Proving Equivalence Between Imperative and MapReduce Implementations Using Program Transformations

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

Institute of Theoretical Informatics
Karlsruhe Institute of Technology, Germany

Distributed programs are often formulated in popular functional frameworks like *MapReduce*, *Spark* and *Thrill*, but writing efficient algorithms for such frameworks is usually a non-trivial task. As the costs of running faulty algorithms at scale can be severe, it is highly desirable to verify their correctness.

We propose to employ existing imperative reference implementations as specifications for *MapReduce* implementations. To this end, we present a novel verification approach in which equivalence between an imperative and a *MapReduce* implementation is established by a series of program transformations.

In this paper, we present how the equivalence framework can be used to prove equivalence between an imperative implementation of the *PageRank* algorithm and its *MapReduce* variant. The eight individual transformation steps are individually presented and explained.

1   Introduction

Today requirements on the efficiency and scale of computations grow faster than the capabilities of the hardware on which they are to run. Frameworks such as *MapReduce* [6], Spark [14] and Thrill [3] that distribute the computation workload amongst many nodes in a cluster, address these challenges by providing a limited set of operations whose execution is automatically and transparently parallelised and distributed among the available nodes.

In this paper, we use the term "*MapReduce*" as a placeholder for a wider range of frameworks. While some frameworks such as Hadoop's MapReduce [13] strictly adhere to the two functions "map" and "reduce", the more recent and widely used distribution frameworks provide many additional primitives – for performance reasons and to make programming more comfortable.

Formulating efficient implementations in such frameworks is a challenge in itself. The original algorithmic structure of a corresponding imperative algorithm is often lost during that translation since imperative constructs do not translate directly to the provided primitives. Significant algorithmic design effort must be invested to come up with good *MapReduce* implementations, and flaws are easily introduced during the process.

The approach we proposed in our previous work [2] and will refine and apply in this paper supports algorithm engineers in the correct design of *MapReduce* implementations by providing a transformation framework with which it is possible to interactively and iteratively translate a given imperative implementation into an efficient one that operates within a *MapReduce* framework.

The framework is thus a verification framework to prove the *behavioural equivalence* between an imperative algorithm and its *MapReduce* counterpart. Due to often considerable structural
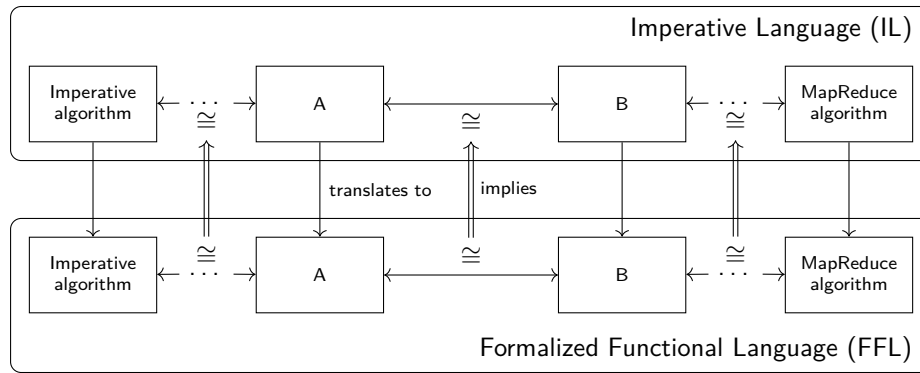
Figure 1: Chain of equivalent programs is translated into formalised functional language

differences between the two programming paradigms, our approach is interactive: It requires the specification of intermediate programs to guide the translation process. While it is not in the focus of this publication, our approach is designed to have a high potential for automation: The required interaction is designed to be as high-level as possible. The rules are designed such that their side conditions can be proved automatically, and pattern matching can be used to allow for a more flexible specification of intermediate steps.

We present an approach based on program transformation rules with which a *MapReduce* implementation of an algorithm can be proved equivalent to an imperative implementation. From an extensive analysis of the example set of the framework Thrill, we were able to identify a set of 13 transformation rules such that a chain of rule applications from this set is likely to succeed in showing the equivalence between an imperative and a functional implementation.

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* [12].

The main contribution of this paper is the demonstration of the abilities of the framework to establish equivalence between imperative and *MapReduce* implementations. We do this (1) by motivating and thoroughly explaining the rationales and the nature of the transformation rules and (2) by reporting on the successful application of the framework to a relevant non-trivial case study. We have chosen the *PageRank* algorithm as the demonstrative example since it is one of the original and best known *MapReduce* application cases.

