Relational Equivalence Proofs Between Imperative and MapReduce Algorithms

Bernhard Beckert, Timo Bingmann, Moritz Kiefer,
Peter Sanders, Mattias Ulbrich, and Alexander Weigl

Institute of Theoretical Informatics
Karlsruhe Institute of Technology, Germany

Abstract. Distributed programming frameworks like *MapReduce*, *Spark* and *Thrill*, are widely used for the implementation of algorithms operating on large datasets. However, implementing in these frameworks is more demanding than coming up with sequential implementations. One way to achieve correctness of an optimized implementation is by deriving it from an existing imperative sequential algorithm description through a sequence of behavior-preserving transformations.

We present a novel approach for proving equivalence between imperative and deterministic *MapReduce* algorithms based on partitioning the equivalence proof into a sequence of equivalence proofs between intermediate programs with smaller differences. Our approach is based on the insight that proofs are best conducted using a combination of two kinds of steps: (1) uniform context-independent rewriting transformations; and (2) context-dependent flexible transformations that can be proved using relational reasoning with coupling invariants.

We demonstrate the feasibility of our approach by evaluating it on two prototypical algorithms commonly used as examples in *MapReduce* frameworks: k-means and PageRank. To carry out the proofs, we use a higher-order theorem prover with partial proof automation. The results show that our approach and its prototypical implementation enable equivalence proofs of non-trivial algorithms and could be automated to a large degree.

1 Introduction

Motivation. Frameworks for functional programming for distributed programs, such as *MapReduce* [10], Spark [25] and Thrill [4] address the challenges arising in the implementation of large-scale distributed algorithms by providing a limited set of operations whose execution is automatically parallelized and distributed among the nodes in a cluster. However, designing efficient algorithms in these frameworks is a challenge in itself. A good starting point for a distributed algorithm is an existing imperative algorithm which is then translated into a *MapReduce* framework. This initial program could be taken from a textbook on algorithms or could be a sequential implementation from an existing code base that is to be optimized. However, the translation into *MapReduce* can be non-trivial, and the original algorithmic structure is often lost during the translation since imperative constructs do not translate directly into the functional *MapReduce* primitives. Implementing efficient algorithms using *MapReduce* frameworks can thus require significant and elaborate alterations to a given imperative algorithm.

By proving the equivalence of the original imperative algorithm and its *MapReduce* version, one can verify that no bugs have been introduced during the translation. While such proofs do not directly provide correctness guarantees for the *MapReduce* algorithm, they transfer correctness results from the imperative version to the *MapReduce* implementation. The transferred correctness properties can be formal proofs whose reach then extends to the distributed implementation, but can also be informal arguments, e.g., if the algorithm is a well-known, simple textbook reference implementation or if it has been successfully applied previously.

In this paper, we use the term "*MapReduce*" in a broader sense than implied by the two functions "map" and "reduce". While some frameworks such as Hadoop's MapReduce [24] module are programmed strictly by specifying these two functions, the more popular and widely used distributed frameworks provide many additional primitives for performance reasons and to make them easier to program with. Theoretically, these additional primitives can be reduced to only map and reduce operations [6], but this overly complicates the program description and is generally not used in real-world applications.

Contribution of this paper. We present an interactive verification approach with which a *MapReduce* implementation of an algorithm can be proved equivalent to an imperative implementation (to the best of our knowledge this is the first framework for the purpose of such equivalence proofs, see Sect. 6). Proofs are conducted as chains of individual, smaller behavior-preserving program transformations.

One novelty of the approach is that it brings together two approaches for equivalence reasoning: (1) proving equivalence by means of a series of uniform context-independent rewriting transformations; and (2) proving equivalence by means of relational deductive program verification using coupling invariants. We show how the approaches can be applied alternatingly. We identified a catalogue of 13 individual rules. Correctness of 10 of those rules was proven formally using the *Coq* theorem prover.

Our approach has a high potential for automation. The required interaction is designed to be as high-level as possible. The proof is guided by user-specified intermediate programs from which the individual transformations are derived. The rules are designed such that their side conditions can be proved automatically and we describe how pattern matching can be used to allow for a more flexible specification of intermediate steps.

We describe a workflow for integrating this approach with existing interactive theorem provers. We have successfully implemented the approach as a prototype within the interactive theorem prover *Coq* [22] and evaluated the feasibility of our approach by applying it to the k-means and PageRank algorithms. These two are prototypical algorithms commonly used as examples in *MapReduce* frameworks, because they exhibit the most common patterns found in large-scale distributed data processing applications. By showing that our approach can be applied to these two examples, we demonstrate that it can be extended to a much larger set of applications.

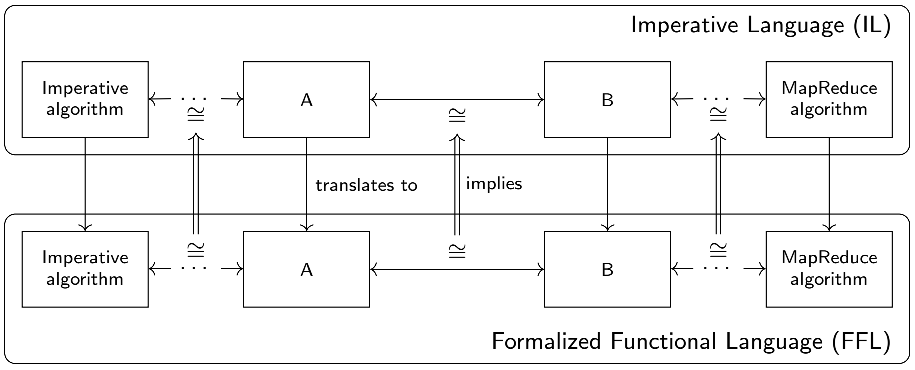

Fig. 1: Chain of equivalent programs is translated into formalized functional language

Interleaving different types of proofs. We came to the important insight that the proof task requires the interplay of two kinds of sub-proofs:

(1) Uniform, context-independent and pattern-driven transformations that can locally change one program into another with a severely changed control flow. Rewriting techniques can be applied to perform such proof steps.
(2) Context-dependent but flexible equivalence proofs that preserve the control flow, but change the data representation. Relational deductive reasoning using coupling invariants allows us to conduct such proof steps.

