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 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* (*UDF*s).

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 criteria 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 [9], 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, which we denote as $\{\!\{\cdot\}\!\}$.

Presburger Arithmetic. We define a fragment of first-order logic over the integers, whose syntax is specified in Figure 1. Disregarding the tuple expressions $((pe\,,\overline{pe}) \mid \boldsymbol{p}_i(e))$, the

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

Figure 1 Terms of the Augmented Presburger Arithmetic

resulting first order theory with the usual $\forall, \exists$ 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 ([9, 6]), even when combined with Boolean logic ([7, 2]). *Cooper's Algorithm* [4] is a standard decision procedure for Presburger Arithmetic[1]. We extend this language by adding a *tuple constructor* $(pe\,, \overline{pe})$ and a projection operator $\boldsymbol{p}_i(e)$, and call the extended language *Augmented Presburger Arithmetic*. The decidability of Presburger Arithmetic, as well as Cooper's Algorithm, can be naturally extended to the Augmented Presburger Arithmetic.

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

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*) [10].

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 $\boldsymbol{c}$ to range over integer numerals (constants), $\boldsymbol{b} \in \{\texttt{true}, \texttt{false}\}$ to range over Boolean constants, and τ to range over basic types and record types.

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

Semantic Domains. We interpret the integer and Boolean primitive types as *integers* ($\mathbb{Z}$) and *booleans* ($\mathbb{B}$), respectively. We use $[\![\cdot]\!]$ (semantic brackets) throughout the paper to denote the semantics of program constructs; for types, the semantics is a set of all the values pertaining to it. So $[\![\texttt{Int}]\!] = \mathbb{Z}$, $[\![\texttt{Boolean}]\!] = \mathbb{B}$.

The interpretation of both primitive types is denoted $\mathrm{T} = \mathbb{Z} \cup \mathbb{B}$. The interpretation of all possible types (including mixed Cartesian products of the primitive types) is denoted by $\mathcal{T} = \bigcup_n \mathrm{T}^n$.

An *RDD* type is interpreted as a *bag* (an unordered set allowing repeating elements). We write $[\![RDD_{\boldsymbol{\tau}}]\!] = ([\![\boldsymbol{\tau}]\!] \hookrightarrow \mathbb{N})$, meaning that an RDD value $\boldsymbol{r}$ of type $RDD_{\boldsymbol{\tau}}$ is interpreted as a bag of values from $\boldsymbol{\tau}$, that is $[\![r]\!] \in ([\![\boldsymbol{\tau}]\!] \hookrightarrow \mathbb{N})$. We let $[\![RDD]\!] = \bigcup_{\tau \in \mathcal{T}} [\![RDD_\tau]\!]$, the semantic domain of RDDs over all possible record types $\tau \in \mathcal{T}$.

[1] A remark on complexity: Cooper's algorithm has an upper bound of $2^{2^{2^{pn}}}$ for some $p > 0$ and where n is the number of symbols in the formula [8]. In practice, our experiments show that Cooper's algorithm on non-trivial formulas returns almost instantly, even on commodity hardware.

2.2 Functional Model

Operations. *RDD*s are analogous to database tables and as such the methods to query the *RDD*s are inspired by both *Relational Algebra* (*RA*) [3, 1] and Spark [11]. *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 UDFs (see next), which allow *extended projection* as well as greater flexibility in executing complex operations on elements of different types.

UDFs. A special feature of Spark is allowing some of its standard operations to be *higher order functions* — they take a function and apply it to an RDD in a specific manner defined by the operation. For example, the operation *fold* can be applied to an RDD containing integer numbers by providing a function that adds to two integers to the *fold* transform, yielding the sum of all the numbers in the bag. Such a function is called a *"User-Defined Function"*, or *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 B). Each UDF has a *definition*, which takes the syntax $\lambda\overline{v}.\ e$, where $\overline{v}$ are names of variables used as arguments. For example, $addMod10 = \lambda x, y.\ x\%10 + y\%10$. This function's *signature* would be $addMod10\colon \mathtt{Int} \times \mathtt{Int} \to \mathtt{Int}$, that is, it takes two `Int` arguments and produces one `Int` value. The types are usually omitted from the definition for brevity, but when the types are not clear from the context, we may write them as annotations: $addMod10 = \lambda x\colon\mathtt{Int}, y\colon\mathtt{Int}.\ x\%10 + y\%10\colon\mathtt{Int}$. We could also write it as a function with single argument like this: $addPairMod10 = \lambda z.\ p_1(z)\%10 + p_2(z)\%10$, where it would have the signature $addPairMod10\colon(\mathtt{Int} \times \mathtt{Int}) \to \mathtt{Int}$. For readability, we sometimes give names to the projections and write them implicitly as $addPairMod10 = \lambda(x, y).\ x\%10 + y\%10$; this is just a shortcut.

