x & \text{otherwise} \end{cases} &\stackrel{?}{=} -\begin{cases} A' & A' < -x \\ -x & \text{otherwise} \end{cases} = \begin{cases} -A' & A' < -x \\ x & \text{otherwise} \end{cases} \stackrel{A=-A'}{=} \begin{cases} A & -A < -x \\ x & \text{otherwise} \end{cases} = \begin{cases} A & A > x \\ x & \text{otherwise} \end{cases} \end{aligned}$$

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

### Basic proof method

The inductive claim is generalized for the class of programs discussed here, of programs performing a single *aggregate* operation after a series of *map*, *filter* and *cartesian* operations (without self-products, that is without variable renaming). Those definitions lead to the following lemma:

► **Lemma 3.** *Let  $R_0 \in RDD_{\sigma_0}$ ,  $R_1 \in RDD_{\sigma_1}$ , and denote their representative elements  $\varphi_0, \varphi_1$  respectively. We assume  $\varphi_0, \varphi_1$  were composed from *map*, *filter* and *cartesian* product (without self products). Let there be two fold functions  $f_0 : \xi_0 \times \sigma_0 \rightarrow \xi_0$ ,  $f_1 : \xi_1 \times \sigma_1 \rightarrow \xi_1$ , two initial values  $init_0 : \xi_0$ ,  $init_1 : \xi_1$ , and two functions  $g : \xi_0 \rightarrow \xi$ ,  $g' : \xi_1 \rightarrow \xi$ . We have: if*

$$FV(\varphi_0) \simeq FV(\varphi_1), \text{ denoted } FV \quad (1)$$

$$g(init_0) = g'(init_1) \quad (2)$$

$$\forall \bar{v}, A_{\varphi_0} : \xi_0, A_{\varphi_1} : \xi_1. g(A_{\varphi_0}) = g'(A_{\varphi_1}) \implies \quad (3)$$

$$g(f_0(A_{\varphi_0}, \varphi_0[\bar{v}/FV])) = g'(f_1(A_{\varphi_1}, \varphi_1[\bar{v}/FV]))$$

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

**Induction length and relation to size of bags** The proof of lemma 3 assumes the sets of free variables to be isomorphic, otherwise the induction termination is not well defined. One possibility is that the size of participating RDDs may not be equal. As a workaround, suppose we take a valuation from the union of *FV*s, using  $\perp$  values if needed. Let there be two equal fold expressions under lemma's 3 premises, and a valuation  $\bar{v}$  in the valuation sequence returning a non- $\perp$  value in  $R_0$ , and  $\perp$  valuation to  $R_1$ . We get:

$$\begin{aligned} g(A_{\varphi_0, i}) = g'(A_{\varphi_1, i}) \quad \wedge \quad g(f_0(A_{\varphi_0, i}, \varphi_0[\bar{v}/FV(\varphi_0)])) &= g'(f_1(A_{\varphi_1, i}, \perp)) = g'(A_{\varphi_1, i}) \\ \implies g(f_0(A_{\varphi_0, i}, \varphi_0[\bar{v}/FV(\varphi_0)])) &= g(A_{\varphi_0, i}) \\ \implies \varphi_0[\bar{v}/FV(\varphi_0)] \neq \perp &\quad \forall A, c. f_0(A, c) = A \end{aligned}$$

In that case, we see in the last transition that the lemma's conditions are fulfilled only if the fold functions are constant, namely the intermediate value returned is never changed by

it. However, fold UDFs which are constant have no application in practice, thus we ignore programs containing them.

Lemma 3 shows that an inductive proof of the equality of folded values is *sound*. The meaning is that given any two folded expressions which are not equivalent, the lemma always reports them as non-equivalent. We show a constraint on 3, for which the method is also complete.

**Examples** Refer to appendix ??.

### Completeness

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

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

► **Example 3 (Non-injective modification of folded expressions).** Non-injective transformations can weaken the inductive claim, resulting in failure to prove it. As a result, lemma 3 fails to prove the equivalence of the following two programs.

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

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

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

$$\forall x, A, A'. A = 0 \Leftrightarrow A' \bmod 3 = 0 \implies (A + x \bmod 3) \bmod 3 = 0 \Leftrightarrow (A' + x) \bmod 3 = 0$$

fails. To illustrate, suppose that in the induction hypothesis we have  $A = 1, A' = 2$ . Then the hypothesis that says  $A = 0 \Leftrightarrow A' \bmod 3 = 0$  is satisfied, but it cannot be said that  $(A + x \bmod 3) \bmod 3 = 0 \Leftrightarrow (A' + x) \bmod 3 = 0$  (take  $x = 1$ :  $((1 + (1\%3))\%3 = 2, (2 + 1)\%3 = 0)$ ).