In previous work [12,18], we have explored how equivalence proofs can be conducted effectively if the two programs to be compared exhibit a similar control flow. In this case, the relational reasoning approach (2) using coupling predicates that logically manifest the relation between the two program states at given synchronization points proved very successful, and often runs fully automatically. The reason for this is that if coupling predicates can be used, only the relationship between two current states needs to be captured in logic – functional properties (what the code actually computes) need not be formalized. The less similar the two compared programs are, the more difficult and less effective becomes this type of reasoning. However, differences in the control flows can often be bridged by aligning them through behavior-preserving rewriting rules of type (1) (like loop unrolling, loop peeling, function inlining) which are applied prior to the verification [17].

The presented approach now generalizes this idea by interleaving steps of both types. This permits us to apply relational reasoning even for the semantically larger gap between implementations of different programming paradigms. Both types of rules (context-independent rewriting and context-dependent relational reasoning) are necessary for such equivalence proofs: Rewriting rules can modify the control flow (and sometimes the data representation), but – being pattern-driven – are very inflexible in the input they can operate on. Relational reasoning on the other hand is not local; thus, it can be very flexible to show differently laid out programs are correct – as long the control flow is generally kept.

Overview of the approach. The main challenge in proving the equivalence of an imperative and a *MapReduce* algorithm lies in the potentially large structural difference between two such algorithms. Existing relational verification approaches (like [13,12,19,23]) exploit the fact that the two program versions to be compared are structurally similar, which allows the verification to focus on describing and proving the similarity of the implementations rather than describing what they actually compute. To deal with the complexity arising from the large structural differences, the equivalence of imperative and *MapReduce* algorithms is not shown in one step, but as a succession of equivalence proofs for structurally closer program versions.

To this end, we require that the translation of the algorithm is broken down (by the user) into a chain of intermediate programs. For each pair of neighboring programs in this chain, the difference is comparatively small and can usually be reduced to one isolated transformation.

The imperative algorithm, the intermediate programs, as well as the *MapReduce* implementation, are given in a high-level imperative programming language (IL). IL is based on a While language and supports integers, booleans, fixed-length arrays and sum and product types. It does not support recursion. Besides the imperative language constructs, IL supports *MapReduce* primitives. Given that we have stated previously that *MapReduce* programs tend to be of a more functional nature, it might seem odd at first to not use a functional language for specifying *MapReduce* algorithms. However, most existing *MapReduce* frameworks are not implemented as separate programming languages but as frameworks offered as APIs on top of imperative languages (Java for Hadoop, Scala for Spark, or C++ for Thrill). Thereby sequential parts of *MapReduce* algorithms can still be implemented using imperative language features.

Each program specified in the high-level imperative language is then automatically translated into the formalized functional language (FFL) described in Sect. 2. FFL is a deterministic language which might seem surprising given that for performance reasons *MapReduce* frameworks often do not guarantee determinism. However, the source of non-determinism in *MapReduce* algorithms are non-associative and non-commutative functions used with the "reduce" primitive. If all used reducer functions are associative and commutative, then the resulting algorithm is deterministic (some frameworks only require associativity). Reducer associativity and commutativity are assumptions for the transformation into FFL which are not verified within our approach. Often, the justification is trivial, e. g., for addition on natural or rational numbers. For non-trivial cases, Chen et al. [8,9] present formal verification approaches.

The equivalence proofs are conducted on programs in FFL. An overview of this process can be seen in Fig. 1. For each pair of neighboring programs in the chain, a proof obligation is generated that requires proving their equivalence. These proof obligations are then discharged independently of each other (using the workflow described in Sect. 4). Since, by construction, the semantics of IL programs is the same as that of corresponding FFL programs, the equivalence of two IL programs follows from the equivalence of their translations to FFL. Figure 2 shows two example IL programs for calculating the element-wise sum of two arrays.

The implementation of our approach based on the *Coq* theorem prover has only limited proof automation and still requires a significant amount of interactive proofs. We are convinced, however, that our approach can be extended such that it becomes highly automatised and only few user interactions or none at all are

```
Function SumArrays(xs,ys)
begin
    sum ← replicate($n$, $0$);
    for $i \leftarrow 0$ to $n - 1$ do
        sum[$i$] ← xs[$i$] + ys[$i$];
    end
    return sum;
end
```

```
Function SumArraysZipped(xs,ys)
begin
    sum ← replicate($n$, $0$);
    zipped ← zip(xs,ys);
    for $i \leftarrow 0$ to $n - 1$ do
        sum[$i$] ← fst(zipped[$i$]) +
          snd(zipped[$i$]);
    end
    return sum;
end
```

Fig. 2: Two IL programs which calculate the element-wise sum of two arrays.

required – besides providing the intermediate programs. Further challenges include the extension of our approach to features such as references and aliasing which are commonly found in imperative languages.

Structure of this paper. In Sec. 2, we lay the formal groundwork for our approach by defining the programming language used for equivalence proofs and the notion of program equivalence used in this paper. Sec. 3 describes the two kinds of program transformations that we have identified and the techniques for proving equivalence using these transformations. The technical framework for equivalence proofs and the potential for automation are described in Sec. 4 and their its evaluation is in Sec. 5. In Sec. 6, we discuss work related to the ideas presented in this paper. Finally, we conclude in Sec. 7 and consider possible future work.