We allow the definition of these functions to be *parametric*, by using two levels of λ. The outer level denotes the parameters, which can be provided to obtain a regular function. For example, a function that adds 1 to an integer is written as $\boldsymbol{f} = \lambda x.\ x+1$, where a function that adds any constant is written as $\boldsymbol{g} = \lambda a.\ \lambda x.\ x + a$. The function is *curried*, so that applying it to parameter values of the appropriate types produces a function (by *beta-reduction*): $\boldsymbol{g}(1)$ is identical to $\boldsymbol{f}$.

2.3 Syntax

The syntax of SparkLite language is defined in Figure 2.

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

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

Figure 2 Syntax for SparkLite

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

$\text{P1}(R_0\!:RDD_{\texttt{Int}}, R_1\!:RDD_{\texttt{Int}})$:
$A = \texttt{filter}(isOdd)(R_0)$
$B = \texttt{map}(doubleAndAdd(1))(A)$
$C = \texttt{cartesian}(B, R_1)$
$v = \texttt{fold}(0, sumFlatPair)(C)$
$\texttt{return}\ v$

Figure 3 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 [5]. Variables are immutable by this construction. Programs have no side effects, do not change the inputs, and always return a value. The *program signature* will consist of its name, its input types and return type: $\boldsymbol{P}(\overline{\mathcal{T}_i}, \overline{RDD_i})\!:\mathcal{T}_o$

Example program. Consider the example 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 announced with a list of input RDDs (R_0, R_1) ($Prog$ rule). Instead of writing $Let\ l_1\ in\ Let\ l_2\ in\ \ldots$, 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, 3 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

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

Figure 4 Semantics of SparkLite. $\mathit{Prog} = \boldsymbol{P}(\overline{\boldsymbol{r} : \mathit{RDD}_\tau}, \overline{\boldsymbol{v} : \tau})\ =\ \overline{\mathit{Fdef}}\ \ \overline{\mathit{PFdef}}\ \ E$. `unOp` and `binOp` are taken from Figure 2: $\texttt{unOp} \in \{-, \neg, \boldsymbol{\pi}_i\}$, $\texttt{binOp} \in \{+, *, /, \%, =, <, \wedge, \vee, (,)\}$

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 total of all this elements is stored in the variable v. The returned value is the integer variable v. The program's signature is $P1(\mathit{RDD}_{\texttt{Int}}, \mathit{RDD}_{\texttt{Int}})\colon \texttt{Int}$.

2.4 Operational Semantics

Program Environment. We define a unified semantic domain $\mathcal{D} = \mathcal{T} \cup \mathit{RDD}$ 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.

Data flow. We start with an initial environment function ρ_0 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.

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 λ-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.

$$
\begin{array}{lll}
\phi_P(\mathit{Let}\ x = \eta\ \mathit{in}\ E, k) & = & (t_{E'}[t_\eta/x], m),\ \mathit{where}\ (t_{E'}, n) = \phi_P(E', k), (t_\eta, m) = \phi_P(\eta, n) \\
\phi_P(e, k) & = & (e, k) \\
\phi_P(\mathtt{map}(f)(r), k) & = & (f(t), m),\ \mathit{where}\ (t, m) = \phi_P(r, k) \\
\phi_P(\mathtt{filter}(f)(r)) & = & \mathit{ite}(f(t) = \mathit{tt}, (t, m), \bot),\ \mathit{where}\ (t, m) = \phi_P(r, k) \\
\phi_P(\mathtt{cartesian}(r_1, r_2), k) & = & ((t_{r_1}, t_{r_2}), m),\ \mathit{where}\ (t_{r_1}, n) = \phi_P(r_1, k), (t_{r_2}, m) = \phi_P(r_2, n) \\
\phi_P(\mathtt{fold}(f, e)(r), k) & = & ([t]_{e,f}, m),\ \mathit{where}\ (t, m) = \phi_P(r, k) \\
\phi_P(r, k) & = & \begin{cases} (\mathbf{x}_r^{(k)}, k+1) & r \in \overline{r} \\ (r, k) & \mathit{otherwise} \end{cases}
\end{array}
$$

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

Figure 5 Compiling SparkLite to logical terms (ϕ). Semantic brackets ($[\![\cdot]\!]$) were omitted.

Semantics of operations. In Figure 4 the semantics of all RDD expressions, including aggregation, are explicitly stated.

Notes. In the *bag semantics*:

- In *map*, if f maps y to x, 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 f on it n times.
- *fold* is *well defined*: 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 $\boldsymbol{f}$, the order in which *fold* picks elements from the bag does not affect the result.