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 the two algorithms. To deal with this, 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 neighbouring programs in this chain, the difference is comparatively small such that the pair can be used as start and end point of a single program transformation step induced by a rule application.

The approach uses two programming languages: One is the imperative input language (IL) in which the imperative algorithm, the intermediate programs, as well as the target *MapReduce*

implementation are stated. Besides imperative language constructs, IL supports *MapReduce* primitives.

Each program specified in the high-level imperative language is then automatically translated into the formalised functional language (FFL). The program transformations and equivalence proofs operate on programs in this functional language. The translation of the original, the intermediate and the *MapReduce* programs form a chain of programs. For each pair of neighbouring programs in the chain, a proof obligation is generated that requires proving their equivalence. These proof obligations are then discharged independently of each other. 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. An overview of this process can be seen in Fig. 1. 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 required – besides providing the intermediate programs. To enable this automation, one of the design goals of the approach was to make the matching of the rewrite rules as flexible as possible. Further challenges include the extension of our approach to features such as references and aliasing which are commonly found in imperative languages.

Structure of the paper. The remainder of the paper is structured as follows: After introducing the supported programming languages IL and FFL in Sect. 2, a description of the two different kinds of transformation rules applied in the framework follows in Sect. 3. The case study on the *PageRank* example is conducted in Sect. 4. After a report on related work in Sect. 5, the paper is wrapped up with conclusions in Sect. 6.

2 Foundations

This section introduces the programming language IL used to formulate the programs, and briefly describes the language FFL used for the proofs.

The high-level imperative programming language IL is based on a while language. Besides the usual `while` loop constructor, it possesses a `for` loop constructor which allows traversal of the entries of an array. The supported types are integers (`Int`), rationals[1] (`Rat`), Booleans (`Bool`), fixed length arrays (`[T]`) and sum (T_1 `+` T_2) and product types (T_1 `*` T_2). Since arrays are an important basic data type for *MapReduce*, a number of functions are provided to operate on this data type. Table 1 lists the IL-functions relevant for this paper. Besides the imperative language constructs, IL supports a number of *MapReduce* primitives. In particular, a lambda abstraction for a variable v of type T over an expression e can be used and is written as ($v : T$) `=>` e. IL does not support recursion.

Given that allegedly *MapReduce* programs tend to be more of a functional than an imperative nature, it might seem odd that we use IL also for specifying the *MapReduce* algorithm and not a functional language. However, most existing *MapReduce* frameworks are not implemented as separate languages, but as frameworks built on top of imperative languages. This implies

---

[1]In the current implementation, rationals are implemented using integers with the operators being uninterpreted function symbols.

Function	Explanation
replicate(n, x)	returns an array of length n whose entries hold value x.
range(a, b)	returns an array containing the values $\langle a, \ldots, b-1 \rangle$.
zip(xs, ys)	returns for two arrays of equal length an array of pairs containing a value of each array.
map(f, xs)	returns an array of the same length as xs that contains the result of applying function f to the values in xs.
fst(p), snd(p)	returns the first (second) component of a pair p of values.
group(xs)	transforms a list of key-value pairs into a list of pairs of a key and a list of all values associated with that key.
concat(xss)	returns the concatenation of all arrays in the array of arrays xss.
flatMap(f, xss)	= map(f, concat(xss))
reduceByKey(f, i, xs)	= map((k,vs) => (k, fold(f,i,vs)), group(xs))

Table 1: Relevant built-in functions of IL.

```
fn SumArrays(xs: [Int], ys: [Int]) {          fn SumArraysZipped(xs: [Int], ys: [Int]) {
  var sum := replicate(n, 0);                     var sum := replicate(n, 0);
  for(i : range(0, length(xs))) {                 zipped := zip(xs, ys);
    sum[i] := xs[i] + ys[i];                      for(i : range(0, length(xs))) {
  }                                                  sum[i] := fst(zipped[i]) + snd(zipped[i]);
  return sum;                                     }
}                                                 return sum;
                                                }
```

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

that the sequential imperative constructs of the host language can also be found in *MapReduce* programs. Sequential parts of *MapReduce* algorithms are realised using imperative programming features, while the computational, distributed parts are composed using the *MapReduce* primitives. Figure 2 shows two behaviourally equivalent implementations of a routine that computes the sum of the entries of two Int-arrays.