2 Formal Foundations and Program Equivalence

In this section, we briefly describe the language FFL, introduce a reduction big-step semantics for FFL and discuss the notion of equivalence for FFL programs.

The primary design goal of FFL is the capability to represent both imperative and *MapReduce* programs written in IL. To achieve this, we follow the work by Radoi et al. [21] and use a simply typed lambda calculus extended by the theories of sums, products, and arrays. Furthermore, the language also contains the programming primitives usually found in *MapReduce* frameworks. We also want to limit the number of primitives included in FFL while still retaining expressiveness. This simplifies proving general properties of FFL and proving the correctness of rewrite rules. We accomplish this by building upon the work of Chen et al. [7], who describe how to reduce the large number of primitives provided by *MapReduce* frameworks to a smaller core.

Two new primitives iter and fold were added to translate imperative loops directly. Compared to transforming imperative programs into a recursive form, this allows a translation closer to the original program formulation. The fold operator is used to translate bounded for-each iterator loops into FFL. The evaluation of the expression fold f v_0 xs starts with the initial loop state v_0 and iterates over each value of the array xs updating the loop state by applying f. General while loops are translated using the iter function. $\mathsf{iter}\ f\ v_0$ is evaluated by repeatedly applying f to the loop state (which is initially v_0) until f returns unit to indicate termination. Program terms incorporating iter need not evaluate to a value since the construct allows formulating non-terminating programs.

The big-step operational reduction semantics [16] of FFL is defined as a binary relation $\Rightarrow_{bs}$. Note that, since FFL is based on lambda calculus, programs in FFL as well as values are FFL expressions. The semantics predicate is thus a partial, functional relation on FFL-terms.

Definition 1. *An FFL term t* evaluates *to an FFL term v if $t \Rightarrow_{bs} v$ holds. A term t is called* stuck *if there exists no v such that $t \Rightarrow_{bs} v$. Terms that evaluate to themselves are called* values*.*

A formal definition of the syntax and semantics of FFL can be found in [2]. The evaluation of a program t in an input state (i.e., for an argument tuple a) resulting in a output state v (a result tuple) can be formalized as the reduction evaluation of the application of the program to the arguments: $\langle t, a\rangle \Downarrow_{bs} v \ := \ \mathsf{app}(t, a) \Rightarrow_{bs} v$.

The semantics of FFL is deterministic. This may seem odd because most *MapReduce* frameworks take considerable leeway from fully deterministic execution in the name of performance. For example, some operations may be evaluated in a non-deterministic order depending on how fast data arrives over the network leading to non-determinism if these operations are not commutative and associative. However, non-determinism in *MapReduce* algorithms is usually not desired, and the problem of checking whether or not a *MapReduce* algorithm is deterministic is orthogonal to proving that it is equivalent to an imperative algorithm. We thus consider a deterministic language model to be suitable for our purposes and defer checking of determinism to other tools such as those developed by Chen et al. [8,9].

Since FFL includes the potential for run-time errors such as out-of-bound array accesses but does not include an explicit error term, the step-relation $\Rightarrow_{bs}$ is not total. The absence of an explicit error term also has the consequence that one cannot distinguish between non-termination and runtime errors according to the definition of program equivalence in Definition 2.

The introduction of the semantics relation allows us to define a notion of program equivalence for FFL terms.

Definition 2. *Two well-typed FFL terms s and t are called* equivalent *if they (a) are of the same type τ and (b) evaluate to the same values v. We write $s \cong_\tau t$ in this case. Using $\vdash t : \tau$ to denote that the closed FFL term t has type τ, this definition can be formalized as follows:*

$$\begin{aligned} s \cong_\tau t \quad := \quad & \vdash s : \tau \wedge \vdash t : \tau \ \wedge \\ & \forall v.\, (s \Rightarrow_{bs} v) \Leftrightarrow (t \Rightarrow_{bs} v) \end{aligned} \tag{1}$$

This definition of program equivalence also enforces *mutual termination* [11], i.e., the property that equivalent programs either both terminate or both diverge. In particular, two non-terminating terms of the same type are equivalent.

Example 1. Figure 3 shows two transformations of the function `SumArrays` (see Fig. 2) into FFL. In Fig. 3 (a), the loop is translated using fold, and in Fig. 3 (b)

```
fold(λsum. λi.
    write(sum, i, xs[i] + ys[i]),
  replicate(n, 0),
  range(0, n))
```

(a)

```
snd(iter(λ(i, sum).
    if i < n
      then inr (i + 1,
              write(sum, i, xs[i] + ys[i]))
      else  inl unit,
  (0, replicate(n, 0))))
```

(b)

Fig. 3: Translation of function `SumArrays` (see Fig. 2) into FFL using `fold` and `iter` (where inl and inr denote the left and right injection into a sumtype).

using the more general iter. In both cases, it can be observed that the local variables i and sum become λ-bound variables of the translation of the enclosing block, in this case the loop body.

The first translation has the initial state replicate$(n, 0)$, an array of length n with all values set to 0, and it iterates over the indices in the array $([0; 1; \ldots ; n - 1])$, updating the array *sum* in each iteration using the *write* function of the McCarthy theory of arrays.

The translation in Fig. 3 (b) starts from the initial loop state $(0, \mathsf{replicate}(n, 0))$. In each iteration, an *if*-clause is used to check if the loop condition still evaluates to *true*. If that is the case, the index is incremented and *sum* is updated, otherwise the program exits the loop as indicated by inl unit and evaluates to the current loop state.

3 Program Transformations

With the reduction of imperative and *MapReduce* implementations to the common language FFL, we are able to prove equivalence between two programs by constructing a chain of single, isolated program transformations. We categorize the transformations by their dependence on the surrounding context. A *context-independent* transformation is an uniform transformation as it replaces only one isolated subterm in the program by an equivalent term. This replacement has no effects on other parts of the program and has only conditions on the replaced subterm. In contrast, *context-dependent* transformations do not replace individual terms but require many small changes throughout different parts of the programs.

