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

We present a novel approach for verifying the equivalence of Spark programs. Spark is a popular framework for writing large scale data processing applications. Such frameworks, intended for data-intensive operations, share many similarities with traditional database systems, but do not enjoy a similar support of optimization tools. Our goal is to enable such optimizations by first providing the necessary theoretical setting for verifying the equivalence of Spark programs. This is challenging because such programs combine relational algebraic operations with *User Defined Functions* (*UDF*s). We define a model of Spark as a programming language which imitates Relational Algebra queries in the bag semantics and allows for user defined functions expressible in Presburger Arithmetics. We present a sound and complete reasoning technique for verifying the equivalence of an interesting class of Spark programs.

1 Introduction

The rise of cloud computing and Big Data in the last decade allowed the advant of new programming models, with the intention of simplifying the development process for large-scale needs, letting the programmer focus on business-logic and separating it from the technical details of data management over computer clusters, distribution, communication and parallelization. The first model was MapReduce [12]. It allowed programmers to define their logic by composing several iterations of map and reduce operations, where the programmer provided the required map and reduce functions in each step, and the framework was responsible for facilitating the dataflow to the provided functions. MapReduce gave a powerful, yet a clean and abstract programming model. Later on, other frameworks were developed, allowing programmers to write procedural code while reliving the programmer from the need to handle distribution and error recovery. One such framework is Apache Spark [22], in which programmers keep writing code in their programming language of choice, (e.g., Scala [3], Java [1], Python [2], or R [17]) but utilize a special object provided by Spark, called *resilient distributed dataset* (*RDD*), providing access to the distributed data itself and to perform transformations on it, using the cloud resources for actual computing. The architecture of Spark comprises of a single master node, referred to as the *driver*, and *worker* nodes in a clustered computer environment. All the nodes have access to the program code, but the driver orchestrates its execution using the underlying Spark framework, abstracting away communications, error recovery, distribution, data partitioning, and parallelization. The access to the data is via the *RDD* API. An *RDD* can be thought of as a simple database table, but which provides support for *User Defined Functions* (*UDF*), greatly increasing its expressive power. Spark programs handle a family of common tasks involving large datasets, such as log parsing, database queries (via *SparkSQL* [5]), training algorithms and different numeric computations in various fields. Due to this, many Spark programs share several properties: they are mostly short and relatively simple to read and understand. We believe that thanks to this nature of Spark programs, the problem of verifying a program's properties, or even program equivalence as we focus on in this paper, may become feasible, even decidable in an interesting class of Spark programs.

Main Results. The main contributions of this paper can be summarized as follows:

1. We present a simplified model of Spark using a simple programming language called SparkLite, in which UDFs are expressed as simply typed λ-calculus restricted to Presburger Arithmetics. The operations on *RDD*s correspond to operations in relational algebra, with additional aggregate operations expressions, augmented with the added expressivity provided by UDFs.
2. We define an interesting subclass of SparkLite in which checking program equivalence is decidable, called Agg^1_{sync}. Interestingly, programs in Agg^1_{sync} go beyond relational algebra by allowing UDFs and aggregates. The decidability of the equivalence is proven by observing that common SparkLite aggregates are 'closed' in the sense that composed operations could be simulated by a single operation.
3. We show that the equivalence of arbitrary SparkLite programs is undecidable.
4. We also define a method for soundly checking the equivalence of arbitrary SparkLite programs.

Should be reconstructed

Motivation. The natural connection between frameworks such as Spark and the traditional relational databases leads to a broader definition of the Query Equivalence problem in databases. The new frameworks allow running queries which are similar to relational algebra, yet at the same time are more expressive due to UDFs and aggregations. In addition, queries are written in the form of programs. We establish this natural connection formally by presenting two semantics for Spark programs, one which is operational (Section 3) and one which is denotational (Section 4), based on a translation of the program to a symbolic representation. The verification technique applied on the symbolic representation readily reproduces classical results on equivalence of conjunctive queries (Section 5), but mainly, provides a new direction for the analysis of queries, and in particular for query equivalence. One such direction is analyzing aggregates, which are also expressed using UDFs.[1] For programs containing aggregates, we provide a syntactic classification of programs, with sound equivalence testing procedures. In addition, a semantic, verifiable property of programs yields a class (Agg^1_{sync}) for which we present a sound and complete equivalence testing procedure.

finish

pldi paper on synthesis

The PL angle?

complete

finish

Related Work. Program and Query Equivalence have long been active research interests. Our model of Spark and its connection to Relational Algebra relies on classical results on Query Equivalence by [**?**, 21]. In particular, these results were extended to bag semantics, for example in [7, 9, 10, 14]. Our method of verifying aggregate terms using induction has roots in the analysis of programs inductive invariants, for example in . A similar work on combining logic with aggregation can be seen in [16]. For Spark and other similar frameworks, optimization efforts were focused on system optimization and efficient distribution and parallelization, . To our knowledge, there is no general scheme for writing optimizers for

[1] As opposed to a finite set of aggregate operations common in commercial databases, for example

data-intensive frameworks such as Spark.

Overview. Section 2 provides necessary preliminaries for the rest of the paper. In section 3 we give a complete formalization (syntax and semantics) of SparkLite. In section 4 we describe the term semantics of SparkLite problem. In section 5 we present the framework for equivalence checking. In 5.1 we provide a decision procedure for verifying equivalence of SparkLite programs without aggregate expressions. We continue in Section 5.2, where we discuss SparkLite programs with aggregate expressions: we present a sound equivalence verification technique for two classes (Agg^1, Agg^n) as well as a description of a class of programs, called Agg^1_{sync}, for which the technique can be made complete. In 5.3, we show that the general problem of program equivalence in SparkLite is undecidable. Throughout the paper, missing proofs of non-immediate results can be found in the appendix.

