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, 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 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 [5], 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 $((e\ ,\ e) \mid \boldsymbol{p}_i(e))$, the resulting first order theory with the usual $\forall, \exists$ quantifiers is called the *Presburger Arithmetic*,


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


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

Figure 1 Terms of the Augmented Presburger Arithmetic

and it is decidable [1]. We shall name the theory including the tuple expressions as the *Augmented Presburger Arithmetic*. *Cooper's Algorithm* [2] is the decision procedure for the standard Presburger Arithmetic[1]. 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*) [6].

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 integers and boolean primitive types as *integers* ($\mathbb{Z}$) and *booleans* ($\mathbb{B}$), respectively. 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 $\mathcal{T} = \bigcup_n \mathrm{T}^n$.

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

Interpretation operator We use $[\![\cdot]\!]$ (semantic brackets) to denote the mathematical interpretation of an expression. We shall see in the next subsections that it is 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.

[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 [4]. 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*) [] 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 []. For example: $sumMod10\colon \texttt{Int} \times \texttt{Int} \to \texttt{Int}$ 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: $sumMod10 = \lambda x\colon \texttt{Int}, y\colon \texttt{Int}.x\%10 + y\%10\colon \texttt{Int}$. Note that $sumMod10$ has two arguments, but we could also write it as a function with single argument having the following signature: $sumPairMod10\colon (\texttt{Int} \times \texttt{Int}) \to \texttt{Int}$, 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 λ. For example, we could define a function that adds 1 to an integer: $\boldsymbol{f} = \lambda x\colon \texttt{Int}.x + 1$, but we could also make it more generic and flexible by writing $\boldsymbol{g} = \lambda a\colon \texttt{Int}.\lambda x\colon \texttt{Int}.x + a$. This is an example of a *parametric function*. Parametric functions can be transformed to regular functions by beta-reducing the first lambda abstraction: $\boldsymbol{g}(1)$ which is identical to $\boldsymbol{f}$.

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 $\boldsymbol{v}, \boldsymbol{b}, \boldsymbol{r} \in \texttt{Vars}$. Expressions range over $\boldsymbol{e}$. 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 $\boldsymbol{f}, \boldsymbol{F} \in \texttt{LambdaExpressions}$. 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.

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: $\boldsymbol{P}(\overline{\mathcal{T}_i}, \overline{RDD_i})\colon \mathcal{T}_o$

Let: $isOdd = \lambda x\colon \texttt{Int}.\neg x\%2 = 0$
$doubleAndAdd = \lambda c\colon \texttt{Int}.\lambda x\colon \texttt{Int}.2 * x + c$
$sumFlatPair = \lambda A\colon \texttt{Int}, (x, y)\colon \texttt{Int} \times \texttt{Int}.A + x + y$

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

Figure 2 Example SparkLite program

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 ($\boldsymbol{P} = P1$) 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* ..., 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 $P1(RDD_{\texttt{Int}}, RDD_{\texttt{Int}})\colon \texttt{Int}$.

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:

$$\mathcal{E} = \texttt{Vars} \to \mathcal{D}$$

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 The *environment function* $\rho \in \mathcal{E}$ denotes the environment of the program. The initial environment function ρ_0 maps all input variables and function definitions: For $x \in \overline{\boldsymbol{r}} \cup \overline{\boldsymbol{v}} \cup \overline{Fdef} \cup \overline{PFdef}$, $\rho_0(x) = [\![x]\!]$, where $[\![\cdot]\!]$ is used to express the interpretation of the input in the semantic domain. We denote $[\![\boldsymbol{v}]\!](\rho) = \rho(\boldsymbol{v})$ as the *interpreted value* of the variable $\boldsymbol{v}$ of type $\mathcal{D}$. The semantics of expressions are straight-forward and we provide the semantics with the current environment ρ (using $[\![\cdot]\!](\cdot)$). In Figure 4 we specify the behavior of $[\![\cdot]\!](\cdot)$ on the body of the program.