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

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 the Augmented Presburger Arithmetic. This term is called the *program term* and denoted $\Phi(P)$ for program P, specified in Figure 5. The variables of the term are taken from the input RDDs. We take the previously analyzed example program from Figure 3. The Φ function is defined using the function ϕ whose purpose is to maintain unique variable names. ϕ is applied recursively on the expression returned by the program. We simplify by running ϕ_{P1} on each line of the program, top-down:

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

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 notation, the term for a non-input RDD, computed in a SparkLite program, is also called a representative element.

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

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

$P1$ and $P2$ have the same program term (the constant 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 is 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. Therefore, for each program term $\Phi(P)$ we consider the *set of free variables*, $FV(\Phi(P))$. 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, $FV(\Phi(P1)) = \{\mathbf{x}_{R_0}\}$, and $FV(\Phi(P1)) = \{\mathbf{x}_{R_1}\}$. For programs to be equivalent, there must be an *isomorphism* between the sets, mapping each free variable to a single free variable with the same source input RDD. An exception to the rule of having an isomorphism of the free variables are *trivial programs*, that always return the empty RDD, which has no multiplicity.

Formalization of the Term Semantics for SparkLite. Let P be a SparkLite program. We use standard notations $\overline{r}$ for the inputs of P, and r^{out} for the output of P. The term $\Phi(P)$ is called the *program term of* P as before. We write the set of free variables of the program term $FV(\Phi(P))$ as a vector: $(\mathbf{x}_{r_{j_k}})_{k=1}^{n_P}$, where n_P is the number of free variables. A vector of valuations to the free variables is denoted $\overline{x} = (x_1, \dots, x_{n_P})$ and satisfies $x_k \in r_{j_k}$ for $k \in \{1, \dots, n_P\}$. 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:

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

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 $\overline{r}$, we get the bag equal to r^{out}. For a program that returns a basic type and not an RDD, $TS(P) = \Phi(P)$.

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

$$TS(P)(\overline{v}, \overline{r}) = [\![P]\!]([\![\overline{v}]\!], [\![\overline{r}]\!])$$

Proof. See appendix C. ◂

4 Verifying Equivalence of SparkLite Programs

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

1. If $\Phi(P) = \varnothing \wedge \Phi(Q) = \varnothing$, output **equivalent**.
2. Verify that:
$$FV(\Phi(P)) = FV(\Phi(Q))$$
If not, output **not equivalent**.
3. **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:
$$\exists \overline{v}.\Phi(P)[\overline{v}/FV(\Phi(P))] \neq \Phi(Q)[\overline{v}/S(FV(\Phi(Q)))]$$
   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

4.1 Verifying Equivalence of SparkLite Programs without Aggregations

Program equivalence problem formalization in TS semantics. Given two programs P, Q receiving as input a series of RDDs $\overline{r} = (r_1, \ldots, r_n)$. 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 $\forall k \in \{1, \ldots, n\}.S(\mathbf{x}^{(k)}_{r_{j_k}}) = \mathbf{x}^{(i)}_{r_{j_i}} \wedge j_k = j_i$

The PE problem becomes the problem of proving the following:

$$\begin{array}{ll} (*) & FV(\Phi(P)) \simeq^{\mathcal{S}} FV(\Phi(Q)) \\ (**) & \forall \overline{x}.\ \Phi(P)[\overline{x}/FV(\Phi(P))] = \Phi(Q)[\overline{x}/\mathcal{S}(FV(\Phi(P)))] \end{array}$$

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

▸ **Lemma 1** (Decidability for programs without aggregations). *Given two* SparkLite *programs P and Q which do not contain aggregate operations,* 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 the Augmented Presburger Arithmetic. For RDDs we provide an algorithm in Figure 6, which is a decision procedure. The correctness of the algorithm follows from the equivalence of the TS semantics and the operational semantics defined in 2.4. The algorithm generates an equivalence formula from the program terms of P and Q, which is a formula in the Augmented Presburger Arithmetic. Thus, its decidability again follows from Proposition 1. ◂

Examples. **Note:** 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. Finding the isomorphism $\mathcal{S}$ between variable names is done automatically in all examples, 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, $\Phi(P) = p_1(\phi_P(r^{out}))$.

▸ 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.

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

Non trivial programs. Both programs may return an non-empty RDD of integers.

Free variables: $FV(\Phi(P1)) = \{\mathbf{x}_R\} = FV(\Phi(P2))$. The sets of free variables are equal.