2 Preliminaries

In this section, we describe a simple extension of Presburger arithmetic [20], which is the first-order theory of the natural numbers with addition, to tuples of integers, and state its decidability.

Notations. We denote the sets of natural numbers, positive natural numbers, and integers by $\mathbb{N}$, $\mathbb{N}^+$, and $\mathbb{Z}$, respectively. We denote a (possibly empty) sequence of elements coming from a set X by $\overline{X}$. We write $ite(p, e, e')$ to denote an expression which evaluates to e if p holds and to e' otherwise. We use $\perp$ to denote the *undefined* value. A *bag* m over a domain X is a multiset, i.e., a set which allows for repetitions, with elements taken from X. We denote the *multiplicity* of an element x in bag m by $m(x)$, where for any x, either $0 < m(x)$ or $m(x)$ is undefined. We write $x \in m$ as a shorthand for $0 < m(x)$. We write $\{x; n(x) \mid x \in X \wedge \phi(x)\}$ to denote a bag with elements from X satisfying some property ϕ with multiplicity $n(x)$, and omit the conjunct $x \in X$ if X is clear from context. We denote the *size* (number of elements) of a set X by $|X|$ and that of a bag m of elements coming from X by $|m|$, i.e., $|m| = \Sigma_{x \in X} ite(x \in m, m(x), 0)$.

Presburger Arithmetic. We consider a fragment of first-order logic (FOL) with equality over the integers, where expressions are written in the rather standard syntax specified in Figure 1.[2] Disregarding the tuple expressions ($(pe\,, \overline{pe})$ and $\boldsymbol{p}_i(e)$), the resulting first-order theory with the usual $\forall$ and $\exists$ quantifiers is called the *Presburger Arithmetic*. The problem of checking whether a sentence in Presburger arithmetic is valid has long been known to be decidable [13, 20], even when combined with Boolean logic [6, 18].[3] For example, *Cooper's Algorithm* [11] is a standard decision procedure for Presburger Arithmetic[4]. Check boolean

In this paper, we consider a simple extension to this language by adding a *tuple constructor* $(pe\,, \overline{pe})$, which allows to create k-tuples, for some $1 \leq k$, of primitive expressions, and a projection operator $\boldsymbol{p}_i(e)$, which returns the i-th component of a given tuple expression e. We extend the equality predicate to tuples in a pointwise manner, and call the extended logical

[2] We assume the reader is familiar with FOL, and omit a more formal description for brevity.
[3] Originally, Presburger Arithmetic was defined as a theory over natural numbers. However, its extension to integers and booleans is also decidable. (See, e.g., [6].)
[4] The complexity of Cooper's algorithm is $O(2^{2^{2^{pn}}})$ for some $p > 0$ and where n is the number of symbols in the formula [19]. However, in practice, our experiments show that Cooper's algorithm on non-trivial formulas returns almost instantly, even on commodity hardware.

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

Figure 1 Terms of the Augmented Presburger Arithmetic. c, v, and b denote integer numerals, integer variables, and boolean variables, respectively. % denotes the modulo operator.

First-Order Functions	$Fdef$	::=	$\texttt{def}\ \boldsymbol{f}\ =\ \lambda\overline{\boldsymbol{y}:\boldsymbol{\tau}}.e:\boldsymbol{\tau}$
Second-Order Functions	$PFdef$	::=	$\texttt{def}\ \boldsymbol{F}\ =\ \lambda\overline{\boldsymbol{x}:\boldsymbol{\tau}}.\lambda\overline{\boldsymbol{y}:\boldsymbol{\tau}}.e:\boldsymbol{\tau}$
Function Expressions	f	::=	$\boldsymbol{f} \mid \boldsymbol{F}(\overline{e})$
RDD Expressions	re	::=	$\texttt{cartesian}(\boldsymbol{r},\boldsymbol{r}) \mid \texttt{map}(f)(\boldsymbol{r}) \mid \texttt{filter}(f)(\boldsymbol{r})$
Aggregation Exp.	ge	::=	$\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$
General Expressions	η	::=	$e \mid re \mid ge$
Let expressions	E	::=	$Let\ \boldsymbol{x}\ = \eta\ in\ E \mid \eta$
Programs	$Prog$	::=	$\boldsymbol{P}(\overline{\boldsymbol{r}: RDD_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v}:\boldsymbol{\tau}})\ =\ \overline{Fdef}\quad \overline{PFdef}\quad E$

Figure 2 Syntax for SparkLite. The syntax of basic expressions e is defined in Figure 1.

language *Augmented Presburger Arithmetic*. The decidability of Presburger Arithmetic, as well as Cooper's Algorithm, can be naturally extended to the Augmented Presburger Arithmetic. Intuitively, verifying the equivalence of tuple expressions can be done by verifying the equivalence their corresponding constituents.

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

3 The SparkLite language

In this section, we define the syntax of SparkLite, a simple functional programming language which allows to use Spark's *resilient distributed datasets* (*RDDs*) [22].