The programs specified in IL are then automatically translated into FFL, the functional language based on λ-calculus in which the equivalence proofs by program transformation are conducted. We follow the work by Radoi et al. [11] and use a simply typed lambda calculus extended by the theories of sums, products, and arrays. Moreover, to allow the translation of both imperative and *MapReduce* IL code into FFL, the language also contains constructs for loop iteration and the programming primitives usually found in *MapReduce* frameworks. FFL is formally defined as a theory in the theorem prover *Coq* which allowed us to conduct correctness proofs on the rewrite rules.

Without going into the intricacies of the details of the translation between the formalisms, the idea is as follows: Any IL statement becomes a (higher order) term in FFL in which the currently available local variables *acc* make up the λ-abstracted variables. The two primitives iter and fold serve as direct translation targets for imperative loops. The fold function is used to translate bounded for loops into FFL. The iterator loop for(x : xs) { f } in IL is translated into FFL as the expression $\lambda acc.\, \mathsf{fold}(\hat{f}, acc, \hat{xs})$ in which $\hat{f}$ and $\hat{xs}$ are the FFL-translations of f and xs. This starts with the initial loop state *acc* and iterates over each value of the array $\hat{xs}$ updating the loop state by applying $\hat{f}$. The more general while loops cannot be translated using fold since that always has bounded number of iterations. Instead, while is translated using the iter fixed point

operator. The loop `while(c) { f }` translates as $\mathsf{iter}(\lambda acc.\ \mathsf{if}\ c(acc)\ \mathsf{then}\ \mathsf{inr}\ \hat{f}(acc)\ \mathsf{else}\ \mathsf{inl}\ \mathsf{unit})$ and is evaluated by repeatedly applying $\hat{f}$ to the loop state until $\hat{f}$ returns unit to indicate termination. iter is a partial function; if the loop does not terminate it does not yield a value. The semantics of `FFL` is defined as a bigstep semantics. A program together with its input parameters are reduced to a value. Ill-typed or non-terminating programs do not reduce to a value. Details on the design of `IL` and `FFL` can be found elsewhere [2].

The design of `FFL`'s core follows the ideas of Chen et al. [4] who describe how to reduce the large number of primitives provided by *MapReduce* frameworks.

3 Transformation Rules

For the examples shipped with the *MapReduce* framework Thrill, we analysed how the steps of a transformation of an algorithm into *MapReduce* would look and detected typical recurring patterns for steps in a transformation. We were able to identify two different categories of transformation rules that are needed for the process:

1. Local, *context-independent rewrite rules* with which a statement of the program can be replaced with a semantically equivalent statement. Such rules may have side conditions on the parts on which they apply, but they cannot restrict the application context (the statements enclosing the part to be exchanged).

2. *Context-dependent equivalence rules* that cannot replace a statement without considering the context. Some transformations are only valid equivalence replacements within their surrounding context. These are not pattern-driven rewrite rules, but follow a deductive reasoning pattern that proves equivalence of a piece of code locally.

Context-independent rules are good for changing the control flow of the program. In Sect. 4.2, we will encounter an example of a rule which replaces a loop by an equivalent `map` expression. The data flow, while differently processed, remains the same. The context-independent rules are a powerful tool to bridge larger differences in the control structure between the two programming paradigms. These changes must be anticipated beforehand and cannot be detected and proved on the spot. We identified a total of 13 rules that allow us to transform imperative constructs into *MapReduce* primitives. (See App. B of [2] for a complete list.) Their rigid search patterns make context-independent rules less flexible in their application.

Context-dependent rules, on the other hand, are suited for transforming a program into a structurally similar program (they do not/little change the control flow); the data flow may be altered however using such rules. It is comprehensible that a change in the data representation is an aspect which cannot be considered locally, but requires analysing the whole program.