For example, consider the IL programs in Fig. 2. In the left IL program, the loop iterates over two separate arrays xs and ys of the same length. In the right IL program, the loop iterates over a single array that represents the *zipped* version of xs and ys. Inspection of the FFL versions from Fig. 3 shows that a program transformation unifying both programs requires two changes to individual subterms: (a) the initial loop state, and (b) adaption of the read and write references.

We use two complementary techniques for proving the correctness of a transformation depending on whether it is context-independent or context-dependent: The equivalence of programs related by context-independent transformations is proven using rewrite rules (Sect. 3.1) while the equivalence of programs related by context-dependent transformation is shown using coupling predicates (Sect. 3.2).

$$\begin{array}{lcl} \mathsf{fold}(\lambda acc.\,\lambda x.\,f(acc, g(x)), & & \mathsf{fold}(\lambda acc.\,\lambda y.\,f(acc, y), \\ \quad i, & \leftrightsquigarrow & \quad i, \\ \quad xs) & & \quad \mathsf{map}(g, xs)) \end{array}$$

Side conditions: $acc \notin FV(f), x \notin FV(f), y \notin FV(f), x \notin FV(g), acc \notin FV(g)$

Fig. 4: Rewrite rule for separating a loop body into two functions f and g such that the evaluation of g is independent of all other iterations and can be computed in parallel. $FV(g)$ is the set of free, unbound variables in the term g.

3.1 Handling Context-Independent Transformations Using Rewrite Rules

Intermediate programs are mostly linked by uniform context-independent transformations on isolated subterms. Instead of performing and proving these local transformations manually, we can capture them into generalized rewrite rules. That equivalence is preserved when these generalized rewrite rules are applied, needs to be proven only once. By maintaining and using a collection of local transformations that have been proven correct, we can lower proof complexity and later increase the computer assistance and automation.

A rewrite rule describes a bidirectional program transformation that allows the replacement of a subterm within a program. It is composed of two patterns and a set of side conditions which are sufficient for the transformation to preserve program equivalence. A pattern is an FFL term containing metavariables.

To apply a rewrite rule on a program, we have to identify a subterm of the program that (a) matches the first pattern and (b) satisfies the side conditions. The transformed program is obtained by the instantiation of the other pattern with the matched metavariables. Since the sets of bound metavariables in the two patterns can be different, some metavariables may not be uniquely instantiated, leading to a degree of freedom in the translation. We will discuss the practical implications of this in Sect. 4.3.

While there is no hard limit on the complexity of the side conditions that can be part of rewrite rules, it is desirable to use side conditions that are simple and easy to check. This prevents the application of rewrite rules from producing auxiliary complex proofs due to complex side conditions. In our experiments we only encountered the following three different kinds of side conditions:

1. Two arrays xs and ys have the same length, i.e., $\mathsf{length}(xs) \cong_{\text{int}} \mathsf{length}(ys)$.
2. t is not stuck.
3. $x \notin FV(t)$ where $FV(t)$ is the set of free variables in the term t.

Sect. 4.3 discusses how these side conditions could be discharged automatically.

To illustrate the kind of rewrite rules used in the equivalence proofs described in this paper, we present two of the most commonly used rewrite rules in detail. To demonstrate the feasibility of formal correctness proofs for rewrite rules, we have proven the correctness of most (10 out of 13 rules) of our rules in *Coq*. A full listing of all FFL rewrite rules can be found in the long version of this paper [2]. The first rule, shown in Fig. 4, decomposes the loop body of a `fold` expression into two separate functions f and g, where g is independent of other iterations. Thus, g can

$$
\begin{array}{lcl}
\begin{array}{l}
\mathsf{fold}(\lambda acc.\, \lambda(i, x). \\
\quad \mathsf{write}(acc, i, f(i, x, acc[i])), \\
\quad ys, \\
\quad xs)
\end{array}
& \rightsquigarrow &
\begin{array}{l}
\mathsf{fold}(\lambda acc.\, \lambda(i, v).\, \mathsf{write}(acc, i, v), \\
\quad ys, \\
\quad \mathsf{map}(\lambda(i, vs). \\
\qquad (i, \mathsf{fold}(\lambda x'.\, \lambda x.\, f(i, x, x'), ys[i], vs)), \\
\qquad \mathsf{group}(xs)))
\end{array}
\end{array}
$$

Side conditions: $acc \notin FV(f), x \notin FV(f), x' \notin FV(f), i \notin FV(f), vs \notin FV(f)$

Fig. 5: Rewrite rule for grouping loop iterations which access the same index of an array.

$$
\begin{array}{ll}
& \mathcal{C}(i_0, i'_0) \\
\wedge & (\forall i, i', j.\, \mathcal{C}(i, i') \implies \mathcal{C}(f(i, xs[j]), f'(i', xs'[j]))) \\
\implies & \mathcal{C}(\mathsf{fold}(f, i_0, xs), \mathsf{fold}(f', i'_0, xs'))
\end{array}
$$

Fig. 6: Coupling invariant rule for `fold` for a coupling predicate $\mathcal{C}$. Free variables are implicitly universally quantified.

be computed in parallel using a `map` operation. This rewrite rule illustrates that rewrite rules used in proofs can often also function as guidelines for parallelizing and distributing imperative algorithms.

The second rule, shown in Fig. 5, is similar to the previous rule in that it tries to separate independent parts of the loop body so that they can be executed in parallel. However, in this case, the part that is extracted is only independent of other iterations that access different indices. The `group` operation can be used to group all accesses to the same index. Using `map` one can then calculate the new values for each index in *xs* in parallel and update *ys* with those new values.