3.1 Syntax

The syntax of SparkLite is defined in Figure 2. SparkLite supports two primitive types: *integers* (`Int`) and *booleans* (`Boolean`). On top of this, the user can define *record types* $\boldsymbol{\tau}$, which are Cartesian products of primitive types, and *RDD*s: $RDD_{\boldsymbol{\tau}}$ is (the type of) bags containing elements of type $\boldsymbol{\tau}$. We refer to to primitive types and tuples of primitive types as *basic types*, and, by abuse of notation, range over them using $\boldsymbol{\tau}$. We denote the set of (syntactic) variable names by `Vars`, and range over it using $\boldsymbol{v}$, $\boldsymbol{b}$, and $\boldsymbol{r}$, for variables of type integer, boolean, and record, respectively.

A program $\boldsymbol{P}(\overline{\boldsymbol{r}: RDD_{\boldsymbol{\tau}}}, \overline{\boldsymbol{v}:\boldsymbol{\tau}})\ =\ \overline{Fdef}\quad \overline{PFdef}\quad E$ is comprised of a *header* and a *body*, which are separated by the = sign. The header contains the name of the program ($\boldsymbol{P}$) and the name and types of its input parameters, which may be *RDDs* ($\overline{\boldsymbol{r}}$) or integers ($\overline{\boldsymbol{v}}$). The body of the program is comprised of two sequences of function declarations ($\overline{Fdef}$ and $\overline{PFdef}$) and the program's *main expression* (E). $\overline{Fdef}$ binds function names $\boldsymbol{f}$ with first-order lambda expressions, i.e., to a function which takes as input a sequence of arguments of basic types and return a value of a basic type. For example,

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

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

defines `isOdd` to be a function which determines whether its integer argument `y` is odd or not. $\overline{PFdef}$ associates function names $\boldsymbol{F}$ with a restrict form of second-order lambda expressions, which we refer to as *parametric function*.[5] A parametric function $\boldsymbol{F}$ receives a sequence of basic expressions and returns a first order function. Parametric functions allow to instantiate an unbounded number of functions from a single pattern. For example,

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

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

The program's main expression is comprised of a sequence of *let* expression which bind general expressions to variables. A general expression is either a basic expression (e), an RDD *expression* (re) or an *aggregate expression* (ge). A basic expression is a term in augmented Presburger arithmetics (see Section 2). The *RDD* expression $\texttt{cartesian}(\boldsymbol{r}, \boldsymbol{r}')$ returns the cartesian product, in the bag semantics, of *RDDs* $\boldsymbol{r}$ and $\boldsymbol{r}'$. The *RDD* expressions `map` and `filter` generalize the *Select* and *Project* operators in *Relational Algebra* (*RA*) [4,8], with *user-defined functions* (*UDFs*): $\texttt{map}(f)(\boldsymbol{r})$ evaluates to an *RDD* obtained by applying the UDF f to every element x of *RDD* $\boldsymbol{r}$, with the same multiplicity x had in $\boldsymbol{r}$. $\texttt{filter}(f)(\boldsymbol{r})$ evaluates to a copy of $\boldsymbol{r}$, except that all elements in $\boldsymbol{r}$ which do not satisfy f are removed. The *aggregation expression* `fold` is a generalization of aggregate operations in SQL, e.g., `SUM` or `AVERAGE`, with *UDFs*: $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ accumulate the results obtained by iteratively applying f to every element x in $\boldsymbol{r}$, starting from the *initial element* e and applying f $\boldsymbol{r}(x)$ times for every $x \in \boldsymbol{r}$.

▸ Remark. To ensure that the meaning of $\texttt{fold}(\boldsymbol{e}, f)(\boldsymbol{r})$ is well defined, we requires that f be a commutative function[6]. Also, as is common in functional languages, we assume that variables are never reassigned.

Need to cleanup the example below by rewriting it into the running example that we will use later on.

▸ **Example 1.** Consider the example SparkLite program in Figure 3. From the example program we can see the general structure of SparkLite programs: First, the functions that are

[5] Parametric functions were inspired by the *Kappa Calculus* [15], which contains only first-order functions, but allows lifting them to larger product types, which is exactly the purpose of parametric functions in SparkLite.

[6] Formally, we refer to the following notion of commutativity, unlike the traditional definition: $f : X \times Y$ is commutative if $\forall x, y_1, y_2. f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$

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

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: $\texttt{unOp} \in \{-, \neg, \boldsymbol{\pi}_i\}$, $\texttt{binOp} \in \{+, *, /, \%, =, <, \wedge, \vee, (,)\}$

used as UDFs in the program are declared and defined: $\mathit{isOdd}, \mathit{sumFlatPair}$ defined as Fdef, and $\mathit{doubleAndAdd}$ 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 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 *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 $\mathit{sumFlatPair}$, 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 $P1(\mathit{RDD}_{\texttt{Int}}, \mathit{RDD}_{\texttt{Int}})\colon \texttt{Int}$.

Noam: I am here

3.2 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.