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	$Fdef$	::=	$\texttt{def}\ \boldsymbol{f}\ =\ \lambda\overline{\boldsymbol{y}:\tau}\ \ e:\tau$
Parametric Functions	$PFdef$	::=	$\texttt{def}\ \boldsymbol{F}\ =\ \lambda\overline{\boldsymbol{x}:\tau}.\lambda\overline{\boldsymbol{y}:\tau}\ \ e:\tau$
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 3 Syntax for SparkLite

$$
\begin{array}{lcl}
[\![c]\!](\rho) & = & c \\
[\![\boldsymbol{v}]\!](\rho) & = & \rho(v) \\
[\![\texttt{uOp}\,e]\!](\rho) & = & uOp\,[\![e]\!](\rho) \\
[\![e\,\texttt{binOp}\,e]\!](\rho) & = & [\![e]\!](\rho)\,binOp\,[\![e]\!](\rho) \\
[\![\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e]\!](\rho) & = & ite\,([\![e]\!](\rho),[\![e]\!](\rho),[\![e]\!](\rho)) \\
[\![\texttt{map}(f)(r)]\!](\rho) & = & \{\!\{\rho(f)(x) \mid x \in \rho(r)\}\!\} \\
[\![\texttt{filter}(b)(r)]\!](\rho) & = & \{\!\{x \mid x \in \rho(r) \wedge \rho(b)(x)\}\!\} \\
[\![\texttt{cartesian}(r_1,r_2)]\!](\rho) & = & \{\!\{(x_1,x_2) \mid x_1 \in \rho(r_1) \wedge x_2 \in \rho(r_2)\}\!\} \\
[\![\texttt{fold}(a_0,f)(r)]\!](\rho) & = & \begin{cases} \rho(f)([\![\texttt{fold}(a_0,f)(r'),a]\!](\rho) & \rho(r) = \rho(r') \cup \{\!\{a;1\}\!\},\ where\ a \in \rho(r) \\ v & \rho(r) = \bot \end{cases} \\
[\![Let\ \ \boldsymbol{x} = \eta\ in\ E]\!](\rho) & = & [\![E]\!](\rho[\boldsymbol{x} \mapsto [\![\eta]\!](\rho)]) \\
[\![Prog]\!](\rho_0) & = & [\![E]\!](\rho_0)
\end{array}
$$

Figure 4 Semantics of SparkLite. $Prog = \boldsymbol{P}(\overline{\boldsymbol{r}:RDD_\tau},\overline{\boldsymbol{v}:\tau})\ =\ \overline{Fdef}\quad\overline{PFdef}\quad E$. $\texttt{uOp}$, $\texttt{binOp}$, and $\texttt{terOp}$ are taken from Figure 3: $\texttt{uOp} \in \{-, \neg, \boldsymbol{\pi}_i\}$, $\texttt{binOp} \in \{+, *, /, \%, =, <, \wedge, \vee, (,)\}$

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

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

Figure 5 Compiling SparkLite to logical terms (ϕ).

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.

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 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 $\boldsymbol{f}$, the order in which elements from the bag are chosen in the *fold* operation does not affect the result.

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 the Augmented Presburger 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 the previously analyzed example program from Figure 2. 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:

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

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^{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 ($\bot$), for the empty RDD. Therefore $\mathbf{x}_{r^{in}}$ ranges over $\text{dom}(r^{in}) \cup \{\bot\}$. By abuse of notation, the term for a non-input 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: $RDD_{\texttt{Int}}$, 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`$(\lambda x.1)(R_1)$

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 $FV(\Phi(P1)) = \{\mathbf{x}_{R_1}\}$. For programs to be equivalent, there must be an isomorphism between the sets, mapping each free variable to a single variable with the same source input RDD. An exception to the rule of free-variables isomorphism 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 $\overline{r^{in}}$ for the inputs of P, and r^{out} for the output of P. The term $\Phi(P)$ is called the *program term of* P as before. We write the set of free variables of the program term $FV(\Phi(P))$ as a vector: $(\mathbf{x}_{r^{in}_{j_k}})_{k=1}^{n_P}$, where n_P is the number of free variables. A vector of valuations to the free variables is denoted $\overline{x} = (x_1,\ldots,x_{n_P})$ and satisfies $x_k \in r^{in}_{j_k}$ for $k \in \{1,\ldots,n_P\}$. 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:

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

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 $\overline{r^{in}}$, 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* $P : \boldsymbol{P}(\overline{r}) = \overline{\boldsymbol{F}}\,\overline{\boldsymbol{f}}\,E$ *be a* SparkLite *program,* $[\![P]\!]$ *be the interpretation of its output according to the defined semantics, and* $TS(P)$ *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:*

$$TS(P)(\overline{r^{in}}) = [\![P]\!]([\![\overline{r^{in}}]\!])$$

Proof. See appendix **??**. ◂

4 Verifying Equivalence of SparkLite Programs

The Program Equivalence (*PE*) problem Let P_1 and P_2 be SparkLite programs, with signature $P_i(\overline{T}, \overline{RDD_T}) : \tau$ 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 $[\![P_1]\!]([\![\overline{v}]\!], [\![\overline{r}]\!]) = [\![P_2]\!]([\![\overline{v}]\!], [\![\overline{r}]\!])$.

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 $\overline{r^{in}} = (r_1^{in}, \ldots, r_n^{in})$. 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 $\forall k \in \{1, \ldots, n\}. \mathcal{S}(\mathbf{x}^{(k)}_{r^{in}_{j_k}}) = \mathbf{x}^{(i)}_{r^{in}_{j_i}} \wedge j_k = j_i$

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

$$(*) \quad FV(\Phi(P)) \simeq^{\mathcal{S}} FV(\Phi(Q))$$
$$(**) \quad \forall \overline{x}.\ \Phi(P)[\overline{x}/FV(\Phi(P))] = \Phi(Q)[\overline{x}/\mathcal{S}(FV(\Phi(P)))]$$

where the choice of $\overline{r^{in}}$ 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 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 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, $\Phi(P) = p_1(\phi_P(r^{out}))$.

▸ 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.

1. If $\Phi(P) = \bot \wedge \Phi(Q) = \bot$, 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

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

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

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

Analysis of representative elements:
$\phi_{P1}(R') = 2 * \mathbf{x}_R$, and $\phi_{P1}(r^{out}) = \begin{cases} \varphi & \varphi < 14 \wedge \varphi = \phi_{P1}(R') \\ \bot & otherwise \end{cases} = \begin{cases} 2 * \mathbf{x}_R & 2 * \mathbf{x}_R < 14 \\ \bot & otherwise \end{cases}$.
$\phi_{P1}(R') = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases}$, and $\phi_{P2}(r^{out}) = (\lambda x.x + x)(\phi_{P1}(R')) = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases} +$
$\begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases}$.
We need to verify that:
$$\forall \mathbf{x}_R. \begin{cases} 2 * \mathbf{x}_R & 2 * \mathbf{x}_R < 14 \\ \bot & otherwise \end{cases} = \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases} + \begin{cases} \mathbf{x}_R & \mathbf{x}_R < 7 \\ \bot & otherwise \end{cases}$$

To prove this, we need to encode the cased expressions in Presburger arithmetic. Undefined ($\bot$) 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 arithemtic when appearing in an equivalence formula:*

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

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

2. *Equivalence of functions of conditionals:*

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

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

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

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

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

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

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

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.*
7. *Unnseting of nested conditionals*

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

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 TS(P) with aggregate expressions The terms for aggregate operations are given using special operators (Figure 7). We extend the definition of ϕ accordingly in Figure 8. The $[\varphi]_{.,.}$ operator *binds* all variables in φ. The $\langle\varphi\rangle_{.,.}$ operator is syntactic sugar for $(p_1(\varphi), [p_2(\varphi)]_{.,.})$ 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 **??**.