From the above example we derive the following lemma:

► **Lemma 4.** Under the premises of lemma 3, if we have:

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

and in addition,  $f_0, f_1$  can generate any intermediate value, then we can prove  $(*)$  by induction.

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

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

## XX:14 Verifying Equivalence of Spark Programs

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

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

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

Let the sequence of valuations  $\langle \bar{a}_1, \dots, \bar{a}_n \rangle$  be the generators of the intermediate values  $A_{\varphi_0}, A_{\varphi_1}$ . We take RDDs  $R_0, R_1$  defined as follows for  $j \in \{0, 1\}$ :

$$R_j = \bigcup_{i=1, \dots, n} \{ \{ \varphi_j[\bar{a}_i/FV(\varphi_j)] \} \} \cup \{ \{ \varphi_j[\bar{v}/FV(\varphi_j)] \} \}$$

For this choice of RDDs we have:

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

But, from the assumption:

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

we get :

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

contradiction. ◀

Lemma 4 shows that the inductive process can be applied on injective mappings of folded expressions in order to prove their equivalence.

**A class for which *PE* is undecidable** We show a reduction of Hilbert's 10'th problem to *PE*. We assume towards a contradiction that *PE* is decidable under the premises of lemma 5 with representative elements based also on the *cartesian* operation. Let there be a polynomial  $p$  over  $k$  variables  $x_1, \dots, x_k$ , and coefficients  $a_1, \dots, a_k$ . For each variable  $x_i$  we assume the existence of some RDD  $R_i$  with  $x_i$  elements. We use SparkLite operations and the input RDDs  $R_i$  to represent the value of the polynomial  $P$  for some valuation of the  $x_i$ . For each summand in the polynomial  $p$ , we define a translation  $\varphi$ :

- $x_i \xrightarrow{\varphi} [\text{map}(\lambda x.1)(R_i)]_{0,+}$
- $x_i x_j \xrightarrow{\varphi} [\text{cartesian}(\text{map}(\lambda x.1)(R_i), \text{map}(\lambda x.1)(R_j))]_{0,+}$
- By induction,  $x_i^2 \xrightarrow{\varphi} [\text{cartesian}(\text{map}(\lambda x.1)(R_i), \text{map}(\lambda x.1)(R_i))]_{0,+}$ .  
This rule as well as the rest of the powers follow according to the previous rule. For a degree  $k$  monom, we apply the *cartesian* operation  $k$  times.
- $x_i^0 \xrightarrow{\varphi} 1$ , trivially.
- $am(x) \xrightarrow{\varphi} a\varphi(m(x))$  where  $m(x)$  is a monomial with coefficient 1 of the variable  $x$ , thus we have already defined  $\varphi$  for it.
- $aq(x_{i_1}, \dots, x_{i_j}) \xrightarrow{\varphi} a\varphi(q(x_{i_1}, \dots, x_{i_j}))$  where  $q(x)$  is a monomial with coefficient 1 and multiple variables for which  $\varphi$  was defined in the previous rules.
- $\varphi(p(a_1, \dots, a_k; x_1, \dots, x_k)) = \sum_{i=1}^k \varphi(a_i q(x_{i_1}, \dots, x_{i_k}))$  follows by structural induction on the previous rules.

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

	P1( $R_1, \dots, R_k: RDD_{\text{Int}}$ ):	P2( $R_1, \dots, R_k: RDD_{\text{Int}}$ ):
1	<b>return</b> $\varphi(p) \neq 0$	<b>return</b> $tt$

By choosing input RDDs of the cardinality of  $R_i$  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 root, and if it outputs ‘not equivalent’ then the polynomial  $p$  has some root, where  $x_i = \llbracket R_i \rrbracket$ . Thus we have polynomial reduction to Hilbert’s 10’t problem.

#### 4.2.2 Multiple aggregates

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

► **Example 4 (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 ?? on each element of the resulting tuple, we are able to show the equivalency.