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

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

Figure 5 Compiling SparkLite to logical terms (ϕ).

$$
\begin{array}{lcl}
[\![\mathbf{x}_{r_i}]\!](\overline{v}, \overline{r}) & = & r_i \\
[\![f(t)]\!](\overline{v}, \overline{r}) & = & \{\!\{ [\![f]\!](z) \mid z \in [\![t]\!](\overline{v}, \overline{r}) \}\!\} \\
[\![\mathit{ite}(f(t), t, \bot)]\!](\overline{v}, \overline{r}) & = & \{\!\{ z \mid z \in [\![t]\!](\overline{v}, \overline{r}) \wedge [\![f]\!](z) \}\!\} \\
[\![(t, t')]\!](\overline{v}, \overline{r}) & = & \{\!\{ (z, z') \mid z \in [\![t]\!](\overline{v}, \overline{r}) \wedge z' \in [\![t]\!](\overline{v}, \overline{r}) \}\!\} \\
[\![[t]_{e,f}]\!](\overline{v}, \overline{r}) & = & [\![\texttt{fold}(e, f)(r')]\!][r' \mapsto [\![t]\!](\overline{v}, \overline{r})]
\end{array}
$$

Figure 6 Semantics of terms.

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 $\rho(v) = 280$.

4 Term Semantics for SparkLite

In this section, we define an alternative, equivalent semantics for SparkLite where the program is interpreted as a term in APA. This term is called the *program term* and denoted $\Phi(P)$ for program P, specified in Figure 5. These terms are assigned their own semantics such that the semantics of $\Phi(P)$ is identical to the semantics of P as defined earlier. Special variables are used to refer to elements of input RDDs[7], and a new language construct is added to denote the fold operation. As an example, we take the program $P1$ from Figure 3, and show how to construct $\Phi(P1)$ by recursively applying ϕ_P and simplifying the formula.

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

[7] To avoid overhead of notations, we assume the programs do not contain self-products (for every product in the program, the sets of variables appearing in each component must be disjoint).

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

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. 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:

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

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

▸ **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:*

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

5 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 $[\![P_1]\!]([\![\overline{v}]\!],[\![\overline{r}]\!]) = [\![P_2]\!]([\![\overline{v}]\!],[\![\overline{r}]\!])$.

5.1 Verifying Equivalence of SparkLite Programs without Aggregations

Given two programs P_1, P_2 receiving as input a series of RDDs $\overline{r} = (r_1,\ldots,r_n)$. We assume w.l.o.g. the programs do not receive non-RDD arguments $\overline{v}$.[8] We let $\overline{x} = (x_1,\ldots,x_n)$ be a concrete valuation for all input RDDs representative elements: $\mathbf{x}_{r_i}$ will map to x_i. We denote the substitution of the concrete valuation in a term t over $\overline{\mathbf{x}_r}$ as: $t(\overline{x}) = t[x_1/\mathbf{x}_{r_1},\ldots,x_n/\mathbf{x}_{r_n}]$.

Comparing representative elements. For two program terms to be comparable, they must depend on the same input RDDs. For example, let $P1(R_0,R_1) = \texttt{map}(\lambda x.1)(R_0)$ and $P2(R_0,R_1) = \texttt{map}(\lambda x.1)(R_1)$. $P1$ and $P2$ have the same program term (the constant 1), but the multiplicity of that element in the output bag is different and depends on the source input RDD. In $P1$, its multiplicity is the same as the size of R_0, and in $P2$ it is the same as the size of R_1. $P1$ and $P2$ are therefore not equivalent, for inputs R_0, R_1 of different sizes. Therefore, for each program term $\Phi(P)$ we consider the *set of free variables*, $FV(\Phi(P))$. Each free variable has some source input RDD. In the example, $FV(\Phi(P1)) = \{\mathbf{x}_{R_0}\}$, and $FV(\Phi(P2)) = \{\mathbf{x}_{R_1}\}$.

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

▸ **Proposition 3.** *Let there be two terms t_1, t_2, over input RDDs $\overline{r}$. such that $FV(t_1) \neq FV(t_2)$, and $t_1 \neq \bot \vee t_2 \neq \bot$. Then $\exists \overline{r}. [\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$.*

The program terms may contain $\bot$ expressions, therefore we need to encode the formulas in APA, and remove all appearances of $\bot$, which is not part of its signature. We write a series of universally true schemes for translating terms referencing $\bot$ to APA when appearing in an equivalence formula, including translation of all conditionals to FOL. Iterative application of these rules by structural induction on the equivalence formula transforms it to a formula in APA: without *ite* and without $\bot$.

▸ **Proposition 4** (Schemes for converting conditionals to a normal form). *Let t denote terms, c denotes conditions, and f, g denote functions which are extended to return $\bot$ if one of the given arguments is $\bot$.*

1. *Applying a function with multiple arguments on conditionals and terms without conditionals:*

$$f(ite(c_1, t_1, \bot), \dots, ite(c_n, t_n, \bot), t'_1, \dots, t'_m) = ite(\bigwedge_{i=1}^{n} c_i, f(t_1, \dots, t_n, t'_1, \dots, t'_m), \bot)$$

2. *General conditionals:*

$$(ite(c, t_1, t_2) = ite(c', t'_1, t'_2)) \iff ((c \wedge c' \implies t_1 = t'_1) \wedge (c \wedge \neg c' \implies t_1 = t'_2) \\ \wedge (\neg c \wedge c' \implies t_2 = t'_1) \wedge (\neg c \wedge \neg c' \implies t_2 = t'_2))$$

3. *Comparison to $\bot$:*

$$(ite(c, t, \bot) = \bot) \iff \neg c \vee t = \bot$$

4. *Shortcut: Equivalence of functions of conditionals, when t, t' do not contain ite:*

$$\Big(f(ite(c, t, \bot)) = g(ite(c', t', \bot))\Big) \iff \Big((c \iff c') \wedge (c \implies f(t) = g(t'))\Big)$$