The collection of rewrite rules for context-independent replacements comprises various different patterns. The context-dependent case is different. There exists one single rule which can be instantiated for arbitrary coupling predicates and is thereby highly adaptable. We employ relational reasoning using product programs [1] to show connecting properties. The rule bases on the observation that loops in the two compared programs need not be considered separately but can be treated simultaneously. If x_1 (x_2) are the variables of the first (second) program, and if the conditions c_1 and c_2, as well as the loop bodies b_1 and b_2 refer only to variables of the respective program, then the validity of the Hoare triple

$$\{x_1 = x_2\}\ \mathtt{while}(c_1)\ \{\ b_1\ \}\ ;\ \mathtt{while}(c_2)\ \{\ b_2\ \}\ \{x_1 = x_2\} \tag{1}$$

is implied by the validity of

$$\{x_1 = x_2\} \ \texttt{while}(c_1) \ \{ \ b_1; \ b_2; \ \texttt{assert} \ c_1\texttt{=}c_2; \ \} \ \{x_1 = x_2\} \ . \tag{2}$$

Condition (1) expresses that given equal inputs, the two loop programs terminate in the same final state. Condition (2) manages to express[2] this with a single loop. This gives us the chance to prove equivalence using a single *coupling invariant* that relates to both program states. To show equivalence for context-dependent cases, the specification of a relational coupling invariant with which the consistency of the data can be shown is required.

An example of two programs which are equivalent with similar control structure, yet different data representation, has already been presented in Fig. 2. Both programs can be shown equivalent by means of the coupling invariant $\texttt{sum}_1 = \texttt{sum}_2 \wedge \texttt{zipped}_2 = \texttt{zip}(xs_1, ys_1)$ where the subscripts indicate which program a variable belongs to. Sect. 4.3 demonstrates the application of the rule within the *PageRank* example.

In the formal approach (also outlined in [2]) and the *Coq* implementation, all rules are formulated on and operate on the level of FFL (which has been designed for exactly this purpose). For the sake of better readability we show the corresponding rules here on the level of IL and notate the context independent rewrite rules as

$$\texttt{program}_1 \quad \leadsto \quad \texttt{program}_2 \qquad (\textit{rulename})$$

meaning that under the specified side conditions any statement in a program matching $\texttt{program}_1$ can be replaced by the corresponding instantiation of $\texttt{program}_2$, yielding a program behaviourally equivalent to original program.

The application of rules are transformations only in a broader sense. The approach is targeted as interacting with a user who provides the machine with the intermediate steps and the rule scheme to be applied. It would likewise be possible to instead allow specifying which rules must be applied and have the user specify the instantiations instead of the resulting intermediate programs.

In particular the context-dependent rule is hardly a rewrite rule due to the deductive nature of the equivalence proof. It is not astonishing though that the transformation into *MapReduce* does not work completely by means of a set of rewrite patterns since transforming imperative programs to *MapReduce* programs is more than simple pattern matching process but requires some amount of ingenious input to come up with an efficient implementation.

4 Example: PageRank

In this section, we demonstrate our approach by applying it to the *PageRank* algorithm. We present all intermediate programs in IL and explain the transformations and techniques used in the individual steps. While the implementation executes the actual equivalence proofs on FFL terms, we only present the IL programs here since these are the intermediate steps specified by the user. Where the translation of a transformation to FFL is not straightforward, we also give an explanation of the transformation expressed on FFL terms.

---

[2] (2) is stronger than (1) in general, but is equivalent in case both loops are guaranteed to have the same number of iterations.

4.1   The algorithm

PageRank is the algorithm that Google originally successfully employed to compute the rank of web pages in their search engine. This renowned algorithm is actually a special case of sparse matrix-vector multiplication, which has much broader applications in scientific computing. *PageRank* is particularly well suited as an example for a map reduce implementation and is included in the examples of Thrill and Spark. While more highly optimized *PageRank* algorithms and implementations exist, we present here a simplified version.

The idea behind *PageRank* is the propagation of reputation of a web page amongst the pages to which it links. If a page with a high reputation links to another page, the latter page's reputation thus increases.

The algorithm operates as follows: *PageRank* operates on a graph in which the nodes are the pages of the considered network, and (directed) edges represent links between pages. Pages are represented as integers, and the 2-dimensional array `links` holds the edges of the graph in the form of an adjacency list: the i-th component of `links` is an array containing all indices of the pages to which page i links. The result is an array `ranks` of rationals that holds the pagerank value of every page. The initial rank is set to $Rank_0(p) = \frac{1}{|\texttt{links}|}$ for all pages p.