3.2 Handling Context-Dependent Transformations Using Coupling Predicates

While context-independent transformations are nicely handled using rewrite rules, context-dependent transformations can usually not be captured by patterns and simple side conditions. Coupling predicates provide a flexible and effective solution to proving the correctness of context-dependent transformations – at the cost of requiring more user interactions than rewrite rules. The use of coupling predicates is based on the observation that analyzing two loops in lockstep and proving that a relational property, i.e., the coupling predicate, holds after each iteration is sufficient to prove that it holds after the execution of both loops. Fig. 6 shows the corresponding coupling invariant rule for `fold`. For the purpose of presentation, we ignore the distinction between syntactic terms and the values to which they evaluate. Besides this rule for `fold`, there is a similar rule for `iter`.

One compelling example for using coupling predicates is given in the beginning of this section. The presented program transformation is provable equivalent with the coupling invariant rule from Fig. 2. If these arrays are part of the accumulator in a `fold` or `iter`, capturing this transformation by a rule patterns is not possible: While the transformation of the initial accumulator value can be captured using patterns, this is not sufficient since all references to the accumulator in the loop body also need to be updated. These references can be nested arbitrarily deep inside the loop body and there can be arbitrarily many references. This makes it impossible to capture them by a single pattern which can only bind a fixed number of variables and thereby only make a fixed number of transformations. To make matters worse, it is not even sufficient to just transform the loop itself since the loops are not equivalent: the right loop evaluates to two separate arrays while the other evaluates to an array of tuples. It is thus necessary to prove the equivalence of the enclosing terms under the assumption that the loop in one program evaluates to a tuple of two arrays $\mathsf{pair}(xs, ys)$ while the other loop evaluates to $\mathsf{zip}(xs, ys)$. This assumption can then be proven correct using the coupling predicate stating that this holds after each iteration.

Another commonly found transformation is the removal of unused elements from a tuple representing the loop accumulator. As it was the case for the previous transformation, the loops themselves are not equivalent and it is necessary to prove enclosing terms equivalent using the assumption that the values present in both loop accumulators are equivalent. As before, this assumption can be proven correct using a coupling predicate which states that this holds after each iteration.

4 Transformation Application Strategy

Splitting the translation into a chain of intermediate programs and translating these into FFL leaves us with the problem of proving neighboring programs equivalent. In order to reduce the amount of user interaction required to conduct these basic equivalence proofs, we define an iterative heuristic search strategy to identify the locations within the programs on which the program transformations described in Sect. 3 will be applied. Alg. 1 depicts this search strategy as pseudocode. First, we use the structural difference operation (`Diff`, see Sect. 4.1) to identify subterms P' and Q' whose equivalence implies the equivalence of the full programs P and Q. Second, we start an iterative bottom-up process in which we try to prove the equivalence of the subterms P', Q' and their enclosing terms (`ProveEquivalent`), until we reached the top level programs P and Q. During the bottom-up process, the subterms P' and Q' may be found to be equivalent only in some cases but not in others. But that is fine as long as we are able to prove that the cases in which they are non-equivalent are not relevant in the context in which P' and Q' occur. Thus, we extract the premises under which P' and Q' are equivalent, and bubble them up to the equivalence proof for the parent terms (`AddMissingPremises`, `Widen`, see Sec. 4.2) If we arrive at the top-level terms and cannot prove those equivalent, the proof fails.

4.1 Using Congruence Rules to Simplify Proofs

While the difference between neighboring programs in the chain – which are more closely related – tends to be small, the size of these programs can still be large. This complicates interactive proofs for the user, and can also slow down automated proofs. To reduce the complexity, we prove the equivalence of subterms and then use congruence rules to derive the equivalence of the full programs. A concrete example of a congruence rule is shown in Fig. 7a.

```
input  : Two FFL terms $P$ and $Q$
output : $true$ if $P$ and $Q$ could be proven equivalent
Premises $\leftarrow \{\}$;
$(P',Q') \leftarrow$ Diff($P$,$Q$);
repeat
    equivalent? $\leftarrow$ ProveEquivalent($P'$,$Q'$,Premises);
    if equivalent? then
        return $true$;
    else
        Premises $\leftarrow$ AddMissingPremises(Premises);
        $(P',Q') \leftarrow$ Widen($P'$,$Q'$);
    end
until $P' = P$ and $Q' = Q$;
return $false$;
```

Alg 1. Strategy for individual equivalence proofs between a pair of FFL programs.

$$\frac{xs \cong_{[\alpha]} ys \qquad i \cong_{\mathrm{Int}} j}{\mathsf{read}(xs, i) \cong_\alpha \mathsf{read}(ys, i)}$$

Fig. 7a: Congruence rule for `read`

$$\begin{aligned}&\texttt{Diff}(\mathsf{fold}(\lambda(x, y).\, x + y, 0, xs),\\ &\qquad \mathsf{fold}(\lambda(x, y).\, y + x, 0, xs))\\ &= (\lambda(x, y).\, x + y, \lambda(x, y).\, y + x)\end{aligned}$$

Fig. 7b: Example of applying `Diff`

We have found that a simple structural comparison (`Diff` in Alg. 1) is well suited for finding smaller subterms whose equivalence implies the equivalence of the full programs. `Diff` computes the smallest two subterms such that replacing them by placeholders results in identical terms. An example of `Diff` can be seen in Fig. 7b.

4.2 Missing Premises and Widening

During the iterative bottom-up process in Alg. 1, P' and Q' may turn out to be non-equivalent in some cases. The strategy then tries to extract required contextual conditions (premises) that are sufficient to ensure equivalence of P' and Q' (`AddMissingPremises`). In the next step, we try to prove the equivalence of enclosing terms (`Widen`), which contain P' and Q' as subterms. Additionally, in the widening-step, we take care of the generated premises. These have either to be shown to always hold in the context of `Widen`(P', Q') or in the context of further widening.