5. *Unnesting of nested conditionals:*

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

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

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

Proof. For non-RDD return types, the absence of aggregate operators implies we can use Proposition 1, as the returned expression is expressible in APA. For RDDs we provide an algorithm in Figure 7, which is a decision procedure: If both program terms result in an empty bag (step 1), the algorithm detects it and outputs the programs are equivalent. Otherwise, the algorithm checks syntactically in step 2 that $FV(\Phi(P_1)) = FV(\Phi(P_2))$, and outputs the programs are not equivalent if that is not the case - as justified by Proposition 3. The correctness of the algorithm in step 3 follows from the equivalence of the TS semantics and the operational semantics defined in 3.2 (Proposition 2). The equivalence formula generated does not contain aggregate terms, and applications of *ite* are normalized using the rules in Proposition 4, resulting in a formula definable in APA. The algorithm generates formulas in APA several times: once to verify whether the programs do not return the empty bag for all

1. If $\Phi(P_1) = \bot \wedge \Phi(P_2) = \bot$, output **equivalent**.
2. Verify that: $FV(\Phi(P_1)) = FV(\Phi(P_2))$. If not, output **not equivalent**.
3. We check the following formula is satisfiable:

$$\exists \overline{x}.\Phi(P_1)[\overline{v}/FV(\Phi(P_1))] \neq \Phi(P_2)[\overline{v}/S(FV(\Phi(P_2)))]$$

   If it is satisfiable, return **not equivalent**. Otherwise, the formula is unsatisfiable, return **equivalent**.

Figure 7 An algorithm for solving PE for two programs P_1, P_2 with the same signature

inputs, and second to test for equivalence. [9] From Proposition 1, all the formulas checked by the algorithm are decidable. ◂

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. In addition, we assume that in programs returning an RDD-type, the RDD is named r^{out}, and the programs always end with `return` r^{out}. Thus, $\Phi(P) = \phi_P(r^{out})$.

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

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

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

$$\begin{array}{ll} \phi_{P1}(R') = 2 * \mathbf{x}_R\,; & \phi_{P1}(r^{out}) = ite(\varphi < 14 \wedge \varphi = \phi_{P1}(R'), \varphi, \bot) = ite(2 * \mathbf{x}_R < 14, 2 * \mathbf{x}_R, \bot) \\ \phi_{P2}(R') = ite(\mathbf{x}_R < 7, \mathbf{x}_R, \bot)\,; & \phi_{P2}(r^{out}) = (\lambda x.x + x)(\phi_{P1}(R')) = ite(\mathbf{x}_R < 7, \mathbf{x}_R, \bot) + ite(\mathbf{x}_R < 7, \mathbf{x}_R, \bot) \end{array}$$

We need to verify that:

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

Using Proposition 4, $ite(2 * \mathbf{x}_R < 14, 2 * \mathbf{x}_R, \bot) = ite(\mathbf{x}_R < 7, \mathbf{x}_R, \bot) + ite(\mathbf{x}_R < 7, \mathbf{x}_R, \bot)$ becomes via rule (4): $ite(2 * \mathbf{x}_R < 14, 2 * \mathbf{x}_R, \bot) = ite(\mathbf{x}_R < 7 \wedge \mathbf{x}_R < 7, \mathbf{x}_R + \mathbf{x}_R, \bot)$ which via rule (2) becomes: $(2 * \mathbf{x}_R < 14 \iff \mathbf{x}_R < 7) \wedge (2 * \mathbf{x}_R < 14 \implies 2 * \mathbf{x}_R = \mathbf{x}_R + \mathbf{x}_R)$. See **??** for an implementation. ■

For additional examples, refer to appendix **??**.

5.2 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. The terms for aggregate

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

operations are given using a special operator - $[t]_{i,f}$ - where t is the term being folded, i is the initial value, and f is the fold function. The operator *binds* all free variables in the term t, thus the free variables of t are not contained in the free variables set of the term that includes the aggregate term.

5.2.1 Single aggregate

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

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

▶ **Lemma 3** (Sound method for verifying equivalence of Agg^1 programs). *Let there be representative elements φ_0, φ_1 over σ_0, σ_1. 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 $g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1})$ if:*

$$FV(\varphi_0) = FV(\varphi_1) \tag{1}$$

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

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

Lemma 3 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.

▶ Example 2 (Maximum and minimum). Below is an example of two equivalent programs belonging to Agg^1:

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`$(-\infty, max)(R)$	$R' = \texttt{map}(\lambda x. - x)(R)$
2		`return` $-\texttt{fold}(+\infty, min)(R')$

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:

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

We apply Lemma 3: The two program apply a fold operation on a term of the same RDD R. $init_0 = -\infty, init_1 = +\infty$ and $g = \lambda x.x., g' = \lambda x. - x$, therefore $g(-\infty) = g'(+\infty)$ as required. We check the inductive claim:

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

We assume $A = -A'$ and attempt to prove $max(A, x) = -min(A', -x)$, after normalizing to APA:

$$\begin{aligned} max(A, x) &\overset{?}{=} -min(A', -x) \\ ite(A > x, A, x) &\overset{?}{=} -ite(A' < -x, A', -x) = ite(A' < -x, -A', x) \end{aligned}$$

And we verify the following APA formula:

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

which is true. ∎

Additional Examples. Refer to appendix ??.

Completeness There are several cases in which one or more of the requirements of Lemma 3 are not satisfied, yet the aggregate expressions are equal. The first requirement, $FV(\varphi_0) = FV(\varphi_1)$, is not necessary when the fold applied on the terms are both *trivial*.