In the course of the k-th iteration ($k > 0$) of the algorithm, the rank of each link target is updated depending on the pages that link *to* the page, i.e., the incoming edges in the graph:

$$\Delta_k(p) = \sum_{(o,p)\in\texttt{links}} Rank_{k-1}(o) \qquad\qquad Rank_k(p) = \delta * \Delta_k(p) + \frac{1-\delta}{|\texttt{links}|} \qquad (3)$$

The factor $\delta \in (0,1)$ is a dampening factor to limit the effects of an iteration by weighting the influence of the result of the iteration $\Delta_k(p)$ against the original value $Rank_0(p) = \frac{1}{|\texttt{links}|}$. Our implementation iterates this scheme for a fixed number of times (`iterations`).

Listing 1 on page 194 shows a straightforward imperative IL implementation of this algorithm that realises the iterative laws of (3) directly. It marks the starting point of the translation from imperative to *MapReduce* algorithm. To allow a better comparison between the programs, the programs are listed next to each other at the end of this section.

4.2   A context-independent rule application

The first step in the chain of transformations from imperative to distributed replaces the `for` loop used to calculate the weighted new ranks with a call to `map`. This is possible since the values can be computed independently. The `map` expression allows computing the dampened values in parallel and can thereby significantly improve performance. The rewrite rule used here can be applied to all `for` loops that iterate over the index range of an array where each iteration reads a value from one array at the respective index, applies a function to it and then writes the result back to another array at the same index.

```
for (i : range(0, length(xs))) {
  ys[i] := f(xs[i]);                    ⤳        ys := map(f, xs);              (map-introduce)
}
```

Sufficient conditions for the validity of this context-independent transformation are that `f` does not access the index `i` directly and that `xs` and `ys` have the same length. The first condition can be checked syntactically while matching the rule while the second requires a (simple) proof in the context of the rule application. In our implementation, these proofs are executed in *Coq*.

As mentioned before, `for` loops in IL correspond to `fold` operations in FFL. The rewrite rule expressed on FFL thus transforms $\mathsf{fold}(\lambda acc\ i.\ acc[i := f(xs[i])], ys, \mathsf{range}(0, \mathsf{length}(xs)))$ into $\mathsf{map}(f, xs)$.

The result of the transformation is shown in Listing 2 on page 194. For convenience, in this and the following listings, the modified part of the program is highlighted in colour.

4.3   A context-dependent rule application

In this step, the body of the main `while` loop is changed to first combine the `links` and `ranks` arrays to an array `outRanks` of tuples using the `zip` primitive. In the remaining loop body, all references to `links` and `ranks` point to this new array and retrieve the original values using the pair projection functions `fst` and `snd`. The process of rewriting all references does not fit easily into the rigid structure of the rewrite rules employed in our approach. We thus resort to using a context-dependent rule using a coupling predicates to prove equivalence of the last and the new loop body. Using the coupling predicate

$$\mathtt{newRanks}_1 = \mathtt{newRanks}_2\ \wedge\ \mathtt{outRanks}_2 = \mathtt{zip}(\mathtt{links}_1, \mathtt{ranks}_1)$$

that relates the values in the states of the two programs (we use the subscript indices 1 and 2 to refer to variables in the original and the transformed program) we obtain that the loop bodies have equivalent effects, and hence, that the programs are equal.

The result of the transformation is shown in Listing 3 on page 195.

4.4   Rule *range-remove*

In the next transformation the `for` loop which iterates over all pages as the index range of the array `links` is replaced by a `for` loop that iterates directly over the elements in `outRanks`. The rewrite rule *range-remove* applied here can be applied to all `for` loops that iterate over the index range of an array and only use the index to access these array elements. Again this is a side condition for the rule which can be checked syntactically during rule matching.

```
acc := acc0;                              acc := acc0;
for (i : range(0, length xs)) {           for (x : xs) {                    (range-remove)
  acc := f(acc, xs[i]);                     acc := f(acc, x);
}                                         }
```

The result of the application of rewrite rule is shown in Listing 4 on page 195.

Expressed on the level of FFL, this rewrite rule transforms terms of the form $\mathsf{fold}(\lambda acc\ i.\ f(acc, xs[i]), acc_0, \mathsf{range}(0, \mathsf{length}(xs)))$ into $\mathsf{fold}(\lambda acc\ x.\ f(acc, x), acc_0, xs)$.

4.5   Aggregating link information

The next step is a typical step that can be observed when migrating algorithms into the *MapReduce* programming model. A computation is split into two consecutive steps: one step processing data locally on individual data points and one step aggregating the results. It can be anticipated already now that these two steps will become the *map* and the *reduce* part of the algorithm.