Let: $h : (\lambda(P, N), x. \begin{cases} (P + x, N) & x \geq 0 \\ (P, N - x) & \text{otherwise} \end{cases})$	
1	P1( $R: RDD_{Int}$ ):
2	<b>return</b> fold((0,0),h)( $R$ )
3	P2( $R: RDD_{Int}$ ):
4	$R_P = \text{filter}(\lambda x. x \geq 0)(R)$
5	$R_N = \text{map}(\lambda x. -x)(\text{filter}(\lambda x. x < 0)(R))$
	$p = \text{fold}(0, \lambda A, x. A + x)(R_P)$
	$n = -\text{fold}(0, \lambda A, x. A + x)(R_N)$
	<b>return</b> ( $p, n$ )

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

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

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

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

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

$$\begin{aligned} \begin{cases} N & x \geq 0 \\ N - x & \text{otherwise}(x < 0) \end{cases} & \stackrel{=?}{=} N' + \begin{cases} -x & x < 0 \\ \perp & \text{otherwise} \end{cases} \\ & \stackrel{=\perp \text{ is neutral}}{=} \begin{cases} N' - x & x < 0 \\ N' & \text{otherwise} \end{cases} \\ & \stackrel{=N=N'}{=} \begin{cases} N - x & x < 0 \\ N & \text{otherwise} \end{cases} \\ & = \begin{cases} N - x & x < 0 \\ N & x \geq 0 \end{cases} \end{aligned}$$

And the equivalence follows. ■



► **Lemma 5.** Let  $R_1 \in RDD_{\sigma_1}, \dots, R_k \in RDD_{\sigma_k}$ , and denote the representative elements  $\varphi_i$  for  $i \in \{1, \dots, k\}$ . We assume the  $\varphi_i$  are based only on map and filter operations and cartesian without self-products. Let there be  $k$  fold UDFs  $f_i: \xi_i \times \sigma_i \rightarrow \xi_i$ , and  $k$  initial values  $init_i: \xi_i$ . Let a similar set of RDDs, representative elements, fold UDFs and initial values, with all denotations having a '. Let there be 2 functions  $g: \xi_1 \times \dots \times \xi_k \rightarrow \xi, g': \xi'_1 \times \dots \times \xi'_{k'} \rightarrow \xi$ . We denote a vector  $\bar{v} = FV(\varphi_1, \dots, \varphi_k, \varphi'_1, \dots, \varphi'_{k'})$  of the free variables in all representative elements. We have: if

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

$$\begin{aligned} \forall \bar{v}, A_{\varphi_1}: \xi_1, \dots, A_{\varphi_k}: \xi_k, A_{\varphi'_1}: \xi'_1, \dots, A_{\varphi'_{k'}}: \xi'_{k'}. g(A_{\varphi_1}, \dots, A_{\varphi_k}) = g'(A_{\varphi'_1}, \dots, A_{\varphi'_{k'}}) \implies \\ g(f_1(A_{\varphi_1}, \varphi_1[\bar{v}/FV(\varphi_1)]), \dots, f_k(A_{\varphi_k}, \varphi_k[\bar{v}/FV(\varphi_k)])) = \\ g'(f'_1(A_{\varphi'_1}, \varphi'_1[\bar{v}/FV(\varphi'_1)]), \dots, f'_{k'}(A_{\varphi'_{k'}}, \varphi'_{k'}[\bar{v}/FV(\varphi'_{k'})])) \end{aligned} \quad (2)$$

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

► **Theorem 6** (Basic decidability for PE). For two SparkLite programs  $Q_1, Q_2$  returning a single or a tuple of RDDs, and for two SparkLite programs  $T_1, T_2$  which act on the output of  $Q_1, Q_2$  respectively by applying fold functions on them, we define  $P_i = T_i \circ Q_i$ . The PE problem is decidable if either  $T_1$  or  $T_2$  apply an injective transform on the folded expressions.

**Proof.** A conclusion from lemmas 3, 4, ??, 5 ◀

### Nested aggregations

We saw in 6 a proof of decidability for a fragment of SparkLite programs. In the following subsection we present more complex SparkLite programs on which the theorem applies, the method for proving the equivalence, and the cases on which it fails. Those programs have a value of an aggregate operation used in later aggregations (i.e. 'nested' aggregations). We see that the inductive method is sound in handling those cases, and that under certain conditions, it is complete too.

► **Example 5** (Conditional summation). The following example takes the *sum* of all elements which are greater than the *count* of elements in an RDD.