Analysis of representative elements:

$$\phi_{P1}(R') = 2 * \mathbf{x}_R\,; \qquad \phi_{P1}(r^{out}) = ite(\varphi < 14 \wedge \varphi = \phi_{P1}(R'), \varphi, \perp) = ite(2 * \mathbf{x}_R < 14, 2 * \mathbf{x}_R, \perp)$$
$$\phi_{P1}(R') = ite(\mathbf{x}_R < 7, \mathbf{x}_R, \perp)\,; \quad \phi_{P2}(r^{out}) = (\lambda x.x + x)(\phi_{P1}(R')) = ite(\mathbf{x}_R < 7, \mathbf{x}_R, \perp) + ite(\mathbf{x}_R < 7, \mathbf{x}_R, \perp)$$

We need to verify that:

$$\forall \mathbf{x}_R.ite(2 * \mathbf{x}_R < 14, 2 * \mathbf{x}_R, \perp) = ite(\mathbf{x}_R < 7, \mathbf{x}_R, \perp) + ite(\mathbf{x}_R < 7, \mathbf{x}_R, \perp)$$

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 when the condition holds, 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 cased expressions to Presburger arithemtic when appearing in an equivalence formula:*

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

$$f(ite(cond, e, \perp)) = ite(cond, f(e), \perp)$$

2. *Equivalence of functions of conditionals:*

$$\Big(f(ite(cond, e, \perp)) = g(ite(cond', e', \perp))\Big) \iff \Big((cond \iff cond') \wedge (cond \implies f(e) = g(e'))\Big)$$

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

$$\Big(f(ite(cond, e, \perp)) = e'\Big) \iff \Big(cond \wedge f(e) = e'\Big)$$

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

$$f(ite(cond, e, \perp), ite(cond', e', \perp)) = ite(cond \wedge cond', f(e, e'), \perp)$$

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

$$f(ite(cond, e, \perp), e') = ite(cond, f(e, e'), \perp)$$

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.

6. *Unnesting of nested conditionals*

$$ite(c_{ext}, ite(c_{int}, e, \bot), \bot) = ite(c_{int} \wedge c_{ext}, e, \bot)$$

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 **??**.

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. As mentioned, for programs returning a non-RDD value, $TS(P) = \Phi(P)$. In particular, it may do so by applying the *fold* operation on one or more RDDs. 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.

4.2.1 Single aggregate

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

▸ Example 2 (Maximum and minimum). Below is an example of two equivalent programs representing the simplest class of programs with aggregation:

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

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

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. The equivalence formula is:

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

The two program apply a fold operation on two RDDs of equal size. We use an inductive claim to prove the equivalence:

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

The inductive claim is a formula in the Augmented Presburger Arithmetic. We assume $A = -A'$ and attempt to prove:

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

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

4.2.1.1 Soundness

We present a generalization of the inductive claim for the class of programs discussed here, of programs performing a single aggregate operation after a series of *map*, *filter* and *cartesian* operations. Those definitions lead to the following lemma:

▸ **Lemma 2** (Sound method for verifying equivalence of aggregate terms). *Let* $R_0 \in RDD_{\sigma_0}, R_1 \in RDD_{\sigma_1}$*, and denote their representative elements* φ_0, φ_1 *respectively. We assume* φ_0, φ_1 *were composed from* map *,* filter *and cartesian product (without self products*[2]*). Let there be two* fold *functions* $f_0 : \xi_0 \times \sigma_0 \to \xi_0, f_1 : \xi_1 \times \sigma_1 \to \xi_1$*, two* initial *values* $init_0{:}\xi_0, init_1{:}\xi_1$*, and two functions* $g : \xi_0 \to \xi, g' : \xi_1 \to \xi$*. We have: if*

$$FV(\varphi_0) \simeq FV(\varphi_1), \textit{ denoted } FV \tag{1}$$

$$g(init_0) = g'(init_1) \tag{2}$$

$$\forall \overline{v}, A_{\varphi_0} : \xi_0, A_{\varphi_1} : \xi_1 . g(A_{\varphi_0}) = g'(A_{\varphi_1}) \implies g(f_0(A_{\varphi_0}, \varphi_0[\overline{v}/FV])) = g'(f_1(A_{\varphi_1}, \varphi_1[\overline{v}/FV])) \tag{3}$$

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

Lemma 2 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.

Additional Examples. Refer to appendix **??**.

4.2.1.2 Completeness

There are several cases in which one or more of the requirements of Lemma 2 are not satisfied, yet the aggregate expressions are equal.

- The first requirement, $FV(\varphi_0) \simeq FV(\varphi_1)$, is not necessary when the fold functions applied on the terms are trivial. Suppose:

  $$\forall \overline{v}. f_0(init_0, \varphi_0[\overline{v}/FV(\varphi_0)]) = init_0 \wedge \forall \overline{v'}. f_1(init_1, \varphi_1[\overline{v'}/FV(\varphi_1)]) = init_1$$

  Then:

  $$[\varphi_0]_{init_0, f_0} = init_0 \wedge [\varphi_1]_{init_1, f_1} = init_1$$

  And by Equation (2), equivalence follows. Conversely, when the fold is not trivial, the proof of lemma 2 requires 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. Assuming one set of free variables may be embedded in the other (the embedding is an isomorphism on the smaller set), any non-constant result of the fold function on these additional elements will lead to inequal results. To avoid such peculiarities, we shall consider only classes of programs that satisfy Equation (1).
- If the requirement of Equation (2) is not satisfied, yet the aggregates are equivalent, i.e.