These two steps – premise extraction and widening – are commonly required to prove the equivalence of loop bodies. The example in Fig. 8 illustrates this. Applying `Diff` instantiates P' and Q' with the two loop bodies, as they are the topmost non-equal subterms. A coupling invariant implying that the two loops are started in equivalent states is not sufficient to ensure equivalent loop states after execution since `zip` is only defined for arrays of the same length. Thus, the coupling invariant needs to include the premise that `xs` and `ys` are of equal length.

In some cases, additional premises sufficient for proving equivalence can be found by working backward from missing assumptions in failed proofs. In the example above, proving that the program states are equivalent at the end of each loop iteration assuming that they are equivalent at the beginning will fail due to the missing premise that `xs` and `ys` have the same length. We thus add this premise and try to prove the loop bodies equivalent using that premise. If that is successful, we widen the context to enclosing terms. In the outer context, we attempt to prove that the additional premises are satisfied and derive the equivalence of the full loops based on proved coupling invariant.

4.3 Potential for Automation of Proofs using Rewrite Rules

Since equivalence proofs using rewrite rules are particularly common but also quite repetitive, this section is devoted to their potential for proof automation. A graphical overview of the individual steps can be found in Fig. 9.

1. We perform an approximate matching procedure to generate candidate programs which match the patterns in the rewrite rule.
2. We attempt to prove that these candidates are equivalent to the input programs or otherwise we reject them.
3. We prove that the side conditions hold for these candidates.

By the correctness of the rewrite rule, the candidates are equivalent.

4.3.1 Matching of Rewrite Rules While automatic rewriting systems have been used in the related context of automatically translating imperative algorithms to *MapReduce* algorithms [21], the specific ways in which rewrite rules are used in our approach brings new challenges as well as simplifications.

The challenge lies in the fact that the intermediate programs often do not match the patterns found in rewrite rules directly. There are two typical solutions: normal forms and generalization of patterns. Both are not applicable here. First, there is no suitable normal form of FFL programs. Additionally, both programs are defined by the user, so we cannot assume a specific program structure. Second, the formulation of generalized rewrite rules for matching the large variety of user-defined programs is difficult to obtain and also their correctness proofs are harder to obtain.

The benefit of the programs A, A' (resp. B, B') being provided by the user is that this can reduce ambiguities. In particular, the schematic variables in the two patterns usually overlap to a large degree, but not fully. The matching of the program A against the corresponding pattern can lead to unassigned metavariables, which we need to instantiate with correct choices to prove the equivalence. Now, we have the

```
sum ← $0$;
for $i \leftarrow 0$ to $n-1$ do
    sum ← sum + xs[$i$];
    xs ← F'(xs,ys);
end
```

```
sum ← $0$;
for $i \leftarrow 0$ to $n-1$ do
    zipped ← zip(xs,ys);
    sum ← sum + fst(zipped[$i$]);
    xs ← F(zipped);
end
```

Fig. 8: Two potentially equivalent IL programs operating on two separate arrays (left) and the result of applying `zip` to these arrays (right). `xs`, `ys` are arrays of length n. `F` and `F'` return arrays of the length of their input.

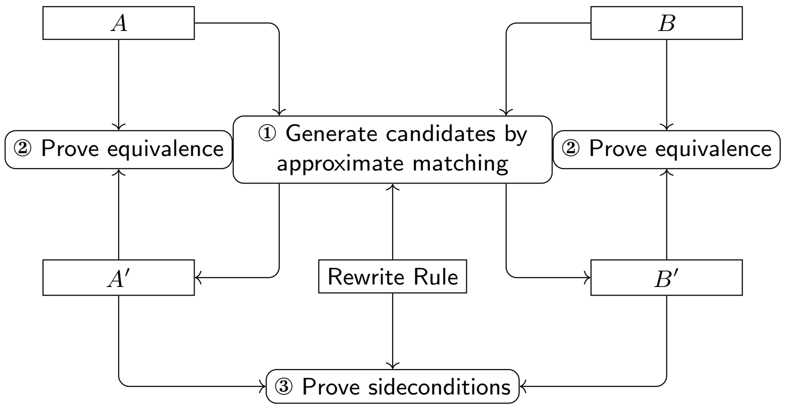

Fig. 9: Workflow for equivalence proofs using rewrite rules. The user has to provide the programs A, A', B, B' and also the rewrite rule. The equivalence proofs ③ are computer-aided in *Coq*.

benefit, that the target program A' is also defined by the user. So, we can obtain missing assignments by matching the other pattern against the other program.

To find the intermediate programs which match the patterns in rewrite rules, an approximate match procedure is used to find assignments for schematic variables in patterns. The approximate matching is an extension of the classical pattern matching with the background knowledge and heuristic of easy-to-prove differences. Applying these assignments to patterns yields candidates for the intermediate program. Once two candidates that match the patterns in a rewrite rule have been identified, it is necessary to prove that (a) the candidates are equivalent to the programs used as the input of the approximate matching procedure, and (b) the side conditions hold and the equivalence of the candidates follows thereby from the correctness of the rewrite rule..

While we have only implemented rudimentary partial automation of the equivalence proof construction, analyzing the *Coq* proofs produced in our experiments has shown that these proofs can be reduced to the correctness of a small number of simple transformations. Proving the correctness of these transformations automatically is feasible and could drastically reduce the need for user interaction.

During our evaluation, one of the most prevalent transformations is *call-by-name beta-reduction* or *lambda abstraction* depending on the direction of the transformation for proving the equivalence (② in Fig. 9). Call-by-name beta-reduction refers to the *beta-reduction* found in programming languages with lazy semantics, which contrary to *call-by-value beta-reduction*, does not evaluate the argument before applying substitution. Since we are working in a call-by-value setting, call-by-name beta-reduction does not always produce an equivalent program. However, the resulting program is equivalent if, for each case where the argument would have been evaluated in the original program, all occurrences in the new program will also be evaluated.