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

$$\begin{array}{lrcl} \textbf{Fold:} & \phi_P([\![\texttt{fold}]\!](f,e)(r),k) & = & ([t]_{e,f},m),\ \textit{where}\ (t,m)=\phi_P(r,k) \\ \textbf{FoldByKey:} & \phi_P([\![\texttt{foldByKey}]\!](f,e)(r),k) & = & (\langle t\rangle_{e,f},m),\ \textit{where}\ (t,m)=\phi_P(r,k) \end{array}$$

Figure 8 Compiling SparkLite to logical terms for aggregate operations.

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

	P1($R : RDD_{\texttt{Int}}$):	P2($R : RDD_{\texttt{Int}}$):
1	`return` $\texttt{fold}(\bot, max)(R)$	$R' = \texttt{map}(\lambda x.0 - x)(R)$
2		`return` $0\text{-}\texttt{fold}(\bot, min)(R')$

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:

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

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

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

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

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 φ_0, φ_1 respectively. We assume φ_0, φ_1 were composed from* map , filter *and cartesian product (without*

self products). Let there be two fold *functions* $f_0 : \xi_0 \times \sigma_0 \to \xi_0, f_1 : \xi_1 \times \sigma_1 \to \xi_1$*, 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*

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

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

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

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

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 FVs, using $\perp$ values if needed. Let there be two equal fold expressions under lemma's 3 premises, and a a valuation $\overline{v}$ in the valuation sequence returning a non-$\perp$ value in R_0, and $\perp$ valuation to R_1. We get:

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

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

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

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

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

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

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:

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

Let the sequence of valuations $\langle \overline{a_1}, \ldots, \overline{a_n} \rangle$ be the generators of the intermediate values $A_{\varphi_0}, A_{\varphi_1}$. We take RDDs R_0, R_1 defined as follows for $j \in \{0, 1\}$:

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

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 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 undecidabile 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 represntative elements based also on the *cartesian* operation. Let there be a polynom p over k variables $x_1, \ldots, x_k$, and coefficients $a_1, \ldots, a_k$. 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 polynom P for some valuation of the x_i. For each summand in the polynom 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 coeffeicient 1 of the variable x, thus we have already defined φ for it.
- $aq(x_{i_1}, \ldots, x_{i_j}) \longrightarrow^{\varphi} a\varphi(q(x_{i_1}, \ldots, x_{i_j}))$ where $q(x)$ is a monomial with coefficient 1 and multiple variables for which φ was defined in the previous rules.
- $\varphi(p(a_1, \ldots, a_k; x_1, \ldots, x_k)) = \sum_{i=1}^{k} \varphi(a_i q(x_{i_1}, \ldots, x_{i_k})))$ follows by structural induction on the previous rules.

We generate the following instance of the PE problem:

	P1($R_1, \ldots, R_k$: $RDD_{\texttt{Int}}$):	P2($R_1, \ldots, R_k$: $RDD_{\texttt{Int}}$):
1	`return` $\varphi(p) \neq 0$	`return` tt

By choosing input RDDs of the cardianlity of R_i equal to the matching variable x_i we can simulate any valuation to the polynom p. If $P1$ returns true, then the valuation is not a root of the polynom p. Thus, if it is equivalent to the 'true program' $P2$, then the polynom p has no roots. Therefore, if the algorithm solving PE outputs 'equivalent' then the polynom p has no root, and if it outputs 'not equivalent' then the polynom p has some root, where $x_i = |[\![R_i]\!]|$. Thus we have polynomial reduction to Hilbert's 10'th 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.

$$\text{Let:}\quad h : (\lambda(P,N),x.\begin{cases}(P+x,N) & x \geq 0\\ (P,N-x) & otherwise\end{cases}$$