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

[2] Self-cartesian products require multiple induction steps. For example, for $R \times R$, the existence of an element (x, y) such that $x \neq y$, implies the existence of an element (y, x). Thus, the number of steps applied each time is determined according to the number of symmetric elements of a certain valuation. Due to lack of space, we omit this discussion from the paper.

then we can get a contradiction by choosing all input RDDs to be empty. Thus, $[\varphi_0]_{init_0,f_0} = init_0 \wedge [\varphi_1]_{init_1,f_1} = init_1 \implies g(init_0) = g'(init_1)$, contradiction. The conclusion is that Equation (2) is a necessary condition for equivalence.

- Suppose $g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1})$ and Equation (3) is not satisfied, that is:

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

Intuitively, as the aggregate expressions are equal, if we could trim the RDDs to the elements corresponding to the valuation $\overline{v}$, and to the elements corresponding to the valuations that generated $A_{\varphi_0}, A_{\varphi_1}$, then we would get a contradiction. We shall formulate this intuition in the next paragraphs.

We proceed with an example 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 2 fails to prove the equivalence of the following two programs.

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

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

$$[x \bmod 3]_{0,+ \bmod 3} = 0 \Leftrightarrow [x]_{0,+ \bmod 3} \bmod 3 = 0$$
$$\forall x, A, A'.A = 0 \iff A' \bmod 3 = 0 \implies (A + x \bmod 3) \bmod 3 = 0 \iff (A' + x) \bmod 3 = 0$$

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

From the above example we derive the first completeness criterion:

▸ **Lemma 3** (Completeness criterion for programs with *map*, *filter* and a single aggregate expression). *Under the premises of Lemma 2, if we have:*

$$FV(\varphi_0) \simeq FV(\varphi_1), \text{ denoted } FV \tag{1}$$
$$g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1})) \tag{2}$$
$$g \text{ or } g' \text{ are injective} \tag{3}$$
$$\forall A.\exists \overline{a_1}, \ldots, \overline{a_n}.A = f_1(f_1(\ldots(f_1(init_1, \varphi_1[\overline{a_1}/FV]), \varphi_1[\overline{a_2}/FV]), \ldots), \varphi_1[\overline{a_n}/FV]) \tag{4}$$

then Equation (3) *in Lemma 2 is satisfied.*

Proof. Suppose w.l.o.g g is injective. Then by Equation (2):

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

We have already shown $init_0 = g^{-1}(g'(init_1))$. The class of programs we consider is restricted to *map* and *filter* only, thus FV is of size 1. Assume:

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

According to 3, let the sequence of valuations $\langle \overline{a_1}, \ldots, \overline{a_n} \rangle$ be the generators of the intermediate value A_{φ_1}. We take RDDs R_0, R_1 defined as follows for $j \in \{0,1\}$:

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

For this choice of RDDs we have:

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

But, from the assumption:

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

we get :

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

contradiction. ◂

▸ **Lemma 4** (Extension of completeness criterion in the presence of *cartesian*). *Under the premises of Lemma 2, if we have:*

$$FV(\varphi_0) \simeq FV(\varphi_1), \text{ denoted } FV \tag{1}$$
$$g([\varphi_0]_{init_0, f_0}) = g'([\varphi_1]_{init_1, f_1}) \tag{2}$$
$$g \text{ or } g' \text{ are injective} \tag{3}$$
$$\forall A, \overline{v}. \exists \overline{a_1}, \ldots, \overline{a_n}. \tag{4}$$
$$A = f_1(f_1(\ldots(f_1(init_1, \varphi_1[\overline{a_1}/FV]), \varphi_1[\overline{a_2}/FV]), \ldots), \varphi_1[\overline{a_n}/FV])$$
$$\wedge \exists V_1, \ldots, V_{|FV|}. \{\overline{a_1}, \ldots, \overline{a_n}, \overline{v}\} = V_1 \times \cdots V_{|FV|}$$

then Equation (3) *in Lemma 2 is satisfied.*

▸ **Lemma 5** (Completeness criterion for non-injective mappings). *Under the premises of Lemma 2, if we have:*