Most other transformations are special cases of constant-folding, e.g., reducing expressions such as $\mathsf{fst}(\mathsf{pair}(a, b))$ to a. Constant-folding does not produce an equivalent program in general if the terms that are being folded are inside the body of a lambda. A sufficient criterion for the resulting program to be equivalent is that the terms being folded are always evaluated.

4.3.2 Proving Side Conditions In Sect. 3.1 we listed the three different kinds of side conditions used in our rewrite rules. The first of those, $x \notin FV(t)$, is purely syntactical and can easily be checked automatically. While proving that a term is not stuck can be difficult in general, in our experiments this could usually be reduced to the term being a value, which again is a syntactical condition. The third kind of side condition states that two arrays have the same length. This can usually be proven recursively by reduction to operations that produce arrays of a specific length, e.g.,

$$\forall n, a, b.\, \mathsf{length}(\mathsf{replicate}(n, a)) = \mathsf{length}(\mathsf{replicate}(n, b)) \ \ ,$$

or to length-preserving operations such as `map`. Note that it can be necessary to strengthen loop invariants to carry this fact through a loop, as explained in Sect. 4.2.

5 Evaluation and Case Study

To demonstrate the feasibility of our approach, we have created a toolchain. The user specifies a sequence of intermediate programs in a simple imperative language. These programs are then automatically translated into a formalization of the previously described functional programming language FFL in *Coq*. In addition to generating proof obligations, our toolchain reduces these obligations using the mentioned structural comparison `Diff`, and it applies congruence rules to reconstruct an equivalence proof of the full programs.

Using this toolchain, we have proven the equivalence of imperative and *MapReduce*s implementations of the *PageRank* algorithm [5] and the *k-means* [20] algorithm in *Coq*. An extensive description of the *PageRank* example including all intermediate programs can be found in [3]. Fig. 10 shows the imperative and the *MapReduce* implementation of *PageRank* that we have used in our experiments. The *MapReduce* implementation of *PageRank* shown here is deterministic when executed on rational numbers due to commutativity and associativity of addition. However, the algorithm is not deterministic when executed using floating point numbers since addition is not associative in this case. We have not attempted the verification of algorithms based on floating point numbers in this work.

While we have created the imperative implementations ot the two algorithms ourselves, the *MapReduce* versions are very close to the implementations accompanying the Thrill [4] framework. This reinforces our claim that FFL is capable of representing *MapReduce* algorithms and is thereby suitable for this approach. In total, the formalization of FFL, the rewrite rules, and proofs of various properties, encompasses about 8000 lines of Coq code. The equivalence proofs of *PageRank* and *k-means* each require about 3700 lines of *Coq* proofs. That includes the automatically generated translation of the chain of equivalent programs (for *k-means* this chain consists of 9 programs while for *PageRank* it consists of 6 programs), which take up large parts of these proofs. The proofs rely on the rewrite rules which we have formalized in *Coq* as well as coupling predicates.

```
Function PageRank(links, numLinks, n)
begin
    ranks ←
        Replicate(numLinks, $\frac{1}{numLinks}$);
    for i = 1 to n do
        ranks' ← Replicate(numLinks, 0);
        for p = 0 to numLinks − 1 do
            contrib ← $\frac{ranks[p]}{Length(links[p])}$;
            foreach q ← links[p] do
                ranks'[q] ←
                    ranks'[q] + contrib;
            end
        end
        for p = 0 to numLinks − 1 do
            ranks[p] ←
                Dampen(ranks'[p], numLinks);
        end
    end
    return ranks;
end
```

```
Function PageRank(links, numLinks, n)
begin
    ranks ←
        Replicate(numLinks, $\frac{1}{numLinks}$);
    for i = 1 to n do
        outRanks ← Zip(links,ranks);
        contribs ←
            FlatMap(
                λ(ls, r).
                    Map(λl.(l, $\frac{r}{Length(ls)}$), ls),
                outRanks);
        rankUpdates ←
            Reduce(+, 0, contribs);
        ranks' ← Replicate(numLinks, 0);
        foreach (l, r) ← rankUpdates do
            ranks'[l] ← r;
        end
        ranks ←
            Map(λr. Dampen(r, numLinks),
                ranks');
    end
    return ranks;
end
```

Fig. 10: Imperative (left) and *MapReduce* (right) versions of the *PageRank* algorithm (the function `Replicate(n,v)` creates an array of length `n` with all elements set to `v`; and `Dampen` is an arbitrary function).

6 Related Work

A common approach to relational verification and program equivalence is the use of product programs [1]. Product programs combine the states of two programs and interleave their behavior in a single program. *RVT* [13] proves the equivalence of C programs by combining them in a product program. By assuming that the program states are equal after each loop iteration, *RVT* avoids the need for user-specified or inferred loop invariants and coupling predicates.

Hawblitzel et al. [15] use a similar technique for handling recursive function calls. Felsing et al. [12] demonstrate that coupling predicates for proving the equivalence of two programs can often be inferred automatically. While the structure of imperative and *MapReduce* algorithms tends to be quite different, splitting the translation into intermediate steps yields programs which are often structurally similar. We have found that in this case, techniques such as coupling predicates arise naturally and are useful for selected parts of an equivalence proof.

Radoi et al. [21] describe an automatic translation of imperative algorithms to *MapReduce* algorithms based on rewrite rules. While the rewrite rules are very similar to the ones used in our approach, we complement rewrite rules by coupling predicates. Furthermore, we are able to prove equivalence for algorithms for which the automatic translation from Radoi et al. is not capable of producing efficient *MapReduce* algorithms. The objective of verification imposes different constraints than the automated translation – in particular, both programs are provided by the user, so there is less flexibility needed in the formulation of rewrite rules.