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

	P1( $R: RDD_{Int}$ ):	P2( $R: RDD_{Int}$ ):
1	$R' = \text{map}(\lambda x. (x, 2))(R)$	$R' = \text{map}(\lambda x. (x, 1))(R)$
2	$sz = \text{fold}(0, f)(R')$	$sz = \text{fold}(0, f)(R')$
3	$B = \text{filter}(\lambda x. x > sz)(R)$	$B = \text{filter}(\lambda x. x > 2 * sz)(R)$
3	<b>return</b> $\text{fold}(0, +)(B)$	<b>return</b> $\text{fold}(0, +)(B)$

The equivalence condition is:

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

Replacing  $sz, sz'$  we get:

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

After formally applying lemma 3 we get that the above is equivalent if and only if  $\phi_{P_1}(sz) = 2 * \phi_{P_2}(sz)$ , that is:

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

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

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

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

Proving the equivalence. ■

This property is reflected in the following proposition:

► **Proposition 4.** *Let there be two SparkLite programs  $P_1, P_2$  and returning aggregated expressions  $[a_i(\bar{r})]_{init_i, f_i}$ . Let there be two other SparkLite programs,  $T_1, T_2$ , also returning aggregated expressions,  $[b_i(\bar{r}', a)]_{init_T, g}$  (the aggregation function is equal in both  $T_1, T_2$ ). PE is decidable for  $T_1 \circ P_1, T_2 \circ P_2$  if and only if PE is decidable for  $P, P'$ .*

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

From proposition 4, it can be concluded that representative elements which are an injection as a function of the folded values have a decidable equivalence checking procedure. In the above example, the expression:

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

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

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

► **Theorem 7.** *Let there be two SparkLite programs  $P_1, P_2$ , returning an expression of the form:  $f_i([\phi_i(\bar{x}, a_i(\bar{y}))]_{init_i, g_i})$ , The PE problem is decidable if exists  $i \in \{1, 2\}$  such that the following expressions are all injective:*

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

## XX:18 Verifying Equivalence of Spark Programs

**Proof.** We apply lemma 3 on the expressions returned by  $P_i$ :

$$f_1(\text{init}_1) = f_2(\text{init}_2) \quad (1)$$

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

$a_1, a_2$  are aggregated expressions:  $a_i = [\psi_i(\bar{y})]_{j_i, h_i}$ . So we apply lemma 3 again. For brevity, we denote:  $\mathcal{F}' = f_i(g_i(A_i, \phi_i(\bar{x}, a_i(\bar{y})))) = \mathcal{F}(A_i, \bar{x})$

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

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

As we know that for at least one of the sides we have injective functions for both applications of lemma 3, then by theorem 6 we have a decision procedure for the equivalence. ◀

Therefore, by using the conditions determined by theorem 7, there is no need to assume the outer aggregation function is equal, as stated in this lemma:

► **Lemma 8.** *Let there be two SparkLite programs  $P_1, P_2$  and returning aggregated expressions  $[a_i(\bar{r})]_{\text{init}_i, f_i}$ . Let there be two other SparkLite programs,  $T_1, T_2$ , also returning aggregated expressions,  $[b_i(\bar{r}', a)]_{\text{init}_{T_i}, g_i}$ . PE is decidable for  $T \circ P, T' \circ P'$  if and only if PE is decidable for  $P, P'$ .*

**Proof.**  $\phi_{P_1} = [a_1(\bar{r})]_{\text{init}_1, f_1}, \phi_{P_2} = [a_2(\bar{r})]_{\text{init}_2, f_2}$  are the program terms of  $P_1, P_2$ , respectively.  $a, a'$  are terms without aggregations. Let  $[b_i(\bar{r}', \phi_{P_i})]_{\text{init}_{T_i}, g_i}$  for  $i \in \{1, 2\}$  be the program terms for  $T_1 \circ P_1, T_2 \circ P_2$ , respectively. The programs are equivalent if and only if the program terms are equivalent. By applying lemma 3 on the outer aggregation, we need to check:

$$g_1(\text{init}_{T_1}) = g_2(\text{init}_{T_2}) \quad (1)$$

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

Verifying the second equation is decidable under the assumptions of theorem 7. ◀

Several examples are given in appendix ??.

---

**References**

---

- 1 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.
- 2 David C Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 1972.
- 3 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: <http://doi.acm.org/10.1145/115372.115320>, doi:10.1145/115372.115320.
- 4 Derek C. Oppen. A 222pn upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323 – 332, 1978. URL: <http://www.sciencedirect.com/science/article/pii/0022000078900211>, doi:[http://dx.doi.org/10.1016/0022-0000\(78\)90021-1](http://dx.doi.org/10.1016/0022-0000(78)90021-1).
- 5 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.