$$FV(\varphi_0) \simeq FV(\varphi_1), \text{ denoted } FV \tag{1}$$
$$g([\varphi_0]_{init_0, f_0}) = g'([\varphi_1]_{init_1, f_1}) \tag{2}$$
$$\forall A, A', \overline{v}. \exists \overline{a_1}, \ldots, \overline{a_n}. \tag{3}$$
$$A = f_0(f_0(\ldots(f_0(init_0, \varphi_0[\overline{a_1}/FV]), \varphi_0[\overline{a_2}/FV]), \ldots), \varphi_0[\overline{a_n}/FV])$$
$$A' = f_1(f_1(\ldots(f_1(init_1, \varphi_1[\overline{a_1}/FV]), \varphi_1[\overline{a_2}/FV]), \ldots), \varphi_1[\overline{a_n}/FV])$$
$$\wedge \exists V_1, \ldots, V_{|FV|}. \{\overline{a_1}, \ldots, \overline{a_n}, \overline{v}\} = V_1 \times \cdots V_{|FV|}$$

then Equation (3) *in Lemma 2 is satisfied.*

▸ **Lemma 6** (Lifting to programs with terms dependent on an aggregation). *Let there be two* SparkLite *programs P_1, P_2 containing two aggregated expressions $a_i = [\varphi_i]_{init_i, f_i}$ for $i \in \{1,2\}$. Let the program terms of P_1, P_2 be $\psi_j(a_j)$ with free variables $FV(\psi_j)$. P_1 is equivalent to P_2 if:*

$$FV(\varphi_1) \simeq FV(\varphi_2), \text{ denoted } FV_\varphi \tag{1}$$
$$FV(\psi_1) \simeq FV(\psi_2), \text{ denoted } FV_\psi \tag{2}$$
$$\forall \overline{x}. \psi_1(init_1)[\overline{x}/FV_\psi] = \psi_2(init_2)[\overline{x}/FV_\psi] \tag{3}$$
$$\forall \overline{v}, A_1, A_2. (\forall \overline{x}. \psi_1(A_1)[\overline{x}/FV_\psi] = \psi_2(A_2)[\overline{x}/FV_\psi]) \implies \tag{4}$$
$$(\forall \overline{x}. \psi_1(f_1(A_1, \varphi_1[\overline{v}/FV_\varphi]))[\overline{x}/FV_\psi] = \psi_2(f_2(A_2, \varphi_2[\overline{v}/FV_\varphi]))[\overline{x}/FV_\psi]) \tag{5}$$

Lemmas 3,4,5,6 show classes of programs in which the inductive argument can be used to prove equivalence of the aggregate expressions[3].

4.2.2 A class for which *PE* is undecidable

must prove multi independent folds which is referenced here!

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 7 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 φ:

- $x_i \longrightarrow^{\varphi} [\texttt{map}(\lambda x.1)(R_i)]_{0,+}$
- $x_i x_j \longrightarrow^{\varphi} [\texttt{cartesian}(\texttt{map}(\lambda x.1)(R_i), \texttt{map}(\lambda x.1)(R_j))]_{0,+}$
- By induction, $x_i^2 \longrightarrow^{\varphi} [\texttt{cartesian}(\texttt{map}(\lambda x.1)(R_i), \texttt{map}(\lambda x.1)(R_i))]_{0,+}$.
  This rule as well as the rest of the powers follow according to the previous rule. For a degree k monom, we apply the *cartesian* operation k times.
- ${x_i}^0 \longrightarrow^{\varphi} 1$, trivially.
- $am(x) \longrightarrow^{\varphi} a\varphi(m(x))$ where $m(x)$ is a monomial with coefficient 1 of the variable x, thus we have already defined φ for it.
- $aq(x_{i_1}, \dots, x_{i_j}) \longrightarrow^{\varphi} a\varphi(q(x_{i_1}, \dots, x_{i_j}))$ where $q(x)$ is a monomial with coefficient 1 and multiple variables for which φ 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$(R_1, \dots, R_k \colon RDD_{\texttt{Int}})$:	P2$(R_1, \dots, R_k \colon RDD_{\texttt{Int}})$:
1	`return` $\varphi(p) \neq 0$	`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 root, and if it outputs 'not equivalent' then the polynomial p has some root, where $x_i = |[\![R_i]\!]|$. Thus we have polynomial reduction to Hilbert's 10^{th} problem.

4.2.3 Multiple aggregates

4.2.3.1 Independent multiple aggregations.

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.

[3] Even when none of the completeness criteria are met, we may be successful in proving the equivalence using Lemma 2. Furthermore, even if the inductive claim is false, by finding the violating intermediate value (or values) and proving that it does not have a generating sequence of valuations with the given fold function and program term, it is possible for the algorithm to propose a stronger inductive claim and attempt to prove it instead.