Chen et al. [7] and Radoi et al. [21] describe languages and sequential semantics for *MapReduce* algorithms. Chen et al. describe an executable sequential specification in the Haskell programming language focusing on capturing non-determinism correctly. Radoi et al. use a language based on a lambda calculus as the common representation for the previously described translation from imperative to *MapReduce* algorithms. While this language closely resembles the language used in our approach, it lacks support for representing some imperative constructs such as arbitrary *while*-loops.

Grossman et al. [14] verify the equivalence of a restricted subset of Spark programs by reducing the problem of checking program equivalence to the validity of formulas in a decidable fragment of first-order logic. While this approach is fully automatic, it limits programs to Presburger arithmetic and requires that they are synchronized in some way.

To the best of our knowledge, we are the first to propose a framework for proving equivalence of *MapReduce* and imperative programs.

7 Conclusion

We have presented a new approach for proving the equivalence of imperative and *MapReduce* algorithms. This approach relies on splitting the transformation into a chain of intermediate programs. The individual equivalence proofs are then categorized in context-independent and context-dependent transformations. Equivalence proofs for context-independent transformations are handled using rewrite rules, while equivalence proofs for context-dependent transformations are based on coupling predicates. We have demonstrated the feasibility of end-to-end equivalence proofs using this approach by applying it two well-known non-trivial algorithms.

While we have hinted at the potential for automating this approach, implementing automation is left as future work. In particular, it would be interesting to explore whether existing tools for relational verification using coupling predicates can be used or if new tools are necessary. Further future work includes extending the approach presented here to support the full expressiveness provided by languages which are used to implement imperative and *MapReduce* algorithms.

References

1. Barthe, G., Crespo, J.M., Kunz, C.: Relational Verification Using Product Programs, pp. 200–214. Springer Berlin Heidelberg (2011)
2. Beckert, B., Bingmann, T., Kiefer, M., Sanders, P., Ulbrich, M., Weigl, A.: Relational Equivalence Proofs Between Imperative and MapReduce Algorithms. ArXiv e-prints (Jan 2018), `https://arxiv.org/abs/1801.08766`
3. Beckert, B., Bingmann, T., Kiefer, M., Sanders, P., Ulbrich, M., Weigl, A.: Proving Equivalence Between Imperative and MapReduce Implementations Using Program Transformations. In: Third Workshop Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation. Electronic Proceedings in Theoretical Computer Science, vol. 268, pp. 185–199. Open Publishing Association (2018)
4. Bingmann, T., Axtmann, M., Jöbstl, E., Lamm, S., Nguyen, H.C., Noe, A., Schlag, S., Stumpp, M., Sturm, T., Sanders, P.: Thrill: High-performance algorithmic distributed batch data processing with C++. In: IEEE International Conference on Big Data. pp. 172–183. IEEE (Dec 2016), preprint arXiv:1608.05634

5. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Comput. Netw. ISDN Syst. 30(1-7), 107–117 (Apr 1998), `http://dx.doi.org/10.1016/S0169-7552(98)00110-X`
6. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R.R., Bradshaw, R., Weizenbaum, N.: FlumeJava: Easy, efficient data-parallel pipelines. ACM SIGPLAN Notices 45(6), 363–375 (2010)
7. Chen, Y.F., Hong, C.D., Lengál, O., Mu, S.C., Sinha, N., Wang, B.Y.: An Executable Sequential Specification for Spark Aggregation (2017), `https://arxiv.org/abs/1702.02439`
8. Chen, Y.F., Hong, C.D., Sinha, N., Wang, B.Y.: Commutativity of Reducers, pp. 131–146. Springer (2015)
9. Chen, Y., Song, L., Wu, Z.: The Commutativity Problem of the MapReduce Framework: A Transducer-based Approach. CoRR abs/1605.01497 (2016), `http://arxiv.org/abs/1605.01497`
10. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008)
11. Elenbogen, D., Katz, S., Strichman, O.: Proving mutual termination. Form. Methods Syst. Des. 47(2), 204–229 (Oct 2015), `http://dx.doi.org/10.1007/s10703-015-0234-3`
12. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 349–360. ASE '14, ACM, New York, NY, USA (2014)
13. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference. pp. 466–471. DAC '09, ACM, New York, NY, USA (2009)
14. Grossman, S., Cohen, S., Itzhaky, S., Rinetzky, N., Sagiv, M.: Verifying Equivalence of Spark Programs, pp. 282–300. Springer International Publishing, Cham (2017)
15. Hawblitzel, C., Kawaguchi, M., Lahiri, S., Rebêlo, H.: Mutual summaries: Unifying program comparison techniques. In: Informal proceedings of BOOGIE 2011 workshop (2011), `https://www.microsoft.com/en-us/research/publication/mutual-summaries-unifying-program-comparison-techniques/`
16. Kahn, G.: Natural semantics. STACS 87 pp. 22–39 (1987)
17. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR – combining static verification and dynamic analysis. Journal of Automated Reasoning (Sep 2017)
18. Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Journal on Formal Methods in System Design (Aug 2017)
19. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SymDiff: A language-agnostic semantic diff tool for imperative programs. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 712–717. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-31424-7_54`
20. Lloyd, S.: Least squares quantization in PCM. IEEE Transactions on Information Theory 28(2), 129–137 (1982), `https://doi.org/10.1109/TIT.1982.1056489`
21. Radoi, C., Fink, S.J., Rabbah, R., Sridharan, M.: Translating Imperative Code to MapReduce. SIGPLAN Not. 49(10), 909–927 (Oct 2014)
22. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), `http://coq.inria.fr`, version 8.6
23. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. ACM Trans. Program. Lang. Syst. 34(3), 11:1–11:35 (Nov 2012), `http://doi.acm.org/10.1145/2362389.2362390`

24. White, T.: Hadoop: The definitive guide. O'Reilly Media, Inc. (2012)
25. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1863103.1863113