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

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

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

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

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

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

$$\begin{cases}N & x \geq 0\\ N-x & otherwise(x<0)\end{cases} \quad =^{?} \quad N' + \begin{cases}-x & x<0\\ \bot & otherwise\end{cases}$$

$$=^{\bot \text{ is neutral}} \quad \begin{cases}N'-x & x<0\\ N' & otherwise\end{cases}$$

$$=^{N=N'} \quad \begin{cases}N-x & x<0\\ N & otherwise\end{cases}$$

$$= \quad \begin{cases}N-x & x<0\\ N & x\geq 0\end{cases}$$

And the equivalence follows. ■

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

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

$$\forall \overline{v}, A_{\varphi_1}{:}\xi_1,\ldots,A_{\varphi_k}{:}\xi_k, A_{\varphi'_1}{:}\xi'_1,\ldots,A_{\varphi'_{k'}}{:}\xi'_{k'}.g(A_{\varphi_1},\ldots,A_{\varphi_k}) = g'(A_{\varphi'_1},\ldots,A_{\varphi'_{k'}}) \implies$$
$$g(f_1(A_{\varphi_1},\varphi_1[\overline{v}/FV(\varphi_1)]),\ldots,f_k(A_{\varphi_k},\varphi_k[\overline{v}/FV(\varphi_k)])) = \tag{2}$$
$$g'(f'_1(A_{\varphi'_1},\varphi'_1[\overline{v}/FV(\varphi'_1)]),\ldots,f'_k(A_{\varphi'_{k'}},\varphi'_{k'}[\overline{v}/FV(\varphi'_{k'})]))$$

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

▸ **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	$R' = \texttt{map}(\lambda x.(x, 2))(R)$	$R' = \texttt{map}(\lambda x.(x, 1))(R)$
2	$sz = \texttt{fold}(0, f)(R')$	$sz = \texttt{fold}(0, f)(R')$
3	$B = \texttt{filter}(\lambda x.x > sz)(R)$	$B = \texttt{filter}(\lambda x.x > 2 * sz)(R)$
3	$\texttt{return fold}(0, +)(B)$	$\texttt{return fold}(0, +)(B)$

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 3 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:

▸ **Proposition 4.** *Let there be two* SparkLite *programs* P_1, P_2 *and returning aggregated expressions* $[a_i(\overline{r})]_{init_i, f_i}$. *Let there be two other* SparkLite *programs,* T_1, T_2, *also returning aggregated expressions,* $[b_i(\overline{r'}, a)]_{init_T, g}$ *(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. $\phi_{P_1} = [a_1(\overline{r})]_{init_1,f_1}, \phi_{P_2} = [a_2(\overline{r})]_{init_2,f_2}$ are the program terms of P_1, P_2, respectively. a, a' are terms without aggregations. Let $[b_i(\overline{r'}, \phi_{P_i})]_{init_T,g}$ 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 $b_1(\overline{r'}, [a_1(\overline{r})]_{init_1,f_1}))$ and $b_2(\overline{r'}, [a_2(\overline{r})]_{init_2,f_2})$ 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:

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

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: $f_i([\phi_i(\overline{x}, a_i(\overline{y}))]_{init_i,g_i})$, The* PE *problem is decidable if exists $i \in \{1,2\}$ such that the following expressions are all injective:*

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

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

$$f_1(init_1) = f_2(init_2) \tag{1}$$

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

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

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

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

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 $[a_i(\overline{r})]_{init_i,f_i}$. Let there be two other* SparkLite *programs, T_1, T_2, also returning aggregated expressions, $[b_i(\overline{r'}, a)]_{init_{T_i},g_i}$.* PE *is decidable for $T \circ P, T' \circ P'$ if and only if* PE *is decidable for P, P'.*

Proof. $\phi_{P_1} = [a_1(\overline{r})]_{init_1,f_1}, \phi_{P_2} = [a_2(\overline{r})]_{init_2,f_2}$ are the program terms of P_1, P_2, respectively. a, a' are terms without aggregations. Let $[b_i(\overline{r'}, \phi_{P_i})]_{init_{T_i},g_i}$ 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:

$$g_1(init_{T_1}) = g_2(init_{T_2}) \tag{1}$$

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

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`.

5 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*, 1929.

6 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.