▸ **Lemma 7.** *Let $\overline{R_i} \in \overline{RDD_{\sigma_i}}, \overline{R'_j} \in \overline{RDD_{\sigma'_j}}$, and denote the representative elements $\overline{\varphi_i}, \overline{\varphi'_j}$. We assume the terms are based only on* map *,* filter *and* cartesian *without self-products. Let there be* fold *UDFs $\overline{f_i : \xi_i \times \sigma_i \to \xi_i}, \overline{f'_j : \xi'_j \times \sigma'_j \to \xi'_j}$, and initial values $\overline{init_i{:}\,\xi_i}, \overline{init'_j{:}\,\xi'_j}$. Let there be 2 functions $g{:}\,\overline{\xi_i} \to \xi, g'{:}\,\overline{\xi'_j} \to \xi$. We have: if*

$$\bigcup_i FV(\overline{\varphi_i}) \simeq \bigcup_j FV(\overline{\varphi'_j}), \text{ denoted } FV \tag{1}$$

$$g(\overline{init_i}) = g'(\overline{init'_j}) \tag{2}$$

$$\forall \overline{v}, \overline{A_{\varphi_i}}, \overline{A_{\varphi'_j}}.g(\overline{A_{\varphi_i}}) = g'(\overline{A_{\varphi'_j}}) \implies g(\overline{f_1(A_{\varphi_i}, \varphi_i[\overline{v}/FV])}) = g'(\overline{f'_1(A_{\varphi'_j}, \varphi'_j[\overline{v}/FV]}) \tag{3}$$

then $g(\overline{[\varphi_i]_{init_i, f_i}}) = g'(\overline{[\varphi'_j]_{init'_j, f'_j}})$

▸ 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 7, we are able to show the equivalence.

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

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

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

We let $g = \lambda(x,y).(x,y)$ and $g' = \lambda(x,y).(x,-y)$. Induction base case is trivial. Induction step:

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

Substituting for A with B, C, we get that we need to prove:

$$ite(x \geq 0, (B+x, C), (B, C-x)) \quad =^? \quad (B + ite(x \geq 0, x, 0), C + ite(x < 0, -x, 0))$$

which is a formula in the Augmented Presburger Arithmetic, whose validity is decidable. ◼

4.2.3.2 Nested aggregations.

In the following subsection we present more complex SparkLite programs, in which the value of an aggregate operation is 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: $RDD_{\texttt{Int}}$):	P2(R: $RDD_{\texttt{Int}}$):
$R' = \texttt{map}(\lambda x.(x,2))(R)$	$R' = \texttt{map}(\lambda x.(x,1))(R)$
$sz = \texttt{fold}(0,f)(R')$	$sz = \texttt{fold}(0,f)(R')$
$B = \texttt{filter}(\lambda x.x > sz)(R)$	$B = \texttt{filter}(\lambda x.x > 2 * sz)(R)$
$\texttt{return fold}(0,+)(B)$	$\texttt{return fold}(0,+)(B)$

The equivalence condition is:

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

Replacing sz, sz' we get:

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

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

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

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

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

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

Proving the equivalence. ∎

This property is reflected in the following proposition:

▸ **Lemma 8.** *Let there be two* SparkLite *programs P_1, P_2 containing two aggregated expressions $a_i = [\varphi_i]_{init_i, f_i}$ for $i \in \{1,2\}$. Let P_1, P_2 contain return aggregated expressions, the terms of which depend on the previous aggregations a_i: $h_j([\psi_j(a_j)]_{init'_j, g_j})$. If:*

$$FV(\psi_1) \simeq FV(\psi_2), \text{ denoted } FV_\psi \tag{1}$$

$$h_1(init_1) = h_2(init_2) \tag{2}$$

$$\begin{aligned}&\forall \overline{v}, A_1, A_2.h_1(A_1) = h_2(A_2) \implies \\ &\qquad h_1(g_1(A_1, \psi_1(a_1)[\overline{v}/FV_\psi])) = h_2(g_2(A_2, \psi_2(a_2)[\overline{v}/FV_\psi]))\end{aligned} \tag{3}$$

Note that Lemma 8 is subject to all previously defined completeness criteria, in conjunction with the completeness criteria applied to the equivalence formula in the induction step. 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.

Several examples are given in appendix **??**.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

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. URL: `http://doi.acm.org/10.1145/362384.362685`, `doi:10.1145/362384.362685`.

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

5 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. URL: `http://doi.acm.org/10.1145/115372.115320`, `doi:10.1145/115372.115320`.

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

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 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`.

9 M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

10 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.

11 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

A Extending Cooper's Algorithm to the Augmented Presburger Arithmetic

Proposition 4. *The theory of formulas over $\mathbb{Z}^n$ with terms in the Augmented Presburger Arithmetic is decidable.*

Proof. Let φ be a quantified formula over $\bigcup_n \mathbb{Z}^n$ with terms in the Augmented Presburger Arithmetic. We shall translate φ to a formula in the Presburger Arithmetic. For any atom $A := a = b$, and $a, b \in \mathbb{Z}^k$ 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 $i \in \{1, \dots, k\}$. Variable quantificaion extends naturally, i.e. $\forall a$ becomes $\forall x_{a,1}, \dots, x_{a,k}$, 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 7 Typing rules for SparkLite