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

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

If two instances of Agg^1 have trivial folds, then Equation (2) in Lemma 3 is a sufficient condition for the equivalence:

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

Conversely, when the fold is not trivial, the proof of Lemma 3 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 is contained in the other, any non-constant result of the fold function on these additional elements will lead to inequal results. To avoid such peculiarities, we shall require for additional classes of programs to satisfy satisfy equal sets of free variables in their aggregate terms. We proceed with an example showing a case which Lemma 3 does not cover.

▸ 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 Agg_1 programs.

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

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

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

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

Despite the fact that Lemma 3 did not cover Example 3, it still belongs to a sub-class of Agg^1 for which an equivalence test method exists.

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

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

$$\forall \overline{v_1}, \overline{v_2}. \exists \overline{v}'. f_0(f_0(init_0, \varphi_0(\overline{v_1})), \varphi_0(\overline{v_2})) = f_0(init_0, \varphi_0(\overline{v}')) \tag{5}$$
$$\wedge\ f_1(f_1(init_1, \varphi_1(\overline{v_1})), \varphi_1(\overline{v_2})) = f_1(init_1, \varphi_1(\overline{v}'))$$

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

▸ **Theorem 4** (Agg^1_{sync} is *decidable*). *Let P_1, P_2 such that $\langle P_1, P_2\rangle \in Agg^1_{sync}$, with input RDDs $\overline{r}$. We denote $\Phi(P_i) = g_i([\varphi_i]_{init_i,f_i})$. Then, $\Phi(P_1) = \Phi(P_2)$ if and only if:*

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

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

Proof. Sound (if): We prove the equality $g_1([\varphi_1]_{init_1,f_1}) = g_2([\varphi_2]_{init_2,f_2})$ by induction on the size of the RDDs $[\![\varphi_1]\!], [\![\varphi_2]\!]$, denoted n. For $n = 0$, $[\![\varphi_1]\!](\overline{r}) = [\![\varphi_2]\!](\overline{r}) = \varnothing$, thus $[\varphi_i]_{init_i,f_i} = init_i$ ($i = 1,2$), and the equality follows from Equation (6). Assuming for n and proving for $n+1$: We let a sequence of intermediate values $A_{\varphi_i,k}$, ($i = 1,2; k = 1,\dots,n+1$), for which we know in particular that $g_1(A_{\varphi_1,n}) = g_2(A_{\varphi_2,n})$, and we need to prove $g_1(A_{\varphi_1,n+1}) = g_2(A_{\varphi_2,n+1})$. We denote $A_{\varphi_i,0} = init_i$, and then we have $A_{\varphi_i,k} = f_i(A_{\varphi_i,k-1}, \varphi_i(\overline{a_k}))$ ($k = 1,\dots,n+1$) for some $\overline{a_k}$. According to Equation (5), $A_{\varphi_i,2} = f_i(A_{\varphi_i,1}, \varphi_i(\overline{a_2})) = f_i(f_i(init_i, \varphi_i(\overline{a_1})), \varphi_i(\overline{a_2}))$ yields $\exists \overline{a'_2}. \bigwedge_{i=1,2} A_{\varphi_i,2} = f_i(init_i, \varphi_i(\overline{a'_2}))$. We can thus use Equation (5) to prove by induction that $\exists \overline{a'_k}. \bigwedge_{i=1,2} A_{\varphi_i,k} = f_i(init_i, \varphi_i(\overline{a'_k}))$, and in particular $\exists \overline{a'_n}. \bigwedge_{i=1,2} A_{\varphi_i,n} = f_i(init_i, \varphi_i(\overline{a'_n}))$. By applying Equation (7) for $\overline{v} = \overline{a_{n+1}}, \overline{y} = \overline{a'_n}$, we get:

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

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

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

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

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

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

▸ Example 4 (Completing Example 3). We have:

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

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

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

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

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

▸ **Definition 5** (The Agg^1_R class). *Let there be a program P with $\Phi(P) = \psi$. We say that $P \in Agg^1_R$ if ψ contains a single instance of an aggregate term $a = [\varphi]_{init,f}$. We write $\Phi(P) = \psi(a)$.*

▸ **Lemma 5** (Lifting Lemma 3 to Agg^1_R). *Let there be two* SparkLite *programs $P_1, P_2 \in Agg^1_R$ with terms ψ_i and aggregate expressions $a_i = [\varphi_i]_{init_i,f_i}$, $i \in \{1,2\}$. P_1 is equivalent to P_2 if:*

$$FV(\varphi_1) = FV(\varphi_2) \tag{8}$$
$$FV(\psi_1) = FV(\psi_2) \tag{9}$$
$$\forall \overline{x}.\psi_1(init_1)(\overline{x}) = \psi_2(init_2)(\overline{x}) \tag{10}$$
$$\forall \overline{v}, A_1, A_2.(\forall \overline{x}.\psi_1(A_1)(\overline{x}) = \psi_2(A_2)(\overline{x})) \implies$$
$$(\forall \overline{x}.\psi_1(f_1(A_1,\varphi_1(\overline{v})))(\overline{x}) = \psi_2(f_2(A_2,\varphi_2(\overline{v}))))\overline{x})) \tag{11}$$

Lemmas 3,5 show that Agg^1, Agg^1_R have a sound equivalence verification method, and Theorem 4 shows that Agg^1_{sync} has a sound and complete equivalence verification method. [10]

▸ **Definition 6** (The Agg^n class). *Let there be a program P with $\Phi(P) = g([t_1]_{i_1,f_1},\ldots,[t_n]_{i_n,f_n})$. $P \in Agg^n$ if $t_1,\ldots,t_n$ do not contain aggregate terms.*

▸ **Lemma 6.** *Let P_1, P_2 be two programs in Agg^n, such that $\Phi(P_i) = g_i(\overline{[\varphi_i]_{init_i,f_i}})$. We have $g_1(\overline{[\varphi_1]_{init_1,f_1}}) = g_2(\overline{[\varphi_2]_{init_2,f_2}})$ if:*

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

▸ Example 5 (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 6, we are able to show the equivalence.