The newly introduced variable `linksAndContrib` stores the (locally for each node) computed rank contribution as a list of tuples. Assume $(\langle s_1, \ldots, s_n \rangle, r)$ is the i-th entry in the array `outRanks`. This means that page i links to page s_j for $j < n$ and has a current rank of r. After the

newly introduced local computation, the entry becomes the list of pairs $\langle(s_1, \frac{r}{|\texttt{links}|}), \ldots, (s_n, \frac{r}{|\texttt{links}|})\rangle$, i.e., the rank is distributed to all successor pages and the data is rearranged with the focus now on the receiving pages.

As in the transformation in Sect. 4.3, a context-dependent transformation is employed to prove equivalence using the following relational coupling loop invariant:

$$\texttt{newRanks}_1 = \texttt{newRanks}_2 \ \wedge$$
$$\forall ij.\ \texttt{fst linksAndContrib}_2[i][j] = (\texttt{fst outRanks}_1[i])[j] \ \wedge$$
$$\texttt{snd linksAndContrib}_2[i][j] = \texttt{snd outRanks}_1[i]/\texttt{length}(\texttt{fst outRanks}_1[i])$$

The result of the transformation is shown in Listing 5 on page 195. Note that the nested loops in the highlighted block no longer perform the computation of the rank updates (`snd links_rank / length(fst links_rank)`), but only aggregate the contribution updates into new ranks. This transformation is a preparation for collapsing the nested loops in the next step.

4.6 Collapsing nested loops

Since the computation of `contribution` has been moved outside in the previous step, the iteration variable `link_contribs` is now only used as the iterated array in the inner `for` loop. This allows collapsing the nested loops into a single loop using `concat`. This rule can always be applied if the iterated value in the inner `for` is the only reference to the values the outer `for` iterates over.

```
acc := acc0;
for (xs : xss) {
  for (x : xs) {
    acc := f(acc, x);
  }
}
```
⤳
```
acc := acc0;
for (x : concat(xss)) {
  acc := f(acc, x);
}
```
(*concat-intro*)

The program with the two loops collapsed is shown in Listing 6 on page 196.

This transformation is succeeded by a step that combines the call to `concat` in the `for` loop and the `map` operation before the loop into a single call to `flatMap`. Its result is shown in Listing 7 on page 196. In FFL, `flatMap` is not a builtin primitive but a synonym for successive calls to `concat` and `map`. This step is thus one which has visible effects on the level of IL, but no impact on the level of FFL.

4.7 Towards *MapReduce*

Now we are getting closer to a program that adheres to the *MapReduce* programming model. The penultimate transformation step restructures the processed data by grouping all rank updates that affect the same page. It operates on the array `newRanks` using the function `group`. The updated result is calculated using a combination of `map` and `fold`. The results are then written back to `newRanks`. The effects of the rule on the program structure are more severe than for the other applied transformation rules, yet this grouping pattern is one that is typically observed in the *MapReduce* transformation process and is implemented as a single rule for that reason.

The corresponding rewrite rule can be applied to all `for` loops that iterate over an array containing index-value tuples and update an accumulator based on the old value stored for that

index and the current value:[3]

```
acc := acc0;                          acc := acc0;
for ((i,v) : xs) {                    var upd := map((i,vs) => fold(f, acc[i], vs),
  acc[i] := f(acc[i], v);                               group(acc));
}                          ⤳          for (x : concat(xss)) {                    (group-intro)
                                        acc := f(acc, x);
                                      }
```

Note that since `acc0` could store values for indices for which there are no corresponding tuples in `xs`, it is necessary to write the results back to that array instead of simply using the result from the `group` operation which would be missing those entries.

The resulting program is shown in Listing 8 on page 196.

4.8   The final *MapReduce* implementation

In the last step, the expression that groups contributions by index and then sums them up is replaced by the IL-function `reduceByKey` which is also provided by many *MapReduce* frameworks. In the lower level language FFL, however, `reduceByKey` is not a primitive function, but a composed expression using map, fold and group, such that this step changes the IL program, but has no impact on the FFL level. The resulting implementation using map reduce constructs is shown in Listing 9 on page 197. It is very close to the *MapReduce* implementation of *PageRank* that is delivered in the example collection of the Thrill framework.