C Proof of the equivalence of the Operational Semantics and TS(P) semantics for SparkLite

Without loss of generality, we do not consider programs with non-RDD inputs, as these are handled in the same manner in both semantics. The proof follows by structural induction on the available operations in a SparkLite program P (defined by the syntactic term E). For brevity, we also assume w.l.o.g. that the RDDs given as arguments to the operations are input RDDs. Reminder:

$$TS(P)(\overline{r}) = \{\!\{ \Phi(P)[\overline{x}/FV(\Phi(P))] \mid \Phi(P)[\overline{x}/FV(\Phi(P))] \neq \bot \wedge \forall k \in \{1, \ldots, n_P\}. x_k \in r_{j_k} \}\!\}$$

- Input RDD : Suppose $E = r, r \in \overline{r}$.

$$TS(P)(\overline{r}) = \{\!\{ \mathbf{x}_r[x/\mathbf{x}_r] \mid \mathbf{x}_r[x/\mathbf{x}_r] \neq \bot \wedge x \in r \}\!\} = \{\!\{ x \mid x \neq \bot \wedge x \in r \}\!\} = \{\!\{ x \mid x \in r \}\!\} = [\![P]\!](\overline{r})$$

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

$$\begin{aligned} TS(P)(\overline{r}) &= \{\!\{\Phi(P)[x/\mathbf{x}_r] \mid \Phi(P)[x/\mathbf{x}_r] \neq \bot \wedge x \in r\}\!\} \\ &= \{\!\{f(\mathbf{x}_r)[x/\mathbf{x}_r] \mid f[x/\mathbf{x}_r] \neq \bot \wedge x \in r\}\!\} \\ &= \{\!\{f(x) \mid f(x) \neq \bot \wedge x \in r\}\!\} = [\![\texttt{map}]\!](f)(r) \end{aligned}$$

- *filter*: Suppose $E = \texttt{filter}(f)(r)$, $r \in \overline{r}$.

$$\begin{aligned} TS(P)(r) &= \{\!\{\Phi(P)[x/\mathbf{x}_r] \mid \Phi(P)[x/\mathbf{x}_r] \neq \bot \wedge x \in r\}\!\} \\ &= \{\!\{ite(f(\mathbf{x}_r) = tt, \mathbf{x}_r, \bot)[x/\mathbf{x}_r] \mid ite(f(\mathbf{x}_r) = tt, \mathbf{x}_r, \bot)[x/\mathbf{x}_r] \neq \bot \wedge x \in r\}\!\} \\ &= \{\!\{ite(f(x) = tt, x, \bot) \mid ite(f(x) = tt, x, \bot) \neq \bot \wedge x \in r\}\!\} \\ &= \{\!\{x \mid f(x) = tt \wedge x \neq \bot \wedge x \in r\}\!\} \\ &= \{\!\{x \mid f(x) = tt \wedge x \in r\}\!\} = [\![\texttt{filter}]\!](f)(r) \end{aligned}$$

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

$$\begin{aligned} TS(P)(r_1, r_2) &= \{\!\{\Phi(P)[x_1/\mathbf{x}^{(1)}_{r_1}, x_2/\mathbf{x}^{(2)}_{r_2}] \mid \Phi(P)[x_1/\mathbf{x}^{(1)}_{r_1}, x_2/\mathbf{x}^{(2)}_{r_2}] \neq \bot \wedge x_1 \in r_1 \wedge x_2 \in r_2\}\!\} \\ &= \{\!\{(\mathbf{x}^{(1)}_{r_1}, \mathbf{x}^{(2)}_{r_1})[x_1/\mathbf{x}^{(1)}_{r_1}, x_2/\mathbf{x}^{(2)}_{r_2}] \mid (\mathbf{x}^{(1)}_{r_1}, \mathbf{x}^{(2)}_{r_1})[x_1/\mathbf{x}^{(1)}_{r_1}, x_2/\mathbf{x}^{(2)}_{r_2}] \neq \bot \wedge x_1 \in r_1 \wedge x_2 \in r_2\}\!\} \\ &= \{\!\{(x_1, x_2) \mid (x_1, x_2) \neq \bot \wedge x_1 \in r_1 \wedge x_2 \in r_2\}\!\} \\ &= \{\!\{(x_1, x_2) \mid x_1 \in r_1 \wedge x_2 \in r_2\}\!\} = [\![\texttt{cartesian}]\!](r_1, r_2) \end{aligned}$$

- *fold*: Suppose $E = \texttt{fold}(e, f)(r)$, $r \in \overline{r}$. We have:

$$\begin{aligned} TS(P)(\overline{r}) &=^{def} [\mathbf{x}_r]_{e,f} \\ &=^{def} [\![\texttt{fold}]\!](e, f)(r) \end{aligned}$$

Equivalence is by definition.