[10] Even when the completeness criterion for Agg^1_{sync} is not met, we may be successful in proving the equivalence using Lemma 3. For example, $[((\lambda x.1)(\mathbf{x}_{r_0}),(\lambda x.1)(\mathbf{x}_{r_1}))]_{0,\lambda A,(x,y).A+x+y} = [((\lambda x.1)(\mathbf{x}_{r_1}),(\lambda x.1)(\mathbf{x}_{r_0}))]_{0,\lambda A,(x,y).A+x+y}$ can be proved by induction, but for $f = \lambda A,(x,y).A+x+y$, $f(f(A,(1,1)),(1,1)) = A+4 \neq f(A,(1,1)) = A+2$ (the choice of valuation does not change the result). Thus, it does not satisfy the completeness criterion.

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

```
    P1(R: RDD_Int):                  | P2(R: RDD_Int):
1   return fold((0,0), h)(R)         | R_P = filter(λx.x ≥ 0)(R)
2                                    | R_N = map(λx. − x)(filter(λx.x < 0)(R))
3                                    | p = fold(0, λA, x.A + x)(R_P)
4                                    | n = −fold(0, λA, x.A + x)(R_N)
5                                    | return (p, n)
```

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

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. ∎

5.3 Proof of undecidability of *PE*

We show a reduction of Hilbert's 10th problem to *PE*. We assume towards a contradiction that *PE* is decidable. Let there be a polynomial p over k variables $x_1, \ldots, x_k$, and coefficients $a_1, \ldots, a_k$. 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. We define a translation φ from polynomials to SparkLite expressions:

- $\varphi(x_i) = [\texttt{map}(\lambda x.1)(R_i)]_{0,\lambda A,x.A+x}$
- $\varphi(x_i x_j) = [\texttt{cartesian}(\texttt{map}(\lambda x.1)(R_i), \texttt{map}(\lambda x.1)(R_j))]_{0,\lambda A,(x,y).A+x}$
- $\varphi(x_i^2) = [\texttt{cartesian}(\texttt{map}(\lambda x.1)(R_i), \texttt{map}(\lambda x.1)(R_i))]_{0,\lambda A,(x,y).A+x}$. This rule as well as the rest of the powers follows directly from the previous rule. For a degree k monom, we apply the *cartesian* operation k times, and fold it with $\lambda A, (x)_{i=1}^k.A + x_1$.
- $\varphi({x_i}^0) = 1$, trivially.
- $\varphi(am(x_{i_1}, \ldots, x_{i_j})) = a\varphi(m(x_{i_1}, \ldots, x_{i_j}))$ where $m(x_{i_1}, \ldots, x_{i_j}) = x_{i_1}^{l_1} \cdots x_{i_j}^{l_j}$, for which φ is defined by induction according to the previous rules.
- $\varphi(p) = \sum_{i=1}^k \varphi(a_i m_i(x_{i_1}, \ldots, x_{i_k}))$, where $p = \sum_{i=1}^k a_i m_i(x_{i_1}, \ldots, x_{i_k})$.

Given a polynomial $p(a_1 \ldots, a_k; x_1, \ldots, x_k)$, we generate the following instance of the *PE* problem:

```
    P1(R_1, ..., R_k: RDD_Int):  | P2(R_1, ..., R_k: RDD_Int):
1   return φ(p) ≠ 0              | return tt
```

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

References

1 Java. http://java.net. Accessed: 2016-07-19.

2 Python. https://www.python.org/. Accessed: 2016-07-19.

3 Scala. http://www.scala-lang.org. Accessed: 2016-07-19.

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

5 Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2723372.2742797, doi:10.1145/2723372.2742797.

6 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.

7 Surajit Chaudhuri and Moshe Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '93, pages 59–70, New York, NY, USA, 1993. ACM. URL: http://doi.acm.org/10.1145/153850.153856, doi:10.1145/153850.153856.

8 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.

9 Sara Cohen. Containment of aggregate queries. *SIGMOD Rec.*, 34(1):77–85, March 2005. URL: http://doi.acm.org/10.1145/1058150.1058170, doi:10.1145/1058150.1058170.

10 Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 155–166, New York, NY, USA, 1999. ACM. URL: http://doi.acm.org/10.1145/303976.303992, doi:10.1145/303976.303992.

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

12 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

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

14 Todd J. Green. Containment of conjunctive queries on annotated relations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 296–309, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1514894.1514930, doi:10.1145/1514894.1514930.

15 Masahito Hasegawa. *Decomposing typed lambda calculus into a couple of categorical programming languages*, pages 200–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. URL: http://dx.doi.org/10.1007/3-540-60164-3_28, doi:10.1007/3-540-60164-3_28.

16 Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, July 2001. URL: http://doi.acm.org/10.1145/502090.502100, doi:10.1145/502090.502100.