Listing 1: Original imperative IL implementation of *PageRank*

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1. / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    for (pageId : range(0, length(links))) {
      var contribution : Rat := ranks[pageId] / length(links[pageId]);
      for (outgoingId : links[pageId]) {
        newRanks[outgoingId] := newRanks[outgoingId] + contribution;
      }
    }
    for (pageId : range(0, length(links))) {
      ranks[pageId] :=
        dampening * newRanks[pageId] + (1 - dampening) / length(links);
    }
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 2: *PageRank* – After applying rule *map-introduce*

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1. / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    for (pageId : range(0, length(links))) {
      var contribution : Rat := ranks[pageId] / length(links[pageId]);
      for (outgoingId : links[pageId]) {
        newRanks[outgoingId] := newRanks[outgoingId] + contribution;
      }
    }
    ranks :=
```

---

[3]The actually implemented version of the rule allows `f` to access not only the values `vs`, but also the index `i` it operates on. See [2] for details.

```
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 3: *PageRank* – After applying a context-dependent rule

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    for (pageId : range(0, length(links))) {
      var contribution : Rat := snd outRanks[pageId] / length(fst outRanks[pageId]);
      for (outgoingId : fst outRanks[pageId]) {
        newRanks[outgoingId] := newRanks[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 4: *PageRank* – After applying *range-remove*

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    for (links_rank : outRanks) {
      var contribution : Rat := snd links_rank / length(fst links_rank);
      for (outgoingId : fst links_rank) {
        newRanks[outgoingId] := newRanks[outgoingId] + contribution;
      }
    }
    ranks :=
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 5: *PageRank* – After aggregating the link information

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [[Int * Rat]] :=
      map((links_rank : [Int] * Rat) =>
            map((link : Int) =>
                  (link, snd links_rank / length(fst links_rank)),
                fst links_rank),
          outRanks);
    for (link_contribs : linksAndContrib) {
      for (link_contrib : link_contribs) {
        newRanks[fst link_contrib] :=
          newRanks[fst link_contrib] + snd link_contrib;
      }
```

```
  }
  ranks :=
    map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
        newRanks);
  iter := iter + 1;
  }
  return ranks;
}
```

Listing 6: *PageRank* – After collapsing nested loops

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [[Int * Rat]] :=
      map((links_rank : [Int] * Rat) =>
            map((link : Int) =>
                  (link, snd links_rank / length(fst links_rank)),
                fst links_rank),
          outRanks);
    for (link_contrib : concat(linksAndContrib)) {
      newRanks[fst link_contrib] :=
        newRanks[fst link_contrib] + snd link_contrib;
    }
    ranks :=
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 7: *PageRank* – After introducing `flatMap`

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var newRanks : [Rat] := replicate(length(links), 0);
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var linksAndContrib : [Int * Rat] :=
      flatMap((links_rank : [Int] * Rat) =>
            map((link : Int) =>
                  (link, snd links_rank / length(fst links_rank)),
                fst links_rank),
          outRanks);
    for (link_contrib : linksAndContrib) {
      newRanks[fst link_contrib] :=
        newRanks[fst link_contrib] + snd link_contrib;
    }
    ranks :=
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 8: *PageRank* – After grouping the input for receiving pages

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var contribs : [Int * Rat] :=
```

```
        flatMap((links_rank : [Int] * Rat) =>
                 map((link : Int) => (link,
                                       snd links_rank / length(fst links_rank)),
                      fst links_rank),
              outRanks);
    var rankUpdates : [Int * Rat] :=
      map((link : Int) (contribs : [Rat]) =>
           (link, fold((x: Rat) (y : Rat) => x + y, 0, contribs)),
         group(contribs));
    var newRanks : [Rat] := replicate(length(links), 0);
    for (link_rank : rankUpdates) {
      newRanks[fst link_rank] := snd link_rank;
    }
    ranks :=
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

Listing 9: *PageRank* – The final *MapReduce* implementation