17 Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):pp. 299–314, 1996.

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

19 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`.

20 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.

21 Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, October 1980. URL: `http://doi.acm.org/10.1145/322217.322221`, `doi:10.1145/322217.322221`.

22 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

A Extending Cooper's Algorithm to the Augmented Presburger Arithmetic

▶ **Proposition 5.** *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

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

Integer ops

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

Boolean ops

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

Tuples

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

UDFs

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

RDD

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

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

Figure 8 Typing rules for SparkLite

C Proof of Proposition 3

w.l.o.g. we $t_1 \neq \bot$ (symmetry). Then $\exists \overline{x}, y. y = t_1(\overline{x}) \wedge y \neq \bot$. We choose input RDDs $\overline{r}$ such that $r_i = \{\!\{x_i; n_i\}\!\}$. If $t_2(\overline{x}) \neq y$ then $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required. Otherwise, the multiplicity of y in $[\![t_1]\!]$ is $\Pi_{i, r_i \in FV(t_1)} n_i$, and in $[\![t_2]\!]$ it is $\Pi_{i, r_i \in FV(t_2)} n_i$. As $FV(t_1) \neq FV(t_2)$ and $n_i > 0$, the multiplicities are different, thus $[\![t_1]\!](\overline{r}) \neq [\![t_2]\!](\overline{r})$, as required. ■

D Proof of Lemma 3

Proof. First we recall the semantics of the `fold` operation on some RDD R, which is a bag. We choose an arbitrary element $a \in R$ and apply the fold function recursively on a and on R with a single instance of a removed. We then write a sequence of elements in the order they are chosen by `fold`: $\langle a_1, \dots, a_n \rangle$, where n is the sum of all multiplicities in the bag R. We also know that a requirement of aggregating operations' UDFs is that they are *commutative*, so the order of elements chosen does not change the final result. We also extend f_i to $\xi_i \times (\sigma_i \cup \{\bot\})$ by setting $f_i(A,\bot) = A$ ($\bot$ is defined to behave as the neutral element for f_i). To prove $g([\varphi_0]_{init_0,f_0}) = g'([\varphi_1]_{init_1,f_1})$, it is necessary to prove that

$$g([\![\texttt{fold}]\!](f_0, init_0)(R_0)) = g'([\![\texttt{fold}]\!](f_1, init_1)(R_1))$$

We set $A_{\varphi_j,0} = init_j$ for $j \in \{0,1\}$. Each element of R_0, R_1 is expressible by providing a concrete valuation to the free variables of φ_0, φ_1, namely the vector $\overline{v}$. We choose an arbitrary sequence of valuations to $\overline{v}$, denoted $\langle \overline{a}_1, \dots, \overline{a}_n \rangle$, and plug them into the *fold*operation for both R_0, R_1. The result is 2 sequences of *intermediate values* $\langle A_{\varphi_0,1}, \dots, A_{\varphi_0,n} \rangle$ and $\langle A_{\varphi_1,1}, \dots, A_{\varphi_1,n} \rangle$. We have that $A_{\varphi_j,i} = f_j(A_{\varphi_j,i-1}, \varphi_j[\overline{a}_i/FV])$ for $j \in \{0,1\}$, from the semantics of `fold`. Our goal is to show $g(A_{\varphi_0,n}) = g'(A_{\varphi_1,n})$ for all n. We prove the equality by induction on the *size* of the sequence of possible valuations of $\overline{v}$, denoted n. In each step i, we show $g(A_{\varphi_0,i}) = g'(A_{\varphi_1,i})$.

Case $n = 0$: $R_0 = R_1 = \varnothing$, so $[\![\texttt{fold}]\!](f_0, init_0)(R_0) = init_0$ and $[\![\texttt{fold}]\!](f_1, init_1)(R_1) = init_1$. From Equation (2), $g(init_0) = g'(init_1)$, as required.

Case $n = i$, assuming correct for $n \leq i-1$: By assumption, we know that the sequence of intermediate values up to $i-1$ is equal up to application of g, g', and specifically $g(A_{\varphi_0,i-1}) = g'(A_{\varphi_1,i-1})$. We are given the i'th concrete valuation of $\overline{v}$, denoted $\overline{a}_i$. We need to show $A_{\varphi_0,i} = A_{\varphi_1,i}$, so we use the formula for calculating the next intermediate value:

$$\begin{array}{rcl} A_{\varphi_0,i} & = & f_0(A_{\varphi_0,i-1}, \varphi_0[\overline{a}_i/FV]) \\ A_{\varphi_1,i} & = & f_1(A_{\varphi_1,i-1}, \varphi_1[\overline{a}_i/FV]) \end{array}$$

We use Equation (3), plugging in $\overline{v} = \overline{a}_i$, $A_{\varphi_0} = A_{\varphi_0,i-1}$, and $A_{\varphi_1} = A_{\varphi_1,i-1}$. By the induction assumption, $g(A_{\varphi_0,i-1}) = g'(A_{\varphi_1,i-1})$, therefore $g(A_{\varphi_0}) = g'(A_{\varphi_1})$, so Equation (3) yields $g(f_0(A_{\varphi_0}, \varphi_0[\overline{a}_i/FV])) = g'(f_1(A_{\varphi_1}, \varphi_1[\overline{a}_i/FV]))$. By substituting back A_{φ_j} and the formula for the next intermediate value, we get: $g(A_{\varphi_0,i}) = g'(A_{\varphi_1,i})$ as required. ◀