```
fn pageRank(links : [[Int]], dampening : Rat, iterations : Int) -> [Rat] {
  var iter : Int := 0;
  var ranks : [Rat] := replicate(length(links), 1 / length(links));
  while (iter < iterations) {
    var outRanks : [[Int] * Rat] := zip(links, ranks);
    var contribs : [Int * Rat] =
      flatMap((links_rank : [Int] * Rat) =>
               map((link : Int) => (link,
                                     snd links_rank / length(fst links_rank)),
                    fst links_rank),
            outRanks);
    var rankUpdates : [Int * Rat] := reduceByKey((x : Rat) (y : Rat) => x + y, 0, contribs);
    var newRanks : [Rat] := replicate(length(links), 0);
    for (link_rank : rankUpdates) {
      newRanks[fst link_rank] := snd link_rank;
    }
    ranks :=
      map((rank : Rat) => dampening * rank + (1 - dampening) / length(links),
          newRanks);
    iter := iter + 1;
  }
  return ranks;
}
```

5   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* [8] 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. [10] use a similar technique for handling recursive function calls. Felsing et al. [7] 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. De Angelis et al. [5] present a further generalised approach.

Radoi et al. [11] 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. [4] and Radoi et al. [11] 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. [9] 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.

6   Conclusion

In this paper we demonstrated how an imperative implementation of a relevant, non-trivial algorithm can be iteratively transformed into an equivalent efficient *MapReduce* implementation. The presentation bases on the formal framework described in [2]. Equivalence within this framework is guaranteed since the individual applied transformations are either behaviour-preserving rewrite rules or equivalence proofs using coupling predicates. The example that has been used as a case study in this paper is the *PageRank* algorithm, a prototypical application case of the *MapReduce* programming model. The transformation comprises eight transformation steps.

Future work for proving equivalence between imperative and *MapReduce* implementations includes further automation of the transformation process.

References

[1] Gilles Barthe, Juan Manuel Crespo & César Kunz (2011): *Relational Verification Using Product Programs*, pp. 200–214. Springer Berlin Heidelberg, doi:10.1007/978-3-642-21437-0_17.

[2] B. Beckert, T. Bingmann, M. Kiefer, P. Sanders, M. Ulbrich & A. Weigl (2018): *Relational Equivalence Proofs Between Imperative and MapReduce Algorithms*. *ArXiv e-prints*. Available at https://arxiv.org/abs/1801.08766.

[3] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm & Peter Sanders (2016): *Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++*. In: *IEEE International Conference on Big Data*, IEEE, pp. 172–183, doi:10.1109/BigData.2016.7840603. Preprint arXiv:1608.05634.

[4] Yu-Fang Chen, Chih-Duo Hong, Ondřej Lengál, Shin-Cheng Mu, Nishant Sinha & Bow-Yaw Wang (2017): *An Executable Sequential Specification for Spark Aggregation.* Available at `https://arxiv.org/abs/1702.02439`.

[5] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2016): *Relational Verification Through Horn Clause Transformation.* In Xavier Rival, editor: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, Lecture Notes in Computer Science* 9837, Springer Berlin Heidelberg, pp. 147–169, doi:10.1007/978-3-662-53413-7_8.

[6] Jeffrey Dean & Sanjay Ghemawat (2008): *MapReduce: Simplified Data Processing on Large Clusters.* Commun. ACM 51(1), pp. 107–113, doi:10.1145/1327452.1327492.

[7] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer & Mattias Ulbrich (2014): *Automating Regression Verification.* In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, ACM, New York, NY, USA, pp. 349–360, doi:10.1145/2642937.2642987.

[8] Benny Godlin & Ofer Strichman (2009): *Regression Verification.* In: *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, ACM, New York, NY, USA, pp. 466–471, doi:10.1145/1629911.1630034.

[9] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky & Mooly Sagiv (2017): *Verifying Equivalence of Spark Programs*, pp. 282–300. Springer International Publishing, Cham, doi:10.1007/978-3-319-63390-9_15.

[10] Chris Hawblitzel, Ming Kawaguchi, Shuvendu Lahiri & Henrique Rebêlo (2011): *Mutual Summaries: Unifying Program Comparison Techniques.* In: *Informal proceedings of BOOGIE 2011 workshop.* Available at `https://www.microsoft.com/en-us/research/publication/mutual-summaries-unifying-program-comparison-techniques/`.

[11] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah & Manu Sridharan (2014): *Translating Imperative Code to MapReduce.* SIGPLAN Not. 49(10), pp. 909–927, doi:10.1145/2714064.2660228.

[12] The Coq development team (2004): *The Coq proof assistant reference manual.* LogiCal Project. Available at `http://coq.inria.fr`. Version 8.6.

[13] Tom White (2012): *Hadoop: The definitive guide.* O'Reilly Media, Inc.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker & Ion Stoica (2010): *Spark: Cluster Computing with Working Sets.* In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, USENIX Association, Berkeley, CA, USA, pp. 10–10. Available at `http://dl.acm.org/citation.cfm?id=1863103.1863113`.